Preface

The standardised hardware description language VHDL has become a widely used language in the electronic design community [24]. It is appropriate to describe hardware at various levels of abstraction that occur in a design flow. The language has proven to be appropriate from switch level modelling to behavioural descriptions.

This thesis considers it useful to enlarge the range of abstraction provided by the language and extends the language to raise its level of abstraction towards system level. Increasing the level of abstraction is regarded as an answer to the question how to improve modelling capabilities and efficiency of the language for large scale designs. The extension adds object-oriented features to VHDL. The extended language has a class-concept that is based on an extension to the existing type concept of VHDL [87], and an inheritance mechanism. Further it supports polymorphism. The extended language becomes a full object-oriented language. The extension comes with a translation concept allowing to integrate object-oriented hardware models in a standard design flow. An important feature of the translation concept is to preserve properties of a hardware description that are intended for a VHDL-based synthesis [125].

The presentation is based on the extension to VHDL, nevertheless, it provides an insight into object-oriented hardware modelling independently from the language extension that is of more fundamental nature. A contribution is to show close similarities between concurrent object-oriented programming, programming of distributed systems, and object-oriented hardware modelling. An analysis of existing languages for system level specification exemplifies the similarities. Among the similarities we focus on the modelling of communication and synchronisation between objects which we believe plays the important role in the design of abstract and re-usable systems. In that context we describe the conflicting between synchronisation constraints and inheritance that is known in concurrent object-oriented programming as inheritance anomaly [111]. A key contribution of this thesis is to provide a solution to the anomaly. Although the solution is formulated for object-oriented hardware modelling using the language extension to VHDL the principle behind it is of a more general nature and can be transferred back to concurrent object-oriented programming [156], where the anomaly originally was detected. In contrast to other approaches to extend VHDL, we do not only provide a re-use concept for data structure and behaviour but we also consider the re-use of synchronisation code which is an essential in the modelling of concurrent systems.

The structuring of the thesis intends a presentation that provides all the relevant information that is necessary to understand the basic problems of object-oriented hardware modelling and to follow the proposed solutions. Especially, the reader is not expected to have any special previous knowledge about concurrent object-oriented programming or object-oriented programming of distributed systems. After briefly presenting the context of hardware modelling we introduce object-oriented design principles. We provide the material from concurrent object-oriented programming and object-oriented programming of distributed systems that turns out to have similarities with object-oriented hardware modelling. As there is no agreement on a common terminology in the object-oriented design community, we give a short definition of the terms we introduce. A glossary of terms can be found in the appendix. Significant parts of the motivation to propose an object-oriented extension to VHDL can be found in an analysis of system level specification methodologies and in the discussion of other approaches to extend VHDL. A very short description of the basic features of VHDL may be skipped by those already familiar with the language.

The language extension to VHDL is introduced with many examples that should allow to understand its main ideas even without a detailed reading of all the previously presented material. The material presented in a discussion on the applicability of the language extension following the introduction of the language and in a discussion about the results of a corresponding case study may be viewed as a large example. The solution of the inheritance anomaly can be considered as an answer to the modelling problems that occurred in that example.

Acknowledgement

The ideas presented in this thesis are the results of my research at Carl von Ossietzky Universität Oldenburg.

I am indebted to my supervisor Professor Dr.-Ing. Wolfgang Nebel for making this research possible. I am grateful for his support and encouragement and I appreciate the freedom that he offered me while pursuing my research.

I would like to express my gratitude to Professor Dr. Dieter Monjau for his interest in this dissertation.

Thanks are due to my colleagues at the VLSI group for providing me with a creative environment and for stimulating discussions, particularly Wolfram Putzke and Martin Radetzki.

I must also take the opportunity to thank Martin Wilmes, Uwe Quiet, Bernd Tangemann, Tiemo Fandrey, and Sven-Olaf Scholz for supporting this thesis by elaboration of a case study and working on a prototype of a translation tool.

Chapter 1

Introduction

Object-oriented specification and design denotes methodologies that are successfully applied in software engineering to build complex software systems [28,44,142,158]. The object-oriented structuring and encapsulation of programs prevents the scattering of related information all over the system. The resulting software becomes robust, reliable, re-usable, and extendable. These are essential qualities for developing large and complex software systems at reasonable cost [128].

The electronic design community is looking for design automation that provides the same characteristics. It needs techniques to improve productivity and to manage large designs. It is a major concern that the reliability of the systems must not suffer from the increasing productivity and that large designs still are controllable in terms of testability and manageability [43].

A promising approach is to use methodologies that promote re-use of tested designs at all levels of abstraction. At system level this includes maintenance during the life-cycle of a product as well as re-use of systems as tomorrow’s subsystems. A form of re-use that becomes more and more important for modelling is to use design components that are purchased as intellectual property. The requirements for designing systems-on-silicon are methodologies that allow technology independent re-use at system level [56].

Like programming, hardware design is about abstraction and decomposition of the design problem. The development of hardware design techniques has been concerned with the introduction of various levels of abstraction [109,140]. Today, the conventional abstraction technique for designing at system level is to use a procedural hardware description language [52]. In the software domain the procedural approach turned out to be inadequate for programming in the large. Enhanced encapsulation and abstraction mechanisms as they are provided by the object-oriented concepts were identified to be necessary for successfully handle large software systems [29]. The object-oriented paradigm as an approach to solve complexity and productivity problems in software engineering motivates the idea to apply it to hardware design.

We believe that only a change of design paradigms for modelling at system level can achieve the required improvements of design productivity. The intended development is to transfer the enhanced encapsulation and abstraction concepts of object-oriented modelling and its benefits to hardware modelling.

It is a goal of the thesis to illustrate how such a change of paradigms can gain the necessary improvements. We show that adding object-oriented features to the hardware description language VHDL performs the desired change of paradigms. We propose a small extension that can be integrated in existing VHDL-based design flows. Different to other proposals to extend VHDL, the approach in the thesis combines the language extension to VHDL with a new modelling methodology. The basic principle of the methodology is to carefully take the interaction between object-oriented modelling and parallel hardware designs into account. The new concept for modelling the interaction is viewed as a keystone for successfully developing robust, re-usable, and expandable hardware designs.

The next chapter covers in outline the main goals of modelling in hardware design. It contains a discussion on the importance of hierarchy and abstraction in the design of robust and re-usable hardware models. The chapter brings out the deficiencies of procedural hardware description languages for system level modelling. It explains the modelling challenges for such systems and the resulting necessity for a change of design paradigms.

Object-orientation is proposed as the new design paradigm for modelling hardware at system level. The principles of object-orientation are provided in Chapter 3. The chapter discusses the usefulness of various object-oriented concepts for the re-use of models. Concepts for modelling parallel and distributed systems and their integration into object-oriented concepts are then introduced. The integration is identified as a key issue in object-oriented hardware modelling.

Chapter 4 analyses the potential of system level specification languages for use in hardware design. Similarly Chapter 5 presents an analysis of hardware description languages and their potential for system level specification. Particularly, the chapter provides a rather complete overview about object-oriented extensions to the hardware description language VHDL. The deficiencies of the presented languages are discussed. In essence, the deficiencies concern the integration of concurrency concepts into object-oriented concepts.

The following three chapters are the core of the thesis. Broadly speaking, they provide a solution to the integration of concepts for concurrency, distribution, and object-orientation in hardware design. The solution is based on an object-oriented language extension to VHDL and a corresponding modelling methodology.

Chapter 6 introduces the language extension. The facilities for object-oriented modelling of parallel hardware systems are presented. This covers a translation concept from the language extension to standard VHDL that allows to integrate the language extension in a VHDL-based design flow. Chapter 7 rounds off the introduction of the language extension by an example. The example illustrates the major themes of modelling concurrency and of object-oriented modelling.

Chapter 8 provides a detailed discussion on these themes. Finally, the chapter presents the solution to the integration of concepts for concurrency, distribution, and object-orientation in hardware design.

Chapter 2

Modelling

In hardware design a model is a representation of a system or a part of it. It allows a designer to form a picture of the system he or she has to deal with. A model has to serve several purposes during the design process. The modelling methodologies have to reflect these purposes. The chapter presents the main goals of modelling in hardware design. The focus is mainly put on the modelling of digital hardware systems. We derive some basic requirements which models must meet and we discuss the basic principles of modelling which are used to achieve these goals.

2.1 Modelling Goals

A general goal of a model is to collect information about a system and to structure the information in a way that it is accessible to people who are interested in the system and its development. Typically there are several persons involved in the design and development of a hardware system with different knowledge and skills. Their vocabularies often do not match and even if they use the same word it might have different meanings. In such a case a model can serve as a way to exchange information between the different persons involved in the system design if the model is described by a well defined formalism which is understood by all the involved persons [158].

A model can be used not only to exchange information about the system at a certain point of time in the design process but also during the whole design process and further on during the life-cycle of the system. The goal of such a model is to serve as a reference during the design process and a documentation of the system during its life-cycle [79].

A model is a representation of a system which is used to describe very different aspects of the system. A modelling technique to show a particular aspect of the system is called *view*. Sometimes we also call the model which shows the particular aspect view. Typical views which are used in hardware design are timing models, behavioural models, delay models, structural models, value systems etc.

It is a quality of a model that it tries to avoid any inconsistencies in the information about the system. To describe what is thought as consistent in a special modelling context a consistency model is used. We could think of such a consistency model as a kind of meta-model. The meta-model deals with information in models and their relation to each other and thus should guarantee the consistency of the different views of a system. Very often such relations in a consistency model can be described as a set of conditions which must be fulfilled.

2.1.1 Specification and implementation

Two kinds of information are normally distinguished in a model. The first one is the information given by the customer of the hardware system. We call this kind of information requirements. The other kind of information is the one added by the designers which we call implementation knowledge or implementation data. Depending on the kind of information a model contains it can be either classified as requirement model or as implementation model. Most of the modelling methodologies make this classification although they may use different terms. A widely used term for the requirements model is specification. Also the modelling of a requirement model is called specification.

Typical requirements in the design of hardware systems concern the functionality of the system, the performance, the area of the system on a chip, the power consumption. Other requirements may contain components which are to be used in the system or definitions of interfaces to other systems in the form of protocols, etc. It is obvious that models of the components, protocols, etc. are required to model these requirements.

The final goal of the modelling is to design a model which contains all the implementation data that is required to produce the system and which is consistent with the specification. A requirement model can be very often replaced by an implementation model during the modelling process. This leads to the notion of transformation. Designing is the transformation of a requirement model into an implementation.

The overall transformation consists of a sequence of iteratively performed transformations. We refer to such a transformation as a design step. In such a sequence an implementation model of a transformation may become a requirement model of the next transformation. After each transformation the resulting model must be consistent with the original one. Again, the consistency can be modelled by a kind of meta-model. The conditions describing the meta-model together with the original model form new requirements.

Other meta-models which are used in hardware design describe the design steps that are required to develop an implementation from a specification. Such meta-models are called *patterns* [67]. A hardware design flow can be viewed as such a pattern.

2.1.2 Exploration of design space

In each design step a designer can build various models which differ in the way the given requirements are met. The transformation techniques used in a design step normally do not automatically guarantee that all requirements are met. This has to be analysed in additional design steps. Due to the complexity it is normally not possible to perform an analysis in all the details. To select a model out of a set of alternative models with only limited information is one of the most problematic issues in design. To analyse various models and to collect as much information as possible about them is called the *exploration of the design space*.

It is a goal of the modelling to enable the exploration of the design space by providing information in a model which is relevant for analysing the requirements. If one model from the design space is chosen in a design step this design decision should be documented as part of the model in a way the decision is understandable. In other words, the design decision should be treated as a special kind of implementation information.

During the design process an initial specification normally undergoes some modifications. Very often it turns out during the modelling that there are some missing requirements which have to become part of the specification. To make the effort not worthless which has been spent so far in the modelling of the system a modelling methodology should support the introduction of the new requirements into the model without needing complex re-design steps of already developed models. The modelling goal is to get a robust model with respect to modifications.

2.1.3 Re-use of models

Re-use of existing models and meta-models is the key idea to reduce the design effort and improve design quality[69]. For example, many hardware systems have requirements considering the same views, e.g., timing behaviour or power consumption. In such cases it makes sense to think about a model of time or of power which can be re-used in many systems. To enable such re-use the modelling methodology must allow to describe such models independently from a concrete target system. We call such a model *target independent*. The example shows that a modelling methodology must allow to generalize about the concrete target system to a whole class of target systems to enable re-use. The *generalization* concept is relevant in many variations for re-use.

There is a great variety of technologies which are used to produce the hardware systems and the technologies are rapidly evolving. Therefore another important concept for re-use of models is to generalize about the technology. A target independent model then represents the *intellectual property* which can be kept when re-targeting the system to another technology. In other words, a hardware modelling methodology should support the modelling of technology independent models for re-use.

There is a wide variety of different specification and modelling techniques in hardware design. Among these techniques hardware description languages (HDLs) are successfully used for many modelling issues in a design flow.

2.2 HDLs in a Design Flow

This section explains how hardware description languages are used in a design flow to model hardware systems. It describes how HDLs are used to meet the different modelling goals.

We denote a well defined language which was designed to document hardware systems, in particular systems consisting of integrated circuits, as hardware description language. Different to more general purpose modelling languages, HDLs have some built in models and meta-models which can be used in many hardware descriptions independently from a concrete system.

Such models are for example timing models, delay models, value models and corresponding conflict solution models, transition models, and models for functional primitives. The generalization about a class of models enables the re-use. The scope a language allows to generalize about with its built in models is just its intended application domain.

It has been mentioned that consistency models are used in hardware design to describe the consistency between various views of a system or between models of different design steps. Basically, there are two concepts supported by HDLs, a verification concept and a validation concept.

Following the verification concept means that the HDL has a formal semantics which allows to reason about system models. The consistency model is a kind of formula and the consistency check itself is a mathematical proof. The problem with such a concept is the in many cases limited model complexity it can handle compared with the validation concept.

Languages which support the validation concept have a *simulation semantics*. Simulation semantics implies that there is a mapping from models which are described by a HDL to an algorithm. Simulation means to execute the algorithm on some stimuli data. Such HDLs are very similar to programming languages except to the fact that they have the built in models for hardware design. One of the most important built in concepts which are provided by HDLs are timing models. They often include concurrency concepts to express parallelism in hardware systems. The consistency models consist of a set of rules describing how to map value models, delay models, and timing models of the different simulation models.

Some HDLs have a *synthesis semantics*. That means that the language contains a meta-model how to transform an existing model which is described by the HDL into a new model by adding information to the model. The consistency model is embedded in the meta-model describing the transformation. Typically, it is just a consistency model which describes a set of possible transformations for a given model. To select a concrete transformation out of many others is the exploration of the design space as mentioned in the previous section.

An interesting aspect of simulatable models is the possibility to communicate about the model by simply looking at the simulation results without studying the model in detail or even without knowing the HDL which was used to describe the model. Such a model can be a good communication interface to a customer [66].

Designers who directly access the models written in a HDL must be able to analyse and understand information of a model even if it is extensive and complex. A designer also must be able to easily add new information to the model and modify the existing model without completely re-analysing or rewriting it. The language must provide corresponding concepts to structure the information, modify it, and manage its complexity.

The next section presents the two main techniques to manage the design complexity.

2.3 Introducing Hierarchy and Abstraction into a Design

The information of a model is decomposed into manageable chunks to break down the complexity of the model. If we first look at specifications this means that a set of interrelated requirements is decomposed into single requirements. The design steps then deals with each single requirement separately. This may allow to design a model in a design step that is an optimal solution with respect to the single requirement, however, in most cases it is only a sub-optimal solution with respect to all requirements if it is a solution at all. There is often no efficient technique which allows to decide how to successfully decompose the model. The design decision how to decompose the system is then based on heuristics or simply the designers experience. Very often a costly search in the design space is required to find an appropriate solution which meets all given requirements.

2.3.1 Hierarchy

If a decomposition technique is iteratively applied to a design then this establishes a hierarchy in the model. We can think of a model consisting of sub-models which in turn consist of sub-models. This establishes a *has-parts-relation* between the (sub-)models [1]. We also call this relation *has-a relation*. Examples are functional decomposition or decomposition of tasks into parallel tasks. Top-down design flows establish the has-a-relation by applying the decomposition technique on the specifications. The same has-a-relation can be established by composition of models from existing (sub-) models[1] in a bottom-up design flow.

After the decomposition or composition the separate sub-models are interrelated to each other. In many design techniques this is the weak point to handle these relations properly. A typical problem is that a local modification in a sub-model causes inconsistencies with other sub-models due to their

1. We subsequently use the term model for both, models and sub-models.

interrelation. This requires a re-analysis and probably a re-design of other sub-models. The decomposition is not stable under local modification of models. An example is the decomposition of a system into a data path and a controller. Introducing new elements to the data path may require a re-analysis and re-design of the controller.

2.3.2 Abstraction

An approach to solve the problem is to provide the information about a model which is required to understand the interrelations to other models and to omit the information which is not relevant for that aspect. We could think of the model with the reduced information as a special view. The view abstracts from the details of a model. We call both the omission of information in a model and the resulting view *abstraction*. An abstraction hides the details of a model from other models. Only abstractions are used to model the relations between models. *Encapsulation* is the relation between an abstraction and the corresponding model with the detailed information.

If an appropriate encapsulation is chosen a model can be re-used. Appropriate here means that the abstraction must contain all information which is required to use the model in various contexts.

Another concept of abstraction is to omit information which is not present at that stage in a design flow. More detailed information has to be added in further design steps. In this sense every model can be seen as an abstraction of the system it represents.

Abstraction along with hierarchy is the key to manage the design complexity. We now discuss examples where hierarchy and abstraction are used in a design flow and how they are supported by HDLs.

2.3.3 Levels of abstraction

Different levels of abstraction are distinguished according to the concepts which are abstracted in the models. The levels correspond quite nicely to design steps in a hardware design flow. A design flow which differentiates six levels of abstraction is given in [140]. The levels are system level, algorithmic level, register transfer level, gate level, switch level, and electrical level. The latter is often also called circuit level.

2.3.4 Bottom-up design

If we look at the levels from a bottom-up perspective we can see that there are concepts to aggregate the basic elements from the circuit and switch level. Encapsulation concepts allow to use the abstractions of the aggregates at the gate level. The concepts are successfully applied in cell libraries of ASIC vendors. A cell in the library abstracts implementation information and at the same time it abstracts behaviour. The models generalize about the timing model. As previously mentioned generalization allows a separation of views which is in this case a timing view and a functional view. A very good example how such a separation can be done at cell level is described in the VITAL standard [88].

The abstractions with the timing view and functional view separated allows their re-use independently from each other. This makes functional decomposition possible which uses the functional view of the gates as submodels. Starting from a functionally decomposed model timing issues can be considered. The timing model can be re-used in scheduling considerations for more complex models which are built as aggregation of gate level models because the timing abstraction is independent from the actual gates. The timing models for the gates only differ in the actual values. For the scheduling it is not relevant which behaviour i.e. which operation is performed by a gate. This abstraction concepts allows low modelling costs of functional and timing models at gate level and a very high re-use of the gate level models.

Attempts to apply these modelling and abstraction concepts to complex parameterized macros in general purpose ASIC libraries failed. Today, only memory cells and analog cells are modelled as macro cells. There are several reasons for that. A macro may have some complex internal state. In that case a simple functional abstraction does not work and it becomes difficult to generalize the timing behaviour so that it is independent from the functionality and can be re-used. Today's HDLs do not provide abstraction mechanisms which allow a designer to understand the behaviour and timing model of a complex macro cell from its abstraction. To re-use such a model either the implementation of the model has to be analysed which increases the cost of re-use or additional documentation about the model has to provide the missing information of the abstraction. The improper abstraction is also an obstacle for the designer to efficiently find the desired model for re-use. Another reason why the modelling of compex parameterized macros fails is the generalization mechanism provided by today's HDLs. The mechanism is based on generic parameters. It requires the designer of generic parameterizable models to think of all possible future application scenarios of the model and to consider them in the model. Abstraction here may introduce quite an overhead into the model and make it more complex and less optimal for a particular application. The modelling effort is very high due to the very large number of possible combinations of actual parameters which have to be considered in a validation of the model. Large parameter lists are difficult to manage by a user. As a result there is only a limited re-use of parameterized macro cells while the modelling effort is high.

To summarize, we can say that there are limits in the useful modelling of complex macro cells due to missing abstraction and generalization concepts in the HDLs.

2.3.5 Top-down design

To overcome the limits of bottom-up approaches abstraction mechanisms are applied in top-down design flows. In a top-down design step information can be added to a model through synthesis. This approach is used for place and route, technology mapping, and logic synthesis. It also can be used for higher levels of abstraction with register-transfer-level synthesis or behavioural synthesis.

In the top-down design flow the original model can be interpreted as an abstraction of the more detailed model which is at a lower level of abstraction. This approach can be used if there is tool support for the automatic synthesis of the abstract model. In that case it is possible to re-use the consistency model which is defined between the abstraction and the synthesis result. At the same time design knowledge is re-used. We can think of this knowledge as a meta-model which is encoded in the synthesis mechanism.

To enable re-use of both, the meta-model and corresponding synthesizable models it is necessary to provide a generalization concept about target technologies. This allows the modelling of technology independent models which can be re-targeted to a new technology by providing generic technology information to the synthesis tool in form of some technology libraries.

As described in the modelling goals, a synthesis mechanism should be robust. That means that a local modification in the abstract model causes only local modifications in the synthesis result at the lower level of abstraction.

2.3.6 Register-transfer-level modelling

Today, technology independent design of synthesizable models is applied at register-transfer-level. In many cases the synthesis concept works quite well at that level[40]. The data model is some kind of re-usable bit-vector representation which normally does not introduce any instability in the synthesis mechanism. What causes problems are modifications of the abstraction which affect the functionality or the timing behaviour of the system. The timing behaviour may change locally due to local modifications of the functionality. However, the modification may cause a global effect in the synthesis mechanism. For example, introducing some components in a data path at register-transfer-level may require the re-analysis and design of the corresponding controller. We can identify two reasons for that. First, the state model is built separately from the functionality although they are interdependent i.e., the encapsulation concept at register-transfer-level is weak. The second reason is that modifications of the functionality may change the timing behaviour of the system. However, this modification of the timing behaviour is not reflected in the abstraction of the functionality. In other words, if we interpret the components in a register-transfer-model as limited resources we could say that there is no appropriate abstraction of the scheduling.

2.3.7 Behavioural modelling

Behavioural modelling and synthesis which is used at algorithmic level provides an abstraction of scheduling. HDLs[2] for modelling at that level have some procedure-like concepts to abstract, encapsulate and decompose behaviour[110]. A procedure models an algorithm. It consists of an interface and an implementation. The interface only allows to pass information to the implementation via parameters and thus encapsulates the model in the procedure implementation. The idea is to look only at the interface to know what the procedure does and where it can be used. In other words, the interface abstracts from the implementation. Basically, the implementation consists of some kind of local variables and a sequence of statements. The variables can be used to model resources which are hidden outside the procedure. Causal relations which are implicitly given by the sequence of statements including some control statements abstract from timing relations concerning the com-

2. Although the focus here is on HDLs we can apply the following considerations to procedural languages in general.

mon access to a resource. The sequence of statements abstracts the scheduling.

Another important concept of modelling at algorithmic level is the possibility to hierarchically decompose procedures. This allows to functionally decompose an algorithm into manageable chunks. Each procedure which models such a chunk can be synthesized separately and then assembled to the complete algorithm.

The advantage of the procedural modelling is the possibility to keep modifications of behaviour in a model local. If an algorithm modelled by a procedure requires a modification it can be performed locally and it becomes effective globally at each place in the model where the abstraction of the modified procedure is used.

An example of an application domain is digital signal processing (DSP). A typical computation in DSP is a filter operation which operates on an input sequence to produce an output sequence. On each sample in a sequence the same sequence of operations is applied. Procedures are appropriate to encapsulate the sequence of operations. The procedure can be re-used for each sample. If modifications of the model are required then the encapsulation keeps modifications of the filter operation locally.

The sequence of operations in the procedure abstracts from the scheduling of the linear operations in the filter. The timing relations between the operations, i.e. unit delays, are modelled as causal relations.

The example shows the possibilities of procedural modelling and at the same time its limitations. It is not possible to encapsulate the internal state of the filter in a procedure because internal variables do not exist after the execution of the procedure. That means that state variables must be modelled as global variables. The disadvantage of global variables and thus the procedural modelling is that modifications of the data cannot be kept locally. The reason why procedural modelling can be nevertheless successfully used in DPS is that typical modifications of a model do not significantly concern the data. All the data is modelled by some numeric types. Modifications only concern some accuracy issues.

The example shows the modelling limitations namely the missing robustness with respect to modifications of data. The example also illustrates that procedural modelling is appropriate to encapsulate functionality and that it allows to abstract simple timing relations like unit delays.

2.3.8 Modelling parallelism

If a model contains more complex timing relations a model consisting only of sequential statements often is not appropriate[66]. This requires a modelling methodology which is well-suited for modelling parallelism. A typical approach is to use a modelling concept which is considered orthogonal to the procedural modelling. Orthogonal means that a model which is decomposed into parallel tasks may contain any hierarchy of procedures. Timing relations between parallel tasks are described by synchronisation and communication models. HDLs which are designed to model at that level of abstraction provide a set of built-in models for communication and synchronisation.

The models abstract mechanisms to synchronize activities in parallel tasks and they abstract the exchange of information between tasks. There is a large number of very different models. On the one hand there are very powerful models like for example a rendezvous-concept and on the other hand there are mechanisms like low-level signals. The powerful models abstract very complex mechanisms. They may require mechanisms like for example routing strategies or queues which are costly to implement in hardware. In many cases such a complex mechanism introduces quite an overhead in the model where the synchronisation model is used. There is no direct mapping of such abstract models to lower level models. This is an obstacle in introducing a synthesis semantics to a language which has such complex built-in models.

For that reason today's HDLs tend to support only low-level synchronisation models. They allow a direct mapping to a hardware implementation. However, these low-level models do not meet the requirements for modelling at system level. The solution is that the hardware designer has to build complex communication and synchronisation models from low-level constructs.

The problem with procedural languages are the missing concepts to encapsulate and abstract user defined[3] synchronisation and communication models. Apart from the data which is transferred in a communication a synchronisation model contains quite complex control information but as mentioned above there is no possibility to encapsulate the control data.

There is also a more general problem of procedural modelling in the context of parallelism. If a synchronisation model is used in a procedure the procedure interface does not contain any abstract information about the model.

3. User defined means the synchronisation and communication is modelled by the hardware designer as opposed to built-in models.

However, the synchronisation model causes an activity outside the procedure namely the synchronisation with some other activities so that the model breaks the encapsulation. In other words, the procedure interface is not any longer an appropriate abstraction of a procedure. We have to conclude that procedural modelling is not orthogonal to modelling parallelism.

2.3.9 Deficiencies of procedural HDLs

Another unsolved issue of today's HDLs is a missing re-use concept for synchronisation models. The idea is that an abstraction should support re-use by allowing to generalize about the data which is used for the communication and the activities which are performed during the synchronisation. The synchronisation model becomes a meta-model.

As a result we can say that today's procedural HDLs are not appropriate to model complex synchronisation. We also can conclude that a modelling methodology and a corresponding HDL should provide an abstraction and encapsulation concept for data and a generalization concept which allows to model re-usable meta-models.

The explanations and examples given in this chapter show that bottom-up and top-down concepts for hardware modelling with today's HDLs both have limitations which restrict their use at system level. The limitations mainly concern concepts for abstraction which allow to break down the complexity of models and which allow the re-use and modification of models. What has to be improved are concepts for generalization and abstraction at higher levels of abstraction.

2.3.10 A new modelling methodology

The goal of the thesis is to develop a methodology for the modelling of robust and re-usable specifications at system level. The methodology is based on a HDL with simulation semantics to support validation of the specifications by simulation. It should be possible to add a synthesis semantics to the HDL which allows to integrate the methodology in a high-level design flow.

The approach in the thesis is to investigate the concept of parallel object-oriented modelling which is used in software engineering and to analyse its suitability for the hardware design at system level. Part of this work is to develop an object-oriented modelling methodology for hardware design and a corresponding HDL.

The approach is motivated by the idea that object-oriented techniques provide concepts for generalization and abstraction which promise to solve some major modelling problems including the ones discussed above. Advantages as they are known from software domain include the design of robust, maintainable, and re-usable models. The re-use concept of object-oriented modelling allows the modification of re-used models. A proper encapsulation concept allows to break down the complexity of models and to reduce its maintenance costs.

While there are great benefits in using object-oriented methodologies, nevertheless we have to be aware that there are still some unsolved modelling problems. They have to be considered when adopting the methodology for hardware design.

2.3.11 Modelling challenges

One of the problems that has turned out is that re-use of models is only possible under certain circumstances and only to a certain extent [105]. The idea to re-use models at system level as pluggable components in more complex models requires well defined interface concepts and a very complex communication structure[4]. An example for such an approach is CORBA (Common Object Request Broker Architecture). The high costs of designing the communication is an obstacle for generally using such a re-use concept for all kind of application. In fact it is mainly used for standard applications.

Another unsolved issue is the question what is the best way to integrate parallelism and object-oriented modelling. There have been a number of very different approaches. However, there is no final answer to that question yet. It is still a topic of research [115].

Although our modelling methodology aims at hardware design we try to address some of these issues in such a general way that the solutions can be used in the software domain too.

2.4 Overview

To introduce the terminology which is used in this book and to give a foundation to understand our modelling methodology the next chapter presents

4. It is interesting to note, that recently an initiative has been started with the Virtual Socket Interface Alliance to introduce such a concept of standardized communication mechanisms between models to hardware design.

object-based and object-oriented modelling concepts. It gives an introduction how to model parallel systems and it discusses the problems when integrating parallelism and object-oriented modelling. Chapter 4 is a survey on specification languages which allow to specify systems at a high level of abstraction. It analyses their potential for use in hardware specification. Chapter 5 is an overview of HDLs. Particularly, it explains the most important features of the hardware description language VHDL and covers the basic concepts of the language from a system level designer's point of view. It explains its main limitations with respect to system level modelling. The chapter also discusses existing language extensions to VHDL which try to overcome the limitations. It explains why there are still some unsolved issues. Chapter 6 presents our modelling methodology which is based on an object-oriented language extension to VHDL. Chapter 7 illustrates the methodology by an example. The example reveals some principle problems of object-oriented hardware modelling that concern communication and synchronisation modelling. Chapter 8 provides very detailed material on how to solve these problems. The concluding chapter provides the results of the earlier chapters in a very condensed form.

Chapter 3

Principles of Object-Orientation

In the previous chapter, we introduced the term object-oriented modelling and announced it as a key issue which shall be investigated in the context of hardware design in this thesis. However, we have so far not explained what the term object-oriented means. It turns out that although many methodologies claim to be object-oriented there is no common understanding of what object-oriented is. It is therefore impossible to give a precise and generally applicable definition of object-oriented modelling. In this chapter we explain what we think are the basic characteristics of object-oriented concepts. We describe the terminology as it is used in the context of the modelling methodology we are going to present in this thesis.

It seems best to start with object-based modelling philosophy on which object-oriented methodologies are based. We then dwell upon object-oriented concepts. We discuss the modelling of parallel systems and try to give an object-oriented view on process-based modelling and more general on object-based parallelism.

3.1 Object-Based Modelling

Object-based modelling concepts evolved from the idea to improve the abstraction of procedural modelling. A first improvement was achieved by putting related procedures together in a module. A module provides the procedures as services to a client. It consists of an interface and an implementation which separates the declaration of a procedure from its implementation[1]. The interface of a module contains the procedure declarations. It is an

1. In the context of procedures and modules we also call the implementation body.

abstraction of the services which a module provides. Only the abstraction is visible to clients the implementation is hidden. The limited visibility from the clients perspective is called *encapsulation boundary* [173] or *abstraction boundary* [172]. At the same time there is a boundary from the implementation's point of view concerning the visibility of entities from within the module implementation. It is the boundary of names, identifiers, references etc., that are visible when looking outward from within the module. This boundary is called *distribution boundary* [173]. Typically it includes the interfaces of those modules of which it is a client.

So far, the sketched module concept only provides services on transient data. After the execution of a service the module does not remember the effects of the execution. It has no memory.

3.1.1 Objects

To overcome this limitation some module concepts allow to declare variables in a module body which are global with respect to the procedure implementations and thus accessible within the module body. We call such a variable *state variable*. Their values are called state. The data structures which implement the state variables and which store the object's state are referred to as *structure*. State variables are hidden from a client. The variables are the memory of the module and the only access to the memory is via the services provided by the module. State variables remember the effects of the execution of the services. The services form what is described as behaviour.

We call such a module which encapsulates structure and behaviour *object*. Structure and behaviour are *properties* of the object. Subprograms which implement the behaviour of an object are often called *method, operation,* or *primitive operation*. We could think in this context of a more general concept to implement the structure of an object. Such a more general concept may allow for example a structure to be implemented by other objects. A state variable turns into a container of an object. If we have this more general notion of a container we call such a container which is part of a structure *attribute*. If the object is inextricably linked to the container we call the object itself attribute. The relation between an object which contains another object to implement its structure and the included object is the has-a relation or has-parts relation that was introduced in Section 2.3.1.

3.1.2 Clientship

Modelling a system with objects means to describe objects and the relationship between them. The basic relationship between objects is the *clientship*. It describes how a client object uses the operations of a server object. An example of such a relation is the has-a relation. In the relation the server is encapsulated by its client. The server can be regarded as a private resource of the client [17]. We call such a server *sub-object* or if we want to emphasize the encapsulation we call it an *exclusive sub-object* of the client [57]. In another clientship an object is a server which might be shared by several clients. We call the clientship *use-relation*.

The interface of an object can be regarded as a contract between the object and its client [172]. The contract specifies both, the object's and the client's responsibilities and probably some general promises about the object. In a general sense, responsibilities of a client are called *preconditions* responsibilities of the object in responding are called *postconditions*. General promises about the object in form of a condition which always holds are called *invariants*. Preconditions, postconditions and invariants are *assertions*. A design approach which uses such assertions to describe a client server contract is called *design by contract* [159]. Assertions typically refer to information which is affected by the execution of an operation. This includes especially a state of the server. To take effect the contract must be visible to a client. As a consequence information about the structure of a server is made visible to the client. In other words, an assertion referring to the internal structure of a server circumvents in a way its encapsulation. It moves the abstraction boundary to include information about internal structure of an object[2].

An important property of assertions is to introduce redundancy into a model. Assertions promise or specify properties of objects which have to be implemented by an object's implementation. Assertions as part of the specification describe what an operation does i.e., its behaviour. The implementation describes how to model the behaviour. To have a solid model it is necessary to check the consistency between these two aspects of the same

2. If information about an object's structure becomes visible outside the object it does not necessarily mean that the structure is directly accessible. This must not be confused. Later on, we shall point out this distinction where it is relevant. In the design by contract approach described above, encapsulation is not violated with respect to accessing the internal state of an object.

behaviour. This can be done statically by analysing the model and dynamically during the execution of the model.

The idea of statically analysing the consistency between assertions and implementation is used as a concept for verification. The assertions become *verification conditions* [20][3]. Checking the consistency means to prove the conditions. In other words, it means to check if the implementation fulfils the specification. Instead of referring directly to the structure of an object the verification conditions refer to information which describes the internal structure and which is part of the interface. The information about the structure is duplicated in the interface and thus redundant information. Redundancy is used to achieve reliability. Although it would be desirable, there is no automatic proof mechanism. In most cases there is at the most tool support for the manual verification of the conditions. This is the reason why such an in theory interesting approach to abstract the behaviour[4] of an object in a specification is not more often used in practice.

Taking the contract point of view on an interface means that a contract is a special kind of abstraction on how to interact with a server. If a client wants to use a service of the object it has to send a *message* requesting the service to the object. This requires a message passing mechanism which allows to identify and address the server. A mechanism to identify objects is to give them distinctive names. These names become part of the object's interface, i.e. its abstraction and can be used to address them. For example, to address an exclusive sub-object it might be sufficient to simply use the object's name as an address.

A message has to be conform to the contract between client and server. The rule that governs the communication between them is called *protocol* and the contract is a part of it. The contract between the client and the server models the clientship.

3. The approach presented in [20] is based on abstract data types (ADTs) which define a set of objects with a set of operations that characterize the behaviour of those objects. The verification conditions are modelled as proof annotations which abstract the implementation of the operations.

4. We shall see later on, how this insight into concepts of abstractions which use conditions can be applied to considerations about the abstraction of concurrent objects and their synchronisation.

3.1.3 State-oriented view

If we regard an object with its internal state as a state machine then we talk about causing an event when a client sends a message and an object may react to an event when it receives a message. In this state-oriented view an execution of an operation which may change the value of the object's attributes performs a state transition. The event which causes the transition is called *trigger event*. Modelling languages which support a state-oriented view may provide dedicated language constructs to model states and state transitions. In other words, they have a built-in meta-model of states. This separates two kinds of states in the modelling: States modelled by the built-in meta-model and states modelled by explicit attributes. State transitions are modelled by the built-in meta-model and by the assignment of values to the attributes. Such a modelling language splits up an operation by separating the built-in state transition from the rest of the operation. We call such a rest of an operation *action*. From a modelling perspective the separation leads to the notion that a message causes a state transition and the built-in state transition meta-model then causes the execution of the corresponding action.

An action may contain assignments of values to attributes and thus may change an object's state. An important difference in the modelling possibilities between built-in state transitions and actions is caused by different distribution boundaries. The distribution boundary of a built-in state transition is the object whereas an action may have a broader boundary. This allows to model side-effects in an action by sending message to other objects.

Another concept of built-in state models is the so-called *activity*. Similar to an action an activity can be viewed as a part of an operation. However, its execution is not caused by a state transition but by entering a state which is modelled by the built-in state model. We call an object which has one or more activities an *active object*. Objects which are not classified as active are called *passive objects*.

The distinction between action and activity becomes relevant if we model parallel systems. In a model of such a system state transitions of the built-in model and corresponding actions are not interruptible by trigger events causing new transitions. Such trigger events are either queued or ignored during the execution of an action. This is different to activities. If an object is in a certain state a trigger event may cause a state transition even if an activity is executed. Such an event is able to interrupt the activity.

In a parallel system a built-in meta-model of states can be used to model just those states which contain the information which is needed only for synchronisation purposes. The resulting state machine is an *implementation*[5] of the protocol that describes the synchronisation. We shall look at the modelling of active objects and protocols in the Section 3.3 in more detail.

Given such a state-oriented view of an object it is possible to transform it in an equivalent model without built-in state model which uses only explicit attributes to model state transitions. Equivalent here means that both models behave in the same way from a clients point of view. In a pure sequential model this transformation is quite straight-forward. In a parallel system it can be very difficult. It requires the explicit modelling of the built-in mechanism which may include such complex mechanisms like queuing of events/messages or an interrupt mechanism for activities. An object which represents its message queues is called meta-object and the modelling style reflective [172].

To perform the transformation vice versa states have to be identified which are modelled by some attributes values and by the protocol of the object. Depending on the set of values which an attribute can contain the resulting state machine may become very large.

So far, we have introduced the term object and we have characterized different views on objects. Corresponding modelling methodologies and languages are object-based if they support the modelling of objects. Such languages allow the decomposition of a system into objects which resemble concepts or objects from the real world. This makes object-based methodologies a good candidate for modelling at system level.

3.1.4 Prototype

In the modelling of a system we very often can identify several objects with the same properties. The objects have the same structure and behaviour. What differs are the actual values of their attributes and their relation to other objects i.e., the concrete context in which they are used. The idea of re-using a description of an object in another context suggests itself. Such an object which can be re-used is called a prototype or exemplar.[104, 106]. A prototype serves as a model for the other objects.

5. It is important to note that the state machine here is not necessarily part of the abstraction i.e., the interface of the object.

3.1.5 Class

The next idea is to abstract from the concrete context an object is used. A key concept is the classification of objects with the same properties. Instead of describing a concrete object a class of objects is modelled by describing the common properties of the objects. In this view a class is a set of objects with the same structure and behaviour. The description of a class is a meta-model which serves as a template for objects. Although this sometimes might be a little bit confusing such a meta-model is also called class. A class has the ability to generate objects or instances. An instantiation declares an object to belong to a certain class and probably to have a name.

A class consists of an interface and may consist of an implementation. They describe the interface and implementation of the respective object. A class which does not have an implementation of its operations is called *abstract class* as it only provides an abstraction of its behaviour. It is obviously not possible to directly generate an instance of an abstract class. As we shall see later on an abstract class may be used to describe the common properties of a set of classes.

The class concept allows to abstract the relations between objects by modelling relations between classes. Again, the basic relation between classes is the clientship. The variants of the relations between classes are analogue to the relations of objects.

A has-a relation between objects is abstracted by a has-a relation between classes. Similar to objects the structure which implements the relation is modelled by attributes. However, the attribute is not an object or a container of an object but a template of it. A class not only abstracts the services its objects provide it also has to describe the services they require. This leads to the notion that certain classes of objects are characterized not merely by services they provide but also by services they require at the same level of abstraction from other objects [17].

A difficult problem in that context is the abstraction of the distribution boundaries. If a class requires a service of an object which is not part of the class's structure to implement its behaviour an abstraction of that object is necessary which can be replaced in various contexts by the required object. There are various different solutions to that problem in different methodologies and languages. The common principle is to use a kind of interface as abstraction of the required object and to link the interface in an instantiation to a concrete object. The interface is part of the client object and abstracts a server object outside the client and makes it visible from within the client.

A more complex situation occurs if the modelling methodology or language allows a link to change dynamically during the execution of the model. The link is typically modelled by a reference mechanism, i.e. by pointers. In such a case the message passing mechanism does not use a static object name to address an object but the reference. A state variable which might store such a reference is called *instance variable*.

An assignment of a new reference to an instance variable might be the result of an operation's execution. Then the operation dynamically changes the distribution boundaries of an object. The unpleasant thing is that the interface no longer abstracts the services an object requires although this might characterize a class as mentioned above.

The advantage of modelling relations between objects by instance variables is the flexibility to modify the relations during the execution of the model. It is even possible to allow the generation of new objects during the execution and to identify them by references which are generated simultaneously to the object. The concept allows for example, to model a use-relation by an instance variable in a client which references the server object. The operations of the server are invoked by messages sent via the reference. Although storing the reference in an instance variable might at first glance look like a has-a relation it is a use-relation because any other clients may contain references to the same server object. To design a class which contains a reference to a server as an instance variable it might be useful to have an abstraction of the server. It might be possible that at that point of time in the design process the corresponding implementation of the server does not exist. If the modelling methodology allows to model the implementation of a class after its abstraction was used in other classes we call such a technique *deferred implementation of classes*.

3.2 Object-Oriented Modelling

As mentioned above, relationships between objects dynamically may change e.g. by instance variables which refer to different objects during the execution of a model, i.e. during run-time. The objects may belong to different classes which use the same abstraction of an operation for different implementations. The particular implementation of the operation which will be executed if a message via the instance variable requests the execution of the operation depends dynamically on the class of the object. The request is dynamically bound to a particular operation implementation. We call such a mechanism *dynamic binding* or *late binding*. The benefit of such a technique is that a client is not required to care about possible variants in the implementation of operations of the server. The abstraction of the server is *polymorphic* with respect to the client. The ability to distinguish different classes of a server object at run-time and to execute the corresponding operation is called *dynamic polymorphism*[6].

The concept is different from genericity which also may provide a technique to model polymorphic objects. Genericity allows to model polymorphic objects by passing the corresponding class descriptions as generic parameters to the model before the execution of the model. During the execution of the model the parameter i.e. the class of an object must not change it is static. We say that genericity provides *static polymorphism*.

Also different from dynamic polymorphism is static operation overloading. An important difference between operation overloading and dynamic binding is that the implementation of an overloaded operation has to be statically determined before the execution of the model (i.e. at compile time). Thus, overloading is not appropriate to model dynamically changing relationships between objects.

3.2.1 Polymorphism

Polymorphism which allows any server to be used in a use-relation of a client allows very flexible modelling. However, there is no guarantee that a server provides the operation which is requested by a message from a client. Such a situation is a run-time error. It is not feasible to rule out such a situation during the modelling. It is only possible to detect such an error during run-time and to initiate some error recovery mechanisms, e.g., to signal an error and to perform a nop operation.

If such a behaviour is not acceptable a modelling methodology or language has to be used which puts some constraints on the use of polymorphism which can be checked at compile time. The constraints are required to guarantee messages are only sent to servers which provide the requested operation. The constraints are imposed by a strong typing.

6. We very often only refer to polymorphism, if we actually mean dynamic polymorphism.

3.2.2 Typing

A type is a set of entities that share the same features with respect to a certain aspect in modelling. A predicate is used to describe the features. In some languages it is possible to split up a type a so called parent type into sub-sets which are called subtypes. A stronger predicate which consists of the parent type's predicate and additional constraints describes the features of the subtype.

If an entity in a model is assigned a type the predicate can be used to interpret the information which the entity contains. The interpretation can be performed statically at compile time. This allows to detect modelling errors in which misinterpretations of information may occur. In other words, it can be avoided that an entity is expected to have a feature which it does not have. The situation is slightly different if a subtype concept is used. The information which the entity of a subtype contains, i.e. its value can be used in any context where the information is expected which is interpreted as the parent type. However, the other way around it is not possible to use any arbitrary value of a parent type in the entity of a subtype. The value may not meet the additional constraints. The problem is that there is no mechanism to generally detect such a situation at compile time. It only can be detected and handled as run-time error.

The concept to make polymorphism more secure is to assign a type to each object and its references, i.e. instance variables. There are several ideas what might be useful features shared by the objects of the same type.

3.2.3 Behavioural compatibility

One idea is that the objects share their properties, i.e., they all have the same behaviour from a client's point of view. The objects may have different implementations of their behaviour but that is not visible for a client in a certain modelling context. Objects of the same type are behaviourally compatible [171]. Polymorphism is restricted to different implementations of the same behaviour. It is possible to predict at compile time what behaviour can be expected by sending a message to an object of a certain type, for example, via a typed instance variable. While the idea is the most advantageous from a security point of view, in most cases it is too difficult to prove the compatibility. Another problem is that behavioural compatibility places strong constraints on polymorphism which may run contrary to the goal of polymorphism to support the modification and extension of objects and classes without modification of their clients.

3.2.4 Compatibility by conformance

Therefore another idea is to have a notion of compatibility which uses weaker constraints. It follows the concept of *conformance* [17]. A class conforms to another class if it can be used in all contexts where the other is expected. It can accept and understand all the messages handled by the other, in other words, it is able to use the same protocol. That means for the type concept that objects of the same type share the same protocol.

It is possible to work out a set of conditions which interfaces of conform classes must meet. An approach is to deduce conformance between classes from special relations between assertions which are part of the interface [64]. A class B conforms to a class A if B subsumes the operations of A and if the following conditions are met: The invariants of B imply the invariants of A. For each operation of A the precondition of A implies the precondition of B and the postcondition of B implies the postcondition of A. This definition assumes that for each operation of A the corresponding operation of B has the same interface, i.e, the same argument list.

It is possible to relax this assumption. To conform it is only required that each argument of an operation of A conforms to the corresponding argument of the operation of B. The rule is called *contra-variant rule*[7]. Each message which calls an operation of A and passes a parameter to the operation can be understood by B and the parameter can be used as an argument in B because the parameter conforms to the argument, i.e., it can be used in the new context of B.

The sketched type concept allows two objects of the same type to have different implementations. The problem with that concept is that in parallel system design parts of the protocol may be modelled by the implementation (compare Section 3.1). Depending on the complexity and the view of the object it can be quite difficult to answer the question if two implementations obey the same protocol.

3.2.5 Signature compatibility

The third idea starts from the assumption that it is often sufficient to approximate the behaviour of an object or class by its signature and to demand the compatibility of the signatures [171]. By *signature* we mean the syntactic structure of an object's or class's interface[8]. Signatures are compatible if one

7. In a language context this rule is normally defined in terms of inheritance (see below)

signature subsumes the other. In other words, the number and profiles of the operations in the interface of one class can be completely found in the interface of the compatible class. Compared to the previous notion of compatibility this approach only takes a part of the protocol into account[9]. Compatibility only depends on the interfaces. Two objects of the same type may have completely different implementations. Signature compatibility does not guarantee the same behaviour of compatible objects or classes nor does it guarantee that they follow the same protocol. It only means that if a message invokes a service in an object there is also a service in a compatible object which can be invoked by the same message.

Each of the presented compatibility concepts establishes a subtype relation on classes. If class B conforms class A then we can think of B as a subtype of A[10]. If two objects conform to each other then they are of the same type.

3.2.6 Inheritance

As mentioned above, polymorphism can be used to allow modifications and extensions of server objects without affecting clients. The mechanism to construct such modified and extended objects is *inheritance*. Polymorphism and inheritance are the concepts which make object-based modelling methodologies and languages object-oriented.

Inheritance means mechanisms for sharing common information of classes by allowing new classes to re-use parts of existing classes. We call such a new class *child* and the existing one from which parts are re-used *parent*. We say the child inherits from its parents. If a child is allowed to inherit properties from more than one class we talk about multiple inheritance. Inheritance sets up an *is-a relation* between parent and child. The child itself may become a parent of another child. The is-a relations between a parent and its descendants form an *inheritance hierarchy.*

8. We also sometimes use the term signature to denote the syntactic structure of an operation’s interface.

9. Considerations of a protocol which depends on an implementation are omitted in [17]. Discussions about assertions as part of the interface are omitted as well. Therefore conformance is mistakenly not distinguished from signature compatibility.

10. We shall see later on that there might be also some kind of indirect subtype relation between the classes instead of the direct relation *class B is a subtype of class A.*

An inheritance mechanism defines how a new class can be modelled by incremental modification of an existing one. The central question of such a mechanism is what kind of information i.e. which parts of a model can be inherited by a class. As we shall see, the answer is a weighing up of flexibility of modification and preserving the encapsulation of a parent class and thus a matter of modelling philosophy.

3.2.7 Inheritance and compatibility

One philosophy is to use inheritance mechanisms which guarantee the compatibility between child and parent. In other words, the idea is to use a mechanism which automatically makes a child a subtype of a parent. Deriving a child does not affect any client of a parent's object. An object of a child can be used in any context where an object of a parent can be used. Such a polymorphic object may contain any object of a class which is derived from the parent including the parent itself. We call such an object *heterogeneous object container.*

In software programming such containers very often are modelled as references to objects of a sub-tree in an inheritance hierarchy. It is then not the class itself but rather the references which indirectly establish the subtype relation.

A similar indirect[11] approach is to introduce a new kind of classes to model heterogeneous object containers. In a typed class concept we call such a new class *class-wide type*. For each class there is an associated class-wide type and vice versa. The objects of the class-wide type are all objects of the associated class and all classes derived from it. Thus, a class-wide type associated with a parent allows the explicit modelling of a context where objects of any child can be used if it is compatible to the parent. In such an approach the subtype relation is established between the child and the parent's class-wide type.

As discussed in the previous section there are different notions of compatibility. Depending on these notions there are various approaches which claim to preserve the compatibility.

11. Indirect with respect to the subtype relation between the child and its parent.

3.2.8 Delegation

A very restrictive approach which tries to preserve behavioural compatibility is to use delegation as an inheritance mechanism. *Delegation* means that a child inherits all the properties of a parent. An instantiation of a child consists of instantiations of its parents and a part which models additional structure and behaviour. An interface of a child subsumes all operations of its parents. If a message which invokes an inherited operation is sent to an object of a child it forwards the message to the instantiation of the parent which has the corresponding operation. The is-a relation becomes more a has-a relation in which a parent is an exclusive sub-object. The difference between delegation and a true has-a relation is that delegation breaks the encapsulation of structure between parent and child. A child has access to the parent's structure to model its operations.

From a state-oriented view the extension of the structure results in a partitioning of the existing states. Each possible value of a new attribute can be represented by a partition in each of the parent's state. Assignments to the new attribute causes a state transition between the partitions which form the new states of the child. If new operations only have a read access to the parent's structure then the new state machine only adds state transitions between the different partitions of a parent's state. New operations do not exit a state of the parent's state machine. If a write access is performed in a new operation then new state transitions between existing states are added to the state machine. Obviously, such an extension requires knowledge about existing states in the parent's state machine and thus it requires information about the implementation. It is important to note that in case of delegation the required information is essentially limited to the parent's structure.[12] Any inherited state transitions and actions are preserved in the child's state machine. This includes transitions between states which model the protocol.

From the considerations above we can conclude that if there is no misuse of the information in the parent's structure by introducing improper transitions then delegation sets up a subtype relation in which a child is behavioural compatible to its parents. Delegation is an incremental modification which is limited to extensions. No replacement of existing information is possible. Delegation achieves compatibility by banning polymorphism.

12. Modelling objects in parallel systems might require some additional information which is not limited to the object's structure as we shall see later on.

3.2.9 Explicitly controlled delegation

An approach to add the missing polymorphism to delegation is to introduce a mechanism to explicitly control delegation in polymorphic operations. Polymorphic operations are operations with different implementations in a parent and a child and which are invoked by the same message. We say the operation of the child overrides the parent's one. A delegation statement in an operation of a child allows to model an access to the structure or behaviour of the parent. If we have the notion of an exclusive sub-object we can think of the delegation as a reference mechanism to the parent. Such an explicit control of delegation allows us to present a modelling methodology for inheritance which achieves compatibility by conformance. The methodology requires an operation of a child which overrides an operation of a parent to conform with the following modelling rules:

- The operation of the child contains an invocation of the corresponding polymorphic operation of the parent.
- There is no write access to the parent's structure apart from the accesses modelled by the delegation to the polymorphic operation of the parent.

The modelling rules guarantee that the methodology preserves the encapsulation of the parent's structure and behaviour when overriding an operation. To override an operation no implementation information from the parent is required. At the same time the rules support conformance between child and parent. This becomes clear if we look at the classes from a state-oriented view. An operation of a child which overrides an operation of a parent can be reduced to three stages: The stage before the delegation, the delegation, and the stage after the delegation. In the first stage the operation performs only transitions between different partitions of a parent's state. In the second stage the operation performs the state transitions and actions of the parent's operation. In the third stage like in the first one only transitions between partitions of a parent's state are performed. In other words, the child's operation only adds some state transitions to the parent's operation which are normally not relevant from the parent's point of view.

There are some exceptions when the protocol depends on the time interval between transition. Then the overridden operation is not any longer in conformance with the parent's operation. Another exception can occur if a class is characterized not merely by services it provides but also by services it requires at the same level of abstraction from other objects. If an overriding operation adds a service request[13] to an object which already serves a request of the overridden operation then it might disturb conformance. To understand why this happens we can look at the distribution boundary as an attribute of the parent. Requesting a service from an object within this boundary means to break the encapsulation of the parent.

3.2.10 Inheritance of the class's interface

Other inheritance concepts support weaker compatibility concepts. In such concepts a child inherits at least the complete interface of the parent. The child is allowed to extend the interface and possibly to modify it. A corresponding implementation is bound to the interface either explicitly or implicitly. Operation overriding can be achieved by binding an operation declaration as part of the interface to a new operation implementation. Such a concept which is based on the inheritance of the interface may allow to separately inherit implementations from different parents. Explicitly binding implementations to the interface can resolve ambiguities which might occur in multiple inheritance.

In case an operation is overridden some inheritance mechanisms allow to modify the interface of the operation which in turn is part of the object's interface. To describe the modifications which are allowed the term signature conformance is introduced. A signature S of an operation is said to conform to a signature P of an operation if the following holds true:

- The number of classes in the argument sequence of both signatures match exactly.
- Each type in S's argument sequence is a subtype of the corresponding type in P's sequence.[14]

The subtype relation often is established by conformance between a child and a parent. In other words, a parent in P's argument sequence has a corresponding child in S's sequence.[15] We illustrate this in Figure 1.

Supposing, the classes U and V appear in the argument sequence of S. Assuming, they are children of the classes X and Y that appear in the sequence of P and they conform to the parent classes[16]. The conformance establishes a subtype relation between them. U and V are subtypes of X and Y and thus S conforms to P.

Starting from this definition of conformance there are basically two philosophies on how to allow the modifications of an operation's interface:

13. This can be a direct request or an indirect request via other objects.
14. We consider a type to be its own subtype.
15. A parent is considered here to be its own child.
16. On conformance of classes see Section 3.2.4.

U $\lhd$ X	class U is a child of class X
V $\lhd$ Y	class V is a child of class Y
U $\Rightarrow$ X	class U conforms to class X
V $\Rightarrow$ Y	class V conforms to class Y
S(U, V) $\rightarrow$ P(X, Y)	signature S conforms to signature P

Fig. 1 Conformance between signatures

The first approach allows to override a parent's operation if the signature of the child's operation conforms to the parent's one. In the child the types (classes) of arguments in re-defined operations are subtypes (children) of types in the parent's operation. The incremental modification varies for class and arguments in the same direction. The inheritance mechanism is called *co-variant*. The co-variant approach is sketched in Figure 2.

```
Class B :     Operation A(X, Y )
  ▽                     ↑ ▽  ▽
Class C:      Operation A(U, V)
```

Fig. 2 Co-variant approach

Consider a class C which is a child of class B. The child class C overrides the operation A of the parent class. Following the co-variant approach A's signature conforms to the signature of the overridden operation A.

The approach allows to incrementally modify related classes mutually. Its disadvantage is that such a modification destroys compatibility between child and parent. In a heterogeneous object container an object of C can be invoked with an argument sequence of B's operation which fails if an argument in C's sequence is a true subtype of the corresponding argument in B's operation. Consider an argument Z that is in class X but that is not in subclass U. A call on operation A of class C with argument Z would fail.

To avoid the problem the second approach is based on a contra-variant inheritance mechanism. That means, to override a parent's operation, the signature of the parent's operation has to conform to the signature of the child's operation. This is illustrated in Figure 3

Again, class C is a child of class B. The child class C overrides the operation A of the parent class. The signature of the overridden operation A conforms to the signature of the re-defined operation A of class C. This allows

$$\begin{tikzcd}
\text{Class B :} \arrow[d, phantom, "\triangledown\ \Uparrow"] & \text{Operation A(} \arrow[d] & U, \arrow[d, phantom, "\vartriangle"] & V\text{)} \arrow[d, phantom, "\vartriangle"] \\
\text{Class C:} & \text{Operation A(} & X, & Y\text{)}
\end{tikzcd}$$

Fig. 3 Contra-variant approach

the modelling of a class C that conforms to class B. The conformance can be used to establish a subtype relation between classes which itself is based on a subtype relation between classes in an argument list. Class C becomes a subtype of class B.

A more restrictive approach is to prevent any modifications of the signature of an overriding operation. Such a modification obeys both the co-variant and the contra-variant rule at the same time. The result of such an inheritance mechanism is signature compatibility.

As already discussed in the considerations about compatibility we can conclude that inheritance mechanisms which are based on the inheritance of the complete interface as described above do not guarantee the same behaviour of parent and child nor do they guarantee that parent and child follow the same protocol. It only means that if a message invokes a service in an object of a parent there is also a service in an object of a child which can be invoked by the same message. Message here particularly includes a typed sequence of attributes.

3.2.11 Modification of assertions

Another modification of the parent's interface which may be allowed by a weaker compatibility concept is the refinement of the preconditions and postconditions. Preconditions may be replaced by weaker ones and postconditions by stronger ones. A weaker precondition is met if the original precondition is met or if some new assertions are fulfilled. A stronger postcondition is met if the original postcondition and some additional assertions are fulfilled. Such modifications preserve the parts of the protocol which are modelled by preconditions and postconditions. Assuming there are no further incompatible modifications of the protocol the modified protocol of a child can accept and understand all the messages handled by the protocol of a parent. The precondition of the parent's operation implies the precondition of the child's operation and the postcondition of the child's operation implies the postcondition of the parent's operation. The modification preserves the conformance between child and parent[17].

This can be illustrated from a state-oriented view. The modifications do not remove anything from the inherited state machine. The extension of the precondition only adds some trigger events to existing state transitions. The new set of trigger events comprises the inherited one. The stronger postcondition still guarantees the assertions of the original postcondition. Thus, the new postcondition guarantees the potential resulting states of the transition performed by an operation to be a subset of the original resulting states.

Such a compatibility concept and thus the inheritance mechanism works well in a sequential model. In parallel system modelling parts of the protocol which are modelled in the implementation may cause incompatibility.

3.2.12 Removing features from a class

Another philosophy is to use an inheritance mechanism which provides as much flexibility as possible in extending and modifying inherited classes. The restriction that a child inherits at least the complete interface of the parent is given up. Such an inheritance mechanism allows to selectively inherit features from a parent. The mechanism may for example allow to remove an operation from the list of inherited features. The inheritance mechanism does not guarantee any kind of compatibility between child and parent. In a system which is modelled with such an inheritance mechanism the inheritance hierarchy can be completely different from the subtype hierarchy. The original idea of introducing a type concept to make polymorphism safe i.e., to prevent run-time errors is given up if inheritance does not impose a subtype relation.

3.2.13 Mixin inheritance

In many modelling methodologies the flexibility to inherit properties from various classes would be useful but the methodologies do not support multiple inheritance due to its complexity. Solutions to resolve potential ambiguities in an inheritance chain e.g. linearization of the ancestor graph violate encapsulation. A solution to that problem is to combine static polymorphism with dynamic polymorphism. The modelling technique is called *mixin inheritance* [31]. The idea is to pass the parent to a class model as a generic parameter. An instantiation of the generic class model generates an actual

17. We do not consider the case here that a failure to meet the preconditions and postconditions occurs and that such an exception is handled by an object. There is obviously no conformance with respect to exception handling.

class. An instance of the class inherits properties of the parent according to the inheritance mechanism and it has the properties described in the generic class description.

3.2.14 Breaking of encapsulation

All the mechanisms show a basic characteristic of inheritance. It breaks the encapsulation of a parent [120]. An awkward break may require a complete re-analysis of a parent to derive a child. This can result in a loss of the benefits of object-oriented modelling.

If we analyse the breaking of encapsulation in the various inheritance mechanisms we can see that inheritance breaks the encapsulation of the structure. To perform an access to an attribute requires knowledge about its function in the context of the object. If the inheritance mechanism allows to remove an operation in a child this obviously breaks the encapsulation of the parent's behaviour. One has to pay for the flexibility of the mechanism with the breaking of encapsulation.

In a parallel system the access to an attribute may require knowledge of the access mechanisms used by other operations of the object. If the mechanisms are part of the operations' implementation then knowledge about the implementation is required which breaks the encapsulation of the operation i.e., the parent's behaviour.

An approach to control the encapsulation is to provide an interface which abstracts the information which can be re-used by a child. Such an interface may be different from the one it provides to its clients, it may for example contain an abstraction of the parent's structure which is not visible to a client but only to a child. In theory the approach solves the problem of breaking encapsulation in inheritance. In practice the problem is to have an appropriate abstraction mechanism especially in case of parallel systems. To be more concrete, a mechanism is missing to abstract the synchronisation to perform the accesses to the attributes.

3.2.15 Résumé

At this point we want to consolidate the material presented in this chapter so far. Object-oriented modelling is about objects and their relations to each other. An object encapsulates its properties, namely, structure and behaviour and abstracts its behaviour in an interface. Objects may be classified into classes which share the same properties. The distinction between object-based and object-oriented modelling is that the latter supports re-use by inheritance and polymorphism.

There are various concepts to introduce inheritance and polymorphism into object-oriented modelling which we have discussed in this chapter. They are described by two main characteristics which are the modelling flexibility and the compatibility between polymorphic operations of derived classes. Both play a key role in re-use concepts. Modelling flexibility is a condition for the modification of behaviour in derived classes. Compatibility, on the other hand, guarantees a certain behaviour in a derived class without the necessity to re-analyse and compare behaviour of inherited classes. Unfortunately, modelling flexibility and compatibility appear to be contrary to each other. The concepts for inheritance and polymorphism differ very much in the degree they support compatibility and provide flexibility. Figure 4 lists the main concepts which we have discussed in this chapter.

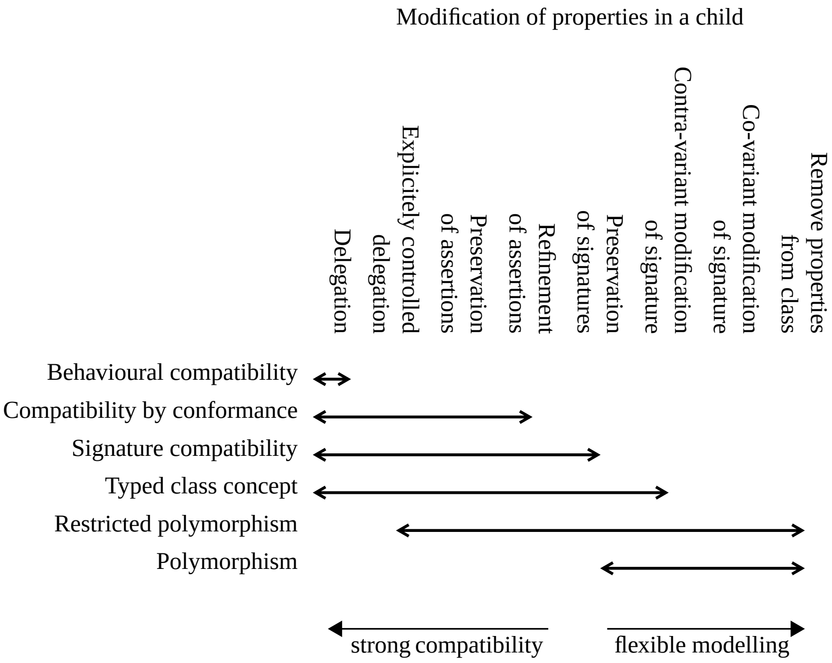

Modification of properties in a child
Delegation
Explicitely controlled delegation
Preservation of assertions
Refinement of assertions
Preservation of signatures
Contra-variant modification of signature
Co-variant modification of signature
Remove properties from class
Behavioural compatibility
Compatibility by conformance
Signature compatibility
Typed class concept
Restricted polymorphism
Polymorphism
strong compatibility
flexible modelling

Fig. 4 Inheritance and compatibility

The classification is into concepts that use delegation, concepts that abstract the interface of a class by assertions, concepts that abstract the interface of a class by a signature, and concepts that allow any kind of modification during inheritance. It is interesting to note that in each category of the classification there is a variant that allows a kind of controlled modification in the modelling. There is a controlled delegation, a refinement concept for assertions following the contra-variant rule, and a contra-variant and a co-variant approach of modifying signatures in derived classes.

Choosing an appropriate concept for inheritance and polymorphism is a weighing of flexibility and compatibility. Compatibility can be characterized according to the different notions of compatibility which have been introduced in this chapter, namely behavioural compatibility, compatibility by conformance, and signature compatibility. Flexibility is defined by the way it is possible to model polymorphic objects. A related aspect that can be seen in Figure 4 is a potential support for a typed class concept in order to make polymorphism more secure.

To give a résumé about inheritance mechanisms we can state that the problem is to find a mechanism which supports polymorphism and at the same time guarantees behavioural compatibility or compatibility by conformance. As it can be seen from Figure 4 there is no inheritance mechanism that supports polymorphism and that automatically generates by itself compatible objects or –in more hardware-oriented terms– pluggable components for re-use.

3.3 Parallelism

So far the chapter has described object-oriented concepts that are addressed by sequential models in which activities are obeyed in a sequential order. We now come to concepts that allow to express concurrency and distribution in models. Concurrency and distribution are new aspects in object-oriented design that have to be carefully analysed. At first sight classes and objects seem appropriate constructions for being parallelized and distributed. Surprisingly, the abstraction and encapsulation of classes that is required for a proper inheritance concept is not as independent from concurrency and distribution as one might expect. The interdependence especially affects the compatibility considerations that we have made in the previous section. Abstraction and encapsulation that considers concurrency and distribution may have an effect on the compatibility concepts that depend on abstraction and encapsulation of polymorphic classes.

In this section we describe various approaches to introduce concurrency into object-oriented programming. Of special interest are abstraction and encapsulation mechanisms and their effect on compatibility issue.

Complex systems very often are discerned as parallel applications from a user's or designer's point of view. Activities[18] of the system occur or at least appear to occur in parallel. There is very often corresponding agreements from different points of view which activities of a system occur in parallel. However, there are various notions what parallel exactly means.

One idea of parallelism is that activities occur simultaneously. Time is an ordering of events[19] in a system which allows to define simultaneity.

A related idea is to have no strict ordering between activities in a system, in other words, there is no sequential series of activities. The ordering reflects the causal relations between the activities. Timing and causal relation become indistinct.

A third idea is that parallelism means arbitrary interleaving of activities in a system. There is no fix sequential series in which activities occur. If a sequential series which was created by arbitrary interleaving is observable in a system such a parallelism is more a kind of pseudo-parallelism in our view.

At an abstract specification level very often only causal and timing relations are important. Wheter an implementation is truly parallel or only pseudo-parallel is not considered at that level of abstraction. Therefore the term *concurrency* is more appropriate here. Concurrent means that activities may be performed truly parallel. With the term concurrent we denote those modelling techniques which allow to model potential parallelism. Concurrency abstracts from an actual parallel or pseudo-parallel implementation[20]. This notion of concurrency is very similar to the definition of concurrent programming in [36].

In the following, we mainly focus on concurrency to model parallel systems. Only where it is necessary to look at the system from a more implementation oriented point of view we consider the various notions of parallelism.

18. In this general view on parallel systems the term activity and the term event are slightly different from the ones used in state-based views of systems.
19. Events are caused by activities, however, unlike activities we do not consider events to consume any time.
20. Naturally, it also abstracts from hybrid forms of parallelism too.

Concurrent modelling provides techniques to express causal and timing relations between activities or events in a system. The relations are modelled by interactions between activities.

There are very different concepts to support concurrency and interaction. In the following considerations we look at the basic ideas behind the concepts from an object-oriented point of view. According to the basic ideas we classify concurrent modelling methodologies.

We have introduced activity as one of the central terms in the context of concurrent modelling. However, we have not yet given a more detailed notion of what an activity might be. If we want to model in an object-oriented way in particular the question arises what can be identified as an activity in an object-oriented model. As we shall see there are several answers to that question with very different implications on the abstraction and encapsulation concepts of the modelling methodology.

After identifying activities the next step is to show the various possibilities to model the interaction between activities. An important aspect is the interrelation between interaction of activities and interaction of objects, i.e. communication and synchronisation of client and server. If we are able to understand these interrelation then the idea to reconcile both interaction concepts suggests itself.

As we shall see the modelling of communication and synchronisation between objects gives a useful application of various interaction concepts. We discuss effects on abstraction and encapsulation of objects and how the concepts may affect inheritance and the ideas on compatibility which have been presented in the previous section.

3.3.1 Concurrency in object-oriented modelling

Activities may occur in a non strict ordering in a parallel system. Breaking the ordering into sequential series of activities[21] gives a useful decomposition. A reasonable modelling concept is to provide some encapsulation mechanism with each series.

3.3.2 Threads

A mechanism which provides encapsulation with a sequential series of activities abstracts the indexing of the series and stores information about some

21. If the activities depend on some conditions a series is strictly speaking a set of possible sequential series.

results of the activities. We call such a mechanism *thread of control* or shortly *thread*. An implementation of the mechanism especially in the software is called thread control block. It contains a *locus of control* which points to the actual activity to be performed. In a state-oriented view we would say state of control instead of locus of control. The control block also contains information about the state of execution in a thread. In many implementations the information may dynamically increase or decrease during the execution of a thread so that the information is stored in a stack. Threads may compete with each other for some resources they require to execute e.g. a processor to run. A *scheduler* or *dispatcher* can be used as a synchronisation agent to resolve the conflict. A scheduler may assign resources to a thread so that it becomes active and it can withdraw the resources from a competing thread which becomes suspended. Thus, a scheduler can break the arbitrary interleaving of activities between different threads.

Depending on the modelling methodology a model may have one or more schedulers to organize the access to resources. Also depending on the methodology the scheduling concept may impose restrictions on number and characteristics of threads in a model. According to these restrictions it is possible to classify the modelling methodologies [170]:

We call a methodology or language *sequential* if it only allows a model to have a single thread of control. The methodology or language is *quasi-concurrent* if it has multiple independent threads but only one active thread at any moment. We call it *concurrent* if it allows a model to have multiple active threads.

Sequential methodologies obviously provide no support for modelling parallel systems. Interleaving of activities which are from a system's point of view considered to be executed in parallel has to be explicitly modelled by the designer. The interleaving must be fine grained enough and the activities have to be small enough that they appear to occur in parallel. Quasi-concurrent methodologies just provide a built-in interleaving mechanism. What is quasi-concurrent from a modelling point of view very much corresponds to pseudo-parallel from an implementation point of view. In quasi-concurrent modelling methodologies activities are very often regarded as atomic transactions. That means, threads may not switch from the state active to the state suspended and vice versa while the system performs an activity. The exception from this rule is an explicit activity to suspend, terminate, or activate a thread. As a consequence encapsulation of resource access in an activity can be used to neutralize resource conflicts. In many modelling scenarios the restriction to have only one active thread is not justified from a modelling point of view but only from an implementation point of view. Then a probably better choice is to use concurrent methodologies which virtually have no restrictions on parallelism and thus on interleaving of activities.

3.3.3 Introducing threads to objects

The way to introduce a thread concept to object-oriented modelling is to make a thread a part of an object. If we make such modelling components part of an object we have to analyse the properties it establishes in an object. We have to think how an object's interface may abstract the newly introduced properties. It must be discussed which of the properties have consequences on compatibility considerations especially with respect to inheritance. The properties and their abstraction also has to be analysed from the point of view of composition.

Such an object which contains a thread may or may not be different in its modelling possibilities from other objects which do not contain threads. An object may have one or more threads of control. In other words, it has a locus of control and a stack or something similar to store information about the thread's state of execution as an attribute. As a thread is part of the object in this view the stack stores at least partly information about the object's state of execution.

The important feature of the new attributes is from an object-oriented point of view that there is no explicit or direct access by writing to or reading from the attributes. There is only an implicit access during the execution of a thread. If such an implicit access is performed by a scheduler which is outside the object that contains the thread the access may potentially break encapsulation. For example, object A executes a thread. Object B requires a resource to execute its thread which is occupied by A. Object B may be able to signal the scheduler probably by a priority mechanism to withdraw the resource from A, i.e to change its state.

Another problematic issue about incorporating threads into objects is that activities of a thread which are part of an operation may be allowed to interact with activities outside the object to which the operations belongs. Such a situation occurs for example if it is allowed to receive a message to execute an operation within a thread. That means such a situation occurs in any *reflective* modelling style.

A problematic interaction between activities is the invocation of two operations of an object which are executed in parallel i.e. in different threads and which are able to interact with each other via a scheduler i.e via indirectly accessing attributes in form of the thread control block. In each case where such an interaction between activities is allowed different values of the thread control block may be indirectly visible outside the object. For example, an operation reads attributes which are modified by another operation executing in parallel. Depending on the locus of control of the other operation during the read access the results of the read operation may differ.

The values i.e. the different loci of control which can be distinguished are modelled by interaction between threads. In other words, an operation is used to model the state of the object. This removes the clear separation between structure and behaviour in an object. We identify this phenomenon to be a reason for modelling problems in object-oriented modelling of parallel systems. Many of the modelling problems which we shall present in the following chapters can be put down to this phenomenon.

From what was discussed in the previous sections about inheritance and compatibility it directly follows that inheritance which allows to modify the structure without preserving the parent's structure causes incompatibility between parent and child. Removing the separation between structure and behaviour i.e. modelling structure by an operation allows to modify structure by overriding an operation. To achieve compatibility the overridden operation has to preserve the parent's structure. Compatibility issues in parallel object-oriented models as they are mentioned in the previous section turn out to be a problem of preserving structure.

The considerations about distinguishing loci of control can be generalised to the objects's state of execution. In objects containing threads the acceptance of messages can be delayed or rejected depending on the object's state of execution. This may make some of the states observable from outside the object. The same holds for a state of execution which can be used to control the locus of control.

Under the aspect of composition the properties which can be described by external visible states must possibly be understood by a client. The understanding can be either achieved by using standardised properties or by specifying the properties in abstract terms as part of the object's i.e., the server's interface. As we shall see later on, the abstractions can be interpreted as some temporal or causal properties of the object's behaviour.

3.3.4 Threads as part of an object

After these general considerations on objects which have one or more threads we look at the different concepts to incorporate threads as part of an object in detail. The extent to which the object-based and object-oriented techniques are supported in the various modelling concepts for objects differs very much. In one modelling methodology an object may support all the object-oriented concepts like abstraction, encapsulation, inheritance and polymorphism. Operations which have access to its state are abstracted in an interface. Threads may be automatically invoked after the object's creation. Such threads allow an object to invoke operations in other objects by its own. In such a modelling methodology a thread also may be used to control when a message which requests the execution of an operation invokes the operation. There is an implicit protection of object's states. The sketched features describe what we call *autonomy.* An object with such features is considered to be autonomous[120].

In another modelling methodology objects may only support some basic encapsulation concepts. The interface may be reduced to an abstraction of the interaction between activities from outside the object with activities inside the object. The interaction is used as a substitute for the operations to access the object's state variables. Inheritance and polymorphism is not supported for such kind of objects. The object has become a *process*[22], the interface is reduced to a set of *entry points* or probably more generally to *interaction points*[23]. It is not any longer an object in the strict sense.

There are many variants in between the two sketched modelling philosophies. Based on these two fundamentally different concepts to make a thread part of an object we distinguish two[24] different approaches to incorporate concurrency within a modelling language:

In the first concept a system only consists of autonomous objects which may execute their threads concurrently.

In the second concept concurrent execution is considered to be independent from objects. Concurrent activities are modelled by creating and execut-

22. From what was discussed above it is perfectly possible that a modelling concept may allow a process to have several threads.

23. Interaction point is used in this thesis as a term to denote abstractions of interactions in both directions, i.e. it includes the abstraction of receiving messages as well as sending them. Entry points only abstract interaction in form of receiving of messages.

24. There is also a classification which distinguishes three different approaches [25] The third approach is described as a combination of both. We argue these distinction is unnecessary because a combination can be viewed as a special case of an orthogonal approach (see below) if we consider only the (passive) objects as real objects.

ing processes or similar objects whose only task it is to introduce concurrency into the modelling methodology. Real world things, structures, and phenomenons are modelled and abstracted as objects which behave passively with respect to concurrency and synchronisation. Such kind of objects are so to speak first class objects in the concept. The superficial independence of first class objects from modelling concurrency leads to the notion of orthogonality between (passive) objects and processes or threads. Orthogonality is sometimes referred to as not pure object-based concurrency [26].

So far we recognized the modification of structure caused by interaction between activities from different threads crucially influence object-oriented modelling of parallel systems. A basic classification of the approaches to introduce concurrency into the object-oriented world was given independently from a specific concept of interaction. Conclusions about interaction between threads were drawn without considering the concrete shape of the interaction.

3.3.5 Interaction between threads

We now look at the different concepts of interaction in more detail. We give special emphasise to the important aspects simplicity and robustness of interaction.

The purpose for using interaction is to model timing and thus causal relations between activities. Different threads although normally executing independently coordinate some of their activities. We refer to such a coordination as synchronisation. It very often involves the exchange of data between the threads i.e. synchronisation associates data communication. A very general notion of an activity is that it is an access to some kind of resource. The causal or timing relations enforce a certain order of access to the resources. For example, a very common order is to postpone an access to a resource during the access of an other thread till it is finished. The interaction excludes an activity from a resource to avoid interference with activities[25].

A modelling concept may either emphasize the aspect of causal relations between activities or the aspect of scheduling the access to resources. This leads to the distinction between two categories of modelling concepts:

25. This kind of interaction is normally called mutual exclusion synchronisation or exclusion synchronisation [36]

3.3.6 Shared memory synchronisation

In the first category approaches offer an abstract model of a resource (template) with a built-in order of access to it. The built-in order protects from uncoordinated access to the shared resource. Thus, it is called protected resource. More generally, each shared resource which requires a protected access is called *critical resource*.

A sequence of activities that performs an access to a critical resource is called *critical section* [164] or *critical region* [81][26]. This leads to another notion of activity. The critical section as a whole can be regarded as an activity. As such a shared resource is often implemented at lower levels by shared memory we refer to it as shared memory approach. For example, all kind of monitors with exclusion synchronisation are in that category. By providing a built-in order the designer is released from modelling a mechanism to enforce the required order. This makes the approach simple and robust from a designer's point of view. The designer simply creates an instance of the model with some concrete information about the actual resource. Its disadvantage is that a resource might require an order of access which is not supported by an abstract model in the modelling methodology.

3.3.7 Message-based synchronisation

In the second category interaction techniques are subsumed which allow a thread to make arbitrary arrangements with another thread about the order of accessing resources. To model such arbitrary arrangements modelling methodologies provide a set of synchronisation and communication primitives. Basically, the primitives allow to send and receive messages[27]. Thus, we call the approaches message based.

Message based approaches allow a direct synchronisation and communication between threads whereas shared memory approaches establish an indirect synchronisation and communication between tasks via protected resources. In many modelling situations activities especially those for synchronisation and communication can only be safely performed if threads or resources are in a defined state or some activities have been performed. The

26. Different to the original mechanism to model critical regions we do not require the resource to be explicitly mentioned as a critical resource in a critical region.

27. Messages between threads do not necessarily correspond to messages between objects

condition to reach such states or the condition of a history to exist is a constraint on synchronisation. Thus, the synchronisation is called *condition synchronisation* and the constraint is called *synchronisation constraint*. Both approaches i.e., message based and shared memory approaches must allow the specification of synchronisation constraints to model condition synchronisation. Mutual exclusion synchronisation to protect a critical region can be used in combination with condition synchronisation to test if the shared resource is in a defined state appropriate to execute certain activities. The critical region is entered only when the test is passed. Such a critical region is called *conditional critical region*.

To discuss various synchronisation and communication mechanisms with special respect to the modelling of conditional synchronisation we slightly extend the meaning of message. We do not limit the term message to direct synchronisation and communication between threads, we also denote requests to access a protected or shared resource as message.

With this extended definition all interaction concepts between threads can be turned down to message passing mechanisms. We now shall describe various message passing concepts. Basically, all the message passing is modelled by primitives for the sending and receiving of messages[28]. The concepts differ in the way they send and receive the messages.

3.3.8 Synchronous message passing

A thread sending a message expects one (or probably more) receivers to receive and accept the message. If the sender does not perform any activity[29] until the receiver accepts the message then the mechanism for sending messages is *synchronous message passing*. Waiting for the receiver to accept the message and to perform activities requested by the message allows to pass information about the sender's state to the receiver if the message concept supports data communication from the sender to the receiver. All messages sent are stored by the receiver in a message-queue until the receiver is ready to accept the message. Depending on the modelling methodology different techniques are used for accepting a message.

The main philosophies are either to use explicit or implicit acceptance mechanisms. An explicit mechanism is to provide primitives which can be

28. In shared memory approaches the receive primitive is part of the built-in mechanism of the protected resource.
29. We also say the sender is blocked if it must not perform any activity.

used as activities in a thread to accept messages. An implicit mechanism as it for example often is used by protected resources associates potential messages with certain sequences of activities. Accepting the message means to automatically execute the sequence of activities by a built-in mechanism. Both, the implicit and the explicit mechanism may allow to specify additional synchronisation constraints to model conditional synchronisation.

Conditions govern the acceptance of messages. Such an approach to model constraints is called conditional acceptance. The receiver queues a message at least until an implicit or explicit accept can be performed and the corresponding synchronisation constraints are met. Typically, the mechanisms to specify synchronisation constraints allow to refer the constraints to the receiver's state or history.

In many modelling situations it is useful to refer also to information from the sender. For example, a producer wants to send variable amounts of data to a consumer which has only limited capacities for storing the data. In other words, it should be possible to pass synchronisation specifications in messages on the basis of the message content [166].

One concept to model such a situation without breaking the encapsulation of the sender is to send the information required by the condition as part of the message and to allow the mechanism for specifying the synchronisation condition to look into the message and read the required information from it before accepting the message.

Another modelling concept is to accept the message and read the required information from the message after accepting it. The information thus can become a part of the receiver's state or history. In a second step a new message is put by the receiver into the message-queue which replaces the original one. The synchronisation conditions for accepting the new message may refer to the new state or history of the receiver which contains the required information from the sender. In other words, synchronisation constraints are partly modelled by activities which are invoked by accepting the corresponding message. The replacement of the message is not visible for the sender. From the sender's point of view the receiver behaves as if the original message were queued. The sender still does not perform any activity until the new message is accepted. We call such a modelling concept *re-queuing*.

If there are several messages in a queue an arbiter which is part of the built-in mechanism of the receiver orders the messages. By allowing a message only to be accepted while no other activity is performed in the receiver an arbiter can model exclusion synchronisation.

There are different strategies for ordering the messages. One strategy is to try to be fair, i.e., not to starve any of the senders. A common model which is considered to achieve this is a first-in-first-out order of the messages[30]. Another strategy is to tag the messages with priorities and to order them according to their priority. The maximum size of the queue is determined by the number of potential senders as each sender can send at most one message in synchronous message passing.

Messages may allow not only to send data from the sender to the receiver but also to pass back data from the receiver to the sender as a reply. Blocking of a sender in synchronous message passing until the requested activities are performed in the receiver and a reply is passed back to the sender makes the sending of a message a remote procedure call (RPC). RPCs abstract synchronisation and communication by allowing a designer to model interaction in the way normal procedure calls are modelled. Underlying mechanisms of the RPCs are transparent for the designer. The limit on RPCs is the missing possibility to create additional concurrency.

3.3.9 Asynchronous message passing

A different concept which supports the modelling of additional concurrency is *asynchronous message passing*. Asynchronous means that a sender sends a message without blocking until the receiver accepts it. The sender can perform any other activities immediately after sending the message. The messages may be stored in a message-queue until the receiver wants to accept it. Another concept of *receiving* a message is to allow a receiver simply to ignore the message. It gets lost. In combination with queuing a message this could mean that under certain synchronisation conditions a receiver is allowed to remove a message from the queue without performing any corresponding activities. For modelling condition synchronisation concepts similar to the ones from synchronous message passing can be used.

Sending without waiting for the receiver to accept the message establishes a one-way message passing from sender to receiver. If such a message is allowed to send data to the receiver it only can pass information about the sender’s state at the point of time when it sent the message. The receiver must not assume when it accepts the message that it contains information about the actual state of the sender it only contains informations from the past[31]. If the sender expects a reply from the receiver the receiver has to

30. Re-queued messages take the position of the original message in the order.

become a sender and perform a send activity as a consequence of the message from the former sender. The former sender has to await this message and accept it. We refer to this approach as *reply scheduling*. One-way message passing gains its flexibility to control all issues of reply scheduling at the expense of abstraction [120].

A difficult question of reply scheduling with one-way message passing is how to make a sender aware of the potential reply messages without breaking encapsulation concepts of the receiver. It seems there is no satisfactory answer to that question yet.

3.3.10 Mixing synchronous and asynchronous concepts

Raising the abstraction in modelling asynchronous message passing can be achieved by using *proxies*. This is a kind of mixture between synchronous and asynchronous modelling. The sender sends its message asynchronously to the proxy instead to the receiver. The proxy performs the actual synchronisation and communication with the receiver. It collects the replies from the receiver to the sender. Whenever the sender requires the information of such a reply it receives it from the proxy. If it is not available from the proxy at the time it is required the sender is blocked until the proxy can pass the information from the receiver to the sender.

There are some more concepts of mixing synchronous with asynchronous message passing e.g., asynchronous transfer of control. Under some synchronisation conditions the sender and receiver behave like in synchronous message passing under some others they behave asynchronously. Such concepts are mainly used to control unexpected behaviour in the interaction between sender and receiver. For example, a time-out mechanism guarantees an original synchronous sender not to be blocked more than a specified amount of time even if the receiver does not accept the message.

Modelling methodologies may at the same time provide synchronous, asynchronous and mixed message passing concepts.

3.3.11 Abstracting synchronisation

Threads as introduced in the previous sections from an object-oriented point of view are parts of objects. In our considerations we carefully stated that modelling behaviour with threads results in the implicit modelling of the

31. In an extreme case the actual state of the sender may be even terminated, i.e. the sender does not exist any more.

objects' structure. It was mentioned that the object's internal state of execution becomes distinguishable from outside the object in form of temporal and causal properties. We now conclude these considerations by investigating the implication of this view in the modelling of abstractions, i.e. in modelling the objects' interfaces.

As stated above, interaction between threads is modelled by various message passing mechanisms. If the interaction is performed between threads in different objects messages are sent between different objects. Keeping the objects' encapsulation makes it necessary to bring together the message passing concepts of the inter-object communication and of the interaction between threads. A message between objects contains messages between threads and the interface abstracts this message consisting of one or more messages. Often there is a one-to-one correspondence between message and containing message so that it is possible to remove the distinction between them. This induces various synchronous and asynchronous message passing concepts in inter-object communication and synchronisation.

Moving the message passing to the inter-object level requires the modelling of abstractions of the messages as part of the object's interface. Such an abstraction typically consists of a message key. It denotes an operation, an entry point, or more general a synchronisation activity which the sender of the message i.e. the client may request from the object. If the modelling methodology supports both synchronous and asynchronous message passing it is useful to make them distinguishable in the abstraction i.e., an abstraction of an object or of a single operation indicates a client if requesting an operation may block the client. The modelling of the corresponding synchronisation constraints which specify the conditions either to accept the message or to block the client is very often understood as part of the interface modelling. It is called interface control. The representation of a set of messages that can be accepted by the object at a given moment is called the *interface control space* [119]. In various modelling methodologies different conceptions exist on how to abstract the interface control space.

3.3.12 Interface control space

Some methodologies make the interface control space part of the abstraction. As stated earlier the synchronisation conditions and thus the interface control space typically depend on the object's state and history. It is possible to interpret the history as temporal or causal property of the object. Abstractions of the state and history are used to specify the interface control space in the object’s interface. In other words, synchronisation constraints which depend on the object’s state and thus on the object’s structure are made visible to a client. Such a state information which is only needed for synchronisation purposes is called *synchronisation state* [27].

We want to briefly illustrate these theoretic considerations by an example of such a methodology that makes the interface control space part of the abstraction. In [166] so called enabled-sets are introduced to model the interface control space. Messages which request to perform certain operations are only accepted when the object is in certain states. They are delayed until the object enters such a state. The specification of the next state in a state transition of the object includes an enabled-set. It is a set of patterns which define the messages that may be accepted for execution in the next state. Such a set can be made visible to other objects. In other words, enabled-sets are sets of message keys which can be associated directly with the state to which they apply. Thus, enabled-sets model those states[32] which contain information needed for synchronisation purposes and make them visible to clients.

The example demonstrates some basic properties of the idea to describe the interface control space part of the object’s abstraction. The concept requires the extraction of the object’s synchronisation. Synchronisation constraints which refer to internal behaviour or structure of the object are abstracted by synchronisation constraints which refer to abstractions of the internal behaviour and structure i.e., they refer to the abstraction of the extraction in form of synchronisation states. In the example, the constraints do not refer to the actual non-reflective part of the object’s structure but to the enabled-sets. That means redundancy is introduced into the object. It is important to note that such a concept does not separate the synchronisation from the implementation. It rather duplicates information which causes synchronisation and thus introduces redundancy into the model.

In a previous section we discussed design by contract as a concept which introduces redundant information into a model. Design by contract allows to duplicate internal state information as part of an object’s interface. Assertions in form of preconditions, postconditions and invariants may refer to this information. If we transfer the idea of duplicating information in an interface for synchronisation purposes we could think of synchronisation constraints

32. In fact enabled-sets are first-class objects which are used as attributes to reflectively model the synchronisation. They may be referred to in messages to other objects. Enabled-sets which are part of an object become visible outside the object.

as a weaker kind of preconditions. A precondition specifies the conditions under which a server accepts a client's request to execute an operation. The conditions rely on redundant information in the interface i.e., duplicated state information or an abstraction thereof. Rather than causing a failure such a client's request is suspended in synchronous communication if the preconditions are not met. The execution is delayed until the preconditions are fulfilled.

Different to the originally introduced kind of preconditions the weaker ones do not specify a client's responsibility in a server client contract[33]. In a sense we could generally interpret synchronisation constraints which model conditional synchronisation by making redundant information part of the interface as preconditions, postconditions and invariants. It is possible to describe temporal or causal relations between the execution of operations by such assertions. For example, the relation between precondition and postcondition itself describes a causal or temporal relation. This view allows us to apply the previously made considerations about modelling of preconditions and postconditions accordingly to the modelling of synchronisation constraints. This includes the considerations about compatibility and inheritance. Another issue of modelling with preconditions and postconditions is how to avoid a consistency problem in the modelling of the objects.

While consistency between an operation's interface which abstracts behaviour and its implementation normally can be automatically checked in a modelling methodology there is no standard solution how to check the consistency between an object's implementation and its abstraction which is referred to by the abstraction of the synchronisation constraints. This becomes obvious if we think about the difficulties of using assertions for verification with automatic tool support [20]. Such a verification is nothing else but a consistency check between specification and implementation. If we think of such a consistency check in derived objects the verification is even more complex. As mentioned earlier, modifications of the implementation caused by inheritance and polymorphism may change the synchronisation constraints. Modelling the resulting modifications of the abstraction in a con-

33. Interpreting such a precondition as a client's responsibility would require a client to be able to read informations about the server's state. The consequences on encapsulation have been discussed previously. The problematic issue here in the context of concurrency is to assure that the affected state information does not change between its reading by the client and the potentially following separate invocation of the operation.

sistent way can become a non-trivial issue. We shall investigate this in more detail in Chapter 8.

If states which govern the acceptance of messages are used to abstract an object's implementation with respect to synchronisation the question arises how does the synchronisation during a state transition work. The typical answer is that no messages are accepted during a transition. The object is not in a state that corresponds to one of the abstract states. In other words, usually such objects perform operations in mutual exclusion and the states are only used to model condition synchronisation.

There are some more techniques to model the synchronisation states separately and in a redundant way as part of the interface. For example, path expressions are used as a kind of regular expression to model a state machine which specifies the order of acceptance of messages. The alphabet of the language that forms the path expressions in principle consists of the object's message keys and the words of the language denoted by the expressions specify the possible invocation histories.

Another interesting example is the use of operators from the so called deontic logic [17] to specify the synchronisation. The approach is based on the use of logic assertions over operation invocation histories. Special operators are used to define the assertions as conditions under which a given operation may be executed. The conditions can be specified in terms of sets of message keys to increase the abstraction of the synchronisation specification by generalisation. It is interesting to note that it is not possible to write the conditions in terms of the object's state variables as the assertions are part of the interface and must not break encapsulation[34]. The assertions are rather defined in terms of primitive history functions which are used to represent the number of times the operations corresponding to a particular set of message keys have been requested[35], activated or completed since system start-up. So the approach indeed introduces redundancy into a model as mentioned earlier.

In all the presented approaches the mechanism of the scheduling of operations which is based on an originally internal state of the object is made vis-

34. There are in fact approaches to *improve* the concept by allowing to reference the internal states in the assertions i.e., by breaking the encapsulation [116].

35. The synchronisation state in this modelling approach may depend on the content of a message queue. Receiving a message without accepting it already causes a state transition. In other words, the receiver performs unconditional acceptance with re-queuing.

ible[36] to the client. However, it is not the scheduling itself. These are two different things which must not be confused. The interface abstracts properties which allow to reason about valid composition of objects. It is not designed to run the scheduling by the clients. The scheduling of the server is transparent to the client. In other words, the public information about the interface control can not be used by a client to satisfy the conditions in an assertion to force the execution of a certain operation. This is a direct consequence of the fact that assertions like the precondition to control the condition synchronisation do not specify a client's responsibility in a server client contract as already mentioned earlier. The assertions depend on the synchronisation state of the server and the client can only get the information from the server by sending a message requesting the execution of another operation. How is it possible enforce the execution of this operation? A second question is how to guarantee that the (synchronisation) state of the object will not alter between the two messages.

An essential characteristic of the presented approaches is that the synchronisation constraints are not only visible to clients but also to children. The expected benefit of such a characteristic is that it is possible to avoid contradictions in the modelling of the child's modifications of the synchronisation constraints without breaking the encapsulation of the parent i.e., to preserve compatibility by only analysing the parent's interface. The interface control is centrally modelled in the parent's interface and the modifications are modelled in the children's interfaces.

3.3.13 Encapsulation of the interface control space

Some other modelling methodologies use a different kind of concept to abstract the interface control space. Modelling the interface control is not part of the interface in such a concept. From what was stated above about composition we can conclude that temporal or causal properties of objects in such a concept should be modelled in a standard way. It allows a designer to understand the properties required for object composition. Hiding information about the interface control space from a client supports request scheduling transparency. That means, the scheduling of the operations based on the objects's internal state and the nature of request to perform an operation is

36. Visible here means either that the information about the object's structure is duplicated in the interface or encapsulation of this information is circumvented. (Compare Section 3.1.2)

transparent to the client. Conditional acceptance is modelled separately for each operation.

A typical concept is to assign each operation a *guard*. A guard is a boolean expression. If it is true then the corresponding operation which is requested by a message or queued in the message-queue may be executed. We call the operation a *guarded operation*. The difference to the example with the deontic logic is that the guard is embedded in the body of the object. The guard expression can depend on the state variables of the object. This avoids redundancy in the modelling of the synchronisation constraints and circumvents problems related to redundant modelling.

In some modelling methodologies guards are also allowed to depend on some information external to the object. If a guard is allowed to depend on the content of a message it is possible for a client to pass synchronisation specifications in messages. As mentioned earlier (compare Section 3.3.8), a re-queue mechanism is required to achieve the same effects if the guard is not allowed to look into the message[37]. The modelling methodology must then support the re-queue to another operation with a different guard.

Each guard separates the states of an object into two subsets with respect to synchronisation: One set containing the states in which the corresponding operation is delayed if its execution is requested and the other set of states in which the requested operation may be executed. Iteratively dividing the subsets by the guards of the operations finally results in a set of subsets of states. The set of subsets represents the state space which is relevant for synchronisation i.e. the interface control space. Each subset stands for a synchronisation state. As the modelling of the states directly is based on the object's states the state transitions are automatically performed as part of the behaviour in the operations by assigning new values to the object's state variables.

Modelling the synchronisation constraints for operations independently from each other allows their scattering over the object's implementation. The constraints in form of guards model a decentralised interface control. Such concepts are criticised for not separating the synchronisation modelling from the implementation.

The main problem is how can a child which requires incremental modifications of the synchronisation avoid inconsistencies with the parent's constraints without re-analysing the parent's implementation to understand the

37. The same is necessary if only limited information is passed from the client to the server. A typical example of such an information is the priority of a message.

synchronisation and thus breaking the encapsulation. The problem can be paraphrased by the question how can guards and related state transitions safely be inherited by a child.

Before we sketch solutions we point out the similarity of the question to the inheritance of pre- and postconditions. A guard corresponds to a precondition. Like in the modelling of preconditions it has to be defined what is considered to be compatible[38] with respect to synchronisation constraints modelled by guards. If we take the definition of compatibility by conformance as it was given in Section 3.2.4 it requires a child to use the same protocol as the parent. It requires the child to distinguish the same synchronisation states and to perform the same state transitions. As the distinction is modelled by guards the child must inherit all the guards and must not modify it for inherited or re-defined operations to preserve the distinction.

This concept is often considered to be too restrictive. To relax it we present a weaker definition of compatibility by conformance. This definition considers a child to be compatible with its parent if it supports the protocol of the parent at least if it is in a certain mode. Mode here stands for a set of states. Compatibility mode is a set of states of a child with each state having a corresponding state in the parent. Each state of the parent can be mapped to state of the compatibility mode and vice versa. The state transitions of the child which are performed by inherited or re-defined operations correspond to the state transitions of the parent's operations on the parent's state. The mapping of the states is isomorphic with respect to inherited or polymorphic operations. This allows a client to use a child in any context where its parent is expected if it is in a compatibility mode. To model such a compatibility mode a new guard of a child is used. The new guard depends on some new attributes. The new guard is accumulated to the guards inherited from the parent. A polymorphic operation in a child may be guarded by the new guard. The actual guard of the re-defined operation is obtained by applying the and operation to the new and the inherited guard. We can think of a refinement of the guard with the new guard being orthogonal to the inherited one. If the new guard is true the object is in compatibility mode provided that

38. The considerations are based on the assumption that pure synchronous synchronisation concept are used for modelling. Extended concepts like asynchronous transfer of control do not allow such general statements about compatibility. They would require each concept to be treated separately which is beyond the scope of this thesis.

the polymorphic operation performs the state transitions within the compatibility mode which correspond to the parent’s state transition.

The presented considerations about compatibility illustrate the problems mentioned earlier which are related to not separating the synchronisation from the implementation. It is not possible for a child to identify the interface control which is required to preserve compatibility with its parent from the parent’s interface. It fails because the child does not know which state transitions in polymorphic operations preserve compatibility. Deduction of conformance between classes from special relations between assertions which are part of the interface does not work in concurrent modelling. To circumvent the problem existing modelling methodologies either restrict inheritance to delegation or relax the notion of compatibility.

As mentioned in Section 3.2.8 delegation preserves the state transitions and thus serves the compatibility even in concurrent modelling [112]. Relaxing the notion of compatibility means that only guard refinement by accumulating guards along the inheritance chain is required in the modelling methodology [65]. This is of course not really a solution to the problem.

3.3.14 Preserving compatibility in concurrent modelling

As a result of this analysis we introduce requirements for a concept which preserves compatibility in concurrent modelling. In such a concept new guards of polymorphic operations must model a compatibility mode and the concept has to integrate a delegation mechanism which guarantees the compatibility of state transitions into a real inheritance concept which fully supports polymorphism. The guards and the delegation mechanism separate the modelling of synchronisation from the implementation of the functionality. Additional levels between the implementation and interface of an object are introduced to model the synchronisation separately from the implementation. We may characterize the tasks of the additional levels to decide which operations may be executed and to change the concurrency constraints. According to [119] we denote such tasks matching phase and state transition phase. With this explicit distinction of the tasks we can assign guards only in the matching phase. Therefore it is the delegation mechanism which implements the state transition phase.

Our modelling methodology for parallel hardware systems which we shall present later on is just designed in such a way that it separates the matching phase and state transition phase from the implementation. It follows the presented ideas to preserve the compatibility so that the methodology obeys the presented requirements.

3.3.15 Re-queuing and compatibility

We stated that a re-queue mechanism is required to pass synchronisation specifications from a client to a server if a guard is not allowed to look into the messages. The question arises how to perform re-queuing of operations in a child. We could think of two basic concepts:

In the first concept a re-queue activity in an operation statically delegates the execution to another operation. That means, if a child inherits an operation which performs a re-queue activity to a polymorphic operation the inherited operation does not re-queue to the new implementation of the child's operation it rather re-queues to the parent's implementation of the polymorphic operation. The problem is that although the re-queued operation preserves the compatibility it does not consider any new synchronisation constraints which may be required by the child. The problem can be solved by dynamically delegating the execution to other operations. In such a concept an inherited operation re-queues to the new implementation of a polymorphic operation. This solves the issue of the new synchronisation constraints, however, the compatibility problem arises. We have to consider that a re-queue activity performs a state transition on synchronisation states. Therefore the transition should be modelled in the state transition phase. We have no solution how to model this by delegation. It is not clear how we can guarantee compatibility in polymorphic operations which use such a kind of state transition. It is not even clear how to separate such a synchronisation modelling from the implementation of the functionality. To conclude our considerations about re-queuing we can state that the use of re-queuing together with inheritance would require a deeper analysis and understanding of the related problems.

3.3.16 Internal concurrency in objects

We started our discussion about interaction with considerations about interaction between threads. Concurrency was introduced into object-oriented modelling by making threads a part of objects. We turned down the communication and synchronisation to message passing between threads. It is interesting to note that during our considerations message passing between threads became indistinct from message passing between objects. This is mainly a consequence of our idea that if scheduling of threads in an object is required then it is mainly a task of the object[39]. Three main concepts of scheduling internal threads and coping with concurrency in objects can be distinguished [170, 120]. The distinction follows the classification of the modelling methodologies given above:

- Sequential objects or processes have a single thread of control. Only explicit acceptance of message is used for synchronisation in such objects. The interface uses entry points to abstract the synchronisation. We can think of the object following a RPC protocol. The scheduler may activate, suspend, or terminate the thread in such an object. A receive primitive in the thread forces the scheduler to suspend a thread and to wait on the receiving of a message. In terms of the RPC it is the specification of a server stub procedure[40]. It is possible to specify some additional synchronisation constraints in a receive statement. The scheduler only activates the thread if the object receives an appropriate message and if the specified constraints are fulfilled. Please note, the specified constraints are only considered by the scheduler if a receive primitive is executed and a corresponding message is received.
- Quasi-concurrent objects or processes may have several threads with at most one active thread. The scheduler interleaves the threads. A typical scheduling strategy for the interleaving is the mutual exclusion synchronisation of the threads. Additional constraints can be specified to model condition synchronisation. Different to sequential processes the additional constraints are re-evaluated and thus considered by the scheduler each time a state transition takes place in the object which might affect the constraints. A monitor using this strategy is a typical example of such an object. A quasi-concurrent process is a widely used kind of object with a lot of variants in the possible scheduling strategies; for example, some strategies allow a client to influence the scheduling by priorities.
- Concurrent objects or processes may have multiple active threads of control. Internal threads may execute in parallel. Additional concurrency can be introduced by creating a new active thread in an object. The internal

39. Please remember, this is a consideration from a modelling point of view i.e., how it is seen by a designer. It is so to say virtual. An actual implementation may use invisible for the designer another scheduling concept e.g., a global one.

40. Likewise, a more general interaction point may be used to abstract synchronisation which corresponds to a client stub procedure.

creation allows the modelling of active objects. Such an object may re-act to external events in form of incoming messages while it internally performs activities in parallel. Modelling of attributes as objects with active threads executing in parallel supports the hierarchical composition of active objects. In principle there is no restriction on active threads. But access to shared resources from internal threads may require a scheduling which suspends one or more threads until the shared resource can safely be accessed. Modelling of concurrent objects does not remove the problem of synchronisation. According to what we stated above it rather tries to shift the responsibility for the synchronisation from the threads to the object. It is not to consider during the modelling of the thread what is its context with respect to the synchronisation. Synchronisation has to be considered as part of the modelling of an object. The same holds for the composition of active objects. Objects are required to synchronise correctly independently from their context e.g., as attributes in a composed object. The objects should be pluggable with respect to synchronisation.

3.3.17 Interference between inheritance and composition

From what we stated about internal concurrency in objects we can say that hierarchical modelling of active objects requires concurrent objects. Composition of such objects requires either standardised interfaces with respect to synchronisation modelling or concepts to abstract temporal or causal properties of an object's behaviour as part of an interface. The composite object for its part has to either have a standardised interface or an interface which abstracts temporal or causal properties. An attempt to achieve this in a design methodology is to provide a set of rules for the composition of objects. The problem with the standardised interface solution is to find such an interface and a corresponding set of rules for any kind of active object.

An abstraction of the interface control space as part of the interface introduces redundancy into a model and thus causes problems with the incremental modification of objects by inheritance. Inheritance interferes with composition of active concurrent objects. The modelling decision is between the hierarchical composition of concurrent objects or compatible incremental modification of concurrent objects.

In the modelling methodology which is presented later on as part of this thesis this conflict is solved by restricting inheritance to quasi-concurrent objects.

3.3.18 Distributed objects

In the presentation given so far the main focus was on how to model and abstract objects and processes which execute concurrently. It was discussed how communication and synchronisation can be modelled as part of the object and abstracted by its interface. The message passing to establish communication and synchronisation was implicitly assumed to be provided by the modelling methodology. This is based on the idea that an object makes its abstraction visible to other objects. The server's interface is visible to a client, i.e. the interface immediately is within the client's distribution boundary. From a modelling point of view use-relations are modelled by instance variables. As previously mentioned, such a modelling concept which allows to dynamically change the distribution boundaries does not model the boundaries as part of an object's abstraction. Omitting the distribution boundaries in an object's abstraction and interface mechanism finally means breaking the encapsulation of objects [17]. For example, an object may model sub-objects by instance variables. It might be possible for such an object to export the values of the instance variables to other objects. References to sub-objects which are exported cause an object to lose control over its state which is partly modelled by its sub-object.

The consequences for a re-use concept are that such an object is not able to protect itself from unintended state transitions caused by a client if the object exports references. This is again such an unwelcome modelling situation which requires the analysis of an object's implementation to re-use it.

The second unwelcome consequence is that certain implementations[41] of such objects with instance variables would cause difficulties determined by the principle structure of the system. This is the case for systems in which the objects are distributed. Distributed means that the objects' implementations do not share the same memory or at least the same address space in a system[42]. In such distributed systems a message passing concept which can handle dynamically changing references to objects would cause an enormous implementation overhead. An issue in this context is how to perform sched-

41. The term implementation is used here in the meaning of Chapter 2, i.e., in the sense of realization of a system.

42. We do not classify distributed systems to be different from parallel systems. We rather consider them to be a special kind of parallel system. In such a sense parallel hardware objects whose state variables are implemented in different registers are also considered to be distributed.

uling to solve resource conflicts if the resource is distributed and its encapsulation is broken.

A common solution is not to support a reference mechanism i.e., instance variables to model relations between distributed objects. In such a concept it is not allowed to pass references as a parameter to operations in a distributed object. Messages cannot be used to pass references between distributed parts of a system.

Such a restriction could be achieved by generally avoiding instance variables and to only allow the modelling of exclusive sub-objects to hierarchically composed objects. To extend the restriction on all objects of a system makes such a language concept simple and uniform. Avoiding instance variables guarantees a strong encapsulation of every object.

The dissimilarity between objects using instance variables and objects with strong encapsulation can be illustrated by looking at how to model an access to nested sub-objects. The difference is in how to call an operation in an innermost sub-object. The reference mechanism only requires each nested sub-object to have an operation which passes the reference of its immediately nested sub-object. It is then no problem to pass the reference of the innermost sub-object across the abstraction boundaries of all enclosing objects although such an access to the innermost sub-object is not explicitly part of their interface. After passing the abstraction boundaries it is no problem to execute the actual operation at the innermost sub-object although an operation to perform the actual operation on the innermost object is not part of the interface of the enclosing objects. The strong encapsulation concept requires each sub-object along the nesting chain to provide an operation which passes the request to perform the specific operation of the innermost sub-object along the chain. The operation of the innermost sub-object is explicitly part of the interfaces of each enclosing object.

It is claimed that avoiding instance variables allows the specification of objects independently from considerations about the object's distribution [46]. The distribution can be specified in further design steps. This is referred to as late binding of parallelism. The orthogonality of object modelling to the distribution of objects is meant to supports their re-usability.

The encapsulation per se in such a modelling approach is independent from the concurrency and synchronisation. In other words, the orthogonality of distributed objects to their distribution does not automatically induce object autonomy. This qualifies the statement about support for re-use.

The reference free communication between all objects uses a copy model of variables and parameters [46]. The limitation with that mechanism is that it does not easily support all kind of objects. It would be a quite complex task to define the semantics for copying active objects. What would it mean to pass an object as a parameter during its execution of an activity? Another issue to be considered especially in an object-oriented language is how to pass polymorphic objects as parameters. In such a situation the system has to provide a way to pass the information about the actual operation implementations from the sender to the receiver. The most simple way to avoid this problem is to only support object-based mechanisms [46].

Another approach is to relax the restriction to generally avoid instance variables in classes and objects and to support passing objects as well as references as parameters to a message [17]. In such a concept the object's structure i.e, whether it contains instance variables or not determines whether it can be sent via messages from one object to another.

The third approach is to support a concept for data types and a separate concept for classes which also may be typed. Such an approach allows to pass data values of a certain type as well as references as parameters to messages. It is only possible to send messages with data values as parameters between distributed objects. That means a distributed object in such a modelling methodology is a special kind of object that internally allows references and the corresponding breaking of encapsulation but externally[43] provides strong encapsulation. It models the distribution boundary for its internal objects.

3.3.19 Explicit message passing via channels

From the discussion on different approaches to use distributed objects we can conclude that the way how to establish distribution boundaries and how to pass information across these boundaries is one of the crucial aspects in modelling distributed parts of a system. We can distinguish two basic opposed concepts on how to use distribution boundaries in distributed objects.

The first concept allows to have a distribution boundary outside the distributed object. Objects from outside and their interfaces can be made visible inside a distributed object. This is an elegant concept from a modelling point of view. Low level protocol layers which may be required for the communication between the distributed objects are abstracted. A designer is not required to care too much about the distribution of the objects. In a strong

43. Externally with respect to other distributed objects.

markedness of the concept a modelling methodology provides the same mechanisms for communication with objects internal or external to the distributed object. Such a property is known as communication transparency [17]. From an implementation point of view such a concept is hard to realize. An implementation has to provide the interconnection between distributed objects. However, making such an interconnection efficient is a difficult problem. It requires a detailed knowledge about the communication between distributed objects and it is very difficult to automatically extract such knowledge from a model[44].

A second concept is to restrict the distribution boundary to the distributed object itself. No object outside the distributed object is directly visible from within the distributed object. Only an interface to a protocol mechanism is visible which runs the communication between the distributed objects. Modelling methodologies may use interaction points to form such an interface. Such an interaction point abstracts an operation and the communication mechanism required to invoke the operation. Low level protocol operations might be used to built operations of a higher protocol layer. Finally, the higher level protocol operations are the operations of the distributed objects. It is important to note that an interaction point does not abstract an object but only an operation[45]. A clientship relation between distributed objects is not modelled as part of the objects participating in the relation. It is rather modelled by interconnecting the interaction points of the distributed objects. The interconnection often is modelled as part of an enclosing object but there are also modelling methodologies and languages which allow the interconnection to be modelled separately from the rest of the model. The flexibility of the latter approach which allows a simple replacement of objects[46] without touching the rest of the model comes with the disadvantage of breaking the encapsulation of hierarchical models. Basically there are two concepts to interconnect interaction points. The first is to directly connect the points.

44. We can accordingly apply here the considerations discussed in Section 3.3.12 about the difficulties to check the consistency between abstraction of synchronisation constraints and their implementation.

45. In fact, an interaction point may even only abstract some low level communication primitives for sending some data between distributed objects which then must be used to build higher level protocol operations.

46. Configuration management is part of the language. Especially in hardware design this allows to use multiple versions of objects at different level of abstraction in different design steps without modifying the actual model of the entire system.

Events caused on one interaction point are immediately visible on the others. A second concept is to interconnect the points and thus the distributed objects via *channels*. An interaction point is connected to a channel. Events on the interaction point occur on the channel or vice versa, events on the channel are observable at the interaction points. A channel abstracts a communication mechanism[47]. It can be seen as a special kind of predefined object or a resource shared by distributed objects. Several communication mechanisms are supported by various channel concepts. There are uni-directional and bi-directional concepts. While bi-directional concepts provide a higher level of abstraction uni-directional channels are uncomplicated from an implementation point of view. If the protocol is kept simple enough uni-directional channels may allow very efficient implementations. However, they require explicit reply scheduling which normally breaks encapsulation as shown in a previous section.

Channels may be able to resolve access conflicts if more than one interaction point is allowed to generate an event on a channel. Some channels may provide message queues for buffering more than one event on the channel. Generally, channels viewed as objects have a state which may change due to events on the channel. This establishes causal and temporal relations between events on connected interaction points so that we can conclude that channels have or model timing behaviour.

An important aspect is how to abstract the behaviour of a channel and how to abstract the behaviour of higher level protocol mechanism finally relying on the channel’s behaviour. Suddenly, the channel not only abstracts the communication mechanism but also a service of a higher level protocol. This is a problematic issue. A low level communication mechanism does not necessarily support object-oriented concepts and even if such concepts are supported the difficulty of composing concurrent objects and abstracting the behaviour of the composed object remains. Especially, all the difficult issues about abstracting synchronisation which have been discussed in previous sections have to be considered as modelling protocols is just about modelling synchronisation.

Another aspect of channels abstracting services of higher level protocols and thus finally abstracting operations of distributed objects is that the implementation of the service finally relies on the objects connected to the channel. In other words, a channel has no control over the services it offers. This

47. In this view an interaction point abstracts a channel.

violates the idea of encapsulation in object-oriented modelling and results in a loss of robustness of the model against modifications.

We can conclude from the observations made above that modelling of distributed objects is a difficult issue. Providing communication transparency causes implementation and efficiency problems. On the other hand, using low level communication facilities not following object-oriented principles jeopardizes the robustness of a model an makes re-use and maintenance of distributed objects problematic.

3.3.20 Conclusion

We conclude this chapter by summarising that there is no modelling methodology that provides an object model which allows the composition of concurrent and possibly distributed objects with arbitrary complex synchronisation concepts and which at the same time allows incremental modification of such objects without breaking encapsulation. This statement repeats and extends the résumé given about object-oriented modelling. Identifying the interface modelling and the related introduction of redundancy as a basic issue allows to think about modelling methodologies which are based on a compromise between flexible inheritance and complex composition.

The modelling methodology for hardware systems which we shall present as part of this thesis will consider this issue in detail. We are going develop a deeper understanding of the theoretic considerations presented in this chapter by discussing concrete modelling scenarios for composition and inheritance.

Chapter 4

Object-Oriented Methodologies for System Level Specification

As mentioned in Chapter 2 making re-usable and with respect to modifications robust models seems to be an important precondition for successfully designing complex systems. Deficiencies of today's procedural HDLs were sketched which disable the modelling of such robust and re-usable models at a high level of abstraction. It was stated that concepts for behavioural modelling with low level synchronisation concepts for the modelling of parallel systems are not appropriate for designing specifications at system level. It was considered to replace the algorithmic modelling concept by an object-oriented paradigm for modelling at system level.

In this chapter we survey several system level specification languages with particular emphasis on their support for object-oriented principles as they have been presented in the previous chapter. We consider the possibility of each language for hardware specification [152]. All of the languages we are going to present in this chapter originally were designed to model software systems. From the great number of system level specification languages a selection is made which covers a representative spectrum of specification methodologies, techniques and formalisms. The spectrum includes state-oriented concepts, declarative modelling concepts, functional approaches, formal languages, and process-oriented concepts. We especially analyse those languages from the software domain which have been proposed for hardware modelling [35,48,59,75,76,108,123,127,129].

The chapter presents the main concepts of the languages StateCharts, SDL, Estelle, ML, HML, Z, ObjectiveZ, LOTOS, CSP and OCCAM.

4.1 State-Oriented Specification Languages

This section concerning state-oriented modelling concepts is about modelling methodologies which support the modelling of states to describe a system as built-in feature of the language. Such languages are based on well known notations to describe automata.

4.1.1 StateCharts

StateCharts [77] is such a state-oriented specification language. It is based on finite-state machine (FSM) modelling. An original FSM consists of a set of states, a set of transitions between these states, a set of inputs, and an output function. A FSM reads the inputs and performs state transitions depending on the input and the actual state of the FSM. Input is modelled by events and conditions. The output function depends on the actual state and possibly on the input. The temporal and causal relations of events are just modelled by the state transitions.

The output concept of FSMs is slightly modified in StateCharts. States and state transitions can be associated with output events called actions. An action is performed, i.e., the events are generated by executing the corresponding transition, entering, exiting, or being in a associated state[1]. In the terminology introduced in Chapter 3 we would call actions either actions or activities. The input and output of a system are interpreted as external events.

State transitions which depend on a condition are called conditional connectors. In our terminology we would call them guards. A special kind of guard mechanism is used in StateCharts to extend the modelling of temporal behaviour. Timing constraints can be specified which guard the execution of activities [48].

The main extension of StateCharts is its support for hierarchy and concurrency. Hierarchy means that activities may be modelled as a state machine. Performing such an activity means to run the state machine. The state machine associated with the activity hierarchically contains sub-states and transitions which model the state machine. Leaving a state which runs a state machine forces the state machine to immediately terminate.

1. We refer to the original terminology of StateCharts here [77] which does not distinguish between actions associated with state transitions and actions caused by being in a state.

Concurrency means that a state may be associated with more than one activity. The activities and thus any corresponding state machines are executed concurrently. Events generated by one state machine can be used as an input in a concurrently executing machine. In other words, the state machines can synchronise with each other by asynchronous message passing. StateCharts also support a shared-memory based communication concept. The sender stores the result of an action. One or more receivers interpret the stored information as a condition. However, there is no mechanism to explicitly protect the access to such a shared resource[2].

StateCharts as they are described above are predominantly tailored for control. Aspects concerning information which is represented by data in a system is separated from the control part of the system. The data is just the information stored as a result of an action. To model such aspects data-flow oriented languages are proposed which have to be integrated into the StateCharts concept [48].

However, even if the separate modelling language provides concepts for data abstraction and encapsulation it does not solve the data modelling deficiencies of StateCharts. There is an integration problem. The separation breaks any encapsulation of state machines in a state. Accessing information from the data-flow model means to access data outside the encapsulation. The separation also establishes a consistency problem between the state-oriented and the data-oriented model. There is no automatic adaptation if one of the models is modified. The modelling concept is not robust against modifications.

StateCharts not only lack an appropriate encapsulation concept there are also problems with abstraction of state machines. A state with some associated transitions as source or target does not sufficiently abstract complete state machines. It does not abstract the memory-based communication mentioned above nor does it abstract events and the corresponding causal and temporal relations between these events.

Although StateCharts does not support object-oriented concepts like encapsulation, abstraction, and inheritance, extensions of StateCharts are used in object-oriented methodologies and notations. [144,141]. To explain this contradiction we have to look at the task of such an adapted StateCharts model. Its task is to model a view, the so called dynamic model of the sys-

2. In actual modelling scenarios such access conflicts are typically solved by a quasi-concurrent implementation of the state machines which automatically provide mutual exclusion.

tem. In other words, the task is just to separate the control-oriented aspects from the rest of the model. The dynamic model can be used to describe the synchronisation states of objects so that it explains aspects of a system which are not properly described by abstractions in other views of the system[3]. In other words, the model is used to eliminate deficiencies in views which strictly follow object-oriented principles[4]. The consistency issue is not solved by such notations. The other unsolved problem is the missing robustness against modifications.

4.1.2 Variations of StateCharts

We conclude the considerations about StateCharts by some remarks on variations. StateCharts use a graphical notation to describe the extended automata. So do many of the variations which are subsumed under the term hierarchical concurrent finite state machine (HCFSM). However, we do not limit our consideration to such graphical representations. We also regard grammar based specifications of such extended automata [121] as variants of StateCharts. Accordingly, all the considerations presented above apply to them.

4.1.3 SDL

Another state-oriented language is SDL (Specification and Description Language) [42,97]. It is probably the best known specification language from the telecommunication area. SDL is an ITU Standard. It is intended for specification at a very high level of abstraction[60]. The language is based on asynchronously communicating processes. The processes or more precisely the processes' bodies are described as state machines which model behaviour. Structure can be modelled by so called enclosed definitions. They can be used to define variables which are enclosed. In the terminology of the thesis they can be used to define state variables which are encapsulated in the process.

3. Please note, the use of a StateCharts model as an abstraction that means as a part of an object's interface is problematic as the modelling of the guards (conditional connections) would break encapsulation. The corresponding problem was discussed in detail in Chapter 3.

4. The interpretation we present is different from the one in [66] which considers object-oriented modelling as a collection of various modelling concepts including StateCharts and their integration as different view in a model.

The state machines and the processes communicate with each other via signals[5]. Instances of the signals are sent between the instances of the processes. A receiver queues signals to allow asynchronous synchronisation. A state machine consists of states with associated triggers and state-transitions. A trigger describes signal sets. If a process receives a signal which is in a trigger associated with a current state of a state machine it performs a corresponding state-transition[6]. The state-transition is able to manipulate variables or to output signals. SDL provides various techniques to describe state-transitions for example, it is possible to describe them by an algorithm.

The language uses a very elaborated type concept which includes the typing of processes. On the basis of this typing it is possible to dynamically create new instances of process types during the execution of the model.

In the terminology of the thesis we would say that SDL provides quasi-concurrent classes and objects or processes with asynchronous message passing. Messages correspond to signals causing events and state machines reacting to the events are used to model synchronisation states. Functional behaviour can be modelled by actions, the so called state-transitions. The language allows to resolve resource conflicts with its encapsulation of structure in combination with quasi-concurrency.

To abstract the behaviour of a process SDL provides so called gates. Gates are the interface of a process. In our terminology we would call them interaction points. That means, there is no abstraction of timing or causal relations of events part of a process’s interface.

The language supports the modelling of hierarchy and the structuring of processes by so called block type diagrams and instances. A block instance contains processes or other block instances. The communication between the components can be described in a block type diagram by communication paths. Basically, communication paths transfer signals between gates. This allows the use of gates as interfaces of block instances. The communication paths are called channels in SDL. Channels in SDL are a second[7] channel concept. It provides techniques for the aggregation and refinement of channels. Opposite to signals channels also have a timing behaviour.

To address the needs of easy code re-use in the new version of the language (SDL-92) [97] there exists an inheritance mechanism on types called

5. In our terminology we would call a signal a channel.

6. The term state-transition used in SDL is different from the term transition we use in this thesis.

7. Second, if we think of signals as the other channel concept in SDL.

specialization. There exists a set of constraints on how to specialize in order to preserve compatibility. A default specialization mechanism is to use delegation to achieve compatibility. It is possible to overcome this limitation by explicitly re-defining some properties of a type. For example, in the context of processes it is possible to re-define triggers and state-transitions but no states. If a modelling guideline advises to restrict modifications to the state-transitions the presented inheritance concept does not guarantee compatibility but at least it supports the modelling of compatibility by conformance between two processes.

Given these basic language concepts we can observe from an object-based view that there are two kind of classes and objects. Process types allow the modelling of quasi-concurrent classes and objects and block type diagrams or instances model concurrent classes or objects. With the default specialization mechanism there is an attempt to support compatibility. The problematic issue with that concept is that it breaks encapsulation especially in modelling specialized processes where properties of a state machine are re-defined. We can conclude from what we mentioned about composition in concurrent objects in Chapter 3 that the lack of standardised interfaces or concepts to abstract temporal and causal relations of blocks prevents effective re-use of models by composition. It also causes a problematic interference between inheritance and composition. From that we can conclude that although it is possible to specify a system in an object-oriented manner by using SDL-92 at a high level of abstraction re-use is still an unsolved problem.

SDL is one of the specification languages which was investigated in several research projects for its suitability to specify and implement hardware and hardware/software systems [71,73,76, 101]. The cited research results confirm the suitability of the language to model at system level. However, apart from advantages there is still a set of unsolved modelling issues.

One of the general problems when using a language and tools from the software domain is its missing integration into a hardware design flow. Particularly in SDL there is no link between a SDL description at that high level of abstraction and further synthesis steps. For example, it is not possible to simulate synthesized parts of the system within the SDL system description. As synthesis tools do not accept a SDL description there has to be a change in the description language and therefore a break in the design methodology while going through the synthesis steps. This is the reason why simulation of synthesized parts within the system description is not possible.

There have been some attempts to solve this problem by building translation tools which translate SDL into a hardware description language accepted by synthesis tools [71,73,75,76] or by simulators [107]. A similar approach is to translate SDL via an intermediate format called SOLAR[101] or HDF [129] to an HDL. But within these approaches there are problems still unsolved. The tools do not support the oo-features of the new language definition. The translators which produce a HDL model for further synthesis steps do not support the dynamic language constructs like creation and deletion of a process. Even at simulation there are some restrictions concerning the dynamic language features. The tool which translates into a HDL model for simulation purposes [107] accepts the CREATE construct in SDL only if a maximum number of parallel process instances is defined by the user.

Another problem is the modelling of simple communication and synchronisation techniques in SDL. For example, the modelling of a global clock is not provided in the language. The queuing of signals is not appropriate for modelling a clock. In the system described in [101] the translation of the high-level communication constructs of SDL into VHDL is the selection of predefined protocols by the designer and the association of them to SDL signals. Similarly in another approach [73] libraries of protocols written by the designer are used in RTL or algorithmic specifications to map communication constructs of SDL to VHDL. Generally, the lack of synchronous communication features requires explicit modelling of such communication and synchronisation objects.

As described in [76] in this approach SDL communication constructs are only used for system level specifications. In lower specification levels the SDL processes are hooked together by schematic entry tools which produce HDL-netlists as outputs. Only the internal behaviour of hardware components is modelled by a SDL processes. The method explained in [129] presents a hardware-data-flow-model (HDF) to model hardware. The mapping of a HDF-description into SDL introduces a special technique for modelling in SDL. In this technique each hardware component is modelled by a SDL process. This process collects the inputs by running through a fix sequence of states. The output is then calculated in the last state. In this approach there is no longer any correlation between SDL specification and real hardware, for example, the states of a SDL description of a non sequential circuit do not have any counterpart in the hardware.

We can conclude the considerations on SDL for object-oriented hardware design by two synoptic statements. Although SDL provides powerful object-oriented modelling features there are some unsolved language issues concerning encapsulation, composition, and its interference with inheritance. The integration of the language in a hardware design flow with tool support is only feasible for subset of SDL not containing the oo-features inheritance and polymorphism and not containing low-level communication mechanisms.

4.1.4 Estelle

There is another specification language well known in the area of telecommunication which is similar to SDL. This is Estelle [94]. Like SDL it is based on extended finite state machines. A pascal-like notation is used to describe the state machine and its extensions. The extensions concern concurrency and actions. The actions are specified as part of so called transitions[8]. A state machine is encapsulated in a module[9]. State machines communicate with each other by asynchronous message passing which is called interaction in Estelle. Interaction is abstracted by interaction points in a module's interface. An interaction point in Estelle corresponds quite well to the term interaction point as it used in this thesis. A slight extension of the concept of interaction points is that each interaction point is assigned a message queue due to the asynchronous message passing concept. The module concept in Estelle allows a module to be hierarchically decomposed into sequential or concurrent sub-modules. Such a module is able to dynamically modify the communication structure which determines the interactions of its sub-modules. Thus, a module is a concurrent object with an abstraction which does not contain any information about temporal or causal relations of its events i.e., its interactions.

As Estelle is intended for protocol specification these relations are just the crucial aspects of the model. A modelling approach in Estelle is to reflect protocol levels in the module hierarchy. A service primitive of a protocol can be abstracted by an interaction and decomposed into several interactions in sub-modules and vice versa. At the top level of a protocol hierarchy such a service primitive is called *feature* in intelligent network specification. A feature is modelled by the relations of interactions of the sub-modules. It is just the relations that make up the modelling information about the feature. However, the information is not part of the abstraction. Generating or identifying

8. In the terminology used in this thesis an Estelle transition is a transition together with its corresponding action.

9. In fact, the module is a process.

a set of interactions as a representation of a feature requires an analysis of a module implementation.

Adding features to an existing model means to incrementally modify behaviour of a concurrent object with all the modelling difficulties including the compatibility problems discussed in Chapter 3. Especially the unintended interference of new features with existing features is a problematic compatibility issue. Splitting the representation of a feature into several interrelated interactions complicates this compatibility issue. Due to a non-deterministic nature of Estelle a situation may occur where an old transition and a new transition which both model different features may be executed. A non-deterministic choice among them is an unintended interference. The compatibility problems of features caused by interference is a well known phenomenon in intelligent network specification where it is called *service interference* or *feature interaction*[10] [32].

While Estelle does not provide object-oriented concepts like inheritance which supports incremental modification as part of the language there are proposals of a specification style which supports the integration of additional features into an existing model [32]. The basic concept is only to allow modification by adding new transitions and states to a model. Existing behaviour may be changed by disabling old transitions and adding new ones. This is to preserve compatibility as good as possible. Modifications are only allowed at the transition level. It is not possible in the specification style to modify or redefine parts of a transition which model an action. Limiting the replacements to the transition level allows to develop a technique how to detect and resolve feature interaction. Due to the abstraction concept of the language mentioned above all modelling steps to add features require the analysis of the implementation of the existing model i.e., they break encapsulation.

So far, Estelle was discussed from a general system level modelling perspective, we now bring together a number of matters relating to hardware design. One obstacle for using Estelle in hardware design is a missing link to the synthesis steps following a system specification. First attempts to provide such a link are made in the COSMOS system [100] however, it seems they are based only on some first theoretical considerations. This approach does not cover any oo-techniques or other approaches to support re-use by incremental modification.

A problematic issue in designing hardware is how to translate dynamic features of Estelle into another hardware description language. Another mod-

10. Feature interaction is not to be confused with interaction.

elling issue is the non-determinism mentioned above. While in protocol specification a non-deterministic automaton is appropriate to model an unreliable service provider [83] it is less suitable to specify a system to be implemented in hardware. Because Estelle is not a simple finite state-machine but an extended finite state machine it is not possible to apply one of the existing algorithms [85] to translate the non-deterministic finite state machine into a deterministic one which would be much more appropriate for hardware implementation.

We can summarize the considerations about Estelle with two interesting observations which make the deeper look into Estelle concepts worthwhile: It is not only language features which solve basic modelling problems like compatibility issues but also an appropriate specification style. The second important observation is that to restrict modifications to transition level reduces the complexity of compatibility considerations.

4.2 Functional Programming Languages

A different approach is to use an existing functional programming language for system specifications. As a system often can be described in a functional way it seems reasonable to use a functional programming language to specify a system. Functional means that functions are first-class objects in the language. As in the other languages mentioned above which are used in the software domain the missing link between the system level description and the synthesis tools causes difficulties in the design process.

4.2.1 ML

There has been done some work to use the functional language ML (Meta-Language) [78,114] for system specification purposes and to integrate it into a hardware design flow. In most cases system development is done interactively in ML and the interaction between the ML-system and the designer is based on the read-eval-print dialogue a concept known for example from LISP.

Basically in ML a system is modelled by values containing information about the system's state and functions which are used to describe the values. Values are identified in a system by so called value identifiers or variables which are bound to the values. ML uses a type concept on values and value identifiers to detect incorrect bindings during specification. Executing a functional model means to evaluate new values by applying functions on existing values and bind them to new identifiers or re-bind them to existing ones. To achieve a high level of abstraction in modelling such functions, ML knows so called higher order functions. These are functions which may take functions as parameters or results. This is the way how to relate properties of a system modelled by functions to other functions i.e. other parts of a system. A concept of higher order functions introduces complex expressions to designate functions compared to typical non-functional languages. To make such a complex mechanism safe a type concept on functions is used. In the terminology of this thesis the predicate describing the features of such a function type describes just the function's profile. The type can be viewed as an abstraction of a function which may be shared and thus re-used by many functions.

ML has two concepts for structuring and encapsulation of value identifiers and functions. The first concept is based on abstract types. An abstract type is a data type with a set of functions defined on it. The functions are called its interface. Describing values and binding them to value identifiers of the abstract type is only possible by using functions of the interface. The mechanism encapsulates the values and bindings but not the value identifiers.

The second concept which is considered to supersede the abstract types is based on modules. There are two types of modules, structures and so called functors [114].

A structure basically is a collection of types, values and other structures. A what is called in ML signature can be used to abstract and encapsulate the items in a structure. It provides type information about the items in a structure. It corresponds quite well to the term signature as it is used in this thesis[11]. Encapsulating a structure within another structure allows the modelling of hierarchy.

A functor is a mapping from structures to structures. Such a mapping is a technique to describe new structures as a mapping from existing ones. We can think of a functor as a polymorphic or generic structure.

Such a concept of polymorphism supports re-use at module level. A second concept for polymorphism supports re-use at function level. It is based on so called polytypes. Polytypes allow to define value identifiers whose values may range over a collection of types. Thus, we can think of polytypes as similar to a class-wide type mechanism. However, there is no hierarchy fol-

11. It corresponds quite well if we consider function types as an abstraction of functions as mentioned above.

lowing inheritance relations between classes or types. ML does not provide an inheritance concept. Even very similar types have to be constructed completely new as different types.

For example, in a program it might be useful to bind an integer value to an value identifier as well as a real number. In that case in ML one has to construct a datatype number with constructors int and float. The meaning is that a value identifier of type number can either be bound to int or to float. If the original declaration of number has to be extended by adding a constructor complex to number the datatype has to be re-defined, the constructor complex has to be added, and the functions using the datatype number have to be modified. Delegation of functionality as part of such a modification is supported by modelling the modified functions as higher order functions applied to the original functions.

To conclude the considerations on abstraction and re-use in ML we can state that ML only provides limited support to extend an existing model without touching the original model.

After the discussion of general modelling and re-use concepts of the languages we now consider timing and concurrency aspects which are relevant for hardware specification at system level. It turns out that these are the weak points of the language. Constructs to model the timing of a system are missing in ML. Although the language provides some constructs for binding identifiers simultaneously to values there is no real concept for concurrency. There is no language support to express timing or causal relations in the context of concurrency. This makes it questionable if ML is appropriate to model hardware systems which typically consist of interacting concurrent system components.

4.2.2 HML

We discussed the modelling concepts of ML in that detail because there exists a proposal to use an extension of ML for hardware modelling. It is called HML [123]. HML claims to provide important features for hardware modelling that are lacking in conventional HDLs. Especially higher-order functions and the polymorphism concept described above are considered to support modelling for re-use at an high level of abstraction.

HML has an extended type system compared to ML which is designed to flag design rule violations statically at compile time. Anyhow, it is a goal to support the verification process of a system description. The idea is that hardware models written in HML are similar to structural descriptions in (interactive) theorem provers which are based on ML.

However, the difficulty that still remains is the missing support for expressing timing and concurrency in ML and thus in HML.

4.3 Formal Languages

Formal modelling methodologies and languages follow the idea that it is not sufficient to validate a model by running an executable model but that it is necessary to verify properties of a system by proving them. As stated above, the functional language ML provides proof support with its strict semantics.

4.3.1 Z

The concept of a functional programming language is extended in the specification language Z [33] by using relations instead of functions. Both, the syntax and the semantics are defined in a formal way. The meta language to specify the semantics is based on the Zermelo-Fraenkel axiomatization of set theory. One of the main goals of the language is to allow a designer to prove the correctness of a Z specification in a mathematical way by the means of its formal semantics.

The Z notation is a calculus which describes a set, which is the state space, by using relationships. In difference to set theory, a type concept is added to Z. A state space is described by so called state schemas. A schema consist of variable declarations which describe constraints on the variables and it consists of properties described by predicates. Mapping of variables to types which describes the variable declaration as part of a schema is called signature[12] in Z. Using schemas as types in variable declarations of other schemas allows to hierarchically compose a state space. A second composition concept is schema inclusion. It means that variables and predicates of schemas become parts of a new schema by merging declarations and conjoin predicates. Another composition technique of two schemas is schema conjunction. It merges the predicates and conjoins the predicates to form a new schema.

Operations can be described to work on the state space. Such an operation is a relationship between input variables output variables and a pair of states,

12. The term signature is used differently from the one introduced earlier in this thesis.

a state before and a state after the operation [160]. From a state-oriented view such an operation describes state transitions. Like states, operations are described by schemas.

We can state that the language Z provides powerful techniques for modifying and combining schemas as described above. Abstraction and encapsulation of schemas is a more problematic issue. If we look at signatures[13] as an abstraction of a schema it abstracts properties of variables but it is not possible to abstract variables themselves. This is a limitation in the abstraction of states of the state space by a signature of a state schema. The same holds for the abstraction of some internal state information in an operation.

In summary, a Z specification describes a relationship between initial states and final states. It is possible to have a relationship between one initial state and several final states. Even an infinite number of states is possible. In other words, Z has non-deterministic operations. This allows one to specify non computable functions for which finite algorithms to compute the final states do not exist. Therefore simulators of a Z specification are not available. Simulation is only possible if the relationship between initial and final state is restricted to functions.

Successfully developed hardware starting from a Z specification was designed by refinement of the specification [127]. This refinement was done manually. After each refinement step it was verified that the refinement matches the original specification. The mapping of the final Z specification to the real hardware was done manually. The verification of the refinement steps has to be done by mathematicians who are familiar with the calculus and proof techniques. The success story is the motivation for discussing Z as a language for hardware specification at system level. The conclusion is that if certain aspects of a hardware system with limited complexity have to be analysed and corresponding properties to be proven it can be a very useful language in the hands of experienced mathematicians. Due to complexity limitations in the proofs the language is not appropriate to model a complete system with all relevant aspects of hardware modelling. Looking at the abstraction and encapsulation concepts of the language we also can conclude that it does not support for specifying abstract re-usable models well.

13. To be more precise a name of a schema and its schema type i.e., its signature.

4.3.2 Object-Z

Object-oriented modelling concepts claim to support the modelling of re-usable models at a high level of abstraction. Object-Z is an extension to the Z language to facilitate specification in such an object-oriented style [49]. It applies a class concept to the type concept of Z i.e., a class is used as a type. A collective definition of a state schema with associated operations is the definition of a class. The operations work on a state space defined by the state schema.

As in Z operations declare modifications of state variables as part of their signature i.e., as part of their abstraction. From an object-oriented point of view this breaks encapsulation. In Object-Z it is possible to introduce additional state variables in a state schema whose modifications are implicitly declared as part of signatures. It is possible to interpret such a mechanism as a concept to preserve the encapsulation of state variables.

Inheritance is modelled as a kind of schema inclusion. Principally, it is the same mechanism as described in the section about Z. Inherited definitions and those declared in the derived class are merged and inherited state schemas and those declared in the derived class are conjoined. Operation schemas with the same name are conjoined. The interesting thing about this concept is the way it preserves compatibility. Existing structure of a class i.e., its existing declarations may be restricted by adding properties to them. Operations which override inherited operations may restrict existing relationships between states i.e., they may add properties to the state transitions.

Looking at the definition of compatibility by conformance given in Chapter 3 we can state that the derived class subsumes the operations of the inherited class. The invariants of the derived class imply the invariants of the inherited class. For each operation of the inherited class the postcondition of the derived class implies the postcondition of the inherited class. The only problem is that the preconditions of the inherited class do not imply the preconditions of the derived class. To meet the conditions we could think of a modelling style which does not allow to put any restrictions on inherited preconditions. In combination with such a modelling style the inheritance mechanism would provide compatibility by conformance. The problem is that if such a modelling style allows modifications of the preconditions analysis of the inherited class is required to be sure not to restrict the inherited preconditions and this breaks encapsulation.

If we try to compare this mechanism with the basic inheritance concepts presented in Chapter 3 it appears to be a variation of delegation. Re-defined operations delegate parts of the operation to the inherited operation. The reason why such a delegation mechanism works without any compatibility problems is that it does not imply any special timing or causal relations between the inherited and the derived part of the operation. Conjunction as delegation basically means that the operation has the old and new properties at the same time[14].

To express causal relations special operations are used to describe the relation explicitly. For example, the sequential operator from Z can be used to describe sequential composition of operations. As we mentioned earlier in Chapter 3 in the context of synchronisation an object's behaviour often depends on its history. To express such properties Object-Z provides a predicate over histories of objects expressed in temporal logic. We directly can apply the considerations and conclusions about modelling of history information in objects which we have presented in Chapter 3.

We conclude the considerations on Object-Z by remarking that the language would require formal semantics of Z to be extended for Object-Z if the language is used to formally describe hardware systems.

4.4 Process-Oriented Languages

In Chapter 3 we presented the basic ideas of object-based modelling. We introduced a process to be a special kind of object with interactions replacing operations and with interfaces which consist of a set of interaction points. We now look at languages which use such processes as first-class objects.

4.4.1 LOTOS

We saw in previous sections that quite a few languages proposed for hardware specification at system level originally were developed for specifying protocols. In this section we are going to discuss another language used in the telecommunication world for describing protocols. It is LOTOS (Language of Temporal Ordering Specifications) [83,150]. The language is based on process algebra and is therefore treated as process-oriented language in

14. Actually, there is a slight difference between conjunction and a parallel operator which is a means of the language to express parallelism. The difference concerns inputs and outputs with the same basename. Conceptually, they can be used to communicate with each other in a parallel operator. Inheritance just omits such a communication.

this survey. LOTOS also has well established theories for formal verification. Therefore it also would have been possible to discuss it in the previous section about formal languages.

A LOTOS model describes a system by the observable interaction with its environment. An observable interaction consists of so called events and the ordering of the events. The event concept of LOTOS corresponds quite well to the notion of event used in this thesis. However, the states between the corresponding state transitions are not part of a LOTOS model. They are not directly observable from outside a system. In other words, it is the history of the events that implicitly models a system's state. The system implementation is virtually a black box. The ordering of events is just the temporal or causal relations between events. A restriction in the relations is that two events never may occur simultaneously. With a concept to explicitly express parallel behaviour we can classify LOTOS as a quasi-concurrent language.

Behaviour is modelled by processes which are used to describe sub-systems. A process specifies possible sequences of events that may be observable to other processes or to the system's environment. The possible sequences are described by so called behaviour expressions. With such an expression it is basically possible to express sequential occurrence and alternative occurrence of events in a sequence. Specification of alternative occurrence of events may introduce non-deterministic behaviour if an actually occurred sequence may follow more than one specified alternative.

The process has an interface which consists of so called gates. The interface can be viewed as an abstraction of the process. The gates model what is called in the terminology of this thesis interaction point. An event corresponds to the activation of a gate. If we observe such a system we could interpret the activation as synchronous sending or receiving of messages.

It is possible to specify behaviour as interleaving to model (quasi-)concurrency. This allows to compose quasi-concurrent processes[15]. The processes can synchronize at gates i.e., they synchronize at events. We can think of a kind of multi-way rendez-vous mechanism here where a process sends a message which is received synchronously by other processes. It is possible to either hide the synchronizing events between processes from the rest of the system or to make them observable from their environment. Making them observable allows a modelling style called constraint oriented specification which designs a system as a collection of concurrent processes. Each process

15. LOTOS distinguishes between process definition and instantiation. To simplify the explanations we use the term process for both.

describes a certain ordering of events which is interpreted as a constraint of the observable event which is used for synchronisation. Extending the system is possible by adding processes which model additional constraints. The rendez-vous mechanism via the observable event automatically includes the new constraint in the model.

It is only such composition techniques used as part of a modelling style which allows the re-use and the extension of existing models in LOTOS. There are no object-oriented concepts like inheritance in LOTOS. Due to introducing new sequences of events composition techniques may introduce service interference[16] which must be detected and solved [47]. Again, we can observe that abstracting behaviour by interaction points is not sufficient for the modelling of re-usable models without breaking encapsulation.

A proposal to use LOTOS for hardware specification [59] mentions the ability of LOTOS to specify concurrent systems across many levels of abstraction. However, its examples are restricted to gate-level circuits. Due to the abstraction and encapsulation concept the presented composition techniques of the proposal are not appropriate for modelling at system level.

Although there are still unsolved issues we can state that LOTOS with modelling concepts like concept oriented specification and the ideas on specification and detection of service interference contributes interesting approaches to the general discussion about design for re-use at system level.

4.4.2 CSP and Occam

A language which like the previously discussed LOTOS can be classified as process oriented as well as formal is CSP (Communicating Sequential Processes) [82]. A language which is closely related to CPS is Occam [92] and its successor Occam 2. It is based on the model of CSP for communication and for parallel execution of processes. Both languages are synchronous concurrent languages.

Although the name CSP suggests that the language only allows the modelling of sequential processes the CSP model for parallel execution of processes supports the modelling of concurrent processes. A process is hierarchically composed of other processes. To compose a process out of existing ones it is possible to specify the existing ones either to execute sequentially, in parallel, or alternatively. The most elementary processes a

16. The principle reason for service interference and the related problems were discussed in Section 4.1.4.

model finally consists of are provided by the language as primitives. They model assignment of values to variables within a process or across process boundaries as input and output primitive.

A process can be abstracted by its interface which basically consists of interaction points which abstract the communication of a process that is performed by its input and output primitives. From a modelling point of view a process appears like a sub-program which can be called in other sub-programs sequentially, concurrently or alternatively to other sub-programs in the calling sub-program. In Occam the interface is not only an abstraction boundary but also a distribution boundary. That means, from inside a process an interaction point abstracts a channel which is used to communicate between concurrently executing processes. The channels are used to model the structure of the communication between the processes. It is not possible for a client to directly address its server. The channels are only uni-directional which requires a client to explicitly model reply-scheduling and which at the same time offers the possibility to explicitly model the concurrency between client and server. However, if we look at this from an object-oriented point of view the reply scheduling in one-way message passing breaks encapsulation as discussed in Chapter 3.

The interaction concept between the processes is based on a rendez-vous mechanism. An input or output command is blocked until the corresponding output or input command is executed. It is possible to allow several inputs at the same time by composing the input primitives alternatively. Alternatively means that one of the processes is non-deterministically chosen. This introduces a non-deterministic behaviour in a specification. To avoid this it is possible to attach each process in an alternative a guard. Such a guard consists of an input primitive and an optional boolean expression. The remaining non-determinism can be removed by assigning priorities to the alternatives.

Combining the modelling of guarded processes in an alternative with the sub-program perspective of modelling results in an object-based modelling concept. It is mentioned in [82] that the construction of subroutines as processes operating concurrently with its user process offers the possibility of passing commands to the subroutine. This has the effect of multiple entry points and therefore can be used like a Simula class instance. In the terms of the thesis this means that guarded processes can be used to model operations of an object. The alternative construct models mutual exclusion synchronisation. The input primitive of the guard specifies the message which can be sent by a client to invoke the operation and the boolean expression of the guard may be used to model user defined synchronisation. In such a reflective modelling style an operation modelled by a process is a kind of thread.

Further essential concepts of like inheritance and polymorphism which would make the object-based modelling style object-oriented are not provided. There are no concepts to support re-use and modification of existing models.

There are proposals to use Occam for hardware specification [35,108] and to use Occam 2 for hardware software co-design [63]. The unbuffered communication mechanism is explicitly mentioned by the inventor of the CSP language in [82] as a mechanism which matches closely physical realizations as wires cannot store messages. Missing timing concepts in the language restricts the synthesis and simulation of hardware to delay-insensitive control circuits in [35]. That means the correct operation of the circuit does not depend on any assumptions about delays in the wires or operators of the circuit. Any timing relations are to be explicitly modelled as causal relations. Links to other synthesis tools which offer other synthesis techniques are not provided. This is different in the co-design approach [63] where timing information is added to a model by tags. To make the methodology more suitable for hardware specification an event driven simulation strategy which runs on a simulator developed for that purpose is used for simulation. Synthesis tools translate the specification including the tags into assembler code and a HDL description. We can state that this is a successful concept to adapt specification techniques from the software domain and to modify it for hardware specification. From the hardware point of view it puts a modified version of modelling language from the software domain at top of an existing hardware design flow. The approach provides a very flexible and at the same time simple technique to model concurrency and a very elegant concept to model communication. With this primary focus of the modelling language the approach does only provide limited support for re-use and extension of existing models.

4.5 Imperative Programming Languages

Imperative languages which describe a program as a sequence of statements with each statement changing the system's state are well suited to define an algorithm. With procedures as a concept to abstract, encapsulate, and decompose behaviour imperative languages are appropriate for behavioural modelling at algorithmic level in hardware design.

We saw earlier in Chapter 2 how such procedural languages successfully can be used at algorithmic level in specific application domains like digital signal processing. Programs written in a sequential imperative programming language are very often used to write an executable specification which allows to evaluate signal processing algorithms. Modelling systems or parts of a system outside these very specific domains at algorithmic level shows deficiencies of procedural modelling. One of the main limitations mentioned in Chapter 2 is a missing robustness with respect to modifications of data in a model due to missing encapsulation concepts. Another deficiency stated in Chapter 2 concerns the use of sequential statements to describe complex timing relations in a system. To overcome the limitations some procedural languages are extended by mechanisms to express parallelism i.e. to supersede the strict sequential ordering of statements. Removing the strict ordering of statements removes the strict ordering of activities in a model and thus introduces concurrency. (Compare Chapter 3)

An example of such a language is the previously introduced language Occam. If we consider processes as procedures which are used to model statements then Occam can be classified as procedural and thus imperative programming language. Occam extends the pure sequential ordering of statements by allowing statements to occur alternatively or in parallel. With its use of processes as first-class objects Occam also provides an appropriate encapsulation concept for data.

In Chapter 2 missing concepts for re-use of synchronisation models were mentioned as a third deficiency of procedural languages. This corresponds to the conclusions drawn in Section 4.4.2 that there is only limited support for re-use and extensions of models in Occam.

4.5.1 Ada

Another imperative programming language which provides concurrency as part of the language is Ada [36,95]. Especially, it provides quite a number of high level communication and synchronisation mechanisms. Concurrency is modelled by processes which are called tasks in Ada. Their interface consists of entry points. They abstract synchronisation mechanisms which require the explicit modelling of activities to accept messages. The tasking mechanism allows to classify the imperative language also as object-based. Different to Occam, a concept separate from tasking is used to model procedures. In other words, the procedural features are not extended to object-based features. Ada is an object-based language independently from the fact that it is also a procedural and thus an imperative language. Ada is also an object-oriented language. It extends the procedural features of the language by an encapsulation concept for data and a related inheritance mechanism on the data. The language does not introduce object-oriented concepts by extending the object-based features of the language. Tasks are used to model objects from an object-based point of view and variables of certain types are used to model objects from an object-oriented point of view. While the object-based objects correspond to our notion of active objects the object-oriented objects have a more passive characteristic. It has been reported several times [3,17,170] that an integration of such an active object with the object-oriented concept, particularly inheritance, cannot be satisfactorily achieved. Based on these considerations we shall investigate the basic reasons behind the decision not to introduce object-oriented concepts as an extension of the object-based features later on from a more general point of view. (Compare Chapter 8)

We shall see that the analysis of the reasons plays an important role in our effort to develop a methodology for system level modelling of hardware. It motivates the investigation of object-based features in existing hardware design methodologies and languages with special respect to the possibilities of extending the methodologies and languages by inheritance.

Several proposals have been made to use Ada as a hardware description language. Among the publications on that topic, [30] in particular confirms the suitability of Ada for behavioural modelling. Modelling at system level is evaluated in [55]. Different to most of the other publications the studies in [30,55] explicitly refer to the latest standard of Ada [95] which includes the object-oriented features mentioned above. Nevertheless, only object-based features and related synchronisation concepts have been subject of investigation. Research how to apply the object-oriented features of the language in hardware design to make the models robust and re-usable is still to come.

Although there is an approach [157] which translates sequential Ada models to a hardware description at behavioural level it is an unsolved issue how to synthesize concurrent Ada models with high level synchronisation mechanisms to hardware. In other words, there exists no integration of Ada models written at system level in a hardware design flow.

4.5.2 Summary

We conclude this section by stating that apart from some domain specific applications sequential imperative languages cannot be considered as specification languages. Extending procedural languages to provide concurrency and encapsulation of data removes some obstacles for modelling at system level, however, extension and re-use of existing models remains still an unsolved issue.

Chapter 5

Object-Oriented Hardware Design Methodologies

In the previous chapter we introduced various methodologies and languages which are used in the software domain for system level specification. It has been carefully analysed how object-oriented principles are supported. An important topic which has arisen is the integration of the methodologies in a hardware design flow. In this chapter we take the complementary approach to find an object-oriented specification methodology for modelling at system level. Instead of using languages and methodologies from the software domain and adapt them to hardware design we survey existing HDLs and look at proposals to modify them for system level design.

This requires an analysis of the languages with special respect to their support of object-based and object-oriented principles. Their application domain in terms of abstraction levels has to be identified. Concrete deficiencies for modelling at system level already referred to in Chapter 2 have to be recognized for each HDL.

Among the various HDLs we are particularly concerned in VHDL which offers a number of interesting techniques to describe hardware at various levels of abstraction. We shall present the basic modelling concepts of the language before we discuss problems and limitations of VHDL for modelling at system level. We then discuss a large number of proposals how to overcome the limitations and how to solve the modelling problems. All proposals correspond on the idea of extending VHDL by object-based or object-oriented features. However, they differ very much in the language features of VHDL which are taken as a basis for the object-oriented extension. With the considerations of Chapter 3 about different concepts for object-orientation and concurrency as a background we shall analyse each proposal in detail.

5.1 Survey on HDLs

We start this chapter on object-oriented design methodologies with a survey on existing HDLs. It is not the goal of this survey to give a complete overview on HDLs but rather to identify some general main aspects of the languages which allow us to argue why VHDL may be an appropriate candidate for being extended to a system level language.

From the multitude of HDLs some have been chosen for this overview which are designed for modelling at abstraction levels above register-transfer level. However, high level languages for very specific applications domains are not considered, for example from the DSP domain. With HardwareC [103] and DACAPO III [140] two languages from the academic world are part of the survey. HardwareC was chosen because of its wide-spread use in various academic tools for hardware and hardware/software co-design. DACAPO III is mentioned in this overview because it offers some very interesting high level design concepts and the language is claimed to be appropriate for system level design [140]. Verilog [124] and VHDL [87] as today's most important and successful[1] HDLs are also considered in the survey.

It is interesting to note that all HDLs mentioned take some concepts or notations from existing programming languages. DACAPO III follows Pascal [99] and MODULA [96], VHDL's syntax is very similar to Ada [95], and Verilog is in several respects a bit C-like [102]. Differences in the ancestors may already indicate differences in the modelling power of the HDLs. HardwareC as the first language to be presented in this overview is not only C-like but is almost an extension of C.

5.1.1 HardwareC

HardwareC [103] is a procedural language which is based on C. It is extended by concepts which are useful to describe hardware namely concurrency and a concept to specify typical constraints of a hardware system like timing constraints or resource constraints. The language also introduces a possibility to model views which describe structural relationships and physical interconnections between the components of a system.

HardwareC allows to model an executable specification of a system. The simulation of such a model can be used to validate the functionality of the

1. Successful means widely used and accepted by the designers. This does not automatically mean superior to other HDLs from a technical point of view.

specification. However, the developers of the language claim HardwareC not to be a language primarily for simulation but for synthesis. It has specific constructs for the design of synchronous circuits, i.e., circuits controlled by a clock.

Basically, a system modelled in HardwareC is a collection of processes. Such a collection explicitly models coarse grained concurrency. Inside a process the language follows the imperative and procedural modelling style of C. The imperative style with its sequential statements is enriched by two concurrency concepts which allow to model a fine grained concurrency. One is based on the so called parallel compound statement and the other on the data-parallel compound statement.

The first one groups the operations of statements to be executed in parallel. We could think of each component of such a statement as a thread. It is the designer's task to assure that there are no access conflicts to shared resources from within a parallel compound statement. Although in many cases hierarchical nesting of concurrency should be avoided in the modelling of concurrent systems [80] the nesting of such parallel compound statements together with sequential statements shows itself to be useful for the modelling of protocols across several protocol layers.

The other concept abstracts from specifying each timing relation between statements explicitly either as sequential or parallel. A data-parallel compound statement is modelled as a sequence of statements. The statements can be executed in parallel subject to existing data dependencies. In other words, causal relations implied by the sequence of statements are analysed and transformed into temporal relations. The transformation is actually part of the synthesis, that means, concurrency is introduced in a synthesis step. The degree of parallelism is determined by the synthesis system which is used to design the system, i.e., it is not language but implementation[2] defined.

This raises the issue of portability and re-use. The concurrency concept introduces a synthesis semantics to HardwareC. The exact semantics depends on the specific synthesis tool and thus is implementation defined. The missing portability due to different semantics has to be considered when re-using a model.

The parallel compound statements allow the modelling of concurrent processes in HardwareC. They can be structured or hierarchically composed

2. Implementation defined in the sense that it depends on the implementation of the synthesis step and in the sense that it depends on the synthesis result which can be referred to as an implementation.

by so called blocks. A block is a kind of module which can be used to encapsulate processes or other blocks. A block or process is abstracted by its interface in a so called declaration. The interface is at the same time abstraction boundary and distribution boundary. It abstracts the communication and synchronisation of processes and blocks via shared variables or channels. A channel is called in HardwareC channel variable. It is abstracted in the interface by an interaction point which is called channel[3]. The communication mechanism and the protocol which is abstracted by a channel and thus by an interaction point is a rendez-vouz concept. The communication mechanism which is based on shared variables is abstracted in the interface by a global port. The global port represents the shared variable. The access protocol of ports i.e., how to avoid or resolve access conflicts on the shared resource is left to the designer. An abstraction of a block or process does not contain any information on these protocols or on protocols using the channels.

This analysis of the process mechanism in the terminology of this thesis allows us to conclude from what was mentioned about explicit message passing via channels in Chapter 3 that robustness and re-usability of blocks and processes is a problematic issue in HardwareC.

We now briefly mention some other language features relating to abstraction. HardwareC allows to specify timing behaviour of a model. This can be achieved by annotating statements of a model with timing constraints. However, there is no concept to abstract this information in an interface. Another language concept which lacks almost completely abstraction is the type concept. HardwareC has three major variable types which only allow modelling at bit level. In other words, the language only supports a simple bit vector based value model. This makes the use of the language questionable above register transfer level and prohibits the use at gate level or below. The restricted value model also does not appropriately support the modelling of communication structures which are based on buses.

Re-use is supported in HardwareC by so called template models which abstract certain properties of a model. The abstracted information is passed as a generic parameter of type integer to a concrete instantiation of the model. Such a re-use concept is applicable to macro cells as described in Chapter 2, however, it is important to note that it does not provide the required abstraction concepts for synchronisation described above and thus is not appropriate for re-use of complex models at system level.

3. We have to carefully distinguish between the term channel used in this thesis and the term channel as it is used in HardwareC.

We conclude the considerations about HardwareC for system level modelling by some observations. Although it is important that there is no gap between a specification at system level and further design steps, i.e., there should be mechanisms which allow a specification to be synthesized to lower levels of abstraction, it turns out to be a problematic concept from a portability and re-use aspect to use a language whose semantics is an implementation defined synthesis semantics.

Another observation is that even very elaborated concepts for modelling concurrency at various granularities on top of a procedural language do not improve the modelling capabilities at system level with respect to a model's robustness and re-usability. The key issue remains as in all the other modelling methodologies presented in the previous chapter how to model and abstract the communication and synchronisation.

5.1.2 DACAPO III

According to its developer the HDL DACAPO III [140] is a language which allows the modelling at various levels of abstraction and which provides special support for modelling at system level. It is a procedural language extended by a module concept which allows to encapsulate procedures. Modules are units which only exist at the top-level of a system description[4]. For hierarchically structuring a system it is possible to nest procedures.

So called explicit variables and a special interface mechanism provide a procedure with characteristics of a process. Explicit variables store values even if the procedure does not execute any statements. The procedure interface may contain so called implicit parameters. An implicit parameter has the property that its actual parameter has always the same value as the formal parameter. This allows to pass information across an abstraction boundary during the execution of a procedure.

A procedure is sometimes referred to as procedure-object because it can be viewed as an object with one operation which can be invoked by a procedure call. A procedure call is a request to perform the operation. If the operation already is executing the request is cancelled. In case of multiple requests one is non-deterministically chosen. Such a strategy models exclusion synchronisation with respect to external operations calls. A procedure with more than one operation which provides mutual synchronisation is a so called export procedure. An export procedure has an interface which abstracts all its

4. Such a top level unit might model a library.

operations. In fact, it is an abstract data type. What is missing in export procedures is a concept to model condition synchronisation. There is no concept to synchronize callers by queuing or even re-queuing the requests. It is possible to model critical regions but there is no modelling support for conditional critical regions.

The explanations about synchronisation in procedures imply that the language supports the modelling of concurrency. Concurrency is introduced in a model by a concurrent compound statement and a parallel compound statement. A concurrent compound statement consists of a list of statements. The statements may be executed in an arbitrary order. A parallel compound statement consists of a list of assignment statements assigning values to some variables or parameters. All the assignments are executed synchronously i.e. at the same time.

Despite these synchronisation concepts, procedures may communicate with each other via shared variables. The implicit parameters mentioned above can be used to abstract the variables. For synchronisation explicit statements can be used which suspend the execution of a sequence of statements until the implicit parameters and thus the shared variables have certain values or perform certain transitions i.e., changes of values. This implements a synchronous communication.

Asynchronous communication can be implemented by an interrupt mechanism. Each procedure may contain an interrupt service routine. If an interrupt is sent by a procedure each procedure containing a corresponding interrupt service routine suspends its execution and executes the routine. Afterwards it continues its originally performed activities. From an object-based view we could think of an interrupt as an operation. Sending an interrupt means to broadcast an operation request to all objects with a corresponding operation. The operation is executed in mutual exclusion. It is possible to model condition synchronisation by explicitly enabling or disabling the interrupts in each procedure. However, there are two severe restrictions from an object-based point of view. The operation cannot have any parameters or return values and it is not abstracted in the object's i.e., the procedure's interface.

Both communication concepts in combination with the concurrency mechanisms support the modelling of protocols. The procedures with implicit parameters can be used to model and encapsulate the services and the service access points of a protocol. However, a procedure's interface does not contain any abstract protocol information. This has a negative effect on robustness and re-usability of procedures. (Compare Chapter 3)

The features of the language presented so far considered important aspects for modelling at system level for which it does not matter if it is a hardware or software system. We now present features which are hardware specific and thus are different from the ones of the specification languages presented in the previous chapter.

The language has a simulation time. It can be used to model the timing behaviour of a system. It is possible to delay the execution of statements until the simulation time has reached a certain value. It is also possible to delay the assignment of values to variables. That means, the value to be assigned to a variable is calculated during the execution of the statement. The actual assignment is delayed until the simulation time has reached a certain value. The presented timing model can be successfully used at register-transfer-level and below to model transition delays of circuits. A generic parameter concept allows to abstract such properties of a component in an interface. However, there is no mechanism to abstract more complex timing relations referring to some timing values in components modelled above register-transfer level.

It is interesting to note that the presented timing model can be used to model concurrency by scheduling assignments to occur at the same simulation time. This may cause problems when used in combination with system level modelling concepts for synchronisation. Consider for example an export procedure which models an abstract data type to provide mutual exclusion synchronisation for accessing a critical resource. The operations which are intended to model the critical region are guaranteed not to execute concurrently. However, the operations may contain delayed assignments to critical resources which are modelled as part of the export procedure. These assignments are not prohibited from becoming effective concurrently. We can say a delayed assignment may violate a critical region.

Compared to many other HDLs DACAPO III has an advanced type concept with records, arrays, and enumeration types to support data abstraction at various levels of abstraction. Part of this type concept is a predefined value model for modelling at RT-level and below which supports the representation of unknown values or different signal strength.

If we resume the previously presented concepts of the language from a system level modelling point of view with special respect to re-use capabilities we can state that the language offers rudimentary features for system level modelling which could be improved by more elaborated synchronisation mechanisms. Corresponding abstraction mechanisms would be required for re-use concepts. Another aspect related to re-use is the non-determinism of the language which raises an unsolved portability issue.

Finally, an important result of the considerations on DACAPO III modelling features which can be generalized to other HDLs was to shown how a timing model can implicitly introduce concurrency which may unintentionally conflict with system level synchronisation concepts.

5.1.3 Verilog

Verilog [124,165] is a HDL which is intended to be appropriate for the modelling at a wide range of levels of abstraction. Like in HardwareC many language aspects of Verilog appear to be similar to C concepts at least at first glance. Different to HardwareC which is defined by a synthesis semantics the dynamic aspects of Verilog are defined by the way a Verilog simulator works. In other words, it is based on a simulation semantics.

Verilog describes a system as a set of modules. A module may consist of hierarchically nested modules and so called behavioural modelling constructs. The constructs are used for imperative modelling of activities. Concurrency is introduced in the modelling concept by so called processes. A process could be thought of as a thread. Another way to introduce concurrency in a model is by using fork and join statements which create and remove additional threads in a process. In the terminology of the thesis we could say that a module that contains Verilog processes is a concurrent process. The corresponding module interface of such a module forms an abstraction boundary and a distribution boundary.

Communication between modules and processes is performed by some kind of shared variables which are called nets. The interface of a module can be connected to such shared variables via so called module ports. That means, from inside a module ports abstract nets. Assigning a value to such a net makes the value visible in any module connected to that net. Access conflicts to nets are resolved by the built-in value model of Verilog. It is a bit level model which models various signal strengths. There is no abstraction level above that value model.

Synchronisation between the processes is achieved by so called event control statements and wait statements. Executing such a statement in a thread means to suspend the execution of the thread until an event in form of a value transition on a net has occurred or in the case of the wait statement some conditions are met.

The simulation semantics provide a timing model based on simulation time. The timing model allows to delay an assignment of a value to a net. The changed value may appear at some time specified in the future. This is achieved by suspending the thread which executes the assignment statement for a specified amount of simulation time. Different to for example DACAPO III this timing model does not implicitly introduce concurrency. It is not possible from within one thread to specify two different events to occur concurrently.

Concepts for abstraction and re-use are only at the module level. It is possible to pass some generic parameters to a module. The parameters are limited to timing values which are used to abstract delay values and to integer values which are used to abstract some bit widths. From these limitations it is obvious that re-use concepts in Verilog focus mainly on gate level modelling with macro cells.

Generally, we can conclude that the rudimentary value and type concept with its missing abstraction mechanisms is an important aspect which makes Verilog unsuitable for system level modelling.

5.1.4 VHDL

As stated at the beginning of this chapter we are going to consider the features and modelling techniques of VHDL [4,87] in more detail. In this section we give an overview about the main concepts of the language. To understand all aspects of the following chapters a detailed knowledge of the language is required. Providing such a knowledge is beyond the scope of this section. Nevertheless, it is an attempt to explain the main points of the language in order to support a better comprehension of the ideas which we are going to presented in the following chapters. The VHDL expert may forgive some of the simplification used in the explanations.

VHDL's semantics is based on a simulation semantics which is described in the language reference manual (LRM) [87]. It describes how a system description behaves in a simulator. To perform such a simulation it is necessary make the model known to the simulator. This is achieved in two steps. In a first step parts of the model which are called design units are analysed. If no errors are detected in such a design unit it is put into a repository of the simulator which is called library. It is possible for models to refer to other models in such a library. Special configuration management features of the language allow to describe a version of a model by forming a model with design units from various libraries. In the second step which is called elaboration the model is put together by the simulator according to the information from the configuration and it is prepared for simulation.

From a modelling perspective libraries can be regarded as modules at the top level of a system. The design units introduce two other module concepts below the top level modules into the language. The design units entity and architecture provide one module concept and the design units package and package body the other. In both concepts one design unit is the interface of the module and the other is the module implementation or body. We first have a look at the entity and architecture.

According to the LRM the design entity is the primary hardware abstraction. Regarding entities and architectures as modules means that an entity is an interface which abstracts corresponding implementations in architectures. Each entity may have one or more corresponding architectures. For an architecture the entity is an abstraction boundary as well as a distribution boundary[5]. The module concept of entities and architectures allows to hierarchically nest them. This is not possible by directly modelling an entity and architecture as part of another architecture but only by referencing them in a design unit from within an architecture or by referencing them by the configuration mechanism as part of another architecture. Thus, an entity and architecture in a library is a kind of template which can be instantiated in other architectures.

Beside instantiations of other entities and architectures an architecture may contain so called signals and processes. A process consists of a sequence of statements which are sequentially executed during simulation. The processes in an architecture are executed concurrently. A process may have some local variables to store its state. The abstraction of a VHDL process is a label which gives the process a name. This is at the same time the abstraction boundary of the process. Nothing inside the process is visible outside nor is there any interface which could abstract objects from inside the process. In fact, a VHDL process strictly speaking is not a process in the terminology of this thesis. It is rather a thread with some internal variables and a strong encapsulation. Its distribution boundary is the distribution boundary of its surrounding architecture.

As stated at the beginning of this section VHDL has a simulation semantics. It requires a process to execute its sequential statements in so called simulation cycles. Each process contains synchronisation points in form of so

5. As we shall see later on there are some reference mechanisms which allow to expand the distribution boundary with respect to packages.

called wait statements to synchronize its activities with other processes. Statements of a process between two synchronisation points are all executed within one simulation cycle. After executing the statements and reaching a synchronisation point the execution of a process is suspended at least until the next simulation cycle starts. Only if each process in a system model has reached a synchronisation point the next simulation cycle starts in which again some of the processes execute their statements between two synchronisation points.

The signals in an architecture are used for communication between the processes. They can be viewed as a kind of shared resource which stores a value of a certain type. An assignment statement in a process sends a request to the signal to store a value. Such an assignment statement is always a request to schedule the assignment to become effective in a future simulation cycle. Conceptually, the shared resource queues the request for execution at some time in the future[6]. Before the scheduled simulation cycle starts it execution the request is processed. The signal which processes a request is said to be active. The value of the request becomes a so called driving value. If an access conflict occurs because there is more than one driving value for a signal a strategy can be defined how to resolve the conflict. A function can be defined which calculates the effective value to be stored in a signal from the driving values. The effective becomes the current value of the signal. If the value is different from the old one then an event[7] is said to have occurred on the signal.

This mechanism implies some limitations on the conflict resolution strategy. It is not possible to delay requests and it is not possible to refer to a previous stored value. In other words, any conflict solution based on the idea to give the simultaneous requests a timing relation does not work. A process can receive information from a signal by reading its current value. Different to a write access to a signal there is no delay in reading a signal value. The effective value is immediately accessible within the same simulation cycle. From the explanations above we can conclude that it is not possible to communicate between processes via signals within one simulation cycle.

6. In VHDL terms this means a set of drivers which are defined by a signal assignment statement queue transactions. The queue is the projected output waveform. It represents a driver.

7. The meaning of the term event in the VHDL context is slightly different from the one previously used in this thesis. Processing a request, i.e., to have an active signal corresponds better to the previously introduced term event.

A way to pass information between processes within one simulation cycle is to use shared variables. A value written to a shared variable can immediately be read by other processes. However, the language does not guarantee the synchronisation of such an access to a shared variable by multiple processes within one simulation cycle. The behaviour is non-deterministic and a model using such a concept is non portable[8].

To synchronize communication it is possible to interleave communication with process synchronisation. It is possible to specify conditions at a synchronisation point which must be true for a process to resume its execution in the next simulation cycle. The conditions can refer to current values of signals or some events on certain signals. That means a process can signal another process to continue its execution in a future simulation cycle by assigning some values to signals so that its conditions at the synchronisation points i.e., at the wait statements are fulfilled.

At that point we shortly discuss the modelling problems arising from such a synchronisation concept before we continue to present other language features of VHDL. The inter process communication based on signals could be classified as asynchronous. A sender is not blocked until a receiver reads the signal. It is not even guaranteed that the receiver reads the value sent by the sender. There is no automatic reply mechanism associated with reading a value from a signal. That means, modelling communication with signals requires explicit reply scheduling. As we know from the general considerations on reply scheduling Section 3.3.9 this causes problems with a proper encapsulation concept.

If we take a look at the synchronisation mechanism from a state-oriented view we can state that each synchronisation point corresponds to one or more synchronisation states depending on the conditions which are part of the wait statement. However, there is no abstraction of the states in a process's interface as there is no interface at all. In other words, to perform communication via signals it is necessary to break the encapsulation of a process and analyse its implementation. This makes processes with a non-standard synchronisation concept very hard to re-use. We can conclude from this short discussion on synchronisation in VHDL that it is very problematic to use VHDL processes as part of an object based modelling.

As stated at the beginning of this section an entity provides an interface which abstracts the implementation in an architecture. It is possible to define

8. The shared variable concept as it is described here is subject to future changes in the standardisation process of the language [91,176].

signals as part of the interface which can be used in an architecture as an abstraction of signals in other design units. This is useful when hierarchically nesting them as described above. In other words, it is possible to aggregate hardware models by encapsulating them in an architecture and by interconnecting them with signals. The ports of an entity serving as an interface are mapped to the signals. However, the interface mechanism lacks a concept to abstract the synchronisation aspects of the implementation. There is also no abstraction of the processes which an architecture contains. From what was discussed about the missing abstraction concepts for processes and the missing abstraction concepts for communication and synchronisation we can conclude that re-use at entity level is also a problematic issue.

So far, we have looked at the module concept based on entities and architectures. We now discuss the second module concept in VHDL which is based on packages. Packages are modules which only exist in a top level module, i.e., in a library. It is not possible to nest packages within other packages or design units but it is possible to reference packages in other design units. This makes the elements in a package accessible to other design units.

Basically, a package may contain some type declarations, constants, and subprograms[9]. A subprogram is abstracted by its interface in a package. The subprogram body is described in the package body which provides a strong encapsulation and abstraction. Furthermore, a package may contain signals and shared variables which can be used for process intercommunication. Extending a distribution boundary by referencing a package and then use a signal in the package for communication and synchronisation is a problematic issue from a modelling point of view. It annuls the abstraction concept behind an entity.

Simulation in simulation cycles as defined by the simulation semantics introduces a concept of concurrency and a corresponding timing model into the language. This timing model which consists of a sequence of simulation cycles executed after one another is extended by a quantitative simulation time. Each simulation cycle is assigned a value of the simulation time in ascending order[10]. The simulation time allows to exactly specify the simulation cycle in which a signal assignment is scheduled to become effective. In

9. In the context of VHDL we use the term subprogram rather than procedure because a procedure denotes a special kind of subprogram in VHDL.
10. It is possible for several simulation cycles to have the same simulation time. Within a sequence of cycles with the same simulation time the cycles are distinguished as the first delta cycle, the second delta cycle, and so on.

VHDL like in DACAPO III it is possible within one process to schedule several signal assignments to become effective at the same time and thus to introduce an additional concept of modelling concurrency.

As mentioned before, VHDL in many aspects is similar to Ada. One of the common features is an elaborated type concept and a strong typing. The type concept of VHDL supports data abstraction with enumeration types, arrays, and records. VHDL also allows the modelling of subtypes. As stated in Section 3.2.2 a subtype has a predicate which consists of the parent's predicate and additional constraints to describe the features of the subtype. An explicitly modelled resolution strategy to solve the access conflicts to signals is one of the features which can be added to a subtype in VHDL. This allows the modelling of user-defined value models as they are for example used for modelling at RT-level and below. Compared to other HDLs with predefined value models as part of the language, like for example DACAPO III and Verilog which are limited to gate level and switch level, the user-defined models of VHDL allow the modelling of different models each optimized for a certain level of abstraction.

VHDL with its subprograms is a procedural language. Subprograms are used to model behaviour. They can be invoked from within processes or from within other subprograms. We now consider two issues concerning abstraction aspects of subprograms. The first issue is overloading. Subprograms are said to be overloaded if they have the same subprogram designator, i.e., the same name. VHDL requires any ambiguities due to the same name to be resolved at analysis time. For this resolution VHDL uses information from the subprogram's interface like e.g., the type, the name, and the order of its parameters. It is possible to abstract similar behaviour by the same name but not by the same interface.

The second issue concerning subprograms is the abstraction of timing behaviour. Like in many procedural languages it is possible to pass constants and variables as parameters to a procedure[11] during invocation. As the procedures are invoked from within processes which execute their statements sequentially and which do not pass any information about variables across the process' boundaries it is sufficient to pass the results back to some variables after the procedure has executed all its statements. This is different for signals which can be within the distribution boundary of various processes. If a procedure contains a synchronisation point either directly or by calling another procedure containing a synchronisation point we can think of the

11. A procedure is a special kind of subprogram in VHDL.

procedure consuming simulation time. It consumes at least one delta cycle. It means that a signal may change its value during the execution of the procedure. The interface of subprograms takes this fact into account by using a different mechanism to pass signals as parameters to subprograms. The parameter itself is regarded as a signal which is associated to the actual signal in a subprogram call. Any access to the parameter abstracts an access to the actual signal. For example, an read access reads the current value of the actual signal. It does not read the probably outdated value of the signal from the time when the subprogram was invoked. Similarly an assignment to the parameter is scheduled as an request to store a value in the actual signal at a certain specified time which can be the next delta cycle at the earliest. The specified time is not necessarily the simulation cycle when the subprogram finishes its execution.

A consequence is that a procedure may cause effects in a model at a point of time when it does not exist any more. An abstraction of such a timing behaviour is not part of the procedure interface. The only indication that such strange effects may occur when calling a procedure is an interface that contains signals as parameters. To understand them it is necessary to break encapsulation of procedures and to analyse their implementation.

After the considerations on the abstraction of behaviour in VHDL we briefly mention a mechanism to abstract structure. VHDL provides generic parameters which can be used to pass static information across entity and architecture boundaries. It is possible to abstract in an architecture from structural information which can be represented by some values like for example array ranges. The application domain is especially at gate level and register transfer level. The mechanism allows to abstract information in cells or macro cells like bit widths or delay values. The entity and architecture is used as a template to which the actual values are passed as parameters in a concrete instantiation. It is a major concept to support re-use at these levels of abstraction. The general ideas behind these concepts already have been discussed in detail in Chapter 2. The limited usefulness of these ideas for re-use purposes at higher level of abstraction has been considered and reasons were given why it is not sufficient for a general re-use concept to use genericity.

To keep the mechanism simple and to make static analysis of a model possible the genericity concept of VHDL has some restrictions. The packages do not have a genericity concept[12]. The genericity concept is restricted to values. It is not possible to pass any type information or even behaviour in form of subprograms via a generic parameter to an entity and architecture.

One of the purposes of the language is to serve as a notation to document hardware systems. This especially includes hardware with long life cycles as it for example occurs in avionic or military systems. This makes it necessary for VHDL to allow a description to abstract from concrete implementations and implementation technologies and thus to support re-use of models across tools and target technologies. This requirement is met by minimizing implementation dependent features in the language semantics. An annex to the reference manual lists potentially non-portable language constructs. They concern interaction with the simulation environment like for example input and output commands and they concern the few non-deterministic features of the language like for example the use of shared variables for communication between processes within one delta cycle. Using modelling guidelines and coding styles which exclude the use of these features wherever possible makes a VHDL description compared to other descriptions highly portable and thus re-usable across multiple tools and environments.

The short introduction into the main features of VHDL concludes with some considerations on the language support for object-based modelling concepts. We discuss the limitations of the language with respect to object-oriented modelling and identify some of the features which would be required in the language in order to support system level modelling. Some basic ideas on that topic can be found in [126].

The basic principles of object-based modelling have been presented in Section 3.1. The central concept is the modelling of objects. An object is regarded as a module which encapsulates structure and behaviour. The properties of an object are abstracted in its interface. As stated above, in VHDL the design entity is the primary hardware abstraction. If we look at the entity as an abstraction of a hardware object an architecture is the corresponding object body. It contains signals and processes which are threads according to the terminology of the thesis. The object's interface in form of an entity does not contain an abstraction of the behaviour. Ports do not abstract operations which can be requested to be executed by messages. They rather abstract signals. The signals then can be referred to in synchronisation conditions of synchronisation points. Thus, ports as part of an entity may be viewed with reservations as an abstraction of synchronisation operations between processes. Considering the synchronisation operations as a special kind of interactions would allow to interpret a port a restricted version of an interaction

12. This would be only of interest if it were allowed to instantiate packages in other design units.

point. An entity is reduced to an abstraction of the interaction between activities from outside the architecture from activities inside the architecture. From that point of view an entity architecture pair is a concurrent process. Nevertheless, an entity encapsulates the object's properties but it does not really abstract them.

The structure of an object is modelled by signals inside the entity and by variables inside the processes i.e. they are the attributes of the object. From a modelling point of view the attributes of different objects and in the case of the processes even the attributes in different processes do not share the same memory or address space. This is in accordance with the fact mentioned above that the entity is a static distribution boundary. We can regard entity architecture pairs as distributed objects.

The problems of dynamically changing references in distributed systems which has been discussed in Section 3.3.18 is solved in VHDL by not supporting any reference mechanism between entities. In other words, there is no language feature which would allow the modelling of instance variables.

A message passing concept which has been identified as an object-based principle in Section 3.1 is not part of the language. Messages have to be modelled from signals using the synchronisation operations in processes. We could say message passing between entities in an object-based sense requires a reflective modelling style.

In the considerations about object-based principles in VHDL we considered entities and architectures as objects. However, there are some more language features worth to be considered from an object-based point of view. If we look at the package mechanism we can state that it is mainly a module concept to improve procedural modelling. It provides a strong concept to abstract and encapsulate behaviour. Additionally, it allows to model structure as part of a package. A package may contain type declarations to describe a set of entities[13] that share the same features, for example some subprograms declared in the package. The weak point with the types in a package is the missing encapsulation concept. The same is for global signals which can be used to model a state of a package. There is no encapsulation mechanism which prevents another design unit from directly accessing the state of the package. Another problem occurs if we regard packages as classes or objects. There is no possibility to declare or instantiate such objects in a design unit.

13. Entity is not to confuse here with a VHDL design entity. The VHDL language reference manual denotes such entities as objects.

From these considerations on object-based principles we can draw some conclusions what language features and concepts could be useful to make VHDL object-based and object-oriented and define some requirements on how to extend the language for system level modelling [22,23]. The first idea is to improve the object-based capabilities of what we identified as objects. For entities this could mean to introduce the missing abstraction mechanism for behaviour. This has to come with an improved message passing mechanism. An approach could be to extend the port mechanism which only abstracts structure. As entities are executed concurrently the additional message passing has to be combined with some underlying synchronisation concepts. Independently from whether the concept is synchronous or asynchronous, mechanisms have to be defined to avoid access conflicts in objects. Such mechanisms could be based on mutual exclusion synchronisation for example. What is also required is a concept to specify conditional synchronisation in an object. The art is in integrating such concepts into the existing synchronisation and communication concepts of VHDL. A goal of such an extension could be to avoid the introduction of any non-deterministic features into the language to be conform with the existing language philosophy.

Identifying entities as distributed objects raises the question if a reference mechanism should be provided which allows to pass references to objects as parameters to operations. Such a mechanism would change the static distribution boundaries into dynamic ones. The consequences and problems of dynamic distribution boundaries for distributed objects have been discussed in Section 3.3. As an alternative to a reference mechanism a copy model has been mentioned. However, entities are active objects and it would be a complex task to define semantics for copying them. Related issues, like for example the problem how to pass polymorphic objects as parameters with such semantics have been discussed in Section 3.3.18.

A major feature for object-oriented modelling that is missing in VHDL is inheritance. Inheritance means to allow incremental modification of the structure and behaviour of a class or an object. Therefore adding an inheritance concept to entities depends on how such an extension to VHDL models and abstracts behaviour. In any case it is a problematic issue to combine it with the existing possibility to model behaviour as one or more processes in an architecture. Any modification or interaction with that behaviour requires a derived class to break the encapsulation of the parent's processes due to the missing abstraction of processes. The same is for structure which is modelled within such processes. If there is an inheritance mechanism on entities the question arises how could it support polymorphism. VHDL provides a mechanism for a kind of static polymorphism with its configuration mechanism which allows the selection of an architecture for an entity out of a set of architectures. This configuration mechanism needs an integration in the inheritance concepts which is problematical because the mechanism breaks encapsulation by explicitly referring to internal structure of an architecture. The second question is how to provide dynamic polymorphism. As mentioned above this issue is very much related to the question how to model distributed objects.

The considerations about extending the language so far are based on the idea to identify an entity as an object. A language extension also could be based on the idea that the package concept should be improved to support object-oriented principles. As a main weakness of packages from an object-based point of view the missing concept to encapsulate structure has been mentioned. This could be overcome by a concept in which structure can be implemented in a package body and the subprograms specified in the package have access to that structure. To support incremental modifications a mechanism could be introduced which allows to extend existing packages by new child packages. Child packages are packages which have access to the parents' implementation. What also would be required for such an extension is a mechanism how to instantiate packages in other objects and how to reference them. Extending the selected name mechanism and introducing some dynamic features to it could be an approach how to reference such packages. Due to missing activities as part of such packages they could be considered as passive objects according to Section 3.1.3 and thus support an orthogonal approach for modelling concurrency.

An alternative approach may use the type declaration which can be part of a package to describe a set of objects. Subprograms in the package could be used to model the behaviour of such objects. The type describes a class and the subprograms model the operations of that class. Message passing is reduced to a simple operation call in a sequential context, i.e., in a process. In such a sequential context a reference concept already provided by VHDL with its access types could be applied to model dynamic distribution boundaries of objects. The access types could be used as instance variables in a class. At the same time a copy semantics in the sense mentioned in the considerations about distributed objects would be possible. The semantics could be extended from the semantics of assignment statements as they are already part of VHDL. With signal assignments it would be possible to pass the objects across process borders and thus to use them in a concurrent model.

Also required for modelling concurrency are synchronisation concepts more abstract than the resolution mechanism of VHDL. Concepts for modelling mutual exclusion synchronisation and condition synchronisation appear to be appropriate for this kind of synchronisation. Inheritance would be added to such an extension which is based on modelling classes as types in packages by defining an inheritance concept for types and some extension mechanisms to packages. Such a mechanism could be combined with a child package concept. Polymorphism in such an extension would mean that various implementations of the same subprogram exist for different derived classes.

There are some more features which could be added to the existing language and which might be useful for modelling of re-useable systems at a high level of abstraction. One of these features is genericity. The existing concept which supports generic parameters as part of an entity is limited to passing some constants to an entity. It could be extended to support the passing of more complex information like types or behaviour to all kind of design units.

When extending VHDL to add object-oriented features the art is in selecting a small subset of the presented features which fits into the existing language philosophy and which converses all the important aspects mentioned like abstraction, encapsulation, inheritance, and synchronisation of concurrency.

5.2 Object-Oriented Extensions to VHDL

In the last section we carefully considered the basic possibilities to add object-oriented features to VHDL from a theoretic point of view. Since the beginning of VHDL quite a few more or less concrete proposals have been made to extend VHDL in an object-oriented way. Finally, they all can be put down to some of the theoretic considerations made above, however, they differ to the extent they support them. The proposals also differ very much in their notation of the extensions which is adapted from various object-oriented programming languages.

This section presents an overview about the proposals. Some other meanwhile rather incomplete surveys are given in [51,152]. Comparisons and classifications between alternative principles for extending VHDL are presented in [5,12,53,117,118].

5.2.1 VHDL++

The first proposal we are going to present is called VHDL++. As its name suggests it is based on C++ constructs which are added to VHDL. The extension exists in various variants [70,72,74,76]. What we discuss here is a conglomerate of the various dialects.

The main extension of VHDL++ is the presentation of a class concept which introduces a new encapsulation mechanism in the language. As in C++ the class consists of a private section and a public section to determine the visibility and accessibility of class-attributes. In other words, it is possible to circumvent encapsulation and to make attributes directly accessible from outside the object. Similar to C++ the operations or in a C++-terminology the methods are defined as procedures within the class. The procedures implicitly have access to the attributes. Different to C++ instance variables are not supported as attributes. VHDL++ does not support a reference mechanism which would allow to calculate references to objects as results of method calls. It is not possible to dynamically change the distribution boundaries of objects. The selected name mechanism is only extended by a mechanism to statically reference the methods of an object. Its notation corresponds to the notation which is used to denote a subprogram in a package. The notational similarities are no accident because the basic idea behind the concept is a variant of the approach to use type declaration which are part of a package to describe a set of objects. The subprograms in the package model the methods of the objects.

A class of that package based approach is called general class in VHDL++. It can be used to declare individual objects by an object instantiation. From the basic concepts behind the extensions it follows that an instantiation only can occur at places in a model where it also would be possible to declare a signal. A model using such objects of general classes is claimed to be principally appropriate for synthesis.

A short example is given to illustrate the modelling with VHDL++. It models a bounded buffer with a get and a put method[14]. Put stores an item in the buffer and get removes[15] the oldest item from the buffer. A bounded

14. The example originally comes from concurrent object-oriented programming where it is often used to show certain aspects of different object-oriented programming languages. To simplify the comparison between them and the proposals mentioned in this thesis we take the class and operation names of the example from [111]

15. Remove means to read and remove items.

buffer appears to be appropriate for the purpose because it models a typical producer-consumer situation which often occurs in client server relationships between objects. It requires the modelling of synchronisation constraints. A bounded buffer cannot execute a get method if the buffer is empty. Likewise it cannot execute a put method if the buffer is full. Another interesting aspect of the model is that it also requires synchronisation between multiple clients and the server for example in form of mutual exclusion synchronisation.

```
class bbuffer is
private
    buf : array (1 to buffersize) of item_type;
    buf_in, buf_out : integer;
public
    procedure put (item : item_type) is
        constant : max_number_in_buffer : positive := buf'last(1);
    begin
        ...
        buf ((buf_in mod max_number_in_buffer) + 1)) := item;
        buf_in := (buf_in + 1) mod (2 * max_number_in_buffer);
    end;
    procedure get (item : item_type) is
        constant : max_number_in_buffer : positive := buf'last(1);
    begin
    ...
        item := buf((buf_out mod max_number_in_buffer) + 1);
        buf_out := (buf_out + 1) mod (2 * max_number_in_buffer);
    end;
end class;
```

The bounded buffer is implemented by an array which is hidden to the clients in the private part of the class definition. This private part also contains two pointers referencing to the point in the array where to store and where to read the next item[16]. The implementation does not explicitly model the synchronisation information if the buffer is empty or full. This information is implicitly encoded in the values of the pointers[17].

The public procedures model the methods to access the bounded buffer. There are two interesting aspects to mention about the procedures. The class

16. Actually, there is an offset of 1 between the array indices and the pointers due to the modulo calculations.

17. This is the reason why the pointers to the array range between 0 and (2 * max_number_in_buffer) - 1.

definition does not provide an interface. As there is no interface there is no abstraction of the protocol i.e., the synchronisation information. It is not only an open issue how to abstract synchronisation information it is also not clear how to implement synchronisation constraints. Therefore in the listing of the implementation the corresponding parts are omitted and marked by some dots. In other words, there is no language concept for modelling and abstracting synchronisation between server and clients. The second aspect is that there is no language support for modelling synchronisation between multiple clients. For example, it is not defined how to specify methods which are mutually exclusive as it would be required here.

We conclude the considerations about the implementation with remarks on some language details. It is noticeable that the parameters of the methods do not have a direction nor a VHLD class like constant, signal, or variable. It is also remarkable that it is possible to implicitly define an array type[18] in the private part of the class.

A general class as it is described above can be used as a starting point for an inheritance mechanism. Like in C++ it is possible in VHDL++ to inherit attributes and methods and if necessary to override methods. For example, it would be possible to define a class x_buf which inherits the methods put and get. The specification of that buffer may require that it has an additional method last which removes the last item which was put into the buffer. The problem is that a method last in a derived class must not access the private part of its parent. That means it would have been necessary in the definition of bbuffer to define the array buf public[19]. Unfortunately this would break encapsulation. Nevertheless, assuming the attribute is visible in a derived class it would be possible to define x_buf.

```
class x_buf : bbuffer is
public
    procedure last (item : item_type) is
        ...
    end;
end class;
```

18. In the example, we changed the original squared brackets of the implicit array type to round brackets to make it consistent with VHDL and to allow the application of a translation mechanism from VHDL++ to VHDL which is part of the VHDL++ approach.

19. A mode protected as it is available in C++ to make attributes visible only to derived classes is not part of VHDL++.

The class definition defines that x_buf inherits all public methods and attributes of the class bbuffer.

VHDL++ not only has the package based class approach but also an entity based one. The language allows to add port information to a class. The ports become the interface between an object of that class and the rest of the system. In that case it is necessary to define protocols[20] to perform the methods of the class instead of procedures. A protocol in VHDL++ is a sequence of stimuli at certain ports of a class. A class with protocols and ports is called *component class*. Such a class may serve as a wrapper for a conventional entity or architecture. For example, a component class could be used to describe an implementation s_buf of the bounded buffer. It implements the bounded buffer as a shift register. s_buf adds port information to the class and inherits the public methods put and get of bbuffer but it does not inherit the implementation of the methods. Instead, protocols are used to defined how to perform the access to the ports.

```
class s_buf : bbuffer is
port
    clock : in bit;
    sreg_in : in byte;
    sreg_out : inout byte;
    shift : in bit; -- shift on = '1' , shift off = '0'
    nr_in_buffer : inout : integer; -- number of items in buffer
protocol put (item : item_type) is
    begin
        sreg_in <= to_byte(item);
        shift <= '1';
        nr_in_buffer <= nr_in_buffer + 1;
        cycle (clock); -- wait one clock cycle
    end;
protocol get (item : item_type) is
    begin
        -- shift first item to the end of the register:
        shift <= '1';
        for i in max_number_in_buffer downto nr_in_buffer + 1 loop
            cycle(clock) -- wait one clock cycle
        end loop;
        -- remove item:
        shift <= '0';
        item := to_item (sreg_out);
```

20. The term protocol is used here as a language construct of VHDL++

```
            cycle(clock);
            shift <= '1';
            nr_in_buffer <= nr_in_buffer -1;
            cycle(clock);
            -- shift remaining items into original position:
            for i in nr_in_buffer downto 1 loop
                sreg_in <= sreg_out;
                cycle(clock);
            end loop;
            shift <= '0';
        end;
    end class;
```

In the component class there is not any longer a separation between public and private part. The protocol models the interface mechanism and is publicly visible. The actual implementation of the component which is abstracted by the component class is encapsulated and only accessible from the protocols via the ports. The example shows that for modelling the protocol it is necessary in VHDL++ to break the encapsulation. It is necessary for the protocol operation to know the clock cycle when to read or write the item via the port. It is noticeable that it is not possible for the protocol to have its own state. Modelling of a state is only possible as a port of the component class. Making the attribute nr_in_buffer and the instruction shift a port allows the protocol to define the timing behaviour i.e., the protocol defines the put operation to execute within one clock cycle and the get operation within max_number_in_buffer + 1 clock cycles. In fact, the protocol is a transformation from the protocol used to communicate via the ports to the protocol which consists of calling a method and waiting for a return value. Although object-orientation originally would require the encapsulation of the shift operations and the attribute nr_in_buffer such an encapsulation would impose a different protocol via the ports with restrictions on possible protocol transformations in the VHDL++ protocols. What happens if a precondition of a method is violated is still an unsolved issue. The example illustrates the difficulties of the idea to separate the protocol from the behaviour of a class.

The VHDL++ approach comes with a translation mechanism of the object-oriented constructs to VHDL. The main idea is the translation of the general class definition into a record definition. The methods or member functions are translated to functions or procedures with an additional argument of the record type. Access to the attributes are changed into access to the additional argument. Object instantiations are translated to data declarations for example global signal declarations. As an example the translation of the bounded buffer description is given. The class is translated into a record type and the methods into normal procedures.

```
type bbuffer_record is record
    buf : implicit_array_of_item_type(1 to buffersize);
    buf_in, buf_out : integer;
end record;
procedure bbuffer_put ( signal bbuffer_data : inout bbuffer_record;
                        constant item : in item_type);
procedure bbuffer_get ( signal bbuffer_data : inout bbuffer_record;
                        variable item : out item_type);
```

The example shows some differences to the original transformation proposals and examples in [74] in order to avoid illegal VHDL as a transformation result. The main differences concern the required separation of subprogram specification and body and the transformations of the subprograms’ parameters into constants, variables, or signals. Especially, making the object a signal parameter requires the modification of all assignment statements in the subprograms.

```
procedure bbuffer_put ( signal bbuffer_data : inout bbuffer_record;
                        constant item : in item_type) is
constant : max_number_in_buffer : positive := bbuffer_data.buf'last(1);
begin
    ...
    bbuffer_data.buf ((buf_in mod max_number_in_buffer) + 1)) <= item;
    bbuffer_data.buf_in <=
            (bbuffer_data.buf_in + 1) mod (2 * max_number_in_buffer);
end;
```

The transformation changes the semantics of the original method put severely because of the signal assignments. An execution of the operation does not change the object’s state it rather schedules some transactions to occur not before the next delta cycle if at all.

Another problem related to the use of global signals in that approach is that it only works properly if the objects are not used from more than one process. The transformation or translation schema implies that the approach allows static overloading of methods but it does not support polymorphism.

Different to the general class there is no detailed description how to translate a component class. It is proposed to translate the class into a component instantiation. The ports of the component are the ports of the component class. The protocols are converted into procedures. The procedures use a signal connected to the ports as an additional parameter. As the signals semantics is already considered in the protocol the difficulties discussed in the context of global class transformation do not occur.

VHDL++ restricts with its object-oriented language extensions the possibility to model concurrency as described above. At the same time it introduces new concepts to model concurrency into the language. A so called concurrent block allows to model a concurrent data flow within a sequential process or subprogram. Such a block specifies a set of statements which occur concurrently. The proposal is similar to an idea in [122] to add an explicit join and fork statement to VHDL which adds concurrency to a sequence of statements. These ideas are very similar to the ideas presented in the surveys about system level specification languages like CSP or LOTOS. As it has been mentioned there, such mechanisms are especially appropriate to model protocols. The static concurrency concept of VHDL would become dynamic. However, in the context of the VHDL timing model and synchronisation concept a bunch of unsolved questions arise. Access conflicts to variables may occur. Concurrent signal assignments from within one process would require an additional resolution mechanism to solve access conflicts or additional drivers. A mechanism to dynamically create and delete drivers would be necessary including the question what would it mean to delete a driver with a set of future transactions. The proposal to introduce the new concurrency concept does not answer the questions and it does not describe how to translate these VHDL++ features into VHDL.

The VHDL++ proposal illustrates principal problems of extending VHDL in an object-oriented manner. It shows the difficulties in integrating the extensions into the concurrency concept of the language. The difficulties concern the new concurrency constructs as well as synchronisation mechanisms. The lack of concepts to model mutual exclusion synchronisation and condition synchronisation has been identified as one of the major deficiencies of the language extension. Although the proposal tries to adapt some powerful concepts from other languages, all in all, the limitations on encapsulation, abstraction, and polymorphism prevent the language extension to support real object-oriented modelling.

5.2.2 VHDL_OBJ

In the previous section we noted that a missing support for polymorphism is a severe limitation for object-oriented modelling. In this section we survey an extension to VHDL which overcomes the restriction not to have polymorphism. The name of the extension is VHDL_OBJ [177].

Like in the previous approach a class construct is added to VHDL. Each class has a declaration which describes the interface of the class and its structure. The methods of the interface are marked as public procedures. The implementation of the methods is described in the body of the class. In other words, the concept separates the specification of a class from its implementation. In a declaration of a class it is possible to inherit features from an existing class. To inherit methods from a parent's class it is necessary to explicitly list them in the class declaration. It is not mandatory to inherit all methods from a parent's class. That means, a derived class does not necessarily support the parent's protocol and an is-a relation does not imply a subtype relation (Compare Section 3.2.12). It is possible to override the implementation of inherited methods. Such methods have to be explicitly marked in the parent's class as virtual. It is decided during the simulation of the model which of several different methods will be executed depending on the class of an object. The possible re-implementation of methods is a means of the language extension to support polymorphism. Although not explicitly mentioned in the proposal for the language extension it can be concluded from a translation mechanism that a class is a new design unit in VHDL_OBJ. Due to the design unit's similarity to packages we could classify the approach as based on packages.

Objects only can be instantiated in a process or subprogram. The proposal does not mention if it is possible to abstract an object in a parameter list of a procedure. This makes it impossible to say something about the distribution boundaries of objects. But it is possible to conclude from the abstraction boundaries of processes that it is not possible to pass any objects between processes. In principle, the object-oriented concepts are only applicable in the sequential context of a process. Consequently, no synchronisation mechanisms between objects are provided. We illustrate the considerations about VHDL_OBJ with the example of the bounded buffer:

```
type bounded_buffer_array is array (positive range<>) of item_type;
class bbuffer is
    variable buf : bounded_buffer_array(1 to buffersize);
    variable buf_in, buf_out : integer;
```

```
        virtual public procedure get (variable item : out item_type);
        virtual public procedure put (item : in item_type);
    end bbuffer;
```

The implementation of the bounded buffer contains the bodies of the methods:

```
    class body bbuffer is
    begin
        virtual public procedure get (variable item : out item_type) is
            constant : max_number_in_buffer : positive := buf'last(1);
        begin
            -- Check precondition of get:
            if buf_in = buf_out then
                … -- Error: Buffer empty, violation of precondition
            else
                item := buf((buf_out mod max_number_in_buffer) + 1);
                buf_out := (buf_out + 1) mod (2 * max_number_in_buffer);
            end if;
        end;
        virtual public procedure put (item : in item_type) is

            …
    end bbuffer;
```

The implementation of the method get does not contain any synchronisation code it rather has a check if a caller violates its preconditions. If the check fails an error has occurred. This follows from the fact that the methods of bbuffer are only called from within a sequential environment. To avoid such errors it would be useful for a caller to be able to check the state of the buffer. This could be achieved by adding a method empty to a derived class x_buf2 of bbuffer which checks if the buffer is empty.

```
    class x_buf2 has
        inherit get , put from bbuffer
    is
        virtual public procedure empty (variable return_val : out boolean);
        virtual public procedure get2 (variable item : out two_items_type);
    end x_buf2;
```

The class x_buf2 inherits the structure of bbuffer. It also explicitly inherits the methods put and get. If required it would be possible in the class body to override the implementation of put and get. The class furthermore adds two new methods empty and get2 to the class. The method get2 simply takes two items from the buffer. The interesting thing about get2 is that it introduces a new precondition, the buffer must contain at least two items to invoke get2. We shall refer to the example of x_buf2 later on.

An object declaration is used to instantiate an object. For example, it is possible to instantiate an object which is a bounded buffer in a process and call its methods from within the process statement part:

```
example : process
    object bbuffer_object : bbuffer;
    variable item : item_type := default_item;
    …
begin
    bbuffer_object~put (item);
    …
end process example;
```

The declaration creates an object. The designator bbuffer_object is a reference to it. The designator of the class bbuffer may also reference to an object of a derived class like x_buf2. This allows the modelling of polymorphic objects. However, there is a bunch of open issues with that concept. For example, as there is no subtype relation between derived classes protocol errors may occur, but there is no concept to handle such errors. What is the exact semantics of an assignment statement between objects, in particular, what happens to spare objects, is there a garbage collection? What happens with references to objects in dynamic scopes which may be removed while the reference still exists.

The proposal not only provides concepts how to extend VHDL it also sketches some ideas how to translate the extensions into VHDL. The translation is based on transforming the class definition into a record definition. The methods become procedures with an additional argument of an access type. That access type is a pointer to an object's virtual table[21]. The table contains a pointer to the actual object. A direct pointer to the object as additional argument of a procedure is not possible because of the strong type concept of VHDL. Objects are implemented by variables which are created by allocators. This use of access types and allocators to implement objects imposes the limitations of using objects only in processes or procedures without the possibility to pass them across process boundaries. We illustrate the transla-

21. We only discuss the translation of polymorphic objects, the non-polymorphic objects use a similar, but, simplified translation mechanism.

tion mechanism by excerpts from the example of the bounded buffer and its derived class x_buf2:

```
package bbuffer is
    type bounded_buffer_array is array (positive range<>) of item_type;
    type bbuffer_type is
        record
        buf : bounded_buffer_array(1 to buffersize);
        buf_in : integer;
        buf_out : integer;
    end record;
    type pointer_bbuffer_type is access bbuffer_type;
    -- derived type
    type x_buf2_type is
    record -- inherits structure from bbuffer:
        buf : bounded_buffer_array(1 to buffersize);
        buf_in : integer;
        buf_out : integer;
    end record;
    type pointer_x_buf2_type is access x_buf2_type;
    -- virtual table which model access to actual object:
    type bbuffer_virtual_class_type is
                (bbuffer_virtual_class, x_buf2_virtual_class);
    type p_virtual_table_bbuffer_type is
    record
        bbuffer_virtual_class : pointer_bbuffer_type;
        x_buf2_virtual_class : pointer_x_buf2_type;
        is_a : bbuffer_virtual_class_type;
    end record;
    procedure put_base
            (variable p_virtual_table: p_virtual_table_bbuffer_type;
             constant item : in item_type);
    procedure put_bbuffer
            (variable oops_p_obj: pointer_bbuffer_type;
             constant item : in item_type);
    procedure get_base
            (variable p_virtual_table: p_virtual_table_bbuffer_type;
             variable item : out item_type);
    procedure get_bbuffer
            (variable oops_p_obj: pointer_bbuffer_type;
             variable item : out item_type);
    procedure put_x_buf2
            (variable oops_p_obj: pointer_x_buf2_type;
                constant item : in item_type);
        procedure get_x_buf2
                (variable oops_p_obj: pointer_x_buf2_type;
                 variable item : out item_type);
        procedure empty_base
                (variable p_virtual_table: p_virtual_table_bbuffer_type;
                 variable return_val : out boolean);
        procedure empty_xbuf2
                (variable oops_p_obj: pointer_x_buf2_type;
                 variable return_val : out boolean);
        procedure get2_base
                (variable p_virtual_table: p_virtual_table_bbuffer_type;
                 variable item : out two_items_type);
        procedure get2_x_buf2
                (variable oops_p_obj: pointer_x_buf2_type;
                 variable item : out two_items_type);
    end;
```

The approach translates each virtual procedure into a procedure which runs the dispatching according to the actual class of the object whose method is invoked. The class information is stored in the table in the field is_a. Thus, the package body implements the dispatching as follows:

```
    package body bbuffer is
        -- procedure which runs the dispatching of the virtual method put:
        procedure put_base
                (variable p_virtual_table: p_virtual_table_bbuffer_type;
                 constant item : in item_type) is
        begin
            case p_virtual_table.is_a is
                when bbuffer_virtual_class =>
                    put_bbuffer
                        (oops_p_obj => p_virtual_table.bbuffer_virtual_class,
                         item => item);
                when x_buf2_virtual_class  =>
                    put_x_buf2
                        (oops_p_obj => p_virtual_table.x_buf2_virtual_class,
                         item => item);
            end case;
        end;
        …
    end;
```

The translation into a package shows that although a two level pointer mechanism is used it is necessary to re-declare inherited virtual methods like put or get in a derived class in order to make them manageable in a dispatching mechanism. It is not possible to re-use the original inherited methods by a delegation mechanism in the translation which would delegate a call of the inherited method to its original implementation.

As a side effect the translation introduces an implicit dynamic binding of operations if they are called from within another operation of the same class and there is no way to suppress such a binding and to make it static. Together with the feature of VHDL_OBJ which allows to remove methods from a protocol in a derived class this is a dangerous property of the language. For example, x_buf2 could implement the get2 method by calling the method get twice. If a new class of x_buf2 is derived which omits the get operation in its protocol the get2 operation which could be still in the protocol would fail. There is no way to detect such situations without re-analysing all implementations of the parents i.e, without breaking encapsulation of the parents.

As it can be seen from the example even if concurrency is omitted the translation leads to an enormous complexity and the result can not be used as a basis for further synthesis steps because of the dynamic constructs. Furthermore, the example illustrates some critical aspects of approaches which provide translation mechanisms from the language extension to VHDL.

5.2.3 OO-VHDL

Another approach of extending the language which also uses an translation mechanism to translate the language extensions into VHDL is presented in [45,162,167,168]. Unfortunately, only the language extension which is called OO-VHDL is discussed in literature and not the translation mechanisms behind it.

OO-VHDL bases on the idea to consider an entity as an abstraction of a hardware object. Due to the missing capabilities of abstracting behaviour in an entity an extension of an entity is introduced in the language extension. The new language construct is called EntityObject and is the core of the language extension. Like an entity an EntityObject may have ports and generics and each EntityObject can be related to one or more architectures. Additionally, operations which model the behaviour of an EntityObject can be declared in an EntityObject to abstract its behaviour. In the specification they are similar to procedures. The body of an operation is implemented in an architecture of the EntityObject. We can illustrate this by the example of the bounded buffer.

```
type bounded_buffer_array is array ( positive range <>) of item_type;
EntityObject bbuffer is
    -- Entity part:
    generic ( buffersize : positive);
    -- Object part:
    operation put (item: item_type);
    operation get (item: out item_type);
end bbuffer;
architecture behaviour of bbuffer is
    instance variable buf : bounded_buffer_array ( 1 to buffersize);
    instance variable buf_in : integer;
    instance variable buf_out : integer;
    operation put (item : item_type) is
        constant : max_number_in_buffer : positive := buffersize;
    begin
        ...
        buf ((buf_in mod max_number_in_buffer) + 1)) := item;
        buf_in := (buf_in + 1) mod (2 * max_number_in_buffer);
    end;
    operation get (item : out item_type) is
        constant : max_number_in_buffer : positive := buffersize;
    begin
    ...
        item := buf((buf_out mod max_number_in_buffer) + 1);
        buf_out := (buf_out + 1) mod (2 * max_number_in_buffer);
    end;
begin
    -- null architecture body
end behaviour;
```

The bounded buffer modelled as an EntityObject abstracts its operations and its size. The size is abstracted by a generic parameter. The EntityObject encapsulates its activity in a corresponding architecture body. It is possible like in the example to have no activity modelled in the body. That means that EntityObjects may be used to abstract both, active and passive objects.

As mentioned above, the size of the buffer is abstracted. The actual buffersize is defined in an object instantiation which is similar to a component instantiation in VHDL.

```
architecture environment of environment_entity is
    ObjectComponent bbuffer
        generic (buffersize : positive)
    end ObjectComponent;
    constant actual_buffersize : positive := 42;
begin
    object bbuffer_object : bbuffer
        generic map (buffersize => actual_buffersize);
    process
        variable object_reference : EO_Handle := bbuffer_object;
        constant actual_item : item_type := … ;
    begin
        send object_reference put (actual_item);
        …
    end;
    …
end environment;
```

In an object instantiation an object gets a name. This name can be used to reference the object when sending messages to the object. Although the objects per se are static it is possible to store references to them in variables of the predefined type EO_Handle and to pass the references as parameters to operations. This changes the static distribution boundaries between entities into dynamic ones. It would be very interesting to see how the problem of implementing dynamically changing references in distributed systems which has been discussed in Section 3.3.18 is solved in OO-VHDL. Unfortunately, there is no information available on that issue. The references can be used in a send statement to request an object to execute an operation like in the example which request the bbuffer_object to execute a put operation.

In case of concurrent requests the object provides a mutual exclusion synchronisation between operations. However, the synchronisation between an activity in an active object and an operation is not explicitly and clearly defined. The synchronisation queues the request and blocks[22] the sender until the operation is executed. The problem with this synchronisation concept is that there is no possibility to define condition synchronisation. This is the reason why any synchronisation code is omitted in the example. If the buffer is empty and a get operation is requested for execution there is no way to

22. There is a possibility to avoid the blocking in certain circumstances, however, we do not consider this feature here in more detail.

queue the request until the buffer contains at least one item. What is also omitted in the queuing mechanism of OO_VHDL is a re-queuing concept.

The example can be used to illustrate a general problem of extending VHDL by any kind of language defined mutual exclusion synchronisation. Say, for example, the implementation of the put operation contains a synchronisation point in form of a wait statement, i.e., the execution of the operation consumes simulation time. As the sender of a message is blocked until the operation is executed the send statement can be interpreted as a synchronisation statement. From the perspective of the sender of the put message the wait statement appears to be a synchronisation point within another synchronisation point. Probably the semantics is that both synchronisation conditions have to be met to allow the calling process to proceed its execution. The simulation time advances in the calling process according to the time consumed in the called ObjectEntity. The more interesting question is what happens to other callers whose requests are queued during the execution of the put operations? How are they synchronized? Does the synchronisation point in the put operation force a synchronisation of callers of other operations even if the other operations potentially do not contain a synchronisation point at all? The synchronisation concept as a whole suddenly depends on the queuing policy of the message queues of EntityObjects.

Another synchronisation mechanism of the language extension is based on a a rendez-vous concept. The difficulties with such a concept in the context of object-oriented modelling have been discussed in previous chapters on other design methodologies and therefore are not considered here any more.

In order to support re-use OO-VHDL provides inheritance mechanisms for entities and EntityObjects. A new entity can be derived from an existing one. It is possible to modify the new entity by adding new ports and generics to it.

A derived entity inherits the architectures of its parents. The language extensions also allow the extension and modification of existing architectures. Such a modified architecture belongs to the same entity as the original architecture. At the same time it belongs to all derived entities according to the inheritance mechanism. It is not only possible to add new declarations and statements to a derived architecture, OO-VHDL also allows to override existing concurrent statements which model the behaviour of an entity. However, such modifications ignore basic object-oriented principles because there is no abstraction of the behaviour and thus any modifications require an analysis of the implementation of the behaviour which breaks encapsulation.

EntityObjects provide the same inheritance mechanisms like entities. That means, it is possible to extend and modify activities of entities by breaking encapsulation principles. Additionally it is possible to extend and modify behaviour modelled by operations. OO-VHDL allows to override operations in an EntityObject. The example of the extension to the bounded buffer looks as follows:

```
EntityObject x_buf2 is new bbuffer
    operation empty ( return_val : out boolean);
    operation get2 ( item : out two_items_type);
end EntityObject x_buf2;
```

As the derived EntityObject modifies the interface the derived architecture which provides the implementation must belong to the derived EntityObject.

```
architecture extended_behaviour of x_buf2 is new behaviour
    operation empty ( return_val : out boolean) is …
    operation get2 ( item : out two_items_type is …
begin
end extended_behaviour;
```

The inheritance mechanism combined with the un-typed reference mechanism allows to model polymorphism. A variable of type EO_Handle can be used to reference various polymorphic EntityObjects and to send them the same message.

The OO-VHDL approach is primarily made for rapid prototyping using a fixed message passing mechanism and therefore there is no close link to further synthesis steps. The problem is similar to translation problems of the SDL high level communication mechanisms.

OO-VHDL can be seen as a typical entity based approach to extend VHDL. Exemplary, the extension and modification of activities in an architecture has been identified as a difficulty in such approaches. Introducing mutual exclusion synchronisation concepts into the language has also to be identified as a principle problem as far as simulation time is concerned. Again, we can note that porting successful modelling concepts from other languages arises difficulties due to the existing timing and synchronisation mechanisms of VHDL.

5.2.4 Protected objects

Shared variables have been introduced into the language to support object-oriented modelling in VHDL [24]. However, making shared variables part of

VHDL causes conceptual problems. As already mentioned above, one of the main problems is that the language does not guarantee the synchronisation of such an access to a shared variable by multiple processes within one simulation cycle. To solve the problem proposals[23] have been made to introduce a mutual exclusion synchronisation for accessing shared variables as part of a revised language [91]. The proposal is based on a monitor concept. Such a monitor is called protected object and its type protected type. A protected type has an interface and an implementation. The interface provides operation declarations which can be used to access data which is encapsulated in the implementation. Such an operation operates atomically on an associated protected object and thus provides mutual exclusion synchronisation. It is called protected subprogram. Different to for example OO-VHDL there is no concurrent activity associated with such an object. In the terminology of the thesis protected objects are passive objects. The example of the bounded buffer could be modelled according to the latest[24] published version of the proposal as follows:

```
type bbuffer is protected
    procedure put (item: item_type);
    procedure get (item: out item_type);
end protected bbuffer;
```

The corresponding implementation encapsulates the attributes of the object.

```
type bbuffer is protected body
    variable buf : bounded_buffer_array ( 1 to buffersize);
    variable buf_in : integer;
    variable buf_out : integer;
    procedure put (item : item_type) is ...
    procedure get (item : out item_type) is ...
end protected body buffer;
```

As it can be seen from the example there is no concept to model condition synchronisation. An earlier proposal [90] mentions a guard mechanism as an optional extension of the language which would allow to specify synchronisation conditions. The resulting guarded operation is called entry. If a request to execute an entry fails because the guard condition is not met the request is queued in an entry queue[25]. The proposal also mentions a re-queue facility of

23. The proposals are the result of a DASC shared variables working group.
24. We are referring to version 5.7 here.
25. In the terminology of the thesis an entry queue is a message queue.

the entry queue. As discussed in Section 3.3.8 such a mechanism provides the concepts for condition synchronisation. In combination with the timing model and synchronisation concept of VHDL some effects have to be considered. An entry call whose synchronisation condition is met may be executed within one delta delay. From the perspective of the process the entry call behaves basically like a normal subprogram call. If the condition is not met the call may be delayed for a certain amount of simulation time until the condition is met. Meanwhile signal values may update. The entry call suddenly behaves like a potential synchronisation point. For a caller it is not predictable whether a call causes a synchronisation with respect to simulation time and corresponding updates of signal values or not. Another problematic effect related to timing is caused by using a wait statement in an entry or protected subprogram. The details have been discussed in the section about OO-VHDL. To avoid these strange effects the latest version of the proposal omits entry queues and requires protected subprograms not to contain wait statements. This guarantees that all calls to protected subprograms are executed within one delta delay with only synchronizing the access to the protected objects. Signals are not affected from such a local synchronisation.

We can state that the proposal to design shared variables as protected objects solves existing problems of shared variables. Protected objects of the proposal support object-based concepts with restrictions in the modelling of timing and deficiencies in the modelling of synchronisation conditions.

The proposal is limited to object-based modelling. Inheritance as a key concept of object-oriented techniques is not supported by the proposal.

5.2.5 Minimally extending VHDL by monitors and inheritance

To overcome the limitations of the object-based approach a proposal has been made to add an inheritance mechanism to the monitor concept for shared variables [174]. The proposal is based on an early version of a monitor concept for shared variables. In this concept subprograms of the monitor are similar to the protected subprograms of the protected object approach. The monitor performs an implicit lock on all in or inout actuals of these subprograms. Different to the previous proposal the synchronisation not only concerns the access to the shared variable but any actual. The difference especially becomes important if wait statements are allowed in monitor subprograms and signals are passed as parameters. What happens to a signal which is passed as a parameter to a monitor subprogram and which has a transaction scheduled during the execution of the subprogram and what happens to signal assignments during the execution of the subprogram from within other processes? The approach does not support the modelling of condition synchronisation.

In the proposal shared variables with their subprograms are identified as objects similar to the protected objects in the previous proposal. Motivated by the idea not to model objects only by variables the extension of the monitor concept to signals, constants, and files is suggested. Unfortunately, there is no explanation how this exactly should work. Therefore we restrict our considerations on the shared variables extensions.

A syntax construct is proposed for the shared variables which introduces a class notation. We illustrate this by an example:

```
type bbuffer is class
    variable buf : bounded_buffer_array ( 1 to buffersize);
    variable buf_in : integer;
    variable buf_out : integer;
    procedure put (object : inout bbuffer; item: item_type);
    procedure get (object : inout bbuffer; item: out item_type);
end class bbuffer;
```

It can be noticed that different to later versions of the protected object approach the variable declarations are part of the type specification and not of the body. Another smaller difference is that the object has to be explicitly passed as a parameter to a monitor subprogram.

Based on the class notion a concept for incremental modification is proposed which supports multiple inheritance. We illustrate this by an example of a class lock:

```
type lock_class is class
    variable object_is_locked : boolean := false;
    procedure lock (object: inout lock_class);
    procedure unlock (object: inout lock_class);
end class lock_class
```

The execution of the operation lock blocks the execution of any operation of the object different from the unlock operation until an unlock operation is executed. We can use this class to build a new buffer lb_buf which can be locked and unlocked:

```
subtype lb_buf is bbuffer, lock_class class
end class lb_buf;
```

The new class lb_buf inherits all the variables and operations from both parent classes. If required an inherited operation could be overridden. The subtype notation of a derived class suggests that there is a subtype relation between a class and its derived classes. One of the interesting questions related to that example is how could a lock and unlock operation be implemented without a possibility to model condition synchronisation in the approach and how could the synchronisation constraints be extended to the inherited operations from the class bbuffer. The proposal does not answer the question.

Although it is possible to override inherited operations it is somewhat unclear how to model polymorphic objects in a concurrent context. A modelling concept for heterogeneous object containers in a concurrent context is missing. There is neither a predefined assignment operation for such objects nor is there a mechanism which would allow to allocate such an object or to reference such an object. Generally, the copying of a monitor object is a problematic issue. What would it mean to copy a monitor which executes an operation and has some other operation requests queued? A reference mechanism for objects on the other hand would require VHDL to allow passing pointers across process boundaries.

If polymorphism is not supported for objects which are used in a concurrent context the difficulties mentioned in Section 3.3 about preserving the subtype relation between derived classes with respect to their synchronisation properties is not relevant for this approach. The subtype notation in the syntax of the extension makes sense.

The proposal is based on a language change with modification in the simulation concept of VHDL. The modifications are so fundamental that it is not possible to implement a translation concept from the language extension to VHDL.

5.2.6 Classification orientation

A classification oriented approach to extend VHDL is described in [37,38]. The term classification orientation is used in [38] to describe language concepts which provide the possibility to classify objects with the same behaviour and structure.

In the approach a class is described by a class declaration which can be seen as an abstract interface of the class. It abstracts the methods and generic parameters of a class. A class declaration also may contain declarations which can be used to model the attributes of a class. The declarations which are part of the class declaration also can be used to declare types and subprograms. Although they are part of the class declaration the declarations are not automatically accessible to clients of a class. The accessibility is controlled explicitly in the declaration by declaring the declarations as either public, private, or restricted. Public declarations are accessible to all clients, restricted declarations to child objects and private declarations are encapsulated in the class and protected from any external access. The bounded buffer example can be used to illustrate this.

```
class bbuffer is
    generic (buffersize : positive);
    private type bounded_buffer_array is
                        array (positive range<>) of item_type;
    private buf : bounded_buffer_array ( 1 to buffersize);
    private buf_in : integer;
    private buf_out : integer;
begin
    public method put (item: item_type);
    public method get return item_type;
end class bbuffer;
```

The introduction of a genericity concept in combination with local type declarations appears to be the most advanced abstraction concept for declaring classes.

The actual implementation of a class can be described in a class body. If a method is called the corresponding method is executed. The caller is not blocked if the method does not have any return values. If the method passes values back to the caller the caller is blocked until the method execution has finished. The method starts its execution independently from other methods already executing or concurrently called. The approach supports the modelling of concurrent objects. The internal synchronisation between concurrently executing objects has to be explicitly modelled in the methods. The explicit modelling of synchronisation is supported by language constructs of the extension which allow to refer to the state of execution of a method in another method[26]. However, the new language constructs do not provide the possibility to queue messages depending on the state which would be required for modelling condition synchronisation.
The implementation of the example could look as follows:

26. In the terminology of the thesis we could say that it is possible to reference implicit attributes of threads.

```
public method get return item_type is
begin
    -- Model mutual exclusion synchronisation:
    if (put'event or put'active) then
        … -- get must not be executed to achieve mutual exclusion
    -- Check precondition of get:
    elsif buf_in = buf_out then
            … -- Error: Buffer empty, violation of precondition
    else
        buf_out <= (buf_out + 1) mod (2 * buffersize);
        return buf((buf_out mod buffersize) + 1);
    end if;
end;
```

The extension uses the notation of a signal assignment to describe an assignment to the attributes of a class. The underlying semantics is not clearly described. From some considerations[27] on a translation mechanism from the extension to VHDL in [39] it can be concluded that this is an open issue in the language extension.

As can be seen from the example the internal state of execution can be referenced by attributes[28]. The attributes 'event and 'active are used to observe the state of execution of the put method[29]. During the execution of a put method the get method must not perform any access to the attributes of the class. The problem is how to model the queuing of the method in such a situation to achieve mutual exclusion synchronisation. A solution would be to use a wait statement to delay the execution. The problems of using such a synchronisation point in a method have been discussed. In the classification orientation approach an additional issue arises in this context. It would be necessary to recognize a method waiting at a wait statement as not being active and it would be necessary to allow to make a wait statement sensitive to the attributes which signal the state of execution of methods.

27. These considerations are not further discussed here as they only very general and vague. They do not explain how to solve the problems of translation concepts which have been presented in previous sections.
28. Attributes in the VHDL terminology like for example 'event are meant here.
29. It is assumed here that only one method can be executed at a time. As the attributes cannot distinguish different invocations of methods it is not clear how to model mutual exclusion synchronisation between multiple invocations of the same method.

The synchronisation concept causes principle problems in modelling a has-a relation between objects by hierarchically nesting objects[30]. Allowing to nest objects would require to make the state information on the execution of methods of a subobject visible to its enclosing object in order to support the synchronisation of the enclosing object. This would break encapsulation because there is no abstraction of the synchronisation modelling of objects.

The approach provides a multiple inheritance mechanism as a technique for incremental modification. It is possible to override inherited methods. The inheritance mechanism allows to inherit private, public, and restricted properties separately from a parent. The consequences of such a mechanism is that selectively inheriting properties does not preserve any kind of compatibility between parent and child. Although the approach allows overriding of methods it does not support polymorphic methods. If we selectively would inherit public properties to extend the bounded buffer this would cause an erroneous model:

```
class x_buf2 use public bbuffer is
begin
    public method get2 return two_items_type;
end class bbuffer;
```

Even if the derived class inherited the required private attributes the mutual exclusion synchronisation as it is modelled in the get method would not work any longer as expected. The get method could interfere with the get2 method. To avoid this a re-implementation of the inherited methods would be necessary.

From the consideration about the classification oriented approach we can state that it provides an object-based language concept for modelling concurrent objects. The considerations emphasize the significance of synchronisation modelling. We can conclude that a mechanism to reference synchronisation states is not sufficient for modelling internal synchronisation of methods. The approach is a concrete example how complex inheritance of synchronisation constraints can become.

5.2.7 The LaMI proposal

The LaMI proposal [21] is an entity based approach to extend VHDL. The idea is that a design entity is an object belonging to a class. As stated earlier

30. It is not really clear from the proposal if it is possible to hierarchically nest objects at all.

VHDL entities lack a concept to abstract behaviour. Therefore a new type of entities is introduced which models behaviour by operations. The operations are then abstracted in the new entity which serves as interface of the object. Such a new entity which abstracts operations must not contain any ports. The operation body is implemented in corresponding architectures. An operation in the LaMI proposal corresponds to a process. The difference to a process is that an operation is started or re-started by calling the operation and that an operation may pass parameters. Only signals are allowed as parameters of operations. Such a restriction simplifies a translation concept from the extension to VHDL. A consequence of only using signals as parameters is that it is not possible to return results of an operation to a caller immediately within a delta cycle. As the abstraction of the operation does not contain any information about the timing and synchronisation behaviour[31] of the operation a caller is not able to determine from the interface if signal parameters already contain return values after the execution of an operation or if the return values are only scheduled for the future, for example the next delta cycle.

The general synchronisation between server and client is defined in a way that a caller of an operation is blocked until the operation has finished its execution if it passes return values back to the caller. Otherwise an asynchronous execution of the operation is performed. Operations in an object can be executed concurrently, however, at most one invocation per specific operation. Messages are queued which request the execution of an operation with the same operation already executing in the object. Such messages are queued until the operation executing at the moment has finished its execution. It is not possible to explicitly control the queuing mechanism, in other words, there is no language concept which supports the modelling of condition synchronisation.

There are some limitations on the message passing concept. Messages to objects only can be sent from an architecture body at the same design level. This limitation avoids the problems of extending the distribution boundaries of objects like for example the ones discussed in the section about OO-VHDL. At the same time, the limitation makes it impossible to model a client server relation between objects that are not in a has-a relation, in other words, it is not possible to invoke an operation if the server is not a sub-object of the client. Obviously this is a sever restriction if one tries to modelling a system as a set of communicating objects.

31. The general problem of using synchronisation points in operations have been discussed earlier.

The structure of an object is stored in an attribute which is called instance variable[32]. Instance variables are encapsulated in the architecture and are only accessible through the operations. Although the assignment statement to assign values to an instance variable uses the notation of signal assignments instance variables are similar to shared variables. Assignments to them become immediately effective. If conflicting accesses to instance variables occur by concurrently executing operations the protected object approach presented in a previous section is proposed as a solution. Each instance variable is a protected object. The bounded buffer example illustrates the approach:

```
Entity bbuffer is
    generic ( buffersize : positive);
    operation put (signal item: in item_type);
    operation get (signal item: out item_type);
end bbuffer;
```

The architecture is used to model the implementation:

```
architecture behaviour of bbuffer is
    type bounded_buffer_array is
                        array (positive range<>) of item_type;
    type po_bbuffer is protected
        procedure po_put (item: item_type);
        procedure po_get (item: out item_type);
    end protected po_bbuffer;
    type po_bbuffer is protected body
        variable buf : bounded_buffer_array ( 1 to buffersize);
        variable buf_in : integer;
        variable buf_out : integer;
        procedure po_put (item : item_type) is ...
        procedure po_get (signal item : out item_type) is ...
    end protected body po_bbuffer;
    instance variable buf : po_bbuffer;
begin
    operation put (signal item : in item_type) do
     begin
        buf.po_put(item => item);
    end operation put;
```

32. Please note, the meaning of the term instance variable in the LaMI proposal that is introduced here is different from the one in the thesis.

```
        operation get (signal item : out item_type) do …
    end behaviour;
```

As the access to the attributes may conflict the protected object approach is used as proposed in the LaMI approach. In the example, it would not work to model each attribute separately as an instance variable. This would guarantee an atomic access to each attribute but it would not guarantee the required consistency between the attributes. The consistency issues force the modelling of the complete access in the operation as a call to a protected subprogram. The object modelling in the LaMI approach becomes a kind of wrapping a model of the buffer which is modelled as protected object.

The difference to directly using the protected object to model the bbuffer is that it is possible to derive a new model by inheritance. All properties of an entity are inherited in a derived entity. The proposal allows to override inherited operations with a new implementation.

The proposal offers a broadcast mechanism in its message passing concept. It is possible to send the same message to all objects that are in a specific class or in one of its child classes. By broadcasting the same message to classes with various implementations of the same operation an exotic kind of polymorphism can be modelled.

In the example of the bounded buffer it would be possible to derive a new entity x_buf2 with the additional operations empty and get2. The problem that occurs is how to implement the operations. To access the instance variable protected subprograms are necessary, but in the example, there is no protected subprogram which checks if the state of the buffer is empty. Even the implementation of get2 could not be modelled by two successive calls of op_get because this would not exclude any interference from other put or get operations. A similar problem occurs if we introduce new instance variables. The proposal does not provide a concept to preserve consistency between the new and some old instance variables.

It can be concluded that it is not the entity or architecture which requires an inheritance mechanism it is rather the protected object. Generally, it is the object providing the actual encapsulation and synchronisation mechanism which requires the inheritance mechanism. It is not possible to look at synchronisation and inheritance independently from each other.

5.2.8 Objective VHDL

The discussion about the modelling of instance variables in the LaMI proposal has shown that even in an entity based approach it would makes sense to provide object-oriented modelling concepts not only for the entities but also for the variables. This is due to the fact that although entities may be hierarchically instantiated in a has-a relation finally all attributes have to be modelled by variables and to extend or modify properties of such variables object-oriented modelling mechanisms are required.

A proposal of an entity based extension which provides such a mechanism for attributes is Objective VHDL [130,133,134,135,136,137,138]. The mechanism is based on an extension of the type system. It is extended by a so called class type. Basically, such a class type is an abstract data type. It encapsulates its structure and models its behaviour by associated subprograms. A class type separates the specification from its implementation in form of a declaration and a body similar to a protected object. The declaration contains attribute declarations and subprogram declarations. Although the attributes are part of the class type declaration they are only accessible through the subprograms listed in the declaration. The body may contain some more attributes and the subprogram implementations. Different to protected objects assignment is defined for class types.

An object of a class type is modelled by instantiating it either as a constant, a signal, a variable of the class type or by a dynamically allocated VHDL object of the class type. For each of them assignment is defined accordingly. Attributes of a class type are either treated as constants, signals, variables, or as objects of another class type according to their instantiation. This has to be considered in the subprograms of class types which model the access to the attributes. It is possible to specify the behaviour of an object which is instantiated as a signal differently from the behaviour of an object which is instantiated as a variable and differently from a constant. If the behaviour is not explicitly specified either for signals, constants, or variables then the behaviour is shared by all three. In such a case the object behaves with respect to the so called common subprogram which models the shared behaviour as if it were a constant[33] independently whether it is a signal, constant, or variable.

The bounded buffer example may provide behaviour for both, signals and variables:

```
type bbuffer is class
    generic (buffersize : positive);
```

33. This must not be confused with explicitly specifying behaviour for constant objects only. The signal, constant, or variable objects are rather treated as if they were passed as a constant parameter to the common subprogram.

```
        class attribute buf : bounded_buffer_array ( 1 to buffersize);
        class attribute buf_in : integer;
        class attribute buf_out : integer;
        for signal, variable
            procedure put (item: item_type);
        end for;
        procedure get (item: out item_type);
    end class bbuffer;
```

As it can be seen from the example it is possible to pass generic parameters to a class type. Attributes are denoted as class attributes. In the example they behave like signals or variables depending on the instantiation of the buffer. In the class type body the procedures put and get require two implementations, one for signals and one for variables. Instantiating the buffer as constant would normally make no sense.

```
    type bbuffer is class body
        for signal
            procedure put (item: item_type) is
                buf ((buf_in mod buffersize) + 1)) <= item;
                buf_in <= (buf_in + 1) mod (2 * buffersize);
            end;
        end for;
        for variable
            procedure put (item: item_type) is
                buf ((buf_in mod buffersize) + 1)) := item;
                buf_in := (buf_in + 1) mod (2 * buffersize);
            end;
        end for;
        procedure get (item: out item_type) is …
    end class body bbuffer;
```

In the implementation of put for variables the state change of the buffer becomes immediately effective whereas in the implementation of the put for signals a state change is only scheduled for the future, i.e., the next delta delay. The only indication in the abstraction that put only may schedule a state change and not actually perform a state transition is the fact that it is specified as behaviour for a signal.

In Objective VHDL it is possible to derive new class types from existing ones. A derived class type inherits the class attributes which are listed in the parent’s class type declaration. It also inherits the subprograms and their implementations in the class type body. Any attribute declarations in the class type body of a parent are not visible to a child. It is possible to add new class attributes and subprograms to a derived class type and to override inherited subprograms with another implementation.

```
type x_buf2 is new class bbuffer with
    function empty return boolean;
    procedure get2 (item: out two_items_type);
end x_buf2;
```

In an implementation of class type it is possible to call a subprogram within another subprogram. For example, it would be possible to implement get2 by calling two times the procedure get, a proper synchronisation provided. Such calls are bound statically, i.e., in derived class types get2 executes x_buf2's implementation of get even if get is overridden in a derived type. The proposal also supports dynamical binding of such calls. A special prefix this is provided for such a call. A call in get2 might look like this:

```
tmp_item := this.get;
```

The subprogram get2 invokes the potentially overridden implementation of get in a derived class type. Such distinctions between dynamically and statically binding subprogram calls can become important for the modelling of synchronisation concepts as we shall see later on (Compare Section 6.2.3).

Deriving a new class from an existing one establishes an is-a relation between the class types but not a subtype relation. Objective VHDL provides an extra mechanism to model a subtype relation on class types. The mechanism allows the declaration of a class-wide type for each class type by using the Objective VHDL attribute CLASS.

```
variable classwide_bbuffer_object : bbuffer'CLASS;
```

As described in Section 3.2.7 a class-wide type includes all objects of the corresponding class type and all objects of classes which are derived from the class type. The class-wide type has the same interface as its corresponding class type. Classwide_bbuffer_object has the procedure put for variables and signals and the common function get in its interface. The common functions empty and get2 are not part of its interface. As objects of derived classes inherit all the properties of the parents a class-wide type may establish a subtype[34] relation. Thus, all objects of a class-wide type and not only

34. Subtype relation is not used here in the strict VHDL terminology but in the sense that derived classes are subsets of the class-wide type sharing common properties especially behaviour.

the objects of the corresponding class type provide the subprogram declarations of the corresponding class type as part of their interface. Class-wide objects of bbuffer'CLASS with the properties of the class type x_buf2 also provide put and get as part of their interface.

From what was stated above it follows that an object of a class-wide type is a heterogeneous object container. If an object is stored in such a container it preserves in principle its type information and the related properties. The object keeps its structure. Although the object also preserves its behaviour the interface to communicate with the object is taken from the class-wide type. In other words, an object that is stored in a heterogeneous object container does not provide all its subprograms as services to clients but only those that are part of the interface of the class-wide type. As mentioned above, this is the same interface as the one of the class type corresponding to the class-wide type. If a subprogram of such an object is executed it behaves according to the properties of its actual type. Combined with the possibility to override behaviour class-wide types can be used to model polymorphic objects.

The extensions to the type concept require type check mechanisms for class types and class-wide types to preserve the strong typing of VHDL. As it is not possible to perform all the checks statically at elaboration time it is necessary to perform some of them dynamically during simulation.

The approach presented so far is based on the same ideas and principles as the proposal to extend VHDL which is part of this thesis and which will be discussed later on. From a semantics point of view there are only a few subtle distinctions between the proposals which we are going to discuss later on. The main difference is that Objective VHDL uses another syntax compared to the proposal of this thesis.

Objective VHDL not only provides the type extension concept but also an inheritance mechanism on entities and architectures. The mechanism allows to derive new entities and architectures from existing ones. If a new entity or architecture is derived, basically, all the features from the parents are inherited. It is possible to add some new features to the entry or architecture, for example, new ports or new concurrent statements. It is also possible to override concurrent statements in derived entities or architectures. As the entities and architectures do not provide an abstraction of their behaviour and as such a concept is not added to Objective VHDL it is not possible to incrementally modify behaviour without breaking encapsulation. Corresponding considerations made in the discussion of the other entity based approaches can be applied accordingly.

As the designers of Objective VHDL are aware of the difficulties of abstracting behaviour at an entity level and providing the corresponding synchronisation mechanisms which have to be integrated into the timing model and synchronisation concepts of VHDL, message passing and synchronisation at entity level is left to the designer. The designer still has to use VHDL communication concepts but with the extended capabilities of the class types for signals. Even with these extended capabilities reflective modelling especially of message passing concepts is a complex task. Therefore proposals have been made to provide tool support for modelling message passing [131,132]. Such an approach avoids the difficulties of the integrating different synchronisation philosophies by separating the model for message passing from the VHDL synchronisation concepts.

In the example it would be possible to model the bounded buffer as an entity/architecture pair.

```
entity bbuffer_entity is
    port (in_channel: in channel; out_channel: out channel);
    procedure put (signal object : inout bbuffer; item: in item_type);
    procedure get (signal object : inout bbuffer; item : out item_type);
end bbuffer_entity;
```

The operations of the class are modelled as procedures. As the state of an entity class is stored in a signal of type class bbuffer the operations perform their access to the entity's state via the parameter object of type class bbuffer. The corresponding architecture contains the subprogram bodies and a process which dispatches the messages coming via the channel.

```
architecture behaviour of bbuffer_entity is
    signal buf : bbuffer;
    procedure put (signal object : inout bbuffer; item : in item_type) is
    begin
        object.put(item);
    end put;
    procedure get (signal object : inout bbuffer; item : out item_type) is
    begin
        object.get(item);
    end get;
begin
    dispatcher : process
    begin
        wait on in_channel'transaction;
        if in_channel.new_message then
```

```
            in_channel.dispatch(in_channel, buf, out_channel);
        end if;
    end process;
end behaviour;
```

Channel is specified as a class type with a procedure dispatch as its subprogram and a class attribute to model messages.

```
type channel is  class
    …
    class attribute msg_attribute: base_message'CLASS:=
                                                init_base_message;
    function new_message return boolean;
    for signal
        procedure dispatch (signal in_channel: in channel;
                            signal object  : inout bbuffer;
                            signal reply_channel: out channel);
    …
    end for;
end class channel;
```

The implementation of the subprogram dispatch receives a new message and executes the polymorphic procedure exec depending on the message received. The unusual situation that a parameter of class type channel is required as parameter in the subprogram dispatch which is just of class type channel is due to the fact that the object which executes dispatch itself has to be passed as parameter to a procedure call in the implementation of the procedure dispatch.

```
procedure dispatch (signal in_channel: in channel;
                    signal object  : inout bbuffer;
                    signal reply_channel: out channel) is
    variable msg: base_message'CLASS;
    variable result_msg: base_message'CLASS;
begin
    in_channel.receive (msg, reply_channel);   --receive new message
    msg.exec (object, result_msg);             --execute the message
    reply_channel.send (result_msg, in_channel);         --reply results
end dispatch;
```

Messages to the buffer and replies from the buffer are modelled as class types which are derived from an abstract class type:

```
type base_message is abstract class
end class base_message;
```

Abstract means that the class type does not require a body. The messages and replies encode in a way the operations of the bounded buffer and the corresponding results.

```
type put_msg is new class base_message with
    class attribute item : item_type;
    for variable
        procedure exec (signal object : inout bbuffer;
                        variable msg : out base_message'CLASS);
    …
    end for;
end class put_msg;
```

The results of get are modelled by a class type get_result.

```
type get_result is new class base_message with
    class attribute item : item_type;
     …
end class get_result;
```

The implementation of the procedure exec in the class type put_msg executes the put operation of the bounded buffer which is modelled as a procedure put in the entity bbuffer_entity. If the channel receives a put_msg it executes the procedure put due to the dispatching mechanism. From a VHDL point of view this requires an extension of the abstraction boundary of the procedure put. This is solved in Objective VHDL by a special construct which makes the subprograms of an entity visible to certain implementations of subprograms in a class type body. The restriction with that mechanism is that the implementations of the subprograms in the class type body are only used within the corresponding entity. To be callable from somewhere else the subprogram requires at least a second implementation in the class type body. In the example of the put_msg the subprograms of the entity bbuffer_entity are made visible to the procedure exec.

```
for entity work.bbuffer_entity    -- make subprogram put visible
    for variable
        procedure exec (signal object : inout bbuffer;
                        variable msg : out base_message'CLASS) is
            …
        begin
            put (object, item);
            …
        end;
        end for;
    end for;
```

The example sketches the various type classes which are necessary to add the message passing mechanism to an entity class. It illustrates the difficulties of high level communication and synchronisation modelling, namely, a proper encapsulation and abstraction of low level protocol layers. The reasons can be identified from the example as immanent in VHDL. Two channels are used to model communication instead of encapsulate the whole communication in one class type because it is not possible to model bi-directional signals without using complex resolution mechanisms of VHDL even if no access conflicts can ever occur. It is also problematic to abstract a channel model from the server client relation in which it is used. In the example, the channel uses the class type bbuffer. That means, there is no clear separation between protocol layers. As a consequence the channel is not re-usable in other server client relations.

An approach which avoids this unwelcome effect is to allow in certain circumstances a procedure which is declared by a declarative item that is not contained within a process statement to have a subprogram call with inout or out mode signal parameters and the parameters not associated with formal parameters of the given procedure[35]. The idea is to allow such an exception for subprograms which are declared in an entity class and which use a homograph of the name of the associated signal as a parameter. In the example the procedure put would have a signal parameter buf instead of object. In the procedure exec of the type class put_msg the procedure put is called with the signal buf as a parameter. The parameter object of type class bbuffer is not any longer required in the procedures of the other type classes. The problematic issue of such an approach is that suddenly signal assignments located somewhere in a model spread drivers of a signal anywhere in a model.

Another approach is to simply use a shared variable of type class bbuffer to model the state of the entity class bbuffer_entity.

The example models one client server relationship. An extension to have several clients is not straightforward. It would be possible to use a different dispatching process which could model mutual exclusion synchronisation. Another option would be to model several dispatching processes. The question is then, how to resolve access conflicts to the object's state. A com-

35. This would weaken the rules in the LRM how to update a projected output waveform. The consequences would be that an analyser is not able to determine the drivers of a process at analysis time.

pletely unsolved issue which is omitted in the example is how to integrate condition synchronisation in such a concept of reflective modelling message passing. It is not clear how entity classes could be useful to solve such problems.

This careful survey how to apply concepts of Objective VHDL discusses modelling approaches and problems which are to be studies in the considerations of the proposal of this thesis due to the similarities in the proposals (Compare Chapter 6). They are used as a starting point for the considerations about modelling concepts with the language extension proposed in this thesis. It shall turn out that there are not only similarities in modelling issues between both proposals but also that there are some basic ideas on how to translate the extension into VHDL which are applicable to Objective VHDL.

5.2.9 The Wright Laboratory report

The proposal presented in the previous section was based on the extension of the type concept of VHDL by introducing class types. In this section we present an alternative way to extend the type concept. The extension concept was proposed in a technical report of the Wright Laboratories [113].

Its basic idea is to adapt concepts from the Ada9X draft[36][18,93] to VHDL. The proposal extends the VHDL type concept by derived types for type extension. The syntactical notation for extending types is taken from Ada. To derive a new type from an existing one the keyword new is used.

```
type derived_type is new parent_type;
```

The derived type is derived from the parent type and has the same properties as the parent type. If the parent type is a record then the derived type is a record with the same record elements. Different to a subtype declaration which also could be used to declare a type[37] with the same properties a derived type declaration does not impose a subtype relation between the parent type and the derived type. Each type may have a set of primitive operations associated with it. The primitive operations of a type include basic operations, predefined operators, and for types declared in a package declaration subprograms with a parameter or result of the type also declared in the package declaration. Additionally a derived type inherits the primitive operations of its parent. For a derived type it is possible to override such primitive

36. The features are meanwhile part of Ada95 [19,95].

37. Strictly speaking in VHDL terms, a subtype is not a new type.

operations which are inherited from a parent. As the derived type is a different type to the parent the overridden operation overloads the parent's one.

Like Ada, the proposal allows to extend a derived type. It is possible not only to add new primitive operations but also to add additional elements to a record. As a curio the proposal allows different to Ada to extend array types although extendable arrays have been identified by the designers of Ada to cause too much implementation cost.

Types which are allowed to be extended are marked by the keyword tagged. Each object of such a type has an implicit element which indicates its type. The element which stores the type information is called tag. It contains the type information of the object during run-time. The type itself and its derived types are called tagged types. In the example of the bounded buffer the buffer is modelled as a tagged type with the put and get operations as its primitive operations.

```
package bbuffer_package is
    type bbuffer is tagged private;
    procedure get (object : inout bbuffer; item : out item_type);
    procedure put (object : inout bbuffer; item : in item_type);
private
    type bounded_buffer_array is array(positive range <>) of item_type;
    type bbuffer is tagged record
        buffer : bounded_buffer_array (1 to buffersize);
        buf_in : integer;
        buf_out : integer;
    end record;
end;
```

The proposal assumes that the latest standardization of VHDL would include an encapsulation mechanism in form of private types. Private types encapsulate the structure of a type. The part before the keyword private is the visible part of a package. In the example, this part includes primitive operations of the bounded buffer. The private part is hidden to the user of the package. It contains a detailed description of the private type, i.e., its structure. In such a way, the attributes of an object which are modelled by the elements of the tagged types are only accessible through the primitive operations. The implementations of the operations are in the package body.

Modelling classes as tagged types allows the instantiation of objects as signals or variables. In the example, only the instantiation as a variable would make sense, because of the interface of the primitive operations. For objects instantiated as signals another set of primitive operations would be required with a signal of type bbuffer as a parameter.

The proposal allows to extend the record by adding new elements to it. The proposal allows, for example, to derive a new buffer type with an additional operation gget which reads an item from the buffer like the get operation. The difference between the get operation and gget operation is that a gget operation only can be executed immediately after a get or a gget operation. This additional synchronisation constraint requires the tracing of the operations' invocation history. The result of that tracing could be stored in an additional attribute. Thus, the derived type could be modelled as follows:

```
package bbuffer_package_gb
    type gb_buf is new bbuffer with private;
    procedure gget (object : inout gb_buf; item : out item_type);
private
    type gb_buf is new bbuffer with record
        after_put : boolean;
    end record;
end;
```

The type gb_buf is derived from bbuffer. It inherits the elements of the type bbuffer. The new element after_put which stores the tracing of the invocations is added to the record. The derived type also inherits the primitive operations put and get. The declaration of the element to trace the invocation history illustrates that there are no language concepts for modelling synchronisation constraints. All kind of synchronisation including mutual exclusion synchronisation has to be modelled explicitly in tagged types. In the example, the interesting question about synchronisation modelling is how to add the new synchronisation constraint without rewriting the inherited operations put and get and without breaking their encapsulation. This is a problem because the inherited operations do not record their invocation for tracing. Such kind of problems are discussed in detail later on (Compare Section 8.1.4). The proposal does not provide solutions for that.

A subtype relation on tagged types is introduced by class-wide types. Each tagged type implicitly has an associated class-wide type. Like in Ada it is denoted by the attribute CLASS. The class-wide type is the union of the corresponding type and all its derived types. A class-wide type is an indefinite type which can be viewed as a kind of unconstrained subtype because its size is unspecified. In VHDL the subtype indication must define a constrained subtype in an object declaration if the size of the base type is unspecified. Constants of such a base type without a constrained subtype in the subtype indication define the constraints by their initial value. Likewise, an object of a class-wide type must have a definite type. This is achieved by initialization with a value of a tagged type. The class-wide object is constrained by the tag of the tagged type. Its tag must not change during the lifetime of the object.

Class-wide types are used to model polymorphic objects. A class-wide object is treated as if it has the type which is indicated by its tag. If a primitive operation is called with a formal parameter of a class-wide type the primitive operation of the type indicated by the tag is executed. If such a call occurs in a simulation the tag is determined during run-time and the corresponding operation is selected. This dynamic selection is called dispatching and is the key mechanism for modelling polymorphism in the language extension.

As a tag of a class-wide object must not change during its lifetime only dynamically created objects are appropriate for modelling polymorphism. Essentially this means, that only parameters and objects designated by access value can be used to model polymorphic objects. In the example, we could model a heterogeneous object container for buffers with a designated subtype which is a class-wide type.

```
type classwide_bbuffer_designator is access bbuffer'Class;
variable polymorphic_pointer : classwide_bbuffer_designator;
…
variable classwide_bbuffer_object : polymorphic_pointer := new gb_buf;
```

Excluding signals from the modelling of polymorphic objects is somewhat contradictory to the statement of the report that signals are the most likely objects for object-oriented programming. The consequences of not being able to model polymorphic objects with signals have been discussed in Section 5.2.2. One main consequence is that there is no concept for modelling communication and synchronisation between concurrently running objects. Ideas to use object-oriented concepts to model the communication like in Objective VHDL are not feasible because communication is based on signals and signals cannot be used to model polymorphic objects.

Additionally to the concepts taken from Ada an inheritance mechanism for entities and architectures is proposed. Similar to the tagged types concept an entity can be marked as expandable by the keyword tagged. Different to the tagged types multiple inheritance is supported for entities. In the inheritance mechanism a derived entity inherits the ports, generics, declarations, and statements of its parents. This is like in all the other entity based approaches. It is possible to modify inherited parts, i.e., to override them. The same mechanism can be applied to architectures in which it is even allowed to remove inherited parts. The inheritance mechanism on entities and architectures is not discussed here any further as all the consequences and effects on encapsulation concepts and compatibility issues have been discussed previously.

We conclude the section by remarking that the type extension mechanism with its combination of strong typing and flexibility very well fits in the existing language concept and philosophy of VHDL. As we shall see later on, the proposal to extend VHDL which is part of this thesis is also a tagged type approach, however, one which avoids the restrictions on modelling class-wide signals.

5.2.10 SUAVE

In the SAVANT and University of Adelaide VHDL Extensions (SUAVE) project proposals for extensions to VHDL for high-level modelling are developed [6,9]. The extensions are intended to remove deficiencies for system level modelling in VHDL which are similar to those mentioned in Section 5.1.4.

The project identifies two main areas which require improvement [13]. Data modelling capabilities need to be improved and a better modelling support for abstraction of concurrency and communication is required. Compared to the other proposals this is not really new. What is different to most of the proposals is that the two areas are treated quite independently from each other in SUAVE. There is a proposal for an object-oriented VHDL to improve support for data modelling [7,10,11,14,16] and there is a proposal for abstraction of concurrency and communication in VHDL [8,15].

The object-oriented extensions develop the tagged types approach of the Wright Laboratories further. The extension provides the concepts of tagged types, derived types, primitive operations, private types, and class-wide types. If we modelled the example of the bounded buffer as a tagged type it would look identical to the code from the Wright Laboratory.

The SUAVE extension removes the restriction of not supporting signals as polymorphic objects. The restriction was caused by the fact that a class-wide type was treated as indefinite or unconstrained type for which we only can declare objects with explicit constraints or initial values. The tag of an object was treated as part of the constraint and after the elaboration of an object declaration it is not possible to change the constraints, i.e, to change the tag. Thus, a class-wide signal was not able to change its tag after elaboration.

A proposal to remove these restrictions in order to support polymorphism of class-wide signals originally was made in [151][38]. A class-wide type is treated as a constrained type with the tag not being a part of the constraint. Object declarations of such a constraint type do not require any further constraints. Such objects are allowed to change their tags during simulation.

SUAVE takes a similar approach to achieve the same effect. It distinguishes if a class-wide object is modelled as constant, variable, or signal. For signals it defines a class-wide type to be an unconstrained type for which we can declare a signal without explicit constraint or initial value. The class-wide object is not constrained by a tag. Its tag may change during the lifetime of the object.

If a class-wide object is modelled as a constant it must be initialized with a value of a specific tagged type. Variables must be declared to have a specific type. Although objects designated by an access value are variables which always have a specific type it is possible to model polymorphism by a variable of an access type which has a class-wide type as its designated subtype.

Supposed we have primitive operations for signal parameters we would have two possibilities to model a heterogeneous object container for a bounded buffer. We could model it as variable of an access type:

```
type classwide_bbuffer_designator is access bbuffer'Class;
variable polymorphic_pointer : classwide_bbuffer_designator;
```

Alternatively it could be modelled as signal of a class-wide type:

```
signal polymorphic_buffer : bbuffer'Class;
```

The first modelling approach allows the modelling of class-wide instance variables. The limitation with objects containing such class-wide instance variables is that they must not be exported across process boundaries. Passing class-wide objects across the boundaries is possible with the latter modelling approach.

Applying the considerations on distributed objects in Section 3.3.18 we can state that SUAVE supports a modelling style which is a compromise between providing instance variables and forcing a reference free communi-

38. The proposal is a predecessor of the one made in this thesis

cation between processes by a copy model. Thus, it combines a high degree of modelling flexibility with implementation efficiency.

In the Wright Laboratory approach a derived type is a new type which is different from its parent. This is the same in SUAVE, additionally, the two types are considered as closely related types in SUAVE. As a consequence a type conversion provides for explicit conversion between the types. There are some limitations on the conversion. Basically, only a conversion from a derived type to a parent is allowed. Such a conversion does not change any attributes in an object including its tag it rather changes its view. Therefore SUAVE refers to such a type conversion as a view conversion. Conversions in the other direction require aggregates which can add record elements to existing records.

A view conversion is a mechanism to control dispatching. It could be used to model dynamic binding of internal calls to primitive operations from within other primitive operations of the same type. It is possible to re-dispatch a call to a primitive operation to another operation. For example, an implementation of the primitive operation get2 of a bounded buffer x_get2 could use two subsequent calls to the get operation[39] which is a primitive operation of x_buf2 inherited from the bounded buffer bbuf.

```
procedure get2 ( signal object : inout x_buf2,
                 signal items : out two_items_type) is
    type x_buf2classwide is x_buf2'Class;
begin
    get(x_buf2classwide (object), items(1));
    get(x_buf2classwide (object), items(2));
end;
```

The view conversion of objects to a class-wide actual results in a dispatching call of get. In a derived type if get is called it is not the implementation of x_buf2 which is executed but the implementation of the derived type. In other words, in a dispatching call of get2 re-dispatching of get occurs because of the view conversion.

The extensions in SUAVE described so far support re-use by providing polymorphism. Additionally to the extensions for dynamic polymorphism there are also mechanisms in SUAVE which allow the modelling of statically

39. The example assumes here that there are no synchronisation problems with that implementation. Different to the example in the previous section the get operation must be modelled as a primitive operation of bbuffer with a parameter of class signal.

polymorphic objects. Genericity provides static polymorphism by abstracting from actual parameters by formal generic parameters. The rudimentary concepts of VHDL for genericity are generalized in SUAVE. It is not only possible to pass generic parameters to entities, architectures, and blocks, it is also possible to pass them to packages and subprograms. The items to be passed as generic parameters include constant parameters, formal types, formal subprograms, and packages. The theory of genericity, its flexibility, the concept to preserve typing, and the mechanisms to make genericity safe are taken from Ada.

The extended genericity concept includes the possibility to model a derived type as a generic parameter of a package. Such an approach can be used to model mixin inheritance. As described in Section 3.2.13 mixin inheritance is a technique which allows to inherit properties from various tagged types but which avoids the complexity of multiple inheritance.

We can illustrate this with the example of the bounded buffer and its derived buffer lb_buf. The properties of lb_buf are described in detail in the section about the monitor extension with inheritance. The buffer has two additional operations lock and unlock which it inherits from another class and which can be used to blocks the execution of any operation of the object different from the unlock operation until an unlock operation is executed.

The package which provides the lock uses the generic parameter to abstract from the tagged type which uses the lock and unlock operations. To preserve type compatibility both, the lock_type[40] and the bounded buffer require a common ancestor.

```
package lock_package is
    generic ( type generic_tp is new common_parent with private);
    type lock_type is new generic_tp with private;
    procedure lock (object : inout lock_type);
    procedure unlock (object : inout lock_type);
private
    type object_locked_or_unlocked is (unlocked, locked);
    type lock_type is new generic_tp with record
        lock_on_object: object_locked_or_unlocked;
    end record;
end;
```

40. lock_type in the example corresponds to lock_class in the monitor example from Section 5.2.5.

The new buffer is derived from an instantiation of the lock_type which has the bounded buffer as an actual parameter:

```
package bounded_buffer_package_lb is
    type lb_buf   is new bbuffer with private;
private
    package lock_buffer_package is new lock_package(bbuffer);
    type lb_buf is new lock_buffer_package.lock_type with null record;
end;
```

The derived type lb_buf inherits the lock and unlock operation from the instantiation of lock_type. The issue how the synchronisation concepts of lock and unlock are extended to the inherited put and get operations is still an open question. We shall discuss this question when we present the language extension of this thesis.

The example also illustrates another interesting language extension. The limitation to use packages only as top level design units in a library is removed. It is possible with some smaller restrictions to hierarchically nest packages in other packages, in entities, architectures, and subprograms. This feature improves thoroughly the encapsulation mechanisms of VHDL.

There are some more extensions on the type concept like abstract and limited types which are taken from Ada. They also in combination with tagged types improve the level of abstraction for modelling data.

Generally, all the language features taken from Ada integrate nicely with existing language features of VHDL. This is not surprising because the original definition of VHDL drew heavily on Ada.

The extension of this thesis is also related to Ada and would co-operate with the presented proposals of SUAVE to improve data modelling. We shall see this later on.

The proposal of SUAVE for abstraction of concurrency and communication is separate from the object-oriented extensions. It takes concepts from system level description languages like SDL and Estelle. The main extensions are abstract communication by channels, adding an abstraction mechanism to processes, and supporting dynamic creation of processes.

The new channel concept is similar to the one from Estelle, i.e., SUAVE provides an asynchronous message passing via its channels. A channel is a new class[41] of objects like constants, signals, and variables. Similar to these classes channels are typed. The type definition looks similar to a file type definition.

41. Class is used here in the LRM terminology of VHDL.

```
type channel_type_name is channel of message;
```

Like signals are used as formal ports in interface lists channels are used as so called formal channels. As channels are only uni-directional they may have the mode in or out in an interface list. In the terminology of the thesis such formal channels are interaction points which abstract synchronisation. The synchronisation is implemented in the channels which have a message queue. Sending to a channel means to put a message into its queue and receiving means to read a message from the queue. To send and receive messages from channels explicit send and receive statements which refer to the channel are used. As the communication is asynchronous a sender continues its execution after putting a message into the channel's queue. If several messages are sent simultaneously to the same channel the messages are put in a non-deterministic order into the queue. Likewise, a receiver reads a message from the buffer and continues its execution. If there is no message in the queue the receiver waits until a message has arrived. Multiple receivers which all read the same channel all receive identical messages in the same order.

If multiple messages arrive on multiple channels SUAVE allows the selection of alternative channels for receiving these messages. It is possible to guard the selection of a message. Guarding the receive of message is a way to model condition synchronisation.

We illustrate this with the bounded buffer example. The synchronisation of the bounded buffer is modelled in a process with the messages as channel variables. The bounded buffer itself is modelled as variable. The type of the variable is x_buf. This type of buffer provides primitive operations to check whether the buffer is in the synchronisation states empty or full. We can interpret the operations as a controlled way to break encapsulation of the buffer. Breaking encapsulation is required because synchronisation is performed externally in the process.

```
process buffer_synchronisation is
    port (  channel get_msg : in get_msg_type;
            channel get_rply : out get_rply_type;
            channel put_msg : in put_msg_type);
    variable bounded_buffer : x_buf;
    variable item : item_type;
begin
    select
        when not (empty (bounded_buffer)) =>
                                        receive from get_msg;
                                        get (bounded_buffer, item);
```

```
                                    send item to get_rply;
          or  when not (full (bounded_buffer)) =>
                                    receive item from put_msg;
                                    put (bounded_buffer, item);
          end select;
     end process;
```

If the process receives a message it selects a receive statement that has a guard that evaluates to true and that has a message in its buffer. The receive statements as new synchronisation points take the place of wait statements in the example. The process performs exclusion synchronisation while the select statement models condition synchronisation. The process is an additional level of abstraction to model the synchronisation separately from the implementation. According to Section 3.3.14, the task of the additional level is to implement the matching phase. A get message is only received if the buffer is not empty and a put message is only received if the buffer is not full. In case there are no messages in the corresponding channels, the process blocks until a message arrives in a channel which is named in a receive statement and whose guard evaluates to true. The receive statement and any following statements are executed as guarded operation. It is not possible to re-queue from the operation to a channel in the select statement. As the receive statements are executed after the evaluation of the guard it is not possible to refer to the content of a message in a guard.

For example, we are not able to model the synchronisation conditions of a buffer with an n_put operation which may put a variable amount of items into the buffer. To check if it is possible to put the items into the buffer it should be either possible to receive the message, read the number of items and to eventually re-queue the operation or to refer to the number of items in the guard before accepting the n_put operation.

The modelling approach in the example of the buffer x_buf is appropriate for the synchronisation modelling of the bounded buffer. The problem is that it does not support re-use. For example, if we think of the derived buffer x_buf2 it becomes obvious that a different process with a new select statement is required as the synchronisation conditions have been modified.

```
     select
          when  not (empty (bounded_buffer)) =>
                                   receive from get_msg;
                                   ...
     or   when  not (empty (bounded_buffer) ) and
                not (one_item_in_buffer (bounded_buffer)) =>
```

```
                                 receive from get2_msg;
                                 …
   or  when  not (full (bounded_buffer)) =>
                                 receive item from put_msg;
                                 …
   end select;
```

The get2 operation only can be executed if two or more items are stored in the buffer. The check of this new synchronisation state has to be performed as part of the select which means to re-implement the matching phase. An additional operation one_item_in_buffer which makes the new state externally visible is also required.

The synchronisation modelling does not integrate with inheritance in SUAVE. Also, the synchronisation mechanism interacts with the original synchronisation and timing concept of VHDL similar to previously discussed rendez-vous concepts. A receive statement introduces a synchronisation point into the language. The consequences of such additional synchronisation points have been discussed earlier.

From the example which models the synchronisation separately in a process another extension of the language can be seen. The extension provides a new kind of process which abstracts concurrency. The new kind of process has a specification which models the abstraction boundary and a body which models the implementation. The specification abstracts the communication by ports and formal channels and other features by generic parameters. For example, the specification of the example abstracts the channels:

```
   process buffer_synchronisation is
      port (  channel get_msg : in get_msg_type;
              channel get_rply : out get_rply_type;
              channel put_msg : in put_msg_type);
   end process;
```

As it can be seen from the example the specification does not abstract the synchronisation.

Such a new kind of process is a template which can be instantiated. An instantiation is similar to a procedure call. It can occur as sequential or concurrent statement. The concurrent statement works as expected. It is equivalent to a block statement containing a process whose declaration part and statement part are taken from the process body. The execution of the sequential instantiation causes an elaboration of the process. After elaboration the process starts its execution. The execution is performed concurrently with the instantiating process. It is not only possible to dynamically create processes during simulation it is also possible to terminate them. As a consequence drivers are dynamically created and removed in a simulation. Creation and removing of drivers is treated as if drivers were connected and disconnected. This integrates this new powerful feature seamless into the existing language philosophy of VHDL.

The dynamic creation of processes allows the modelling of concurrent objects. However, the signals which are used to implement the objects must be guarded signals to allow the drivers to be connected and disconnected. A guarded signal of tagged type or class-wide type would require the definition of a resolution mechanism on the tagged type or class-wide type. That means, the integration of the dynamic concurrency concept with the object-oriented extensions is an open issue in SUAVE.

Generally, we can conclude that the SUAVE approach proposes very powerful features from system specification languages for integration in VHDL. Each of the features can be conceptually integrated with the existing language. However, the integration of the proposed object-oriented features with the proposed features to abstract concurrency which are especially required to model condition synchronisation has been shown to be an unsolved issue.

The proposal of this thesis to extend VHDL which is based on similar principles takes care of the synchronisation modelling and its integration problems. It offers modelling concepts to avoid them as we shall see later on.

5.2.11 Variant records

An idea to extend VHDL which is closely related to the tagged types approaches is the proposal[42] to introduce variant records into the language [50]. A variant record is a record type which may exist in several variants. A special record element indicates which variant is assumed by a record at a given simulation time. The special element is called selector. Depending on the variant, alternative record elements could be included in the record or not.

The terminology for variant records of Pascal [99] points to the relation between the approaches. Pascal uses the term tag field to denote the special

42. Actually, there are some variants on variant records in the proposal which are not further discussed here. For example, the proposal presents an Ada-like approach in which the choice between alternatives is governed by a discriminant, like for example:
type bbuffer_classwide (discriminant: tag_field := bbuffer_tag) is …

record element which indicates the actual variant of the record. The explicitly modelled special record element in a variant record corresponds to the tag of a tagged record. We can illustrate this by the example of the bounded buffer.

```
type tag_field is (bbuffer_tag, gb_buf_tag, lb_buf_tag);
type bbuffer_classwide is record
        buffer : bounded_buffer_array (1 to buffersize);
        buf_in : integer;
        buf_out : integer;
        with tag_field select
            when bbuffer_tag =>
                null;
            when gb_buf_tag =>
                after_put : boolean;
            when lb_buf_tag =>
                lock_on_object: object_locked_or_unlocked;
        end select;
end record;
```

An object of type bbuffer_classwide may exist in three variants of the bounded buffer which are distinguished by the record element tag_field. From the structure of the record it is possible to derive the principle idea how to translate tagged records into VHDL in a pre-processor approach. The following discussion on variant records should also be considered when defining a language extension which can be translated into VHDL. It illustrates main aspects of how to model objects as signals.

An object of a variant record may store values of different record variants in a simulation. We could say it has a polymorphic structure. The interesting question related to the polymorphic structure is what operations can be performed to dynamically change the structure. If it is possible to change the variant by assigning the selector another value then it would be an open issue what happens to the values in the alternative record elements. Therefore the variant record approach proposes to allow only assignments to selectors together with at least assignments to the corresponding alternative record elements. In the most restrictive variant the approach only allows complete record assignments to change the record variant. An aggregate value for a variant record must contain locally static values for the selectors. This allows a compiler to check at analysis time if the alternative record elements correspond to the selector.

The consequences of these rules are quite obvious if they are applied to variables. This is different if signals of variant records are used because assignments to signals affect the projected output waveforms of the corresponding drivers. The rule does not define what happens to transactions scheduled for alternative record elements after the selector changes its value to select other record elements. Are they cancelled from the projected output waveform? Do the drivers associated with the alternative record elements even exist any longer after such a change in the selector? A related question is what happens to wait statements which synchronize on an event in such an obsolete alternative record element? As it was stated in Section 5.2.10 it would be a complex task to define a language extension which allows the dynamic creation and removal of drivers.

This situation is caused by the composite signal semantics of VHDL which defines a composite signal to be a collection of signals. The effect is that signals and drivers are dynamically created and removed in a simulation. The variant record approach proposes to introduce atomic signals in VHDL to solve the problem. Such an atomic signal is treated as a single signal even if it is a record. A composite atomic signal is not a collection of signals. Because there is only a single signal the elements of a composite signal only can be used as a reference within an expression. They cannot be used as a signal itself. For example, it is not allowed to use them in a sensitivity list. Only the atomic signal as a whole can be used in a sensitivity list.

By introducing atomic signals it is possible to define variant record signals to be atomic by default [89]. According to what was stated above the problem of dynamically creating and removing drivers does not exist in atomic variant record signals. The introduction of atomic signals require some more language changes which are not discussed here, for example some modifications of the resolution mechanism would be required for atomic signals.

The proposal considers variant records as predecessors of tagged and class-wide types. The limitations of variant records compared to tagged types are a missing support for re-use. Adding a new variant to a variant record requires not only to add new alternatives to the record type definition but it also requires a modification of each part of the model which distinguishes the different variants in order to take the new variant into account.

The proposal in [50] considers the tagged types to just overcome these limitations by making the selector implicit and by automatically introducing the distinction by the dispatching mechanism.

So far, we have discussed the main proposals for typed based and entity-based object-oriented extensions to VHDL. The following proposals do not provide something substantial new to the discussion on object-oriented extensions. They are mentioned for completeness.

5.2.12 Data modelling extension to VHDL

A variant of the previously discussed proposal to minimally extend VHDL by monitors is the approach to a data modelling extension to VHDL [148,149]. In principle, it is the same idea which only uses another syntax. A new type of record is introduced which is called abstract data type. The example of the bounded buffer looks with the different syntax as follows:

```
type bbuffer<buffersize : positive> is
    record
        protected:
            buf : bounded_buffer_array ( 1 to buffersize);
            buf_in : integer;
            buf_out : integer;
        public:
            procedure put (item: item_type);
            procedure get (item: out item_type);
    end record;
```

As it can be seen from the example, as slight modification to the monitor approach is that the interface of the class distinguishes between protected and public properties of a class. Although not in the example, private properties are also allowed. The consequences of such a distinction especially on compatibility issues have been already discussed. The second modification is that the object is not explicitly passed as a parameter to its operations. The third modification which can be seen from the example is that it is possible to pass generic parameters to an abstract data type, i.e., to a class.

Another modification of the proposal is that it allows to instantiate signals of an abstract data type. In that case according to an example in [147] exclusion synchronisation of the operations is not provided. Instead it is possible to declare a process inside an abstract data type which can be used to model synchronisation. The process has access to the attributes of the object and thus may interact with the operations[43]. With such a process the signal becomes an active object. For example, it would be possible to model the synchronisation of a buffer of class lb_buf which was introduced in Section

43. It is not quite clear how this mechanism works with signals which are ports.

5.2.5 if the lock and unlock operations are called from different processes. The buffer lb_buf has attributes which signal by toggling if a lock or unlock operation has occurred. It also has an attribute to indicate if the buffer is locked or unlocked:

```
type lb_buf is public bbuffer with
    record
        protected:
            signal_lock : bit;
            signal_unlock : bit;
            object_is_locked : boolean;
        public:
            …
    end;
```

The process resolves the access conflict to the attribute object_is_locked which is used to model condition synchronisation in the operations:

```
process_in_an_ADT : process (signal_lock, signal_unlock)
begin
    if signal_lock'event then
        object_is_locked <= true;
    elsif signal_unlock'event then
        object_is_locked <= false;
    end if;
end process;
…
procedure put (item: item_type) is
begin
    if object_is_locked then wait until not object_is_locked;
…
end procedure;
```

Different to normal procedures operations which perform write access from different processes do not cause the VHDL model to instantiate multiple drivers for the object. It rather instantiates multiple drivers for the single record elements of the object[44]. A resolved signal is not required. We could say the resolution mechanism is modelled as part of the process and different

44. It is not clear what happens if the object is an inout port and the write access is performed across distribution and abstraction boundaries of architectures.

to the existing resolution mechanism of VHDL the new one can refer to the previous state of the object to perform the resolution.

While the example models parts of the condition synchronisation it does not provide a concept for the exclusion synchronisation issue. At the same time it introduces the previously discussed problem how to inherit a process and its synchronisation concepts.

It is interesting to note, that it is not possible to distinguish from a procedure’s interface if the procedure is applicable to signal objects, variable objects of both.

The proposal takes some more concepts from previously discussed approaches like for example, class-wide types, polymorphism, abstract classes, or variant records. For these features we refer to the proposals where they have been originally introduced.

5.2.13 Object-orienting VHDL for component modelling

The proposal in [139] is an entity-based approach which adds an inheritance mechanism to entities. A class is an entity which can inherit its ports and generics to derived entities. The derived entities may add new ports and generics to the inherited ones. A detailed investigation of the mechanism is omitted here as similar inheritance concepts have been discussed in previous sections.

An architecture of the class may abstract from specific data structures in an implementation. A binding to an actual type in an instantiation is performed in a separate configuration. The abstraction of the data structure is used in the architecture. This appears to be an extension of the existing genericity concept of VHDL. The difference to the concepts discussed before is that the abstraction is not required to be modelled as part of the entity. For example, in an implementation of the bounded buffer gb_buf which was introduced in Section 5.2.9 it would be possible to model the attribute after_put to be of an abstract data structure. The configuration could define that it is bound to an actual type that is able to trace the history. The subprograms which trace and report the history and which use the attribute as a parameter would be statically polymorphic. The actual implementation of the subprograms in the implementation is selected depending on the binding of the data structure in the configuration.

As the data structure is not abstracted in the entity, using the configuration mechanism of VHDL breaks the encapsulation. (Compare the general remarks on the VHDL configuration mechanism in Section 5.1.4)

5.2.14 Structural VHDL and object-oriented principles

An entity-based approach which introduces an inheritance mechanism to extend the configuration capabilities of VHDL is presented in [54]. A class is modelled as an entity/architecture pair which can be modified and extended by inheritance in the proposal.

A structural description[45] of a system can be modelled in VHDL by components connected to each other via signals. The configuration mechanism of VHDL allows to bind entities and architectures to the components. Component instantiations which are bound to an entity are interpreted as object containers in the proposal. Binding a new entity to the component instantiation by modifying the configuration means to put a new object of a different class in the container. In that view static polymorphism is modelled by configurations. The proposal uses the term configuration based polymorphism. In the proposal the configuration is viewed as a mechanism to model incremental behaviour. The binding derives the behaviour of a component from the entity it is bound to. The has-a relation originally modelled by a component instantiation is re-interpreted as an is-a relation. The interconnection just models a delegation mechanism which delegates incoming events to the instantiation. To simplify the specification of such relations inheritance is introduced. Inheritance means to directly instantiate the structural description of a parent to a child without the additional abstraction boundaries of the components in between. Thus, the inheritance mechanism extends the configuration mechanism for modelling structural VHDL models.

The inheritance mechanism is not reduced to a strong delegation mechanism. Its allowed to modify inherited parts of a structural description by overwriting[46] them. Overwriting is not restricted to behaviour, that means it does not preserve the parent’s structure. The consequences on compatibility issues have been discussed.

We conclude the section mentioning a curio in the syntax of the proposal. The approach uses the keyword tagged which elsewise is used in type based approaches, to mark entities, architectures, and components as potential parents for inheritance.

45. Structural description is a description which lists elements of a system and their interconnection. It must not be confused with the term structure used in the context of objects and classes.

46. The exact meaning of overwriting is not quite clear here. Presumably it means in difference to overloading that the overwritten part is not in the derived class.

5.2.15 Résumé

The analysis of the proposals on extensions to VHDL consolidates the considerations on object-oriented hardware design methodologies. It extracts some general issues which have to be carefully considered when extending VHDL.

It is a major issue to integrate object-oriented extensions to the synchronisation and concurrency model of VHDL. Especially, modelling of condition synchronisation is an open question in the proposals. Adapting concepts from existing system level specification languages emerged to be a difficult problem. It turns out that compatibility considerations which are crucial for re-use concepts and maintenance strategies are not appropriately considered by most of the proposals. Additionally to language extensions corresponding modelling methodologies are required which take advantage of the new language capabilities.

The survey has shown two philosophies to extend the language. One philosophy is to define the language extensions together with a translation mechanism which translates the extensions to VHDL to achieve an easy integration of the new language into existing VHDL-based design flows. The other philosophy is to design the extended language with no considerations for such a translation. The latter approach allows more wide and powerful extensions. In either case the goal is to preserve an upward compatibility of VHDL to the language extension which allows the integration of existing VHDL models into the new design approach.

From these conclusions it seems reasonable to add yet another proposal for an object-oriented extension to VHDL. The proposal comes as part of a modelling methodology. Issues of synchronisation and concurrency are studied in detail. Re-use aspects are considered by introducing modelling guidelines for compatibility which are based on the new language extension.

With respect to the integration of the extension into existing design flows a translation concept is part of the proposal. It is a good compromise between wide extensions and manageable complexity of the translation. More generally, the proposal is an approach to achieve powerful modelling improvements with only minimal language extensions.

Chapter 6

Object-Oriented Extension to VHDL

We now come to the proposal of this thesis how to extend VHDL by object-oriented features. The basic characteristics which make a language or methodology object-oriented have been discussed in Section 3.2. Especially, they include abstraction, encapsulation, inheritance, and polymorphism. Opposite to some of the earlier mentioned language extensions these features are all covered by the proposal we are going to present in this chapter.

We have introduced the need to integrate a system-specification language for hardware systems into existing design flows. The integration is done in the proposal through a translation concept which we present in this chapter.

The language extension is a typed based approach. It uses tagged types to extend the existing type concept of VHDL. It can be viewed in this respects as a predecessor of SUAVE. At the same time the approach of the thesis appears to be a variant of the class types of Objective VHDL in many aspects although it uses a different syntax.

To take advantage of the object-oriented paradigm an appropriate modelling methodology has to complement the object-oriented language. Thus, the proposal for the language extension comes with a modelling style which could be used as part of an object-oriented hardware design methodology. As we shall see in the following chapters, the language extension and the modelling style are well suited to each other.

6.1 Overview

Before we present an elaborate description of the language extension we want to give a short overview. It covers in a rough sketch the main concepts and features of the extension. The overview is intended for the reader to get a notion of models that are described in the language extension to VHDL.

The basic feature that is added to the language is the concept of derived types. A simple form of a derived type is a new type that is a kind of copy of an existing type.

```
type existing_type is tagged record …
procedure operation (parameter : existing_type);

type derived_type is new existing_type …
-- inherits the operations that characterize the existing type:
-- procedure operation (parameter : derived_type);
```

The features of the types including operations which characterize the types are almost identical except the logical distinction that they belong to different types. The derived type inherits the features of the existing type. Especially, it inherits operations which characterize the existing type. The more general form of deriving a type does not only copy a type but extends it in some way. The type which is suited for such an extension is a record type where we can consider extension as adding new record elements.

```
type extended_type is new existing_type with record
    extension: a_type;
end record;
```

The other kind of extension is the addition of new operations which characterize the type or it is the modification of inherited operations.

```
procedure new_operation (parameter: derived_type);
-- new operation which characterizes the derived type
procedure operation (parameter: derived_type);
-- re-defined operation modifies an inherited operation
```

The inherited operations of a derived type behave almost identical to the operation of the existing type. By re-defining the operations it is possible to modify the behaviour.

Using derived types for modelling classes, we are able to model a class hierarchy. To make the class concept object-oriented the language extension introduces dynamic polymorphism to the type concept. Class-wide types establish a means for modelling polymorphic objects.

```
variable polymorphic_object : exisiting_type'Class;
```

The attribute 'Class of the language extension that decorates the type name designates a class-wide type. The polymorphic object that has a class-wide type allows late binding of operations.

```
operation (parameter => polymorphic_object);
```

The implementation of the operation to be executed is determined during run-time and depends on the actual type of the polymorphic object.

The language extension improves the encapsulation concept of VHDL types. It is possible to model abstract data types that make their internal structure only accessible via their operations.

```
type adt is tagged type
private
    private_element : a_type;
end record;
procedure operation_of_adt ( parameter : inout adt);
-- private_element only accessible via operation_of_adt
```

The last feature of the language extension to VHDL we want to mention in this brief overview is the extension of the linear elaboration model of VHDL.

During the modelling of derived types there may occur circular dependencies between packages containing different derived types which cannot be elaborated by the standard VHDL elaboration model.

```
use work.Q.derived_type;
package P is …

use work.P.another_derived_type; -- illegal in VHDL
package Q is …
```

The extended model allows to postpone the elaboration of certain declarations to break these dependencies.

```
use work.Q.derived_type;
package P is …

postponed use work.P.another_derived_type;
package Q is …
```

After this outline of the main language features we hope that the reader has a first impression of the extensions which helps to understand the complete language extension with all its details that are presented in the following sections.

6.2 Type Extension

In the previous chapter we discussed two approaches (compare Section 5.2.9 and Section 5.2.10) which extend the type concept of VHDL by introducing tagged types. We saw how it was possible to derive new types from an existing type. The structure and the primitive operations were inherited by a derived type. It was possible to extend the properties of a derived type by adding new elements to a tagged type, by adding new primitive operations, and by overriding existing primitive operations.

We now present a similar approach which introduces tagged types as a language extension to VHDL.

6.2.1 Tagged types

A tagged type is a special kind of record type which allows type extension. It can be defined in a type declaration of a package declaration. Like any type in VHDL a tagged type is characterized by a set of values and a set of operations namely the operations of explicitly declared subprograms that have a parameter or result of the type, the predefined operators, and the basic operations. Additionally a tagged type is characterized by a set of primitive operations. That means, the primitive operations are the operations which belong to a type. The primitive operations of a tagged type include its predefined operations, its basic operations[1] and procedures[2] which have a parameter of mode in or inout of the tagged type. Primitive operations are declared together with the tagged type in the same package declaration. To distinguish the procedures from the basic operations and the predefined operations we refer to them as user-defined primitive subprograms. With this definition of primitive operations we can state that the properties of a tagged type are characterized by a set of values and primitive operations. This makes a tagged type appropriate for modelling a class. The record elements model the structure of the class and the primitive operations model its behaviour.

The separation of package declaration and body abstracts the behaviour of a tagged type in the package declaration and encapsulates it in the body. It is only allowed to have one tagged type declaration per package declaration

1. Compare Section 3 of the LRM.
2. Although it would be possible to extend the definition of user-defined primitive subprograms to functions it is restricted to procedures here, mainly, to keep the translation mechanism from the language extension to VHDL simple.

in order to have a clearly laid out model structure and to keep the translation from the extension to standard VHDL simple.

For example, we could use a tagged type to model the bounded buffer as a class.

```
package bounded_buffer_package is
    type bounded_buffer_array is array(positive range <>) of item_type;
    type bbuffer is tagged record
        buf : bounded_buffer_array (1 to buffersize);
        buf_in : integer;
        buf_out : integer;
    end record;
    procedure get (object : inout bbuffer; item : out item_type);
    procedure put (object : inout bbuffer; item : in item_type);
end bounded_buffer_package;
```

From the example we can see that a new reserved word tagged is introduced into the language. It indicates that the record is a tagged type.

The record elements model the structure of the class. The subprograms get and put are primitive operations of the tagged type because they have a parameter of mode inout of the tagged type and they are declared in the package declaration. The basic operations of a record type like for example an assignment to bbuffer or an aggregate of bbuffer are also primitive operations of the tagged type. The primitive operations model the behaviour of the bounded buffer, especially, the subprograms put and get. They perform the access to an object of the tagged type. The objects which can be accessed via a put or get operation are variables due to the interface list of the subprograms. The other primitive operations of bbuffer, i.e., its basic operations, may access all classes[3] of objects, i.e., constants, variables, and signals.

6.2.2 Inheritance

The interesting thing about tagged types is that they introduce a possibility to derive a new type from an existing one. To derive a new type we can write:

```
use work.bounded_buffer_package.all;
package bounded_buffer_package_derived is
    type derived_bbuffer is new bbuffer with null record;
end bounded_buffer_package_derived
```

3. The term class is used here in the sense of the LRM.

The type derived_bbuffer is said to be a derived type and bbuffer is the parent type of derived_bbuffer. We also refer to the derived type as child. The derived type is declared in a new package because the language extension does not allow to derive from a type in the same package declaration as it is declared[4]. Generally, two tagged types are required to be declared in different package declarations.

The concept allows a derived type to become parent of another derived type. The parent-child relations between the types establish a tree of types with a root that is the ancestor of all the types in the tree.

A derived type is a new type which is distinct from its parent type[5]. The derived type is similar in many respects to the parent type. It is also a tagged type, i.e., a record and it has the same record elements with the same identifiers and the same subtype indications. If we think of the type as a set of values then the values of derived_bbuffer are a copy of the values of bbuffer.

Although we can view at the new values as copies they belong to a new type. That means, values of the parent type cannot be assigned to objects of the derived type and vice versa. Due to the similarities of the types a derived type is considered a closely related type to its parent type and more generally to any of its ancestors. A corresponding type conversion provides for explicit conversion between the closely related types. In this conversion only the parent type is allowed as a target type. More generally, only conversions towards the root of a tree of tagged types are allowed. The operand of the conversion is an expression of the derived type. The result is a value of the parent type. To convert between objects of derived_bbuffer and bbuffer we could write:

```
variable bbuffer_object : bbuffer;
variable derived_bbuffer_object : derived_bbuffer;
…
bbuffer_object := bbuffer(derived_bbuffer_object);
```

However, we cannot write

```
derived_bbuffer_object := derived_bbuffer(bbuffer_object);
```

The derived type derived_bbuffer must not appear as a target type of a type conversion with the parent type as operand. Please note, it is also not possible to write

4. This rule is introduced into the language extension to limit the complexity of the translation mechanism to VHDL. (See Section 6.5.1)
5. Please note, this is different to a subtype declaration which does not define a new type.

```
    get (bbuffer(derived_bbuffer_object),item);
```

After applying the type conversion[6] in the actual part the value of the actual parameter belongs to the subtype bbuffer denoted by the subtype indication of the formal as it is required for variable parameters of mode inout (Compare LRM 2.1.1.1.). However, the value of the formal parameter object does not belong to the subtype indication derived_bbuffer of the actual parameter derived_bbuffer_object. A formal part in form of a type conversion for explicit conversion from bbuffer to derived_bbuffer is not possible because the derived type is not allowed as a target type in such an explicit conversion. In case of signal parameters a type conversion is not allowed at all (Compare LRM 2.1.1.2). Providentially, such an explicit conversion is not required to make primitive operations which characterize the parent type available for the derived type.

The derived type inherits primitive operations of its parent. That means, that for each user-defined primitive subprogram there exists a corresponding inherited primitive subprogram of the derived type. The inherited primitive subprogram is also considered to be a user-defined subprogram. Thus, we can extend the list of primitive operations of a tagged type which was mentioned above by the inherited primitive subprograms. It has the same name[7] as the corresponding primitive subprogram of the parent. The formal parameter list of the inherited primitive subprogram is obtained from the formal parameter list of the parent's subprogram after replacement of each[8] subtype indication of the parent type with a corresponding subtype indication of the derived type. Corresponding subtype indication means that the type mark in the subtype indication denotes the derived type. The resulting subprogram is implicitly declared immediately after the declaration of the derived type which inherits the subprogram. Default expressions for parameters of the parent type are not allowed in the user-defined primitive operation. This

6. This is different from the view conversion approach of SUAVE discussed in Section 5.2.10. In SUAVE such a call would pass the object derived_buffer_object after a view conversion as an actual parameter to the subprogram because the conversion appears as an actual parameter of mode inout [6].

7. The technical term in the LRM is designator.

8. Please note, although the mode of a parameter is used to define user-defined primitive operations the replacement is independent from the mode declared in the subtype indication. A subtype indication of an interface element of the parent type and mode out is also replaced in a derived formal parameter list.

avoids a type mismatch after the replacement of the subtype indication for the parameters of the parent type[9].

If we apply the inheritance mechanism to the example of the user-defined primitive operation get of bbuffer we obtain the inherited primitive subprogram get of the derived_bbuffer.

```
procedure get (object : inout derived_bbuffer; item : out item_type);
```

The subtype indication bbuffer of the interface variable declaration of the parameter object is replaced by the corresponding subtype indication derived_bbuffer. The primitive subprogram get of the tagged type derived_bbuffer is implicitly defined after the declaration of derived_bbuffer in the package bounded_buffer_package_derived. Now any explicit type conversions in the example are obsolete and we can directly call the primitive operation get with an actual parameter derived_bbuffer_object.

```
get (derived_bbuffer_object,item);
```

We call the actual parameters in such a call to the primitive operation of a tagged type whose corresponding formal parameters are of the tagged type controlling operands[10]. In the example, derived_bbuffer_object is the controlling operand with the corresponding formal parameter object of type derived_bbuffer. To keep things simple, the language extension requires that all controlling operands of a call are of the same type.

As the user-defined primitive operation of a parent type and the inherited primitive operation of a derived type have the same subprogram designator but different parameter type profiles–remember, a derived type is distinct from the parent type it is not a subtype–both primitive operation overload each other (Compare LRM 2.3).

A call on an inherited subprogram is executed as if it were a call to the corresponding subprogram of the parent in combination with some type conversions during its execution to avoid type mismatches between parameters of the derived type and objects of the parent type. Basically, each call on an

9. From a language philosophers' point of view it would be possible to introduce a special kind of implicit conversion from the parent type to the derived type which is implicitly applied to the default expressions in an inherited primitive subprogram. The type conversion would handle the type mismatch. Such an approach was omitted in the language extension because it would not be possible to implement it in the translation concept (See Section 6.5.1).

10. Please note that the definition of controlling operands is independent from the mode of its corresponding formal.

operation with operands of the derived type causes an implicit type conversion from the derived type to the parent type except from calls with controlling operands which are only of the derived type[11].

According to the principle that all controlling operands in a call must be of the same type, each time during the execution of the subprogram when a primitive operation is called to which controlling operands of both types, the parent type and the derived type, are passed as operands, an implicit type conversion from the derived type to the parent type is performed on the operands of the derived type. Likewise any call to a subprogram or basic operation with one or more formal parameters or operands of the parent type and actuals of the derived type which are not controlling operands causes an implicit type conversion during the execution of the inherited subprogram. In such cases for each implicit type conversion an object of the parent type which has no name is implicitly declared in the calling subprogram. Similar to types without a name we call it an anonymous object. It is the target of the implicit type conversion. The anonymous object is composed of record elements from the operand of the implicit type conversion. It has just those named elements which are declared in the parent type. The anonymous object is a signal, a constant, or a variable according to the operand of the implicit type conversion. The anonymous object as result of the implicit type conversion is passed as controlling operand to the primitive operation. Please note that such an implicit type conversion from the derived type to the parent type latently also contains the conversion vice versa. To better understand the mechanism of the new case of implicit type conversion we could think of it as a kind of slice for tagged types.

We want to illustrate this by an example. The bounded buffer bbuffer may have a primitive operation to initialize its values. The procedure body of such an operation may look as follows:

```
procedure initialize (object : inout bbuffer) is
begin
    object := bbuffer'( buf => (others => item_type'left),
                        buf_in => 0,
                        buf_out => 0);
end;
```

11. We also could say that calls in an inherited subprogram are statically bound except from calls with all controlling operands of the derived type.

Assume, the type derived_bbuffer is derived from bbuffer and inherits its primitive operation initialize. The inherited operation has a new formal parameter list:

```
procedure initialize (object : inout derived_bbuffer);
```

The inherited operation overloads the parent's operation. We can call the inherited operation with an actual of type derived_bbuffer. The call is executed as if the parent's operation was called. That means that the assignment statement is executed. The assignment statement is a primitive operation according to the definition given above. It is called with an operand of type bbuffer, which is the expression of the assignment, and an operand of type derived_bbuffer which is the target of the assignment. As the assignment is called with different types an implicit type conversion is performed on the target. The resulting anonymous object which is actually used as the target operand of the assignment has the type bbuffer. The object contains the record elements buf, buf_in, and buf_out of the operand of the type conversion which is the actual parameter object. The type conversion of the aggregate of type bbuffer to the type derived_bbuffer in the assignment is inherent in the implicit type conversion from the derived type to the parent type of the target of the assignment.

To illustrate this, we give a pseudo code description of the anonymous object that is used as a target in the assignment statement.

```
procedure initialize (object : inout bbuffer) is
    alias anomymous_object : bbuffer is (  object.buf,
                                           object.buf_in,
                                           object.buf_out);
begin
    anonymous_object := bbuffer'(  buf => (others => item_type'left),
                                   buf_in => 0,
                                   buf_out => 0);
end;
```

So far we have looked at the possibility of tagged types to derive a new type which has the same properties as the parent type. Very often it is useful to introduce a new type which has additional or slightly modified properties. We present some more advanced features of tagged types which allow to extend an existing type by deriving a new one from it. This new form of derivation makes it possible to add new elements and to add new primitive operations to a tagged type. It also allows to modify the behaviour of an existing type by replacing inherited user-defined primitive operations.

6.2.3 Modification of behaviour

We now introduce the concepts to extend and modify behaviour of derived types. We will start with considering how we can add new primitive operations to a derived type and how they may interact with inherited subprograms.

Consider an example where we want to derive a new bounded buffer xbuf_2 from bbuffer. We have already introduced the buffer in previous examples. It has two additional operations get2 which reads two items from the buffer and empty which indicates if the buffer contains any items. The language extension takes such a modelling situation into account by the possibility to add new primitive operations to a derived type. In the example we perform the extension in two steps. In a first step we derive a type buf2 which adds the new operation get2 to the type.

```
type buf2 is new bbuffer with null record;
procedure get2 (object : inout buf2; items : out two_items_type);
```

The procedure get2 is a primitive operation of buf2. If we do not consider any synchronisation issues for a moment we could implement the operation by calling the inherited get operation.

```
procedure get2 (object : inout buf2; items : out two_items_type) is
begin
    get(object, items(1));
    get(object, items(2));
end;
```

In a second step, we derive the new type x_buf2 from buf2. It adds the new primitive operation empty to the derived type. It inherits the primitive operations from its parent, that means, it inherits the operation get2. It also inherits the implicitly declared operations get and put of buf2.

```
type x_buf2 is new buf2 with null record;
procedure empty (object : inout x_buf2; return_val : out boolean);
```

If we call get2 with an actual parameter of type x_buf2 the inherited operation get of type x_buf2 is called without any implicit type conversion in the implementation of get2[12]. The controlling operand of the call on get which is

12. We can note that this is different to the example in SUAVE where it is necessary to perform an explicit type conversion to achieve dynamic binding by re-dispatching.

the parameter object of type x_buf2 determines which of the overloaded primitive operations is actually called. Therefore it is not the operation of buf2 which is called in get2 of x_buf2. In other words, the call in an inherited subprogram on another user-defined subprogram with controlling operands as actual parameters is not statically bound it is rather dynamically bound to the primitive operation of the derived type. Dynamically means, the subprogram declaration and its corresponding body which is invoked by the call is determined dependent on the controlling operands during the elaboration of the call.

In the example it would make no difference which version of get is invoked as both, the inherited operation of buf2 and the inherited operation of x_buf2 finally perform a call on the get operation of bbuffer. This is different if we replace an inherited get by a new version of get in a derived type. An inherited user-defined operation can be replaced by overriding the operation. Overriding means to declare a homograph of the inherited operation in the package declaration where the derived type is declared and the subprogram specification of the homograph conforms the implicit declaration of the inherited subprogram. The explicitly declared homograph then hides the implicitly declared inherited subprogram, that is, it overrides the inherited subprogram. The explicitly declared subprogram is directly visible and the implicitly declared one is hidden. Such a hidden subprogram will not become visible neither by selection nor directly. To overcome this limitation a predefined attribute 'Parent of the language extension provides a possibility to execute the behaviour implemented in the overridden subprogram if the overridden subprogram is called from within another subprogram of the same tagged type. The execution of the subprogram body of the parent is invoked by a subprogram call which decorates the subprogram name with the attribute 'Parent. The subprogram call executes as if it were a call to the corresponding subprogram of the parent. The subprogram dynamically modifies the interface list during dynamic elaboration by replacing each subtype indication of the parent type by the subtype indication of the controlling operands. It adapts the original interface list of the parent to its use in a child. Type conversions during the execution of the parent's subprogram which may become necessary as a consequence of the dynamic adaptation of the interface list are performed like the other implicit type conversions in inherited subprograms. Each call on an operation with actual parameters of a derived type and formals of the parent type causes an implicit type conversion from the derived type to the parent type except from calls with controlling operands which are only of the derived type.

The call on the parent’s subprogram is statically bound. The corresponding subprogram can be determined during the analysis of the model. It does not change during the execution of the model. A derived type may inherit a subprogram which contains a call decorated with the attribute 'Parent. The call still executes the subprogram which was determined at analysis time.

We want to illustrate this by an example. If we look at the bounded buffer x_buf2 we could override the inherited get operation.

```
use work.bounded_buffer_package_buf2.all;
package bounded_buffer_package_buf2_x_buf2 is
    type x_buf2 is new buf2 with null record;
    procedure empty (object : inout buf2; return_val : out boolean);
    procedure get (object : inout x_buf2; item : out item_type);
end bounded_buffer_package_buf2_x_buf2;
```

Type x_buf2 inherits the primitive operation get which is implicitly declared immediately after the declaration of the derived type x_buf2. The explicit declaration of the primitive operation get conforms to the implicit declaration of get—remember, the formal parameter lists are adapted for inherited subprograms—and thus overrides the implicit declaration. The inherited version of get is hidden. If we use the package in another design unit we can potentially make the primitive operation get directly visible by a use clause. If we call get the new version of get is executed. To execute the behaviour of the overridden subprogram we have to use the attribute 'Parent.

```
use bounded_buffer_package_buf2_x_buf2.all;
…
variable x_buf2_object : x_buf2;
…
get(x_buf2_object, item);           -- call to new version of get
get'Parent(x_buf2_object, item);-- call to the parent's version
```

The decorated name get'Parent statically denotes the parent’s version of get. which actually is the subprogram implementation of buf2.

In an inherited subprogram the default mechanism for calling another inherited subprogram with all controlling operands of the same type dynamically binds the call to the primitive subprogram. To statically bind the call the language extension provides the predefined attribute 'Static. Any procedure call of a user-defined primitive subprogram that is marked by the attribute is referred to as a static call. Such a static call may occur in any user-defined primitive subprogram of a tagged type. The static call is executed by performing an implicit type conversion of the controlling operands to the tagged type whose primitive subprogram originally performed the static call. According to the implicit type conversion the static call invokes the user-defined subprogram of the tagged type. The consequences of the implicit type conversion is that if a derived type inherits the user-defined subprogram which contains the static call it executes the original subprogram body of the parent even if the subprogram which is statically called is overridden in the derived type. To illustrate this mechanism we look at the example of the bounded buffer buf2. It would have been possible to implement the primitive operation get2 by statically calling the primitive operation get.

```
procedure get2 (object : inout buf2; items : out two_items_type) is
begin
    get(object, items(1))'Static;
    get(object, items(2))'Static;
end;
```

To indicate the static call the attribute decorates the procedure calls of get. The tagged type x_buf2 inherits the primitive operation get2 and overrides the operation get. However, if we execute get2 of x_buf2 it is the operation get of the parent which is called after an implicit type conversion to the type buf2 of the controlling operand object. It is not the new version of get which is executed.

Independent from any modifications of get by overriding in derived types inherited versions of get2 will preserve their behaviour. In other words, a static call is a means to make calls to primitive subprograms within other subprograms safe against unintended modifications in derived types.

This is an important issue in compatibility considerations on derived types because there is no abstraction of dynamically bound calls in a subprogram specification of a primitive operation. As a consequence there is no possibility to preserve compatibility in derived types without breaking the encapsulation of inherited behaviour if behaviour is overridden and dynamic calls are arbitrarily used in operation implementations. Generally we can state that it is a good modelling practice to avoid dynamic bindings and to use them only if it is possible to give good reasons for dynamic bindings.

An attribute 'Dynamic is in the language extension as a counterpart to 'Static to keep it symmetric. It can be associated to subprogram calls to primitive operations from within other user-defined subprograms where the language extension provides dynamic calls. The attribute then explicitly indicates that it is a dynamic call. Basically, it uses the tagged type of the subprogram which contains the dynamic call as controlling operands of the call. As described above, this is the default behaviour of calls to primitive operations from within other operations if the controlling operands in an inherited operation of the derived type are of the derived type. So, both calls are equivalent:

```
procedure get2 (object : inout buf2; items : out two_items_type) is
begin
    get(object, items(1));
    get(object, items(2))'Dynamic;
end;
```

Please note, although the attributes 'Parent and 'Static have similar effects by circumventing the execution of overridden operations they are different concepts. Both perform a static binding of calls to primitive operations from within other user-defined subprograms but while the attribute 'Static performs an implicit type conversion of the controlling operands the attribute 'Parent does not. To combine the concepts of both attributes, it is perfectly legal to combine them. In contrast to the attribute 'Static it is not possible to combine the attribute 'Dynamic with the attribute 'Parent. The dynamic binding of 'Dynamic would contradict the static concept of 'Parent.

For example, we could statically call the parent's operation in the procedure get2 of buf2. A static call to the parent's implementation of get would so to say freeze the controlling operands to be of type buf2 in any derived type. The call is executed as if it were a call to the corresponding subprogram of bbuffer.

```
procedure get2 (object : inout buf2; items : out two_items_type) is
begin
    get'Parent(object, items(1))'Static;
    get'Parent(object, items(2))'Static;
end;
```

Any overriding of get in a derived type does not have any effect on the execution of inherited versions of get2 and thus makes it safe from a re-use point of view.

We conclude this section by observing that modification of behaviour is mainly an issue of appropriate type conversions between derived types and binding mechanisms of calls to primitive operations. We can turn down the concepts of the type extension we are going to discuss in the next sections to very similar issues of conversion of types and binding of calls.

To complete the discussion about calling primitive operations from within other primitive operations, about static and dynamic binding, and about type conversions of parameters we compare the concepts of this thesis with the SUAVE approach. With the Ada-like concept the SUAVE approach performs a static binding of calls by default. No distinction between different calling situations are required. View conversions of the parameters are generally performed during the dynamic elaboration of the call. Nevertheless, dynamic binding can be achieved where required in an inherited subprogram by re-dispatching the call. Calling a parent's subprogram is simply achieved by an explicit view conversion of the actual parameter. That means, the SUAVE approach provides the same modelling capabilities as the approach of this thesis. This raises the question why the presented concepts do not adapt such a simpler and more elegant Ada-like modelling approach. The answer is that the Ada-like concept would cause an enormous complexity in the translation mechanism from the language extension to VHDL while the more complex language concepts of this thesis keep the translation simple[13].

6.2.4 Extension of structure

In the previous section we saw how it was possible to extend and modify the behaviour of tagged types by adding new primitive operations to the tagged type and by overriding existing operations. We now introduce a derivation where it is possible to extend the structure of a tagged type by adding additional record elements to a derived type.

Consider for example the bounded buffer gb_buf which was introduced in the previous chapter. It is a child of bbuffer which has an additional primitive operation gget. The new operation removes an item from the buffer like the get operation. The difference between the get operation and gget operation is that a gget operation only can be executed immediately after a get or a gget operation. This is a synchronisation constraint which requires the tracing of the operations' invocation history. To store the history the original structure of the type bbuffer is extended by an additional record element. The keyword with in the type declaration indicates that it is a derived tagged type which may have some additional record elements following the with.

```
type gb_buf is new bbuffer with record
    after_put : boolean;
end record;
procedure gget(object : inout gb_buf; item : out item_type);
```

13. Remember, SUAVE does not provide a translation mechanism to VHDL.

The bounded buffer gb_buf has the record elements buf, buf_in, and buf_out which are inherited from bbuffer and the newly added record element after_put. The buffer inherits the primitive subprograms put and get and adds the operation gget as a new primitive operation. The primitive operations have access to any of the record elements[14] so that the new subprogram gget can read the new record element after_put to perform its condition synchronisation and that gget can read and update the inherited record elements to perform its actual behaviour.

It is possible to add new record elements to the derived type but it is not allowed to remove or replace inherited record elements. As we know from Section 3.2, removing or replacing elements would cause non-solvable problems in any kind of compatibility considerations between derived types. Similarly, a name that denotes a record element of the parent must not be used as a name of a record element that is added in a derived type.

Like for any type declaration the scope of a type declaration for a tagged record type starts after the end of the declaration (Compare LRM 10.3). That means, the name in the declaration that denotes the tagged type is not visible in the declarative region that is associated with the record type. It must not be used as a subtype indication in an element declaration of the tagged record. In other words, it is not possible to model a tagged record that recursively contains a record element that is of the tagged record. However, it is possible that a derived type contains an element that is of its parent type. It is possible to have a has a relation and an is-a relation between two types at the same time.

For example, it would be possible for a derived type of bbuffer to contain an element that is of bbuffer:

```
type backup_buf is new bbuffer with record
    -- backup : backup_buf; -- illegal
    backup: bbuffer;
end record;
```

From the examples it can be seen that the previously introduced concept to derive tagged types is only a special case of the more general mechanism. The notation

14. Actually, any operation on an object of the derived type has access to any record element as there is no special encapsulation mechanism provided with a tagged type. We shall discuss possibilities to achieve such an encapsulation later on (Compare Section 6.3.4).

```
type child is new parent with null record;
```

is simply a short form of

```
type child is new parent with record
    null;
end record;
```

which indicates that an empty list of record elements is added to the derived types. All the concepts that have been introduced in the previous sections can be viewed as general concepts which are applied to the special case where only an empty element list is added to a derived tagged type. A tree of tagged types consists of types with each type containing all the record elements of its ancestors and some additional own elements. With this view on a tree of tagged types it becomes clear why type conversions are only allowed towards the root. A type conversion towards the root simply ignores the additional record elements of the derived types. If type conversions were allowed in the other direction it would not be clear what are the values of the additional elements. In case of an implicit type conversion as part of the elaboration of a subprogram call the additional elements of the derived types are effectively ignored in the association of the actual with the corresponding formal.

At this point we want to consider the effects of the implicit type conversion in more detail. We may have a look at an implicit type conversion in a static call on the operation get of the buffer gb_buf. This call could be part of the implementation of the gget operation.

```
procedure gget(object : inout gb_buf; item : out item_type) is
begin
    if object.after_put = false then
        get'Parent (object => object, item => item)'Static;
        -- implicit type conversion of object
    else
        -- Synchronisation constraint not met
        …
    end if;
end;
```

Passing the parameter object of gget as an actual parameter in the static call to the parent’s implementation of get causes an implicit type conversion from the type gb_buf to the type bbuffer. The conversion ignores the record element after_put of the derived type gb_buf in the association of the formal parameter object with the actual parameter object. Thus, the call is equivalent to an explicit call of get with the element after_put omitted in the association list of the call:

```
procedure gget(object : inout gb_buf; item : out item_type) is
begin
    if object.after_put = false then
        work.bounded_buffer_package.get (
                                object.buf => object.buf,
                                object.buf_in => object.buf_in,
                                object.buf_out => object.buf_out,
                                item => item);
    else
        -- Synchronisation constraint not met
        …
    end if;
end;
```

From the equivalent explicit call it is obvious how the implicit type conversion latently contains the conversion vice versa, i.e., from parent type to derived type. The record elements of the derived type which are not part of the parent are simply not modified by executing the statically bound subprogram.

While the example is based on variables as parameters the principle of the implicit type conversions works for signals too[15]. Each basic signal that is not a sub-element of the parent is not affected by the execution of the static call. Vice versa, any events on these sub-elements during the execution of the call are not observable inside the called subprogram. Consider for example a version of get and gget which uses signals as parameters. A wait statement inside the get operation to model synchronisation which is sensitive to the events on the parameter object would not resume a suspended process if an event occurs on the sub-element after_put of the signal object in the procedure gget.

6.2.5 The tag

So far, the presented concepts allow to determine the type of objects at analysis or elaboration time. As was discussed in Section 3.2 polymorphism is a means for flexibility and re-usability of models. The idea behind polymor-

15. There is only a slight difference to variables concerning the so called tags which is explained below.

phic objects is that they can change their properties during simulation. To introduce a concept for polymorphism on the basis of tagged types it is useful to be able to determine the type of an object at run-time. A tagged type has an implicitly declared record element which stores this type information during run-time. The record element is called the tag following Pascal which uses the term tag field to denote the actual variant of a variant record [99]. The tag of a tagged type is an anonymous element which is hidden, that means, it is not directly visible. The implicitly declared record element has the predefined type tag which is an implementation defined type. It is declared in a package object_oriented_extension.

```
-- library STD; use STD.STANDARD.all;
package OBJECT_ORIENTED_EXTENSION is
    -- predefined type tag:
    type TAG is implementation_defined;
    -- The predefined operator for this type are as follows
    -- function "="      (anonymous,anonymous: TAG) return boolean;
    -- function "/="     (anonymous,anonymous: TAG) return boolean;
end OBJECT_ORIENTED_EXTENSION;
```

The operators = and /= are given in comments. They are predefined operators for the type tag in the language extension and thus implicitly declared in the package. They are defined as one would expect. The equality operator returns the value TRUE if the operands are equal and thus denote the same tagged type. Otherwise it returns the value FALSE. The result of the inequality operator is obtained by applying the logical not operator on the result of the equality operator.

In the object-oriented language extension to VHDL the package OBJECT_ORIENTED_EXTENSION resides in the library STD beside the packages STANDARD and TEXTIO. To automatically make the type TAG visible in every design unit of a model and thus allow the use of tagged types it is assumed in the language extension that every design unit except the packages OBJECT_ORIENTED_EXTENSION and STANDARD implicitly contain the following context item as part of its context clause:

```
use STD.OBJECT_ORIENTED_EXTENSION.all;
```

To keep things simple it is not allowed to overload the predefined type TAG.

With the package OBJECT_ORIENTED_EXTENSION it is possible to declare constants, variables, or signals of type tag which can be used to explicitly store the tag information of tagged types. To read the tag information of a tagged type a predefined attribute 'Tag of the language extension can be used. It denotes the tag of a tagged type. The value of the attribute is of type tag.

For example, it would be possible to test that bbuffer and gb_buf are different tagged types during simulation:

```
constant tag_of_gb_buf : tag := gb_buf'Tag;
constant tag_of_bbuffer : tag := bbuffer'Tag;
…
assert tag_of_gb_buf /= tag_of_bbuffer;
```

With the attribute 'Tag it is also possible to test the tag of a polymorphic object. This is especially interesting as polymorphic objects change their properties during simulation, that means in the context of tagged types they may change their tag. The detailed concept of polymorphic objects and the application of the attribute is explained later on.

We now want to investigate some properties of the tags more closely. The details allow to understand some of the more advanced features of the translation mechanism which is presented in a following section. The properties of the tags are especially used in the modelling of condition synchronisation using polymorphic objects. We touch on some language concepts which require a deeper understanding of the parameter passing mechanism and type conversion concept of VHDL. (Compare LRM 2.1.)

The tag of an object that is of a tagged type identifies the type. It is implicitly assigned to the object with the object’s declaration. VHDL does not allow objects to change their type after elaboration. The same rule applies for objects of tagged types in the language extension. That means, a tag of such an object does not change during simulation.

The tag of an object basically behaves like any other sub-element of the object. However, in assignment statements the tag of the target is not modified due to the fact that an object that is of a tagged type does not change its tag during simulation. In an operation call in which both, the actual and the formal parameter have the same tagged type the value of the tag of the actual is passed by copy or reference to the formal if the parameter is a variable or constant. If the parameter is a signal the reference[16] to the signal which models the tag is passed into the subprogram call. This works if the parameters are associated in whole.

16. Alternatively, the reference to the driver of a signal or both references can be passed into the subprogram call.

Passing parameters that are of a tagged type and which are associated individually functions different. For parameters of class constant or variable an implicit object of type tag which is a constant or variable is created during the dynamic elaboration which is involved in the subprogram call. The implicit object is assigned a value at the start of the subprogram call which indicates the tagged type of the parameter. The implicit object is associated as an actual either in form of an expression or a variable to the tag of the formal parameter. It is only available for use in the subprogram call. The scope of the object is just the statement of the subprogram call. A look at the example of the alternative implementation of a call to get in the implementation of gget which has been presented in the previous section may illustrate the mechanism:

```
procedure gget(object : inout gb_buf; item : out item_type) is
begin
    …
    work.bounded_buffer_package.get (
                                object.buf => object.buf,
                                object.buf_in => object.buf_in,
                                object.buf_out => object.buf_out,
                                item => item);
    …
end;
```

The parameter object of the procedure get is associated individually in the call. The hidden record element tag of the formal parameter is associated to an implicitly created variable of type tag. The variable is initialized with a value which indicates the tagged type bbuffer. Its scope is the procedure call statement.

```
-- implicit variable:
-- variable anonymous : tag := bbuffer'Tag;
work.bounded_buffer_package.get (
                        --  object.tag => anonymous,
                            object.buf => object.buf,
                            object.buf_in => object.buf_in,
                            object.buf_out => object.buf_out,
                            item => item);
-- end of declarative region containing declaration for anonymous
```

After the call the anonymous variable is not any longer required and not available. The whole mechanism is very similar to the implicit declaration of a loop parameter in a loop statement.

Another mechanism is used for passing parameters of class signal that are of a tagged type and that are associated individually. Creating an implicit object during the dynamic elaboration of the call like in the case of variable and constant parameters and associating it with the tag of the formal parameter would not fit in the language philosophy of VHDL which does not support dynamic creation and destruction of signals during simulation. An effect of creating and initializing of signals would be that a signal changes its current value within a delta cycle in the simulation.

The concept to associate signals of tagged types individually is to restrict the association. For such signals a formal parameter that is associated individually is only allowed if all the associated actuals are sub-elements of a signal that has a tagged type which is closely related to the tagged type of the formal parameter. Then the tag of the formal parameter is implicitly associated with the tag of the signal whose sub-elements are associated with the formal.

If the parameters in the example above were signals it would be allowed to associate them individually.

```
procedure gget(signal object : inout gb_buf; item : out item_type) is
begin
    …
    work.bounded_buffer_package.get (
                              --  object.tag => object.tag,
                                  object.buf => object.buf,
                                  object.buf_in => object.buf_in,
                                  object.buf_out => object.buf_out,
                                  item => item);
    …
end;
```

Each actual parameter is a sub-element of the signal object. The type of the signal gb_buf is closely related to the parent type bbuffer. The tags of the signals are associated implicitly.

It would not be allowed to associate the parameter individually if the actual parameters are not sub-elements of the same signal.

```
signal object1 : gb_buf;
signal object2: gb_buf;
signal object3 : integer;
```

```
…
get (   object.buf => object1.buf,
        object.buf_in => object2.buf_in;
        -- Error : actual is not sub-element of object1
        object.buf_out => object3,
        -- Error : actual is not sub-element of a closely related type
        item => item);
```

The statements on properties of tags which occur as operands in primitive operations can be specially applied to assignment statements. If the target of an assignment is a name that denotes an object that has a tagged type then the right-hand side value of such an assignment has the same type as its target, possibly after an implicit type conversion, otherwise the model is erroneous. Therefore, in the assignment operation which assigns each sub-element of the right-hand side value to the corresponding element of its target the right-hand side value of the tag has the same value as the tag of the target.

If the target in a variable assignment statement is an aggregate of a tagged type an implicitly created variable of type tag is used in the expression that is associated with the tag of the aggregate. After the assignment the anonymous variable is not any longer required and not available.

Consider a set of variables that is assigned the elements of a bounded buffer.

```
variable object : bbuffer := bbuffer'(…);
variable buf : bounded_buffer_array(…);
variable buf_in : integer;
variable buf_out : integer;
…
-- implicit variable:
-- variable anonymous : tag := bbuffer'Tag;
(--  tag => anonymous,
     buf => buf,
     buf_in => buf_in,
     buf_out => buf_out)        := object;
-- end of declarative region containing declaration for anonymous
```

The example illustrates how the elements of the object are assigned to the elements of the aggregate including the implicitly defined element tag.

A target of a signal assignment that is an aggregate of a tagged type is treated differently. All expressions in the element associations of the aggregate must be names that denote elements of the same signal. The tag of the aggregate is implicitly associated with the tag of the signal whose elements

are denoted by the expressions in the element association. As the tag of a signal must not change the signal is required to have the same tag as the right-hand side value in the assignment. Finally this means that an aggregate of a tagged type is only allowed as a target in a signal assignment where it could be replaced by a name denoting the signal whose elements are referenced in the aggregate.
This becomes clearer in an example:

```
signal target : bbuffer := bbuffer'(…);
signal rh_side : bbuffer := bbuffer'(…);
…
(-- tag => target.tag,
    buf => target.buf,
    buf_in => target.buf_in,
    buf_out => target.buf_out) <= rh_side;
```

The assignment containing the aggregate could be replaced by one containing a name as a target.

```
target <= rh_side;
```

The presented rules for using tags in association elements of aggregates and element associations in operation calls can be extended to port maps. To extend them the rules for passing signals as parameters that are of tagged types are applied accordingly to the mechanism of port maps.

As we have seen in Section 6.2.2 it is not possible to apply explicit type conversions between derived tagged types in an association list that associates a formal that has mode inout due to the missing conversion from the parent to the derived type. However, what is possible in an association list of a port map is a conversion function between types.

A conversion function may have a single parameter that is of the parent type and a return value of a derived type. As the return value is of the derived type the value which is copied into the tag of the associated parameter indicates the derived type.

To give an example, we could add a conversion function to derived_bbuffer from Section 6.2.2 so that a call of the parent's implementation of the operation get with an object that is of type derived_bbuffer is legal. The type derived_bbuffer was modelled as a derived type of bbuffer.

```
type derived_bbuffer is new bbuffer with null record;
```

A conversion function from the parent type bbuffer to the derived type derived_bbuffer can be modelled:

```
function to_derived_bbuffer (object : bbuffer ) return derived_bbuffer is
constant return_value : derived_bbuffer := (
                                        --  tag => object.tag,
                                            buf => object.buf,
                                            buf_in => object.buf_in,
                                            buf_out => object.buf_out);
begin
    return return_value;
end;
```

An object of derived_bbuffer may call the parent's implementation of the primitive operation get by applying the conversion function on the formal parameter to obtain a value which matches the actual parameter.

```
variable item : item_type;
variable derived_bbuffer_object : derived_bbuffer;
…
get(to_derived_bbuffer(object) => bbuffer(derived_bbuffer_object),
    item => item);
```

The conversion function provides the required type conversion from parent to child. It is important to note, that the call is different from the call of the inherited subprogram get which does not modify the tag of the formal parameter:

```
get(derived_bbuffer_object, item);
```

It rather corresponds to the static call of the parent's implementation of get:

```
get'Parent(derived_bbuffer_object,item)'Static;
```

The situation is a little bit different if a derived type is modelled which extends the parent's structure. Then a conversion function can be used to extend the parent type by values for the additional record elements of the derived type.

Consider the derived type gb_buf with its additional record element after_put:

```
type gb_buf is new bbuffer with record
    after_put : boolean;
end record;
```

The conversion function may look like this:

```
function to_gb_buf ( object : bbuffer) return gb_buf is
    constant return_value : gb_buf := (
                                    buf => object.buf,
                                    buf_in => object.buf_in,
                                    buf_out => object.buf_out,
                                    -- extension part:
                                    after_put => false);
begin
    return return_value;
end;
```

Now, there is a difference between the calls:

```
variable gb_buf_object : gb_buf;
…
get(to_gb_buf (object) => bbuffer(derived_bbuffer_object), item => item);
-- different from:
get'Parent(gb_buf_object,item)'Static;
```

Both calls implicitly adapt the value of the tag in the formal parameter. However, the conversion function additionally modifies the synchronisation information in the record element after_put while the implicit conversion of the static call does not modify it. For example, the call of put using the conversion function to_gb_buf would model an incorrect synchronisation.

```
put(to_gb_buf (object) => bbuffer(derived_bbuffer_object), item => item);
```

The call erroneously assigns the record element which traces the invocation history of put to false. The assignment might remove values from the record element which are assigned in the operation put as part of its synchronisation[17].

Conversion functions to convert tag information are restricted to associations of variable parameters in procedure calls and associations of signals in port maps, according to the general restrictions on applying conversion functions (Compare LRM 2.1.1.2, 4.3.2.2). Especially conversion functions are no means to change a tag of a signal which is passed as a parameter to an operation.

An interesting situation occurs if an object of a tagged type is passed as a parameter to primitive operations in which some implicit type conversions occur. Constant or variable parameters of tagged records may be passed by

17. A corresponding consideration can be made if signals in port maps use conversion functions. The functions might falsify synchronisation information by removing transactions which are relevant for synchronisation from the projected output waveform of the signal.

copy or reference to the subprogram like any parameter whose type is a record. In a call which includes an implicit type conversion of a tagged record the tag of the formal parameter is assigned the value which indicates the target type of the conversion at the start of each call. After completion of the subprogram body the original value of the tag before the call is assigned to actuals which have been involved in the implicit type conversion. In case the parameter is passed by reference the temporary change of the tag in the actual parameter does not have any effect in a simulation and thus is harmless. The rest of the model cannot observe the temporary change. It is as if the object never had changed its tag.

The situation is different and more complex if we look at signal parameters of tagged types which are involved in an implicit type conversion. Any change of the tag at the beginning of a call or after the completion of a subprogram similar to the parameter passing mechanism of constants and variables that are of tagged types would be observable across process boundaries in a model[18]. Such a mechanism would introduce inconsistencies in the timing and synchronisation concept of the language. The corresponding problems of language extensions which allow to pass information across process boundaries within one delta cycle have been discussed in Chapter 5.

The solution of the problem is not to change the tag in an implicit type conversion of signals. However, during the execution of the subprogram body the formal parameter behaves in the subprogram as if it has a tag indicating the target type of the implicit type conversion.

The reason for introducing tags with all the complex behaviour in explicit and implicit type conversion was to be able to determine the type of an object at run-time. With this tag concept as a starting point we now discuss how this information can be used to model polymorphic objects which may change their properties during simulation.

6.3 Polymorphism

The type extension mechanism presented so far allows to derive a new type from an existing one. The derived type inherits the properties of the parent. Thus, it is possible to use tagged types for modelling classes which share common properties and which are related to each other in an is-a relation. The resulting type hierarchy from deriving tagged types is not a subtype hier-

18. As a side effect, such a mechanism cannot be translated into VHDL.

archy the types are only closely related to each other. Such closely related but distinct types do not support the modelling of heterogeneous object containers which may contain object of various classes. What we require is a concept which provides the missing subtype relation in the type hierarchy.

A general approach to establish a missing subtype relation in a typed class concept was discussed in Section 3.2.7 where the concept of class-wide types was mentioned.

We now present how such a concept is introduced into the language extension. It is explained how the concept provides polymorphism which can be used to model heterogeneous object containers. The mechanism how to invoke operations of such containers is analysed. The modelling of polymorphic objects with signals is investigated and some conclusions are drawn on inheritance concepts for signals.

6.3.1 Class-wide types

In Section 3.2.7 we saw that a class-wide type is a type which is associated with each type modelling a class. In the language extension such a class-wide type is implicitly defined for each tagged type. It is associated to the tagged type. The implicit type is defined by a predefined attribute 'Class of the language extension. The attribute takes the name of a tagged type as a prefix and denotes the associated class-wide type.

The class-wide type is characterized by a set of values like any other VHDL type. The set is the union of all values of the associated type and the values of all types derived from the associated type. The resulting subtype[19] relation allows the modelling of a polymorphic object. We refer to such objects as class-wide objects or more specific as class-wide variables or signals. Although class-wide objects may contain values with different structure only the structure of the associated tagged type is visible by selection outside the object. In other words, only the identifiers of record elements of the associated tagged type can be used in a selected name which denotes an element of a class-wide object.

For example, to denote the class-wide type associated with the bounded buffer bbuffer we can write bbuffer'Class. An object of the class-wide type can be used to model a heterogeneous object container which can contain

19. Subtype here is not meant in the strict meaning of VHDL but in the more general sense of Section 3.2.2.

values of the type bbuffer and its derived classes derived_bbuffer, gb_buf, buf2, and x_buf2.

```
variable container : bbuffer'Class := …;
```

The container is a class-wide variable which models a polymorphic buffer, i.e the buffer may change the properties characterizing its type during the simulation. Such a modelling of a polymorphic object is not restricted to the root of a tree of derived types. There is also a class-wide type implicitly defined for each of the derived types. The derived type comprises all values of the sub-tree which has the derived type as a root.

For example, buf2'Class comprises values of the tagged type buf2 and the derived type x_buf2. It does not comprise values of the type bbuffer or gb_buf. The values of buf2'Class are a subset of the values of bbuffer'Class.

It is possible to assign values of the different tagged types to a class-wide object during simulation. Such values of the tagged types are implicitly converted to the class-wide type. The value keeps its structure and the values in its elements. This especially includes the tag. It is just the tag of a class-wide objects which indicates its present structure. The tag so to say provides the interpretation of the information in a class-wide object.

In the example it would be possible to assign a value to the container that is of type gb_buf:

```
container := gb_buf'( --   tag => gb_buf'tag,
                           buf => (others => item_type'left),
                           buf_in => 0,
                           buf_out => 0,
                           after_put => false);
```

The assignment performs an implicit type conversion to the type bbuffer'class. The class-wide variable has the structure of the type gb_buf after the assignment including the record element after_put which is not in the structure of bbuffer. The elements buf, buf_in, and buf_out are visible outside the container and can be accessed via a selected name.

```
container.buf_in := 0;
```

The value in after_put is assigned to the container but it is not visible outside the container.

```
container.after_put := 0; -- illegal assignment
```

The tag of the container which identifies the present structure is implicitly assigned the value which indicates the type gb_buf.

As the type information in a tag of a polymorphic object may change during simulation it may be interesting to read the tag information. This is possible by using the predefined attribute 'Tag of the language extension. If it decorates a class-wide object it denotes the value of its tag. The value is of type tag which is declared in the package
STD.OBJECT_ORIENTED_EXTENSION.

In the example it would be possible to test if the value of the tag denotes the type gb_buf after the assignment.

```
assert container'Tag = gb_buf'Tag;
```

The attribute 'Tag is a function which returns the value of the tag it cannot be used to write values to the tag of a class-wide type or any other type[20].

The implicit conversion of values that are of tagged type allows the assignment of such values to class-wide objects. An implicit conversion in an assignment vice versa is also possible. In that case, the tag of the class-wide expression has to denote the tagged type of the target of the assignment. A dynamic check during simulation tests the condition and if it fails the model is erroneous.

In the example, we could assign the value which is stored in the container to an object that is of type gb_buf.

```
variable container : bbuffer'Class := …;
variable bbuffer_object : bbuffer;
variable gb_buf_object : gb_buf;
…
container := gb_buf'( …);
gb_buf_object := container; -- implicit conversion with dynamic check
-- bbuffer_object := container; -- erroneous: dynamic check would fail;
```

It would be erroneous to store it in an object that is of type bbuffer. To store it in such an object an explicit type conversion is required as it is provided for closely related types.
In the example this would be a conversion to the type bbuffer.

```
bbuffer_object := bbuffer(container);
```

20. Please note, it is not possible to decorate an object of a non class-wide tagged type with the attribute 'Tag at all. The restriction becomes clear if we look at the considerations on tags in a previous section. Among other things it was explained that a signal parameter may behave as if it has different tags at the same time.

Such an explicit conversion of an expression of a class-wide type into a tagged type is only allowed if the tag of the class-wide type denotes a tagged type that is a descendant of the type of the target in the assignment.

It is as if the explicit conversion is applied on the result of an implicit conversion from the class-wide type to the tagged type denoted by its tag. The explicit conversion fails if it is not applied towards the root (Compare Section 6.2.2). All the elements of the target in the assignment are assigned the corresponding elements of the expression. Any additional elements of the class-wide type are simply ignored.

In the example the object gb_buf_object has all the values stored in the class-wide type including the value stored in the hidden record element after_put. The object bbuffer_object only stores the values of the elements buf, buf_in, and buf_out after the explicit conversion. It does not store the value of the record element after_put.

The language extension also allows to assign an expression that has value of a class-wide type to an object of another class-wide type. In such an assignment the value of the tag of the expression has to denote a tagged type that can be converted in an implicit conversion to the class-wide type of the target. Such an assignment requires a dynamic check if an implicit conversion is possible. If the check is successfully passed the values of all elements of the expression including the value of the tag are assigned to the elements of the target. If the check fails the model is erroneous.

Consider the example of the container which is extended by another container:

```
variable container : bbuffer'Class := bbuffer'(…);
variable another_container : gb_buf'Class := gb_buf'(…);
…
-- another_container := container; -- erroneous
-- no implicit conversion from container'Tag = bbuffer'Tag to gb_buf'Class
container := another_container;
```

The class-wide object container is assigned a tag which indicates the type gb_buf. Furthermore all the values of the elements of another_container including the element after_put are assigned to the object container.

Apart from being used as operands in primitive operations class-wide signals may also occur in a sensitivity list of a wait statement[21]. Like for signals of other composite types that are in a sensitivity set the effect on class-wide signals is as if scalar sub-elements become members of the sensitivity set. In case of a class-wide signal those sub-elements become virtually a member of the sensitivity list that are sub-elements of the tagged type indicated by the effective value of the signals's tag including the tag itself. Please note, the sensitivity set may include sub-elements that are not visible outside the class-wide signal. A consequence of the concept is that the sensitivity set dynamically may change during simulation depending on the tag of a class-wide signal.

With this concept a synchronisation point in the language extension may dynamically change its behaviour during simulation similar to primitive operations which are called with class-wide operands. The difference is that the behaviour may change during the execution of a synchronisation point whereas its behaviour must not change after starting the execution of a particular primitive operation.

A class-wide type defines a collection of values which is not constrained in the number and types of its values. Accordingly, a class-wide object represents a collection of objects which is not constrained in the number or type of its objects. At any time in the modelling it is possible to derive new types from the type associated with the class-wide type and thus extend the collection. Nevertheless, a class-wide type is treated as a constrained type in the sense that it is not necessary to explicitly put any constraints on the size or quality of its collection. It is automatically constrained at the end of the elaboration. At that point the information about all potential values of a class-wide type are available and thus the collection can be constrained to these potential values. The extension modifies the original elaboration concept of VHDL. In VHDL declarations in a declarative part are basically elaborated in the order in which they are given in the declarative part (Compare LRM 12.3). In the extension the elaboration of a class-wide type depends on the elaboration of derived types that are derived from the tagged type associated with the class-wide type. However, the language extension does not require the derived types to be declared before the class-wide type. Such a restriction would be against the idea of modelling by extension. The declarations are not elaborated in the order they are given any longer.

A class-wide type is implicitly declared after the declaration of its associated tagged type. Its elaboration as part of the elaboration of a package declaration is postponed after the elaboration of declarations in the package which

21. Referencing only sub-elements of the class-wide signal in a wait statement is also possible. However, the use of sub-elements is restricted by the visibility rules on the structure of class-wide types as they were stated at the beginning of this section.

do not directly or indirectly reference the class-wide type. Indirectly means that in a chain of references where a declaration is referenced in another declaration the class-wide type does not occur. The elaboration of a class-wide type involves first the elaboration of declarative items in packages containing derived types. Only those declarative items are elaborated that do not directly or indirectly reference the class-wide type. Elaborating the items means to elaborate at least partly the package containing the items.

Similar to the original elaboration mechanism of VHDL, the elaboration of those packages involves first elaborating each not-yet-elaborated package containing declarations that are referenced by the packages. In such a partial elaboration of a package only those declarative items are elaborated that do not directly or indirectly reference the class-wide type. Likewise the partial elaboration of any package which is caused by the elaboration of a class-wide type works accordingly. Only those declarative items are elaborated that do not directly or indirectly reference the class-wide type.

During elaboration the above elaboration rules are only applied to declarative items that are not-yet-elaborated.

If dependencies are modelled that do not allow an elaboration according to that rules the model is considered to be erroneous.
Consider the following declarations:

```
package parent_package is
    type parent is tagged record …;
    type ref_to_parentC is record
        reference : parent'Class;
    end record;
    type contains_no_ref_to_parentC is (enum);
    …
end parent_package;

package not_yet_elab_package is
    type another_contains_no_ref_to_parentC is (enum);
    type direct_ref_to_parentC is record
        reference : work.parent_package.parent'Class;
    end record;
    type another_enum_type is (enum);
end not_yet_elab_package;

use work.parent_package.all;
package child_package is
    type elem_type is (enum);
    type indirect_ref_to_parentC is record
        reference : work.not_yet_elab_package.direct_ref_to_parentC;
    end record;
    alias another_contains_no_ref_to_parentC is
    work.not_yet_elab_package.another_contains_no_ref_to_parentC;
    type child is new parent with
        element1 : elem_type;
        element2 : contains_no_ref_to_parentC;
        element3 : another_contains_no_ref_to_parentC;
    end record;
    …
end child_package;
```

The elaboration of the parent_package starts with the elaboration of the tagged type parent. The elaboration of the class-wide type associated with parent involves first the elaboration of the type declarations of contains_no_ref_to_parentC, another_contains_no_ref_to_parentC, the elaboration of the declarations another_enum_type, elem_type, the elaboration of the alias declaration of another_contains_no_ref_to_parentC, the implicit alias declaration of enum[22], the elaboration of the type declarations of child and the class-wide type associated with child. The elaboration is executed in the given order. Then the elaboration of the class-wide type associated with parent is completed. All types of potential values of the class-wide type are elaborated. The elaboration of the package now proceeds with the elaboration of the type ref_to_parentC.

Obviously, changing the order of elaboration makes the elaboration of class-wide types more complex than the elaboration of other composite type definitions, because it is necessary to determine all the potential values in the collection. The information about potential values depends on the tree of the derived types of the tagged type which is associated with the class-wide type.

Such an elaboration only works if the collection of values represented by the type is definite. In Section 6.2.4 we have discussed how to keep the collection of elements in tagged types definite by prohibiting recursive structures.

A similar concept is necessary to prohibit recursive structures in class-wide types. Such a recursive structure would occur if a tagged record had a record element which is of a class-wide type that is associated with the tagged record or that is associated with any of the tagged record's parents. Likewise a recursive structure would occur if any sub-element of the com-

22. Compare LRM 4.3.3.2

posite type has such a class-wide type. In other words, in a collection of values that is defined by the tagged record type the occurrence of a class-wide type that is associated with the tagged record or its parents causes an indefinite recursive structure.

We illustrate such a recursive structure by a variant of the backup_buf presented in Section 6.2.4.

```
type encapsulate_bbuffer_classwide is record
    subelement_of_backup_buf_variant : bbuffer'Class;
end record;
-- Not in the language extension of VHDL:
type backup_buf_variant is new bbuffer with record
    backup1 : backup_buf_variant'Class;
    -- recursively contains backup_buf_variant
    backup2 : bbuffer'Class;
    -- recursively contains backup_buf_variant
    backup3 : encapsulate_bbuffer_classwide;
    -- subelement has type bbuffer'Class and
    -- thus recursively contains backup_buf_variant
end record;
```

To prohibit such recursive structures it is necessary define visibility rules for tagged types that extend the exiting ones for normal record types. In a declarative region that is associated with a tagged type any class-wide type that is associated with the tagged type or with any of the tagged type's parents is not visible within that region. We say, the class-wide type is hidden in the declarative region by its direct subtype relation to the tagged type. Any composite type that has an element of a type that is hidden by a direct or indirect subtype relation to the tagged type is said to be hidden by its indirect subtype relation to the tagged type. The identifier which denotes the composite type is not visible in the declarative region associated with the tagged type. The recursive definition of *hidden by a subtype relation* prohibits the definition of indefinite recursive structures in a tagged type.

The visibility rules are just defined in a way that enables the elaboration of class-wide types according to the previously presented scheme.

In the example, the class-wide types associated with the types bbuffer and backup_buf_variant are not visible in the tagged record type declaration due to their direct subtype relation to the tagged type backup_buf_variant. The record type encapsulate_bbuffer_classwide is hidden by its indirect subtype relation to the tagged type backup_buf_variant.

So far, we have described how a class-wide object may change its tag and accordingly its structure during simulation. We now discuss how other properties in particular the behaviour also may change accordingly during simulation.

6.3.2 Dispatching

The behaviour of a tagged type is modelled by a set of primitive operations which characterize the type. It suggests itself that a class-wide type which comprises values from different tagged types is characterized by the corresponding different primitive operations.

A class-wide type has no primitive operations itself which it passes to derived classes. There is no inheritance mechanism defined on class-wide types at all. The class-wide type is rather characterized by primitive operations of its associated tagged type as follows. It is possible to pass an actual parameter of a class-wide type to the primitive operations of the associated tagged type and to inherited or overloaded primitive operations of derived types. We call such an actual parameter which is associated with a formal parameter of a tagged type in the call of a primitive operation of the tagged type controlling operand[23].

As mentioned in Section 6.2.2 a call of a primitive operation requires that all controlling operands are of the same type. The rule that all controlling operands have to have the same type in a call does not exclude implicit type conversions from a derived type to a parent type on actual parameters as it was described in Section 6.2.2. However, the rule does not allow to mix controlling operands of a class-wide type and a tagged type in a call of a primitive operation.

In the container example it would be possible to call a primitive operation of bbuffer:

```
variable container : bbuffer'Class := gb_buf'(…);
variable item: item_type;
…
get(object => container, item => item);
```

In such a case where class-wide objects are used as controlling operands in an operation call the tags of those controlling operands that are of mode in or inout are used to determine which operation is actually executed as a result of

23. This is just a more general definition of the term controlling operand compared to the one given in Section 6.2.2.

the call. During the dynamic elaboration of a subprogram call the tags are checked to have the same value. If the check fails an error occurs. The subprogram of the tagged type is chosen that is indicated by the tag. The controlling operands[24] are implicitly converted to the tagged type indicated by the tags. The conversion is performed like the other implicit type conversions in calls with tagged types.

In the example, the container has a tag with a value that indicates the type gb_buf during the elaboration of the call. Therefore the primitive operation get of the type gb_buf is executed. The controlling operand container is implicitly converted to the type gb_buf.

We call such a late binding of the operation dispatching. The polymorphic object container may change its associated behaviour during simulation.

After presenting the basic concepts of dispatching we now want to illustrate some of the properties of dispatching by examples.

One of the examples is the call of an assignment operation with a target that has a class-wide type. Its implicit conversions of the right-hand side values to the class-wide type of the target makes an assignment statement a dispatching call of a basic operation. The results of such a dispatching call correspond exactly to the effects of an assignment statement as they were explained in Section 6.3.1.

Another example of a dispatching call on a basic operation is the call of the predefined equality and inequality operations with values of a class-wide type. Both operand types must have the same class-wide type. During the execution of the operation the tags of the operands are checked to have the same value indicating the same type. If the check fails the execution is erroneous otherwise the operation is executed like on any other record type.

After showing dispatching calls on basic operations we illustrate dispatching calls on primitive subprograms. As mentioned before, dispatching works for primitive operations of tagged types that are associated with the class-wide type of the controlling operands. If we look at the container example from above, it is possible to call the operations put and get with an actual parameter of type bbuffer'Class. However it would be an error to call the operation gget with that parameter even if the value of the tag of the parameter indicates the type gb_buf during the dynamic elaboration of the call.

```
variable container : bbuffer'Class := gb_buf'(…);
…
gget(object => container, item => item); -- illegal
```

24. All controlling operands, including those of mode out are converted.

Another erroneous situation occurs if we call an operation with controlling operands of different types. Consider a type e_buf which is derived from bbuffer and which has an additional primitive operation equal. The equal operation returns a result parameter that is true if two objects contain the same number and sequence of items[25].

```
type e_buf is new bbuffer with null record;
procedure equal (    object1 : inout e_buf;
                     object2 : inout e_buf;
                     test_result: out boolean);
```

From the type e_buf another tagged type ce_buf is derived which has an additional copy operation which copies the content of one buffer into the content of the other.

```
type ce_buf is new e_buf with null record;
procedure copy (source : in ce_buf;
                target : out ce_buf);
```

Any calls with controlling operands that are of different types even after applying implicit type conversions according to the rules presented above are erroneous:

```
variable e_container : e_buf'Class := e_buf (…);
variable another_e_container : e_buf'Class := ce_buf (…);
variable ce_container : ce_buf'Class := ce_buf (…);
variable another_ce_container : ce_buf'Class := ce_buf (…);
variable e_buf_object : e_buf;
variable another_e_buf_object : e_buf;
variable ce_buf_object : ce_buf;
variable test_result : boolean;
…
List of illegal calls:
-- equal ( e_container, another_e_container, test_result);
-- illegal:   tags of controlling operands of mode inout are not equal
--            run-time error
-- equal ( e_container, e_buf_object, test_result);
-- illegal:    types of controlling operands are different
-- equal ( ce_container, another_e_container, test_result);
```

25. Please note, the operation is different from the predefined equality operator which not only requires two buffers to contain the same sequence of items but additionally requires to have the same values in their elements buf_in and buf_out.

```
-- illegal:    types of controlling operands are different
--             (although value of tags are equal)
-- equal ( e_buf_object, ce_buf_object , test_result);
-- illegal:    types of controlling operands are different
-- copy ( ce_container,ce_buf_object );
-- illegal:    types of controlling operands are different
```

The legal calls have always controlling operands of the same type and the operands of mode in or inout have the same value of the tag:

```
--List of legal calls:
equal ( ce_container, another_ce_container, test_result);
equal ( e_buf_object, another_e_buf_object, test_result);
copy ( e_container, another_e_container );
```

In the last call it is interesting to note that it is a legal call although the controlling operands have different values in their tag during the elaboration of the call. It is legal because the formal parameter target of the operation copy has the mode out and thus is not required to have the same tag as the formal parameter source which has the mode in.

Modelling polymorphic objects by using class-wide objects and dispatching allows to generalize from concrete objects and their behaviour. In the example of a call on the primitive subprogram equal with class-wide actual parameters the call abstracts from the concrete objects and their behaviour. The details of the properties and the concrete implementations are not relevant or not even known when the call is modelled. From a language point of view that means the call is not required to be in the scope of the tagged type which has the properties which are observable in the call during simulation. In other words it is only the class-wide type that is required to be visible and not any tagged type.

6.3.3 Postponed use clause

In Section 6.3.1 we saw that the extension to the elaboration mechanism of VHDL allows a flexible object-oriented modelling with class-wide types. The flexibility was achieved by a postponement of the elaboration of the class-wide type in the order of elaboration. It is finally elaborated after the derived types it depends on.

We now introduce another extension to the elaboration concept which tolerates some changes in the order of elaboration from the order in which the declarations are given. The goal is to provide a greater modelling flexibility in cases where we do not want to use class-wide types to achieve this effect but only tagged types. We are looking for a possibility to describe a deferred implementation of a class.

Such a flexibility in the order of elaboration is required when two tagged types are made visible in each other's package declaration to model a mutual use-relation between the types.

A general description of such modelling situations where two module specifications try to import each other in a language which uses a linear[26] elaboration model is given in [169]. It describes the so-called with-ing problem which is just the difficulty to model module specifications importing each other. It proposes some solutions for the programming language Ada[27].

Making two types which are in different package declarations visible to each other is not possible in VHDL. In VHDL the elaboration of a package starts with the elaboration of not-yet elaborated packages containing declaration referenced by the package. In case of the two packages containing the two types one of the two packages would start such an elaboration of the not-yet-elaborated other package. This in turn would re-start the elaboration of the package which originally started the elaboration. A chicken-and-egg dilemma. This limitation is not really a problem in VHDL. On the one hand it avoids recursive record definitions in an elegant way. It is not possible to model the following records:

```
type A is record
    element B;
end record;
…
type B is record
    element : A;
end record;
```

On the other hand, if the mutual visibility is required in other declarative items of the packages, for example in interfaces of subprograms, then it is possible to model the items in separate packages. This can be illustrated by the following example:

```
-- use work.Q.B; -- Package Q is not-yet-elaborated
package P is
    type A is …
```

26. Linear means, the order of elaboration corresponds to the order in which the declarations appear in the source code.
27. Interestingly enough, some of the *solutions* describe a *language extension* to Ada.

```
    -- procedure mutual (paramA : A; paramB : B); -- B is not visible here
end P;

use work.P.A;
package Q is
    type B is …
    procedure mutual2 (paramB : B; paramA : A);
end Q;

use work.P.A;
use work.Q.B;
package additional_separate_package_for_A is
    procedure mutual (paramA : A; paramB : B);
end additional_separate_package_for_A;
```

The modelling situation is different in the language extension where it is possible to model tagged types and corresponding primitive operations in a package. Splitting the package and moving an operation into another package to achieve a proper elaboration order would make a primitive operation a non-primitive operation.

As a simple example suppose we want to model part of an ALU with a buffer which stores instructions and data send to the ALU. One of the instructions which we want to sent to the ALU is a store instruction to store some data in the buffer. We could model the buffer and the instruction as a tagged type in packages.

```
package instr_and_data_buffer_package is
    type instr_and_data_buffer is tagged record …
    procedure put (  object : inout instr_and_data_buffer;
                     instr : in store_instr);
    procedure get (  object : inout instr_and_data_buffer;
                     instr : out store_instr);
end instr_and_data_buffer_package;

package store_instr_package is
    type store_instr is tagged record …
    procedure execute_instr (  object : in store_instr;
                               \buffer\ : inout instr_and_data_buffer);
end store_instr_package;
```

Modelling the put and get operation as a primitive operation of the buffer and modelling the procedure execute_instr as a primitive operation of the tagged type store_instr causes the mutual use-relation between the types and thus the mutual dependency of the package declarations. Moving the subprogram execute_instr into another package to break the mutual dependency is not possible because the subprogram would then not be any longer a primitive operation of the type store_instr.

As mentioned before, to solve the problem it is necessary to change the order of elaboration. We saw in Section 6.3.1 how it is possible in the language extension to alter the order of elaboration by using class-wide types. The elaboration is completed only after the elaboration of dependant types.

In the example it is possible to postpone the elaboration of the type of one of the parameters in the primitive operations by using a class-wide type. The type associated with the class-wide type has to be a parent of one of the tagged types in the package. Defining a tagged type instruction as a parent of the tagged type store_instr we can use the class-wide type instruction'Class as type of the parameters in the operations of the buffer.

```
package instruction_package is
    type instruction is tagged record …
    …
end instruction_package;

use work.instruction_package.instruction;
package instr_and_data_buffer_package is
    type instr_and_data_buffer is tagged record …
    procedure put (  object : inout instr_and_data_buffer;
                     instr : in instruction'Class);
    procedure get (  object : inout instr_and_data_buffer;
                     instr : out instruction'Class);
end instr_and_data_buffer_package;
```

The type store_instr is derived from the type instruction. As the package store_instr_package does not depend on the package declaration of instr_and_data_buffer_package any longer it is possible to make the type instr_and_data_buffer visible in the package store_instr_package.

```
use work.instruction_package.instruction;
use work.instr_and_data_buffer_package. instr_and_data_buffer;
package store_instr_package is
    type store_instr is new instruction with …
    procedure execute_instr (  object : in store_instr;
                               \buffer\ : inout instr_and_data_buffer);
end store_instr_package;
```

Calling the operations of instr_and_data_buffer requires a class-wide object as actual which is assigned a value that is of the type store_instr before the call. In the assignment the type of the value is implicitly converted as explained in Section 6.3.1. In the procedure bodies the conversion has to be performed from the class-wide type to the tagged type store_instr. The conversion with the target type store_instr is workable because although it is not possible to make declarations of the package store_instr_package visible in the package declaration of instr_and_data_buffer_package it is possible to make them visible in the package body according to the elaboration rules of VHDL.

```
-- library work;
use work.store_instr_package.store_instr;
-- make store_instr directly visible
package body instr_and_data_buffer_package is
    procedure put (   object : inout instr_and_data_buffer;
                      instr : in instruction'Class) is
    -- store_instr is visible here and can be used for a type conversion
…
end instr_and_data_buffer_package;
```

Using class-wide types to break the mutual dependencies does not mean to circumvent the strong typing of VHDL. As explained in Section 6.3.1 the class-wide type uses type checking mechanisms to guarantee correct typing. However, the checks are not performed during analysis time but during simulation. To provide the strong type checking at analysis time a new language construct which indicates a different elaboration order for tagged types is introduced into the language extension.

The new construct is called postponed use clause. It looks like a use clause preceded by the keyword postponed. It might appear as a context clause of a design unit that is a package declaration or it might appear as a declarative item in a package declaration and has there the same scope as a normal use clause. Each selected name in a postponed use clause identifies tagged types which might be referenced in the scope of the postponed use clause but which are not required to be elaborated at the time the declarations in the package declaration are elaborated. Especially, the packages containing the referenced declaration are not required to be elaborated. Likewise, tagged types identified by a postponed use clause are not required to be analysed, the referenced package is not even required to exist at analysis time of the postponed use clause.

The postponed use clause makes the name of an identified tagged type visible by selection and if there is no homograph of the name visible in the scope of the postponed use clause it also makes the name of the tagged type directly visible. The elaboration of the tagged types is postponed until the package declarations containing the referenced types is elaborated according to the other elaboration rules.

The use of the names of the tagged types is restricted in the scope of the postponed use clause. They are only allowed as a subtype indication in the interface list of a primitive operation[28].

The language extension allows to break mutual dependencies similar to class-wide types. In the example of the tagged type instr_and_data_buffer the type store_instr can be referenced by making it visible through a postponed use clause.

```
postponed use work.store_instr_package.store_instr;
package instr_and_data_buffer_package is
    type instr_and_data_buffer is tagged record …
    procedure put (  object : inout instr_and_data_buffer;
                     instr : in store_instr);
    procedure get (  object : inout instr_and_data_buffer;
                     instr : out store_instr);
end instr_and_data_buffer_package;

use work.instr_and_data_buffer_package.instr_and_data_buffer;
package store_instr_package is
    type store_instr is tagged record …
    procedure execute_instr (  object : in store_instr;
                               \buffer\ : inout instr_and_data_buffer);
end store_instr_package;
```

Referencing the type store_instr in the package declaration instr_and_data_buffer_package does not cause an elaboration of the package store_instr_package. The elaboration is postponed.

Similar to the example of the class-wide type it is useful to make the tagged type store_instr and its primitive operations directly visible in the package body of instr_and_data_buffer_package for example by a normal use clause.

28. Allowing the names to occur elsewhere would make the elaboration concept of the language extension too complex. It would not be possible to find an appropriate translation mechanism to VHDL which re-arranges the elaboration order of the types and resolves mutual dependencies.

```
use work.store_instr_package.store_instr;
-- makes the primitive operations of store_instr directly visible
package body instr_and_data_buffer_package is
    …
end instr_and_data_buffer_package;
```

Like in the example using a class-wide type the approach provides a strong typing, however, the type checking mechanism is performed during analysis and elaboration.

6.3.4 Private type

We have almost completed the introduction of the language extension. We saw how the language extension provides the object-oriented principles of abstraction and modelling by extension.

The object-oriented concept of encapsulation was based on the package mechanism of VHDL. Especially its separation of package declaration and package body allows an effective encapsulation of behaviour of a tagged type. Only the subprogram specifications modelling the primitive operations of the tagged type are made visible outside the package declaration. This is different with the encapsulation of structure of a tagged type. The complete structure is modelled in the package declaration and thus can be made visible outside the package. To overcome this weak point in the language an additional encapsulation mechanism for the structure of tagged types is introduced into the language extension.

The new encapsulation mechanism is based on so called private types. A private type is a tagged type which allows the declaration of record elements only visible in the primitive operations of the record.

Additionally to the normal sequence of element declarations a private tagged type has a private part which is an additional sequence of element declarations. The element declarations in the private part have a limited scope compared to the element declarations in the non private part. The scope of the element declarations in the private part only extends to the primitive operations of the private type and to the primitive operations of tagged types derived from it. In other words, the primitive operations are the only access mechanism to elements declared in a private part of a tagged type. The private part is protected from any uncontrolled access by an user of the tagged type.

Primitive operations which have access to the elements of the private part especially include the assignment operation as a basic operation on a tagged type. The assignment operation assigns each right-hand side value to its corresponding sub-element of the target including all elements in the private part of the target (Compare Section 6.2.5). The assignment in principle works like any other assignment to a composite type, however, there is an important limitation on the use of an assignment operation on private types which follows from the rules above regarding visibility. The target of an assignment that has a private type must not be an aggregation if the assignment is outside a corresponding primitive operation, i.e. outside the scope of the elements in the private part of the target. Even if the names of the hidden elements of the aggregate do not explicitly occur in a positional element association of the aggregate the assignment is not allowed. It is not only the names but the entire structure that is hidden in a private part of a tagged type.

The private part of a tagged type that is not derived from another tagged type is defined as the sequence of element declarations that follow the new keyword private in the record definition.
The following example shows a private tagged record.

```
type private_tagged_record is tagged record
    non_private_element : integer;
private
    private_element : boolean;
    another_private_element : bit;
end record;
```

The private part comprises the declarations of private_element and of another_private_element.

The private part of a derived type is the union of the parents' private parts and the element declarations that follow the keyword private in the definition of the private derived type.

In the example we can derive a private type from the type private_tagged_record.

```
type derived_tagged_record is new private_tagged_record with
    another_non_private_element : bit;
private
    additional_private_element : integer;
end record;
```

The private part of the derived type consists of the declarations of private_element, another_private_element, and additional_private_element.

In Section 3.3.18 it was discussed how references stored in instance variables break the encapsulation of objects. The unwelcome consequences in distributed objects have been considered. A proposal to solve the problem was to avoid the breaking of encapsulation. It was stated that this could be achieved by avoiding any reference mechanism.

To preserve the encapsulation of objects that use the encapsulation mechanism provided by private types and to support the modelling of distributed objects a new kind of composite type is introduced into the language which is called reference-free composite type. A reference-free composite type is a composite type which only contains elements that are of scalar or reference-free composite type. That means, an object that has a reference-free composite type finally is a collection of objects of scalar types. None of the objects in the collection is of an access type. The encapsulation of the private part of a tagged type is preserved by a rule that requires that a private part only may contain elements that are of scalar or reference-free composite type.

The rule avoids an access value to be hidden in a private part of a tagged type declaration. Signals of a private type that are used to model distributed objects are prevented from the problematic effects of breaking encapsulation in the private part. Elements of reference-free composite types in the private part inhibit the unintentional declaration of a signal that is of an access type without requiring the analysis of the private part of a tagged type and thus breaking the encapsulation. This coincides with VHDL in which it is an error if a signal declaration declares a signal that is of an access type (Compare LRM 4.3.1.2).

Signals are used to provide a reference-free communication between processes which are a means to model distributed behaviour and structure in VHDL. Objects modelled in different processes can be viewed as distributed objects. Values which may represent the state of an object are passed from one process to another via signals[29]. The communication between the processes and thus between the distributed objects is based on a copy model as it was mentioned in Section 3.3.18.

Thus, the language extension provides a modelling concept which corresponds to the approach presented in Section 3.3.18 with respect to distribution of objects. The extension does not generally forbid the use of instance variables in classes and objects. It supports both, passing objects and references to objects as messages between objects. As we have already stated in Section 3.3.18, it is only the objects’s structure that determines whether an objects or at least its state can be sent from one object to another. As a conse-

29. The signal itself may be considered here as an object which is used for communication.

quence modelling with private types that are at the same time reference-free composite types provides to a large extent late binding of parallelism.

Like any other tagged type a private tagged type has an implicitly declared class-wide type associated with it. It is characterized by a set of values which is the union of all values of the associated type and the values of all types derived from the associated type. In Section 6.3.1 it was explained that only the structure of the associated tagged type is visible by selection outside the object. This must be stated more precisely in the context of private types. In a class-wide type only the structure of the associated tagged type that is declared in the non private part is visible outside the object. In other words, only the identifiers of record elements of the associated tagged type in the non private part can be used in a selected name which denotes an element of a class-wide object. Any element declared in a private part of the associated tagged type is hidden in a class-wide type. Access to the hidden elements is only possible via a dispatching call that executes a primitive operation which is in the extended scope of an element in the private part of the corresponding tagged type. Only the interpretation of the private part as a private part of the tagged type indicated by the tag of the controlling operand makes the elements of the private part accessible. The interpretation is due to the implicit conversion which is part of the elaboration of the dispatching call.

Hiding elements of a private part in a class-wide object that is a signal also means that their names cannot be used in a sensitivity list of a wait statement outside a corresponding primitive operation. What can be used in such a sensitivity list is the name that denotes the class-wide signal. In that case each scalar sub-element of that signal including the elements of the private part is in the sensitivity set of the wait statement, that means a process may resume as a result of an event on that elements.

Consider an implementation of the bounded buffers bbuffer and gb_buf which completely encapsulate their structure in a private part.

```
type bbuffer is tagged record
private
    buf : bounded_buffer_array (1 to buffersize);
    buf_in : integer;
    buf_out : integer;
end record;

type gb_buf is new bbuffer with record
private
```

```
    after_put : boolean;
end record;
```

None of the record elements is visible outside the tagged record. Only the primitive operations put, get, and gget have access to the elements in the private parts of the objects. A dispatching call on the operations performs the implicit conversion to the tagged types bbuffer or gb_buf and thus makes the elements of the private parts accessible.

```
variable container : bbuffer'Class := bbuffer (…);
variable item, item_type;
…
get(object => container, item => item);
container := gb_buf (…);
get(object => container, item => item);
```

In the first call the elements buf, buf_in, buf_out are accessed in the implementation of the operation after the implicit conversion to the type bbuffer of the controlling operand container. In the second call after_put is accessible additionally to the inherited elements of the private part after the conversion to the type gb_buf.

The rules for modelling private types may seem a bit restrictive but they result from the considerations of modelling distributed objects. The concept especially is designed for preserving encapsulation and thus for supporting late binding of parallelism.

6.4 Extending the Extension

Having introduced the language extensions to VHDL it seems appropriate to list some features which are not part of the extension but which nevertheless might be useful for object-oriented modelling at system level.

They have been proposed in variants by various proposals to extend VHDL (Compare Section 5.2). It is only an exemplary selection of the most interesting proposals on non object oriented extensions for high-level modelling. They concern the following language features: genericity, channels, private types, dynamic processes, and nesting of packages. We briefly discuss each of the features in special regard to the presented language extension. It is investigated how they would interact with the concepts and mechanisms of the language extension.

6.4.1 Genericity

Genericity as it is provided by VHDL is limited to entities, components, block statements and to a certain degree to subprograms. The idea is to remove the limitations and to introduce generic types and to introduce generic clauses in package declarations and subprograms. A corresponding proposal was made in SUAVE [16]. The considerations on genericity in this section follow this proposal.

A generic type abstracts in a design unit, a block statement, or a subprogram from concrete types in an instantiation. It can be declared in a generic clause. It is possible to define some constraints of the generic type in the declaration. The constraints must be kept by actuals which might be associated with the generic type in an instantiation. The proposal in [16] mentions among other constraints the requirement that a type must allow an assignment, or the restriction that a type must be derived from a specific tagged type. If we think about the language extension presented in this thesis a constraint that requires a type to be a reference-free type also might be useful. Following the SUAVE proposal a declaration of a generic type in a generic clause might look as follows:

```
generic (type generic_type is private);
```

A generic type generic_type is declared which has the constraint that the type must have at least an assignment as a basic operation.

```
generic (type derived_generic_type is new parent with private);
```

This generic clause contains a generic type declaration with the constraint that the type must be derived from the tagged type parent. The design unit, block, or subprogram could use the information provided by the constraint. For example, it could call primitive operations inherited by the actual type that is associated to the generic type. The operations are inherited from the parent type according to the constraint. The actual in an association of an instantiation has to be a type derived from parent.

```
type actual_type is new parent with …;
…
generic map (derived_generic_type => actual_type);
```

The information which is provided by the abstraction of a type in form of the constraint is the decisive link between the genericity concept for types and the object-oriented concept of the language extension.

In VHDL it is possible to pass constants as parameters to subprograms. In the proposal to provide a genericity concept generic subprograms that may have generic parameters are added to the language extension. Such a generic subprogram is a template which can be instantiated wherever it is allowed to declare a subprogram. To properly interact with the tagged type concept of the language extension it is important that a generic procedure is not considered as a primitive operation of a tagged type. However, an instantiation of a generic subprogram certainly can be a primitive operation.

Introducing generic clauses in package declarations allows to abstract from constants or types in generic packages. A generic package serves as a template from which packages can be generated in a package instantiation. For example, such a generic package could be used to generalize a lock and unlock mechanism like the one presented in Section 5.2.5. A type lock_class provides two operations lock and unlock which block respectively unblock the execution of an operation of an object. A generic package may abstract from a parent of the tagged type lock_class.

```
package lock_package is
    generic (type generic_type is new root_type with private);
    type lock_class is new generic_type with private
        object_is_locked : boolean;
    end record;
    procedure lock (object : inout lock_class);
    procedure unlock ( object : inout lock_class);
end lock_package;
```

An instantiation generates a package with a derived type providing the primitive operations for the lock mechanism. Assume, the tagged type bbuffer is derived from the root type, then it would be possible to instantiate a new package lock_buffer_package with the type lock_class which inherits the primitive operations of bbuffer and adds the lock and unlock operations as new primitive subprograms.

```
package lock_buffer_package is new lock_package
                                              (generic_type => bbuffer);
```

It is interesting to note here that the newly created type does not automatically have the intended behaviour with respect to the lock and unlock mechanism. Although the new type has the lock and unlock operations the inherited implementations of the primitive operations get and put are not aware of them. Even though the tagged type and the generic type may not interact as intended in the example they successfully can be combined from a language design point of view.

So far we presented a proposal as part of a genericity concept how to add generic parameters to subprograms and package declarations. In a next step it would be possible to allow subprograms and packages themselves to become a generic parameter in other design units, blocks, or subprograms.

Like any other generic parameter a subprogram can be declared in a generic clause. When the formal parameters are associated with the actuals the subprogram is associated with an actual subprogram. In the SUAVE proposal the subprograms are required to have the same parameter and result type profile. If we introduced such a mechanism into the language extension of the thesis the requirement would not guarantee an error-free elaboration of all calls on the subprogram. It might occur that some parameter modes or parameter classes of the formal parameters are different from the actual ones. To make the proposal work with the language extension of the thesis it would be additionally required that in an association of a formal subprogram with an actual the corresponding parameters of the subprograms have the same mode and class.

To give a short example we could think of a generic bounded buffer package which abstracts from the type of the items stored in the buffer. Furthermore, the buffer might provide a primitive operation equal that returns a result parameter that is true if two objects contain the same number and sequence of items. This in principal corresponds to the example e_buf in Section 6.3.2. A subprogram that defines what equality means for the different item types is declared as a generic parameter of the package. It can be used in the operation equal of the buffer.

```
package generic_e_buf_package is
    generic
        (constant buffersize : positive;
         type item_type is private;
         function equal (left,right : item_type) return boolean);
    type bounded_buffer_array is array(positive range <>) of item_type;
    type e_buf is tagged record
        buf : bounded_buffer_array (1 to buffersize);
        …
    end record;
    procedure equal (   object1 : inout e_buf;
                        object2 : inout e_buf;
                        test_result: out boolean);
-- procedure equal can use function equal in its implementation
```

```
…
end generic_e_buf_package;
```

In an instantiation the subprogram is associated with the function which checks the equality of objects that are of the actual parameter for item_type.

```
package e_buf_package is
        new generic_e_buf_package
        generic map (buffersize => 4,
                     item_type => integer,
                     equal => "=");
```

The instantiation mechanism interacts with the object-oriented language extension by generating a package which contains a tagged type and its primitive operations.

The last extension to the genericity concept we are looking at is the modelling of packages as generic parameters. In principle, a generic parameter that is a package is described in a generic clause as a generic package that is so to speak derived from another generic package. Derived from another package means that any actual package that is associated to the formal package in the generic clause must be an instantiation of the generic package from which the formal is derived.

For example, it would be possible to pass instantiations of the generic package generic_e_buf_package as parameters to another generic package.

```
package another_generic_package is
    generic ( package e_buf_package is new generic_e_buf_package
              generic map (<>));
    -- other declarative items
    …
end another_generic_package;
```

In the example the generic parameter e_buf_package is required to be an instantiation of the generic package generic_e_buf_package. In an instantiation of the generic package another_generic_package the type of the items and the function which checks for equality are passed as actuals to the package. Like in the example before, a package is created which contains a tagged type and its primitive operations, however this time it is created inside another package. And again, the tagged type concept can be integrated in the genericity mechanism.

As shown by the example, passing packages as parameters only makes sense if the language allows the modelling of hierarchies of nested packages.

This requires an extension to VHDL as VHDL only allows the declaration of packages at library level.

6.4.2 Packages

In VHDL the design entity is the primary hardware abstraction. A package is a secondary part in a hardware model. Basically it serves as a collection of declarations which are shared by other design units. With the declaration of tagged types in packages they become a means to encapsulate class declarations in the language extension of the thesis.

There are ideas to make an even greater shift in the meaning of packages from a secondary concept to a central concept of encapsulation [9]. This shift is achieved by allowing package declarations to occur as declarative items in declarative parts of entities, architectures, block statements, generate statements, process statements, package declarations, and subprograms.

As it was briefly mentioned in the previous section the tagged type concept in principle interacts with the extended encapsulation concept of packages. The situation is somewhat different if a package is declared in an entity, block statement, generate statement, or process statement. The elaboration concept of such packages has to be extended as their elaboration does not any longer depend only on the elaboration of other packages. Assuming there is such an extended elaboration concept the question is how it would interact with the non-linear elaboration concept of class-wide types in the language extension. Another problematic issue concerns a potential weakening of the encapsulation caused by class-wide types.

Consider a tagged type that is declared in a package at library level together with its associated class-wide type. A derived type might be declared in a package that is a declarative item in an entity, architecture, etc. A class-wide object could be used to export the structure and behaviour declared locally in the entity, architecture, etc. This is against the fundamental encapsulation concepts for these language constructs.

The situation becomes even more complex if packages are allowed to occur as declarative items in subprograms. The dynamic elaboration of subprograms does not interact with the elaboration concept of class-wide types.

Consider a derived type that is declared in a package inside a subprogram and that is derived from a parent type declared outside the subprogram. A value of the derived type is assigned an out parameter that has a class-wide type associated with the parent’s type. After the execution of the subprogram call the caller receives a value of a type whose corresponding type declaration does not any longer exist. Although the information stored in the value would not be interpretable outside the subprogram the value nevertheless could be used in a dispatching call outside the subprogram. The effects of such a call would not be clear.

We can conclude that combining the tagged types approach with the possibility to declare non library-level packages would require additional concepts that remove the problematic issues raised above.

6.4.3 Extension to private types

Another extension to the encapsulation concept of packages proposed in [9] is the introduction of a visible part and a private part in a package declaration. The extension is proposed for the modelling of abstract data types. In the extension, the private part of a package allows to hide implementation details of types that are declared in the visible part of the package. The details may include the structure of the type and the operations characterizing the type. The declarations in the private part of a package are only visible in the corresponding package body. Such a concept would be a more general variant of the private type concept introduced in this thesis.

To illustrate the extension, the example of the bounded buffer bbuffer from Section 6.3.4 could be transformed to the extended version.

```
package bounded_buffer_package is
    type bbuffer is tagged private;
    procedure get (object : inout bbuffer; item : out item_type);
    procedure put (object : inout bbuffer; item : in item_type);
private
    type bounded_buffer_array is array(positive range <>) of item_type;
    type bbuffer is tagged record
        buf : bounded_buffer_array (1 to buffersize);
        buf_in : integer;
        buf_out : integer;
    end record;
end bounded_buffer_package;
```

In the example, the declaration of the structure of the tagged type is completely moved to the private part of the package. Only the bodies of the primitive operations get and put that are implemented in the package body have access to the declarations in the private part, i.e. the structure of the tagged type.

The private type bbuffer is marked as tagged and thus could be derived in another package.

```
use work.bounded_buffer_package.all;
package bounded_buffer_package_gb_buf is
    type gb_buf is new bbuffer with private;
    ...
private
    …
end bounded_buffer_package_gb_buf;
```

From the example the problem with such a modelling approach becomes clear. Although a use clause makes visible declarations of the package bounded_buffer_package directly visible in the package containing the derived type the structure of the parent type bbuffer is not visible to the derived type gb_buf.

As we have stated in Section 3.2.14 breaking the encapsulation of structure is a basic characteristic of any inheritance concepts that is different from pure delegation. However, the extended private type mechanism just prevents the derived type from breaking the encapsulation. What would be required for the private type mechanism to interact with the tagged type concept is a controlled and safe way for a derived type to break encapsulation of the structure of its parent type.

At that point it is interesting to note that the non object oriented language extensions described so far could be used in combination with the tagged type concept to model mixin inheritance which may serve as a surrogate for the missing multiple inheritance. The modelling principles of mixin inheritance were introduced in Section 3.2.13. We just sketch an example of how the concepts may interact to achieve mixin inheritance.

Consider the example of the lock_buffer_package that was introduced in Section 6.4.1. The example associated a tagged type as an actual parameter to a generic package and thus created a derived type in the new instantiation of the package. The problem was that inherited operations required a modification to correctly interact with new primitive operations but the instantiation mechanism for the package does not provide a mechanism for modifying them. The solution to the problem is mixin inheritance which encapsulates the instantiated package in a private part of another package and thus allows the overloading of inherited operations.

```
package lock_package is
    generic (type generic_type is new root_type with private);
```

```
        type lock_class is new generic_type with private;
        procedure lock (object : inout lock_class);
        procedure unlock ( object : inout lock_class);
    private
        …
    end lock_package;
    …
    package bounded_buffer_package_lb is
        type lb_buf   is new bbuffer with private;
        procedure get(object : inout lb_buf; item : out item_type);
        procedure put(object : inout lb_buf; item : in item_type);
    private
        package lock_buffer_package is new lock_package(bbuffer);
        type lb_buf is new lock_buffer_package.lock_class
                                                    with null record;
    end bounded_buffer_package_lb;
```

The derived type lb_buf inherits its properties from the types bbuffer and generic_type. The latter one is an instantiation of the declaration in the generic package lock_package.

The example illustrates how object oriented principles could be combined with other high-level modelling concepts to noticeably improve the modelling capabilities of VHDL.

6.4.4 Extended processes

A third proposal to improve the encapsulation concept of VHDL is to provide an abstraction boundary for processes [13]. Such an idea to add an interface to a process which abstracts the communication with other operations turns out to be basically a shorthand notation for a block containing a process.

A more problematic idea is to allow a dynamic elaboration of processes which might be elaborated as consequence of a sequential statement. The general difficulties of approaches that add concurrency to a sequence of statements in VHDL have been discussed in Section 5.2. The difficulties are not removed by combining the approaches with the tagged type concept. On the contrary, it would add all the unsolved issues on thread synchronisation and encapsulation to the approach that have been discussed in Section 3.3.2.

6.4.5 Communication

A similar problematic category of proposals concerns extensions that add new synchronisation and communication mechanisms to the language. As stated in Section 3.3, communication and synchronisation concepts in various forms of message passing mechanisms play a central role in object oriented modelling of concurrent systems. However, VHDL only provides a very low-level communication concept. In [98] the communication concept of VHDL is even classified as poor for system level description purposes. This is confirmed in Section 5.2 by the investigation of the proposals for communication that are part of object-oriented language extensions to VHDL. Apart from the protected object proposal none of the extensions could provide an abstract communication mechanism with a seamless integration into the existing synchronisation concept.

We now want to investigate proposals to extend VHDL by system level communication mechanisms with special respect to their potentiality for combination with tagged types. The analysis comprises among the protected objects approach, two more proposals for modelling communication at specification level.

The protected object proposal discussed in Section 5.2.4 provides a shared memory synchronisation which is based on a monitor approach. It can be used to establish indirect synchronisation and communication between processes. The proposal is described in detail in [91].

A protected object is modelled as a shared variable that has a protected type. A protected type is modelled by a protected type declaration and a so called protected type definition which is the corresponding body to the declaration. The protected type declaration consists of a set of subprogram declarations which declare the so called protected type subprograms. The protected type subprograms entirely characterize the protected type. They serve as an interface of a shared variable that has the protected type. The monitor mechanism of the protected type provides a mutual exclusion synchronisation between the protected type subprograms of the shared variable.

The subprogram bodies are defined in the protected type body which is associated with each declaration. Additionally to the subprogram bodies the protected type body may contain variable declarations. They are used to represent the shared data of a protected type, in other words, they model the structure of the type. The variables are encapsulated by restricting their scope to the body of the protected type in which they are declared. That means, only protected type subprograms are allowed to access the variables. The mutual exclusion synchronisation of the subprograms prevents the unwanted concurrent access to the shared data.

If the tagged type approach were integrated with the protected object proposal it would be possible to declare variables that have a tagged type in a protected object. The protected type subprograms which provide the mutual exclusion synchronisation would access the variables via the primitive operations. The protected type would be a wrapper that brings mutual exclusion synchronisation to the object of tagged type. This mechanism to provide mutual exclusion synchronisation would not only work for objects of tagged type but also for class-wide objects.

An example for such a modelling style would be a wrapper for an object of bounded buffer bbuffer.

```
type wrapper_of_bbuffer is protected
    procedure put (item : item_type);
    procedure get (item : out item_type);
end protected wrapper_of_bbuffer;
…
use work.bounded_buffer_package.all;
type wrapper_of_bbuffer is protected body
    variable encapsulated_bbuffer : bbuffer;
    procedure put (item : item_type) is
    begin
        put (encapsulated_bbuffer,item);
    end;
    procedure get (item : out item_type) is
    begin
        get (encapsulated_bbuffer,item);
    end;
end protected body wrapper_of_bbuffer;
```

Calling the protected type subprograms put and get of a shared variable guarantees mutual exclusive access to the buffer.[30]

```
shared variable shared_bbuffer : wrapper_of_bbuffer;
…
-- access from within one process:
    shared_bbuffer.put(item);
    …
```

30. Some details on initialization are omitted in the example for simplification.

```
    -- another concurrent access from within another process:
        shared_bbuffer.get(another_item);
```

The example illustrates how a protected type could be used to add mutual exclusion synchronisation to objects of tagged type. At the same time it shows the limitations of such a modelling style. If the tagged type is derived a new wrapper has to be modelled. It is not possible to derive the protected type. Another limitation is that the usage of a wrapper prohibits any reference semantics in the modelling of objects as any access-typed variables are completely encapsulated in a protected object. The modelling of polymorphic objects is restricted by the fact that dispatching calls are only possible indirectly via protected type subprograms.

The most severe limitation in that modelling style concerns the modelling of condition synchronisation. In the example, synchronisation prevents a concurrent access to the shared resource but it does not prevent a caller to call a get operation on an empty buffer or to block a caller until the buffer contains items. There is no guard mechanism and no re-queue mechanism which would allow an appropriate condition synchronisation for such a modelling problem[31]. The protected object proposal does not care about specification or modelling of pre-conditions on operations. Even if it cared about conditions synchronisation the modelling approach would not support re-use of the synchronisation code as it would not be part of the tagged type.

Adding synchronisation to a tagged type the other way round by dragging the protected object into a tagged type unfortunately does not work because protected types may not be used in composite types according to the protected object proposal.

Another idea of integrating the tagged type approach and the protected object proposal would be to have a new protected tagged type which combines features of both types the protected type and the tagged type. Such an idea apart from some notational details would very much correspond to the proposal on minimally extending VHDL by monitors and inheritance which has been discussed in Section 5.2.5. The limitations already have been sketched. Due to the missing assignment operation on protected types and due to the restriction that access type definitions must not depend on protected objects it would not be possible to model any kind of polymorphic objects. Another severe limitation of the new type would be the missing modelling concept for condition synchronisation. In conclusion, the introduc-

31. Such language concepts are considered to be too complex by the protected object proposal for inclusion in a first extension to VHDL

tion of a new type for the integration of tagged types approach and protected object proposal would not have a real advantage over the wrapper approach.

Both integration approaches would be possible, however none of them would be a general approach for modelling communication between processes or objects. The extended meaning of message as it was introduced in Section 3.3.7 also denotes requests to access an object that is wrapped in a protected object but the message passing approach lacks mutual exclusion synchronisation to be general.

In addition to protected objects we also could think of other shared resources that could be used for modelling communication. In Section 3.3.19 we stated that channels can be interpreted as a special kind of predefined shared object.

A message passing mechanism that is based on channels is proposed in [15]. It is a uni-directional channel concept for asynchronous message passing. The pros and cons of such a concept were discussed in Section 3.3.19. Messages are asynchronously sent to a channel and received by an explicit acceptance mechanism from the channel. Messages may contain data information which is sent uni-directionally from the sender to the receiver. The type of such an information is called message type. If the message does not contain any data information the information of the message is the event of sending the message. To avoid that an information sent to a channel gets lost if the receiver is not ready to receive a message the channel concept provides message queues for buffering messages. The message buffer is also used to solve access conflicts to channels. All messages that are sent concurrently to a channel are stored in an arbitrary order in the message queue.

Channels as abstraction of communication omit quantitative timing specification on messages and instead introduce a relative order of events corresponding to the order of messages in the buffer. On the receiver side the potential processing order of the events defines the interface control space. To support the modelling of the interface control space the channel concept has a guard mechanism. The guard mechanism allows to specify synchronisation conditions as part of the explicit acceptance mechanism for messages.

With its support for condition synchronisation the channel concept can be used for the synchronisation modelling of distributed objects. They can be modelled as sequential objects each having a thread that automatically could be invoked after the object's elaboration and that models the condition synchronisation. The automatically started thread receives the messages sent to the object and explicitly calls its corresponding operations while using the guard mechanism to model the synchronisation. If the object is modelled using tagged types it combines nicely with the channel concept for modelling condition synchronisation.

We illustrate this consideration by the example of the bounded buffer bbuffer. As described above the distributed version is modelled as an object that communicates via channels. One channel is used for each operation and for each result of an operation.

```
type get_instr is null channel;
type put_instr is channel of item_type;
type item_as_a_result is channel of item_type;
```

A message that has type put_instr contains data information that has type item_type. Likewise item_type is the message type of the channel type item_as_a_result that is used to model the reply scheduling of the get operation of the bounded buffer.

The channel approach requires a message type that is a composite type not to have an element that is an access type. This requirement can be met by using a private type from the tagged types approach. If the non-private part is empty the type is guaranteed not to contain any access type.

The tagged type itself is extended by a primitive operation which models the synchronisation code and which has to be invoked after the elaboration of the bounded buffer.

```
procedure thread (  object : inout bbuffer;
                    channel get_channel : in get_instr;
                    channel get_channel_reply: out item_as_a_result;
                    channel put_channel : in put_instr);
```

It can be called in a process which contains the buffer as a variable. In the process the thread is invoked after the elaboration of the object before any other object from outside the process may try to communication with the buffer. By modelling the call of the thread as the only sequential statement in the process the object becomes an active object.

```
process
    variable bbuffer_object : bbuffer;
begin
    thread (    object => bbuffer_object,
                get_channel => actual_get_channel,
                get_channel_reply => actual_get_channel_reply,
                put_channel => actual_put_channel);
end process;
```

It is interesting to see that no wait statement is required in the process. A receive statement inside the thread which accepts messages from the channel replaces wait statements as a synchronisation point. Among the receive statement the body of the primitive operation thread contains the complete synchronisation modelling of bbuffer.

```
procedure thread (   object : inout bbuffer;
                     channel get_channel : in get_instr;
                     channel get_channel_reply: out item_as_a_result;
                     channel put_channel : in put_instr) is
    variable item : item_type;
begin
    condition_synchronisation : select
        when  object.buf_in /= object.buf_out =>    -- buffer is not empty
                                receive from get_channel;
                                get (object, item);
                                send item to get_channel_reply;
        when   (object.buf_in - object.buf_out )
              mod ( 2 * buffersize ) /= buffersize =>-- buffer is not full
                                receive item from put_channel;
                                put (object, item);
    end select condition_synchronisation;
end;
```

In the body of the thread operation the messages are received by a select statement. The select statement is a language construct of the channel approach. It selects one of the messages from a channel referenced in the select statement. The statement only selects a message from a channel that has a corresponding guard expression that evaluates to true. In the example the guards test if the buffer is not full or if the buffer is not empty; accordingly put or a get message can be received by the select statement. After selecting a message the statements in the selected branch of the select statement are executed. If a message is received from the get channel the primitive operation get of bbuffer is invoked. The result is sent to the channel get_channel_reply.[32] Likewise the put operation is invoked if a put message is selected.

The example shows how it is possible to use the channel concept to model the synchronisation of distributed objects that have a tagged type. The

32. To keep the example simple, the buffer does not consider the client identity in the reply scheduling of the get operation. We may assume, only one client consumes items from the buffer in the example.

severe limitation of the presented modelling style is the impossibility to derive a new type without re-analysing the existing synchronisation code in the body of the primitive operation thread and completely re-write it. Re-analysing especially means to break the encapsulation of the parent's behaviour. We can conclude that the modelling style which combines the channel approach and the tagged type concept works but only provides a very limited support for re-use[33].

Another channel concept is proposed in [86] and explained in [58]. It describes a language extension which is called VHDL+. The extension is especially developed to support interface based design at system level. The channel concept behind the interface mechanism of the language is a bi-directional message passing concept that uses proxies to abstract its asynchronous message passing.

While the channel may be bi-directional a single message is always uni-directional. Each channel specifies which kind of messages are allowed to be sent in which direction across the channel. That means, different to the previous proposal it is possible to send more than one kind of message via a channel. Each kind of message may transport values that have particular types. Like in the previously presented channel concept a channel is viewed as a shared resource. Each kind of message which can be sent via the channel is viewed as a shared resource that is mutually exclusively accessed by a sender. If a message is sent to a channel while another message of the same kind is still processed by the channel the new message is queued until it can be processed.

A channel is described by a so called interface primary unit[34]. An interface lists the messages which can be sent via the channel. It distinguishes different kinds of connections to interaction points. In the terminology of VHDL+ these kinds of connections are called ends and the interaction points are called interface ports. The messages of a channel must not be sent arbitrarily between different interaction points but only between interaction points that have a connection of certain kinds. In other word, messages

33. Please note, this is not a particular problem of the channel concept but a general problem of active objects that model their synchronisation as part of their user-defined activity. We shall investigate this phenomenon later on in Section 8.1.1 in more detail.

34. Actually, the term channel is not used in the language description at all. Instead interface primary unit or interface is used which in the terminology of the thesis represent a channel.

between interface ports are only possible if the interface ports are connected to certain ends.
An example of an interface to a bounded buffer may illustrate this.

```
interface bbuffer_interface is
    between client, bbuffer_server;
    protocol is
        put_message;
        get_message;
        get_reply_message;
    end;
    message put_message (item : item_type) is
        from client to bbuffer_server;
    end message put_message;
    message get_message is
        from client to bbuffer_server;
    end message get_message;
    message get_reply_message (item : item_type) is
        from bbuffer_server to client;
    end message get_ reply_message;
end interface bbuffer_interface;
```

The ends of the interface are client and bbuffer_server. The protocol lists the messages which can be sent between these ends. The message declarations declare that the messages put_message and get_message only can be sent from client to bbuffer_server and that the message get_reply_message only can be sent from bbuffer_server to client. Further it declares that put_message and get_reply_message are used to send a value of type item_type[35].

With this interface it is possible to encapsulate an instantiation of a bounded buffer in an entity. The entity has interface ports as new interaction points which are connected to the bbuffer_server end of the interface.

```
use interface work.bbuffer_interface;
entity bbuffer_entity is
    interface port (object_interface: bbuffer_server of bbuffer_interface);
end entity bbuffer_entity;
```

35. Like in the previous example, reply scheduling of the message get_message through get_reply_message does not consider the identity of a client.

The architecture which contains the instantiation of the bounded buffer can receive the messages get_message and put_message from object_interface and it can send the message get_reply_message to object_interface.

If we want to combine the VHDL+ approach with the tagged types concept and implement the bbuffer as an object that has a tagged type bbuffer the problem is that it is not allowed in VHDL+ to send and receive messages from within subprograms. In other words, it is not possible to write a primitive subprogram thread like in the previous example to model the synchronisation via the channel.

Instead, it is necessary to model the synchronisation in a process. As a consequence, the actual synchronisation states of the bbuffer must be accessible from within the process. This is achieved by declaring an enumeration type which lists the synchronisation states of bbuffer and by declaring a primitive operation which returns the actual synchronisation state.

```
type bbuffer is …
…
type synch_states_of_bbuffer is (empty, partial, full);
procedure actual_synch_state (object : inout bbuffer;
                              state : out synch_states_of_bbuffer);
```

As a process must not immediately contain the constructs of the language extension that are employed to model the synchronisation it is necessary to encapsulate them in a special construct of VHDL+ that is called activity[36]. The activity is called in a process like a subprogram.

```
architecture behaviour of bbuffer_entity is
    activity encapsulate_synchr (object : inout bbuffer) is
        variable synch_state_of_object : synch_states_of_bbuffer;
        variable item : item_type;
    serial
        actual_synch_state (object,synch_state_of_object);
        case synch_state_of_object is
            when empty =>
                receive object_interface.put_message(item);
                put (object, item);
            when partial =>
                action
```

36. We do not consider the general properties of activities like concurrency concepts etc. in the example. We only use activities for encapsulation without any further function.

```
                    when receive object_interface.put_message(item) =>
                        put (object, item);
                    when receive object_interface.get_message =>
                        get (object, item);
                        send get_reply_message(item);
                end action;
            when full =>
                receive object_interface.get_message;
                get (object, item);
                send object_interface.get_reply_message(item);
        end case;
    end activity encapsulate_synchr;
    process
        variable bbuffer_object : bbuffer;
    begin
        encapsulate_synchr (bbuffer_object);
    end process;
end;
```

In the example, the synchronisation is modelled in the activity encapsulate_synchr. This activity consists of a sequence of sequential statements. The first sequential statement calls the primitive operation actual_synch_state of the tagged type bbuffer and reads the synchronisation state of the object which models the buffer and which is passed as a parameter of type bbuffer to the activity. The execution of the following synchronisation code depends on the synchronisation state. If the buffer is empty the activity waits until it receives a put_message on the object_interface. After receiving the message it performs the actual put operation on the bounded buffer by calling the primitive operation put of bbuffer. It is the receive statement which is only allowed to occur in processes or activities and which must not be used in primitive operations. The receive statement acts as a new synchronisation point in the language extension. In case the buffer is full the activity waits until a get_message arrives. It executes the get operation and sends the result to the interface. The message is sent blocking the sender until it exclusively can access the message get_reply_message of the interface bbuffer_interface via its end bbuffer_server. The end is viewed as a shared resource in the way mentioned above. The interface serves as a proxy in the asynchronous message passing.

If the buffer is not full nor empty the buffer is in the state partial. In that state both messages, put_message and get_message are accepted. Which message is accepted depends on which one arrives first. Such a situation is modelled in VHDL+ by a statement which is called action[37]. It is similar to the select statement of the SUAVE language extension. Depending on the message it receives it performs a sequence of statements associated with the receive statement, i.e., the put and get operations. The difference to the select statement is that it is not possible to guard a receive statement.

The bounded buffer object itself is instantiated as a variable in a process which runs the synchronisation by calling the activity which actually contains the synchronisation code of bbuffer. The impurity of such a synchronisation modelling is that a type of an object is not characterized by its synchronisation code. The synchronisation rather has to be modelled for each object separately in the architecture where the object is instantiated. In VHDL+ activities are only declared in architectures. They cannot be declared in packages. This drastically reduces the possibility to re-use synchronisation code of objects in a model. These difficulties in declaring synchronisation as a property of a tagged type are a major obstacle for modelling by extension.

The investigation of the VHDL+ modelling philosophy shows that it is possible to combine tagged types with the communication mechanisms of VHDL+ as far as language issues are considered but that an integration of the concepts is problematic with respect to modelling styles which support re-use and modelling by extension.

The syntax and semantics of VHDL+ is defined in a way that allows a translation from VHLD+ models to standard VHDL models by a pre-processor. There already exists a tool for that pre-processing [58]. At the same time there is a proposal how to pre-process tagged types and translate them into standard VHDL constructs. (Compare Section 6.5.1). It would be interesting to see if it is feasible to integrate both pre-processor approaches. However, detailed information about the pre-processor mechanisms for VHDL+ which is required for such an investigation is not yet published.

If we summarize the main points of this section we can state that the tagged types approach can be combined with proposals for language extensions that add new communication and synchronisation concepts to the language. The combination would not introduce any syntactic or semantic inconsistencies into the language. The missing point of contact between the object-oriented concepts and the communication mechanisms results in the

37. There is also another receive statement in the language extension which is able to receive a get or a put message and which could be directly used in a process. However, it would not be possible to associate particular sequence of statements which the different messages received.

assumption of many language designers that object-oriented features in form of for example tagged types are orthogonal to communication concepts[38]. However, none of the presented extensions for high-level communication and synchronisation is suitable for modelling the message passing between distributed objects that have tagged type. All concepts abstract the communication between objects. However, the abstractions are only well suited for use-relation between objects but they are not appropriate for the modelling of is-a relations between classes. In other words, the language extensions do not provide any support for the modelling of re-usable and extendable synchronisation code.

6.5 The Translation

The previous sections described the proposal of this thesis to extend VHDL. The syntax and semantics of the language extension have been presented in detail. It was mentioned that the language extension could be integrated in a design flow by a pre-processing step which translates models using the object-oriented features of the language extension into models using only standard VHDL. The language features which have been presented in Section 6.2 and Section 6.3 were identified as appropriate for such a translation mechanism. We now introduce a synthesis semantics which describes the transformation from an object-oriented level of abstraction to an algorithmic level.

6.5.1 The pre-processor approach

We start by making some general remarks on pre-processor approaches to translate a language extension of VHDL. In VHDL and likewise in an extension a model is structured into design units which can be analysed separately into design libraries. The environment in which a design unit is analysed is defined in terms of design libraries by library clauses. Thus, if we translate a language construct of the extension which depends on its environment, i.e.,

38. These reflections on language extensions manifest in different study and working groups of the Design Automation Standards Committee (DASC) of the IEEE Computer Society which is in charge of the standardization effort for VHDL. A working group elaborates a monitor concept, a study group discusses the communication proposals, and another study group studies with the object-oriented language extension.

on the content of some design libraries it is not possible to perform the pre-processing by textual replacements on a per design unit basis. It is rather necessary to process the content of the design units which are extended by the capability to store the results of the analysis of the extensions. The result of the processing then can be written to a set of new VHDL design units.

The new design units may contain some new identifiers automatically generated by the translation process. The generation must take care that the new identifiers do not conflict with existing ones. This could be achieved for example by modifying existing identifiers in a way that allows to distinguish them from the newly added ones.

The limitation of such a pre-processor approach is the missing possibility to directly simulate a model written in the language extension. The introduction of new objects as a translation result and the renaming of exiting identifiers makes it difficult to trace a simulation in the original model. Therefore it is important to generate readable target code as a translation result.

6.5.2 Translation of tagged types

The most central construct in the object-oriented language extension of this thesis is the tagged type. It is a composite type which may contain sub-elements of different types. In other words, it is a special kind of record type. Thus it suggests itself to translate a tagged record as a normal record. The normal record contains all the sub-elements of the tagged record. If the tagged type does not contain any record element then a dummy element is introduced into the record. The implicitly declared tag field of a tagged record is omitted in the normal record. As an object that has a tagged type must not change its type during simulation the value of the tag field can be statically determined at analysis time and modelled as a constant in the translated code where it is required.
The example of the tagged type bbuffer was introduced in previous sections.

```
type bbuffer is tagged record
private
    buf : bounded_buffer_array (1 to buffersize);
    buf_in : integer;
    buf_out : integer;
end record;
```

It translates as follows[39]:

```
type bbuffer is record
    buf : bounded_buffer_array (1 to buffersize);
    buf_in : integer;
    buf_out : integer;
end record;
```

In the target code it makes no difference if an element declaration of a record declaration originally was in a private part or in the non private part of the tagged record declaration.

At that point we want to give some general remarks on encapsulation properties and the corresponding visibility rules of the new language features with respect to translation. Considerations on encapsulation of objects in the target code are not relevant from a code generation point of view. Checking the visibility rules is part of the parsing and the semantic analysis but not of the code generation. That means that if a model passes the parsing and the semantic checks it can be translated without worrying about encapsulation. The target code might contain declarations with an extended scope compared to the scope of the corresponding declaration in the source code but it does not matter as no references in the extended parts of the scope are introduced by a translation.

To come back to the example that means that the record elements of objects of bbuffer are not referenced outside its primitive operations. The operations in the target code that correspond to the primitive operations in the source code are either implicitly declared with the type declaration of the record like its basic operations or they are translated by replacing the subtype indications referencing tagged types by indications referencing the corresponding translated types.

In the example of the bounded buffer the translation is simple as the renaming of identifiers, i.e., the renaming of the type names is omitted.

```
procedure get (object : inout bbuffer; item : out item_type);
procedure put (object : inout bbuffer; item : in item_type);
```

With the record type having the same name as the original tagged type the procedures can be taken without any modifications.

39. To make the translation results in the example as readable as possible the examples omit the renaming of identifiers which was mentioned in the previous section.

6.5.3 Translation of derived types

A derived type is a tagged type that inherits properties from its parents and that might have some additional properties. The properties are expressed in terms of structure and behaviour of a tagged type. We first look at the translation of the structure. A derived type is translated as a record type which contains all the record elements of its parents and possibly some new elements. All record elements means that a potential dummy element of a parent in case of a tagged record containing no elements also appears in the element list of the derived type. The new elements of the derived type possibly require a renaming in the translation to avoid potential naming conflicts with record elements from other derived types which have a common parent. Without renaming such a conflict might occur in the translation of class-wide types as we shall see later on.

With this translation concept the translation of the derived type example gb_buf looks as follows:

```
type gb_buf is new bbuffer with record
private
    after_put : boolean;
end record;
```

The translation result from this declaration is:

```
type gb_buf is record
    buf : bounded_buffer_array (1 to buffersize);
    buf_in : integer;
    buf_out : integer;
    after_put : boolean;
end record;
```

An alternative translation concept for the structure of derived types would be to translate the inherited structure as a new record element that has the parent type. A variant of such a concept was proposed in [175]. In the example the translation result would look as follows:

```
type gb_buf is record
    parent_structure : bbuffer;
    after_put : boolean;
end record;
```

For the modelling of objects that have derived types the alternative would not make much difference. In the following we assume that the original translation proposal is the basis of the rest of the translation schema.

The derived type and its parent are closely related types with an explicit type conversion between them. The explicit type conversion can be translated as a call to a conversion function from the derived type to its parent type. The implementation of the conversion function assigns the values element-wise ignoring those elements which are only in the derived type.

In the example an explicit conversion bbuffer can be translated as a call to the conversion function to_bbuffer which is implemented like this:

```
function to_bbuffer (object : gb_buf) return bbuffer is
    variable result : bbuffer :=   (buf => object.buf,
                                    buf_in => object.buf_in,
                                    buf_out => object.buf_out);
begin
    return result;
end;
```

The function call can replace the conversion in expressions or associations where it is legal to use a conversion. A typical example of such a legal use would be a port map with an association list that contains type conversions.

Among its structure a derived type also inherits the behaviour of its parents. The inherited subprograms are translated by a declaration of the subprograms in which the formal parameter list is modified so that the derived type is characterized by the subprogram. Each subtype indication of the tagged type in the formal parameter list is replaced in the translation by a subtype indication of the derived type.

In the example the put and get operations of bbuffer are inherited by gb_buf. Accordingly the formal parameter list is modified in the subprogram declarations of the translation:

```
procedure get (object : inout gb_buf; item : out item_type);
procedure put (object : inout gb_buf; item : in item_type);
```

During the execution of an inherited primitive operation it might be necessary to perform implicit type conversions to avoid type mismatches.

An example might be an assignment to a local variable that has the parent's type.

```
procedure put (object : inout bbuffer; item : in item_type) is
    …
    variable local_copy_of_object : bbuffer := object;
begin
    …
end;
```

After inheriting the operation put the derived type gb_buf determining the initial value of the local variable in the procedure requires an implicit type conversion from the derived type to the parent.

```
procedure put (object : inout gb_buf; item : in item_type)  is
    …
    variable local_copy_of_object : bbuffer := to_bbuffer(object);
begin
    …
end;
```

In many cases an implicit conversion could be translated as a call to the conversion function which was defined to translate explicit conversions. There are a few situations which require a slightly different translation.

If an implicit conversion is required in an assignment where the type of the right-hand side expression is the parent type and the target has a derived type then the target has to be modelled as an aggregate. The aggregate just contains the elements of the target which have corresponding elements in the derived type. To avoid any ambiguities which may result from this translation a type conversion is applied on the expression.
An example taken from the procedure put may look like this:

```
(object.buf,object.buf_in,object.buf_out):= bbuffer(local_copy_of_object);
```

Similar situations which might not allow the use of conversion functions to translate implicit conversions are association lists of subprogram calls which associate parameters of class signal, parameters of mode inout, or parameters of mode out. In such situations the implicit conversion is translated as an individual association of the elements of the formal with the corresponding elements of the actual. Elements of the actual which do not have a corresponding element in the formal are ignored in the association. It might be necessary to resolve ambiguities in overloaded subprograms which could be introduced by the translation by renaming of some subprograms.

An example might be a call to an operation equal which is similar to the procedure equal introduced in a previous example.

```
procedure put (object : inout bbuffer; item : in item_type) is
    …
    procedure equal (    object1 : inout bbuffer;
                         object2 : inout bbuffer
                         test_result: out boolean) is …
    variable local_copy_of_object : bbuffer := object;
    variable objects_are_equal : boolean;
begin
    …
equal (object, local_copy_of_object, objects_are_equal);
    …
end;
```

A call on the procedure in the inherited subprogram put of the type gb_buf requires an implicit conversion of the object to the type bbuffer. As the parameter mode is inout a conversion function is not appropriate as a translation of the association. Instead, an individual association performs the conversion.

```
equal (    object1.buf => object.buf,
           object1.buf_in => object.buf_in,
           object1.buf_out => object.buf_out,
           object2 => local_copy_of_object,
           test_result => objects_are_equal);
```

Up to now we discussed in this section how to translate primitive subprograms which are inherited by a derived type. The derived type also may modify behaviour by re-defining these inherited primitive operations. In such a case the inherited subprogram is re-named in the translation. We could rename it by adding a suffix _parent to its name.

In the example we could re-define the primitive operations put and get of the derived type gb_buf. Thus the operations are declared as follows:

```
procedure get_parent (object : inout gb_buf; item : out item_type);
procedure put_parent (object : inout gb_buf; item : in item_type);
procedure get (object : inout gb_buf; item : out item_type);
procedure put (object : inout gb_buf; item : in item_type);
procedure gget (object : inout gb_buf; item : out item_type);
```

A call on the overridden subprogram by using the attribute 'Parent can be translated as a call on the re-named operation.

In the example a call on the parent's implementation of the put operation is translated as a call on the re-named inherited subprogram put_parent.

Another predefined attribute of the language extension which can be used to decorate a subprogram call is the attribute static. It forces a call to be statically bound different from the normal dynamic binding mechanism. In a translation of the attribute a simple name denoting the subprogram is replaced by the corresponding selected name. Any implicit type conversions which might be required in the association list of the call are translated according to the mechanisms described above.

The combination of the attributes 'Parent and 'Static can be translated as a call which uses the selected name denoting the overridden operation.

The attribute 'Dynamic as the counterpart of 'Static can be ignored in a translation.

Consider a static call to the overridden version of the get operation in the implementation of the get operation of the type gb_buf as an example.

```
procedure get (object : inout bbuffer; item : out item_type) is
begin
    get'parent(object,item)'static;
    object.after_put := false;
end;
```

The static call with its implicit type conversion is translated as

```
library_name.package_name.get_parent(
                                    object.buf => object.buf,
                                    object.buf_in => object.buf_in,
                                    object.buf_out => object.buf_out,
                                    object.after_put => object.after_put,
                                    item => item);
```

If the procedure is inherited by a type which is derived from gb_buf the call is statically bound due to the selected name which denotes the procedure get_parent of gb_buf. The individual association automatically performs any required implicit type conversion.

6.5.4 Translation of class-wide types

The translation of a class-wide type is different from the translation of a tagged type. This mainly results from the fact that a class-wide type has a tag which may change during simulation whereas the tag of a tagged type is a constant value.

A class-wide type is implicitly defined for each tagged type. It is associated with that tagged type. The inheritance hierarchy defines a sub-tree for such a tagged type with the tagged type as a root. The class-wide type associated with the tagged type is translated as a record type which contains all record elements of all the types that are in the sub-tree that has the tagged type as a root. The record type also contains a tag field as a record element that has a subtype of the predefined type tag.

Before we continue to describe a translation concept for class-wide types we first have a look at the implementation of the predefined type tag. This type is declared in the package OBJECT_ORIENTED_EXTENSION which is part of the language extension. The type tag is declared to be implementation dependent. An implementation could implement it as an enumeration type with each enumeration item corresponding to a tagged type. For each inheritance hierarchy of a tagged type a sequence of enumeration items is defined which is part of the enumeration type definition of tag. A sequence is obtained by traversing the type tree of a tagged type in post order. With such a sequence in post order all enumeration items that correspond to types which have a common root can be described as a subtype of type tag. The benefit of such an ordering is that a class-wide type which is associated with that root can just contain the values of this subtype in its tag field. A translation may chose just such an implementation without any further modifications.

We may illustrate the concept by a possible implementation of type tag for the examples in this thesis.

```
type tag is    (x_buf2_tag, buf2_tag, gb_buf_tag, backup_buf_tag,
                e_buf_tag, ce_buf_tag, bbuffer_tag);
```

Various subtypes of type tag contain possible values for the tag field of corresponding class-wide types:

```
subtype buf2_tag_subtype is tag range x_buf2_tag to buf2_tag;
subtype gb_buf_tag_subtype is tag range gb_buf_tag to gb_buf_tag;
subtype bbuffer_tag_subtype is tag range x_buf2_tag to bbuffer_tag;
```

Actually it is not necessary to declare the subtypes explicitly. It is sufficient to declare them by a constraint in the element declaration of the tag field.

The translation of the class-wide type gb_buf'Class may serve as an example.

```
type bbuffer_classwide is record
    tag_field : tag range x_buf2_tag to bbuffer_tag;
    buf : bounded_buffer_array (1 to buffersize);
    buf_in : integer;
    buf_out : integer;
```

```
        after_put : boolean;
    end record;
```

For each tagged type there are implicit conversions from the tagged type to any class-wide type associated with the tagged type or associated with one of its parents. The conversions can be translated as calls on conversion functions. The translation approach is very similar to the previously described ones for implicit conversions between tagged types. Corresponding conversion functions have to be declared for each tagged type. The new thing in the conversion functions is the value of the tag field. It is passed as a result of the conversion to a class-wide object. The value of the tag field in the result just indicates the type of the parameter of the conversion function. The values in the other elements of the result are obtained by assigning the values of the elements in the parameter to the corresponding elements in the result. Record elements of the result which do not have a corresponding element in the parameter are implicitly assigned some default values[40].

To translate an implicit conversion in an assignment that has a right-hand side value of class-wide type and a target of tagged type a conversion function with a class-wide object as a parameter and a tagged type as the type of the target also has to be declared. In the implementation of this conversion function a check has to test if the tag of the parameter indicates the type of the target. If the check fails a run-time error has to be reported. If the check is passed the values of the elements if the parameter which have a corresponding element in the result can be assigned to the result.

A third kind of conversion in the context of class-wide types is an explicit conversion of a class-wide type to a tagged type. Such a conversion corresponds to an implicit conversion from the class-wide type to the tagged type indicated by its tag field and a conversion between closely related tagged types. Accordingly the conversion function which is used as a translation of the conversion is implemented. The tag field is checked if it indicates the type of the target or a type derived from the target type. If the check fails a

40. Assigning default values has an important effect in signal assignments that have a class-wide target and that perform an implicit conversion. The driver of such a class-wide signal may have scheduled events to occur on some of its elements in the future. A signal assignment may change the tag of a target so that some of the elements do not belong to a valid value of the target any longer. The assignment of the default values remove any future events from the invalid elements.

run-time error occurs otherwise the elements are copied like in the other conversion functions.

A fourth kind of conversions is the conversion between class-wide types. Essentially, this is an implicit conversion to a tagged type followed by an implicit conversion from the tagged type to the class-wide type. The translation idea is similar to the translation in the case of an explicit conversion. A conversion function checks if the tag field indicates a tagged type that can be implicitly converted to the class-wide type of the target. In case the check fails a run-time error occurs otherwise the elements are copied like in all the other conversion functions.

We now want to give examples of such conversion functions which are used to translate conversions between class-wide types and tagged types. The corresponding examples where to apply such conversions are taken from Section 6.3.1.

```
variable container : bbuffer'Class := bbuffer'(…);
variable another_container : gb_buf'Class := gb_buf'(…);
variable bbuffer_object : bbuffer;
variable gb_buf_object : gb_buf;
…
container := gb_buf'( …);
gb_buf_object := container; -- implicit conversion with dynamic check
bbuffer_object := bbuffer(container); -- explicit conversion
another_container := container; --conversion between class-wide objects
```

The first example is a conversion function which is used to translate an implicit conversion from bbuffer to bbuffer'class.

```
function to_bbuffer_classwide(object:bbuffer)
                                        return bbuffer_classwide is
   variable result : bbuffer_classwide;
begin
   result.tag_field := bbuffer_tag;
   result.buf := object.buf;
   result.buf_in := object.buf_in;
   result.buf_out := object.buf_out;
   -- result.after_put := "some initial value";
   return result;
end;
```

The translation of the initial expression with its implicit conversion is as follows:

```
variable container : bbuffer_classwide :=
                                    to_bbuffer_classwide(bbuffer'(…));
```

The second example is the implicit conversion from the class-wide type bbuffer'Class to the tagged type gb_buf which performs a dynamic check during simulation if the tag of the class-wide object indicates the type of the target. The corresponding conversion function for the translation can be implemented like this:

```
function to_gb_buf (object : bbuffer_classwide) return gb_buf is
    constant result : gb_buf :=  (buf => object.buf,
                                  buf_in => object.buf_in,
                                  buf_out => object.buf_out,
                                  after_put => object.after_put);
begin
    assert (object.tag_field = gb_buf_tag)
    report ("Illegal implicit conversion to type gb_buf")
    severity error;
    return result;
end;
```

The implicit conversion in the assignment of the example can be translated by a call on the conversion function.

```
gb_buf_object := to_gb_buf (container);
```

The example of the explicit conversion is very similar to the previous translation of the implicit conversion. The difference is that a whole set of tags are valid tags for the conversion.

```
function explicit_conversion_to_bbuffer (object : bbuffer_classwide)
                                                 return bbuffer is
    variable valid_tag : tag range x_buf2_tag to bbuffer_tag;
    variable result : bbuffer :=   (buf => object.buf,
                                    buf_in => object.buf_in,
                                    buf_out => object.buf_out);
begin
    -- check that tag_field contains a valid tag for the conversion:
    valid_tag := object.tag_field;
    return result;
end;
```

Correspondingly the translation is:

```
bbuffer_object := explicit_conversion_to_bbuffer(container);
```

The conversion function detects any illegal conversion during run-time by a violation of the range constraints in the assignment to the variable valid_tag[41].

The conversion function to translate the implicit conversions between class-wide types uses the same implementation idea as the conversion function to translate the explicit conversion.

```
function to_gb_buf_classwide (object : bbuffer_classwide)
                                        return gb_buf_classwide is
   variable result : gb_buf_classwide :=(
                        -- implicit check in assignment of tag_field
                        -- that it contains a valid tag for the conversion:
                                        tag_field => object.tag_field,
                                        buf => object.buf,
                                        buf_in => object.buf_in,
                                        buf_out => object.buf_out,
                                        after_put =>object.after_put);
begin
   return result;
end;
```

The corresponding translation is:

```
another_container := to_gb_buf_classwide (container);
```

Illegal conversions are detected during run-time by the range-violation in the conversion function.

6.5.5 Translation of a dispatching call

A call on a primitive subprogram with controlling operands of class-wide type dispatches to the subprogram of the tagged type that is indicated by the tag field of the controlling operands.

To translate such a call there is a subprogram for each primitive operation which performs the dispatching. The subprogram declaration is taken from the primitive operation. Each type indication of a parameter that could be associated with a controlling operand is replaced by a subtype indication of the translation of the class-wide type. The declaration is renamed to distinguish it from the corresponding primitive operations and to avoid unwanted

41. The checks can be omitted in cases where the inherent implicit conversion between the closely related types is guaranteed to be towards the root of the corresponding type tree.

overloading. The implementation of the subprogram reads the values in the tag fields of the parameters that have mode in or inout and that are associated with the controlling operands. If there is more than one such parameter a check is performed to ensure that all tag fields contain the same value. If the check fails an error occurs in the simulation. If the check is passed without error the value of the tag field selects the call on the corresponding subprogram in a case statement. In this subprogram call the elements of the parameters that are associated with the controlling operands are associated individually to the formal parameters of the primitive operation. Only those elements from the class-wide actuals that are in the tagged type of the formal are associated. If we compare the mechanism to previously discussed concepts we can see that it just corresponds to a translation of an implicit conversion.

We chose the subprogram as an example which implements the dispatching of the subprogram get of bbuffer if the subprogram is called with a controlling operand of class-wide type bbuffer'Class.

```
procedure dispatching_get  (object : inout bbuffer_classwide;
                            item : out item_type) is
begin
    case object.tag_field is
        when bbuffer_tag => -- dispatch to primitive op get of bbuffer:
                work.….get   (object.buf => object.buf,
                              object.buf_in => object.buf_in,
                              object.buf_out => object.buf_out,
                              item => item);
        when gb_buf_tag => -- dispatch to primitive op get of gb_buf:
                work.….get   (object.buf => object.buf,
                              object.buf_in => object.buf_in,
                              object.buf_out => object.buf_out,
                              object.after_put => object.after_put,
                              item => item);
        when …
    end case;
end;
```

A dispatching call with a controlling operand that has a class-wide type is translated as a call on the corresponding dispatching subprogram. Taking the container from the previous example as a controlling operand in a call on the get operation causes a dispatching call.

```
get (container, item);
```

The translation of that call is:

```
dispatching_get (container, item);
```

As the parameter object is the only formal parameter that is associated with the controlling operand container a check to ensure that all tag fields contain the same value is omitted in the example.

6.5.6 Translation of the attribute 'Tag

The predefined attribute 'Tag of the language extension exists in two versions. In the first version it decorates identifiers denoting a tagged type. In the second version it decorates class-wide objects. In both cases the attribute denotes a function which returns a value of type tag that indicates the type of the tagged type or the class-wide object.

The first version of the attribute is translated by the enumeration literal of the type tag which represents the tagged type used as the prefix of the attribute. For example, bbuffer'Class is translated as bbuffer_tag.

The second version is translated as a call on a function which uses the prefix as an operand and which returns the value of the operand's tag field[42].

```
function tick_tag (object : bbuffer_classwide) return tag is
begin
    return object.tag_field;
end;
```

An if statement might be used to prevent illegal implicit conversions in the previous example:

```
if container'Tag = gb_buf'Tag then
            gb_buf_object := container;
end if;
```

The translation is:

```
if tick_tag (container) = gb_buf_tag then
            gb_buf_object := to_gb_buf (container);
end if;
```

42. If the language extension was used as an extension to VHDL'87 the translation would not properly work for wait statements with an empty sensitivity clause which use the attribute in the condition of the wait statement. This is due to the different rules how to create a sensitivity set in VHDL'87 and in VHDL'93.

The attribute must not have a prefix that denotes an object that has a tagged type and thus there is no translation for such an attribute.

6.5.7 Translation dependencies

So far we looked at the distinct translation mechanisms separately. Each translation step was described nearly independently from others. It was explained which type declarations, subprogram declarations, and function declarations are added to a model during translation. However, it was not mentioned where to declare the new named entities. In this section we describe how to split up a model into separate design units during the translation in order to find a place where to declare the new named entities.

The new named entities have to be declared in a place where they do not produce dependencies which would not allow to find an order of analysis and elaboration for the model. In particular, a class-wide type depends on the type names referenced in the element declarations of the corresponding tagged types. It depends on these type declarations which especially might occur in all the packages containing the tagged type declarations. Such a potential dependency prevents the declaration of the record type which translates a class-wide type to be declared in the package containing the translation of its associated tagged type.

Consider a declaration of the derived type gb_buf where the type of the record element after_put is declared in the package containing the type declaration of gb_buf as an example.

```
use work.bounded_buffer_package.all;
package bounded_buffer_package_gb_buf is
    type invocation_history is …;
    type gb_buf is new bbuffer with record
        after_put : invocation_history;
    end record;
    …
end bounded_buffer_package_gb_buf;
```

The package bounded_buffer_package which contains the type bbuffer must be analysed before the package containing the derived type gb_buf can be analysed. The packages containing the translation of the tagged types must have the same order of analysis. At the same time, the translation of class-wide type bbuffer'Class requires the type invocation_history to be analysed before it can be analysed. The translation of the class-wide type therefore cannot occur in the same package as the translation of the tagged type bbuffer. Otherwise this would introduce a circle in the analysis dependencies.

Another situation which could cause circles in the dependencies occurs if a class-wide type is referenced in the packages containing the corresponding tagged types.

To break such circles in the dependencies, in Section 6.3.1 a complex set of rules concerning the elaboration of class-wide types have been presented. Following the ideas behind these rules it is possible to derive a concept how to split a package containing a tagged type declaration into several packages during the translation.

To translate a package containing a tagged type the primitive operations are translated in a separate package. The remaining declarations including the translation of the tagged type itself that do not directly or indirectly reference the translation of the class-wide type are moved into a package and those which have a reference into another. The package containing the translation of the tagged type additionally contains the conversion functions which are used as a translation of the conversion between the tagged type and its parent types.

The translation of the class-wide type associated with the tagged type is declared in a package which may use the packages containing no references to the class-wide type. Additional to the class-wide type the package also contains the conversion functions required by the class-wide type for the translation of implicit and explicit conversion.

The subprograms of the translation of the class-wide type implementing the dispatching can be declared in the package which contains the declarations with the references to the class-wide type. The package always has to use the package containing the translation of the class-wide types and it might use the package containing the translation of the tagged type.

The packages containing the translation of the primitive operations may use all the other package declarations.

The package bodies of the declarations described above essentially contain the subprogram bodies and probably some additional declarations which are used in the subprogram bodies. If subprograms using the same local declarations are partitioned into different packages it is necessary to have multiple copies of this local declarations in the different packages[43]. The same

43. The special case that a local declaration of a shared variable is referenced is not considered here. Anyway, using such a construct produces highly non-portable code.

situation occurs if an inherited primitive operation uses local declarations in its implementation. It requires local copies of the declarations in the body of the package which contains the translation of the derived type which inherits the primitive operation[44]. In such a case it might be necessary to rename some of the local declarations and their references to avoid naming conflicts with other declarations that are visible in the package body.
Figure 5 illustrates the dependencies between the split packages.

The analysis order for the package directly follows from the use-relation between the packages. Finally, the corresponding package bodies may use all the package declarations mentioned above. Therefore the analysis of the corresponding package bodies is deferred until all package declarations are analysed.

6.5.8 Translation of the postponed use-clause

Another construct of the language extension which influences the order of elaboration is the postponed use-clause which was discussed in Section 6.3.3. A postponed use-clause makes a name of a tagged type visible that is not necessarily elaborated. It allows the type's elaboration to be postponed. The name referenced in such a postponed use-clause is only allowed as a subtype indication in the interface list of a primitive operation. This allows a simple translation of the use-clause as the translation result of the primitive subprograms and tagged-types are split into separate packages anyway. The postponed use-clause is translated by a normal use-clause which references the translation of the tagged type. The reference is possible because the analysis of the packages containing the translations of the primitive subprograms can be deferred until all the packages containing the translations of the tagged types are analysed[45]. This was explained in the previous section in detail (See also Figure 5).

The use clause also has to be added to the package which contains the subprograms for the dispatching as the type referenced in the use clause also occurs in the parameter lists of the subprograms. In rare cases where this use clause causes circular dependencies due to other type declarations in the

44. As explained in Section 6.2.1 primitive operations are only declared in package declarations. Thus, the concept of local copies does not interfere with the dynamic binding of primitive operations in subprograms of derived types.
45. Remember, the only reason why such a deferment is not already modelled in the source code is the requirement that primitive operations must be declared together with the tagged type in the same package declaration.

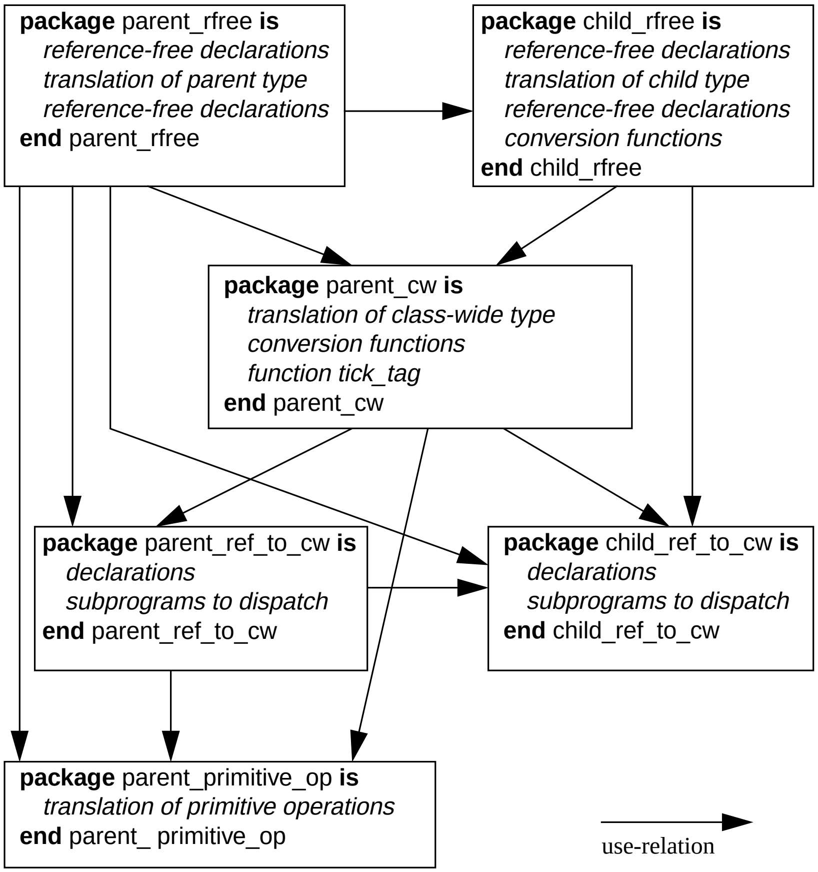


Fig. 5: Dependencies between translated packages

same package, the subprograms for the dispatching have to be modelled in a separate package so that the circular dependency does not exist any longer.

The sections about the translation gave an idea which translation concepts can be used to translate the language extension into standard VHDL. They especially show the feasibility of the ideas. In the last section about translation we want to reflect some properties of the translation especially with respect to synthesis.

6.5.9 Synthesis

The presented translation mechanisms do not introduce any VHDL constructs that are considered to be non-synthesizable by today's synthesis tools. Opposite to statements that claim that polymorphic procedures in a hardware model cannot be supported by synthesis [72] the presented translation mechanism for dispatching calls does not introduce any non-synthesizable constructs.

The decision to translate derived types and primitive operations without introducing any pointer structures for delegation preserves the main characteristics of distributed objects that are modelled as tagged types.

Class-wide types as part of the concept for modelling polymorphic objects can be translated into a data representation which is supported by synthesis. However, the resulting data representation comprises the union of all potential values an object of class-wide type could store even in cases where some values must not occur in the object simultaneously. We could think of some optimized translation strategies which reduce the memory size required by objects of class-wide types. Basically such translation mechanisms could use a variant record implementation at bit-level for class-wide types. All values to be stored in a class-wide object are encoded in a bit-vector representation.

An according mechanism together with a prototypical implementation is described in [163]. A related concept based on the same basic ideas for modelling memory-optimized class-wide types is described for the ObjectiveVHDL approach in [161]. It clearly shows the feasibility of the ideas.

With these considerations on translation of tagged-types, class-wide types, primitive operations and inheritance and dispatching mechanisms we conclude this chapter which introduced a proposal how to extend VHDL by object-oriented features. In the next chapter we discuss the modelling capabilities of the language extension in more detail. We have a closer look at the presented type based inheritance mechanism and we investigate how the new language features interact with the existing concurrency concept, the synchronisation and communication mechanism of VHDL.

Chapter 7

A Modelling Example

The previous chapter introduced an object-oriented language extension to VHDL and characterized its enhanced modelling features. A translation concept was explained which allows to link the object-oriented modelling to a VHDL-based design process.

We now illustrate possible links to an abstraction level above the modelling with an object-oriented HDL. Powerful modelling techniques at that level of abstraction are object-oriented specification methodologies. Such a methodology comprises a notation and a design methodology. The first part of the material in this chapter presents a short survey on one such a notation and design methodology. The second part describes how to modify the design methodology so that it integrates the language extension as a target language of an implementation step in the design methodology.

Starting from a sequential view on an example the modelling aspects of concurrency are introduced step by step into the example. This finally leads to an idea how to model a channel as a possible implementation of a communication mechanism.

The chapter concludes with the results of a case study to demonstrate the feasibility of an object-oriented hardware design flow using the language extension to VHDL.

7.1 Survey on the Object Modeling Technique

There are several object-oriented design methodology which could be adapted for the hardware design process [28,44,142,158]. They differ mainly in their notation and the design steps from a first specification model to an implementation. A compact overview can be found in [84]. One of the well-known methods is the *Object Modeling Technique* (OMT) from Rumbaugh

[142]. In this chapter we chose this methodology to illustrate the design step from the specification to the implementation in the language extension to VHDL. This first section gives a brief overview of OMT.

OMT is an object-oriented specification technique which makes it possible to describe a system at a very abstract level. The design process in OMT starts with an analysis. It characterizes the system and its problems. The result of this step should be a first specification which describes what the system does. It is not necessarily complete, and some of the details have to be added later on. It is important that the main ideas of the system are not hidden by too many details in that early stage of the development. While not constricting the design space, no decision restricting a further implementation should be taken at that time. This means in the analysis phase that the model should not contain implementation details. In further design steps, additional information is added to the description in the form of refinement steps [146]. Such a description consists of up to three different views, called models. The basic view is called *object model* [143]. It characterizes the static structure of the system in terms of objects and classes and their relation to each other. Each object and class is described by its attributes and operations. As objects are instantiations of classes, the attributes of objects contain concrete values. As an example, an object model for a simple processor [154] is shown in Figure 6. Each class is drawn in the diagram as a rectangular box, consisting of three parts. The name of the part is written in the top part, the middle part contains the attributes and the bottom part contains the operations. In the processor, for example, the class processor system has the operation execute instruction. The structural relations between the classes are drawn in the diagram as lines between the boxes. Mainly three different relations can be distinguished. An *association* is a relationship that contains information that is relevant for a certain time during the existence of the described system. A *role* belongs to each member of an association, which is a class or an object. It describes how a member of an association is viewed by the other members. Each role can be named by a *rolename*. The role also indicates how many instances of the class can be associated in a binary association with one instance of the other class. This can be denoted by a number as in the example in the association between the processor system class and the operand class. A special form of an association is an *aggregation*. As a class can be a member of different associations, the role is not assigned directly to a class but to the corresponding end of an association. It describes a *whole-part relationship*. It is marked by a diamond. In the example, an instantiation of the class memory is part of the whole processor system. A

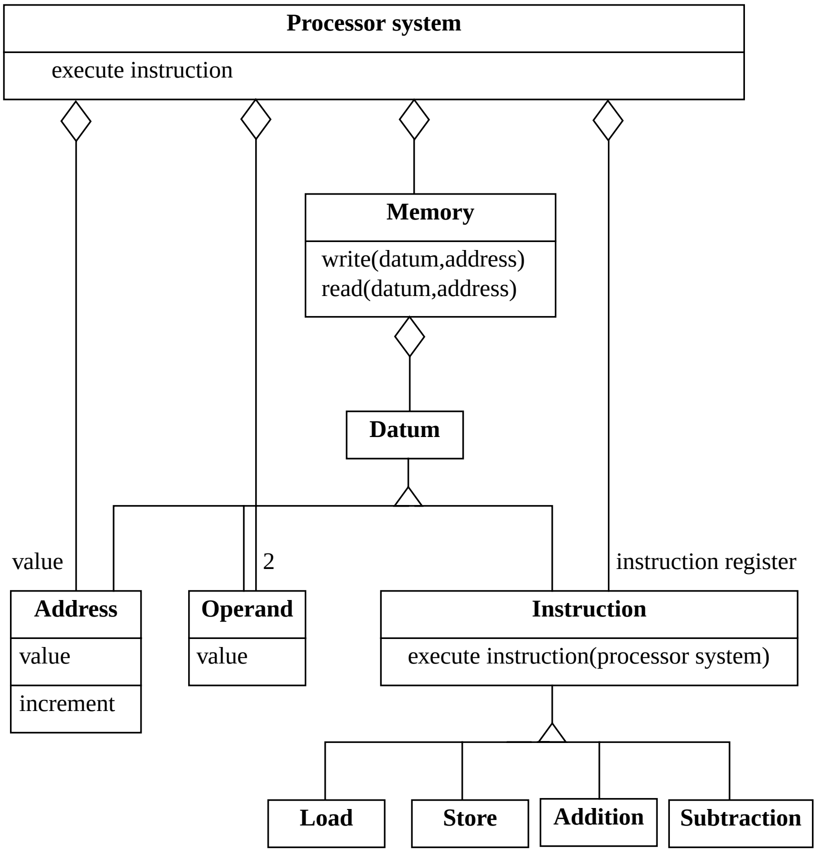


Fig. 6 Object model

third relationship is called *generalisation*. This is the most important relationship concerning the re-use of the system. It describes the relationship between *superclasses* and *subclasses*. The superclass and the subclass in OMT is exactly what we referred to as parent class and a child class. Thus, a subclass inherits all attributes, operations, and associations of a superclass. The notation for this relationship is a triangle. In the simple processor, for example, the class load is a subclass of the superclass instruction. This means it inherits the operation execute instruction from the class instruction as well as the aggregation between processor system and instruction together with the role instruction register. If a new class without additional relations is derived from the class instruction to extend the system specification, then no changes appear in the rest of the object model. For a more detailed description of the classes, their attributes and associations, a data dictionary completes the diagrams of the object model. It contains information about the entities of the object model as text. The dictionary of the processor, for example, contains the entry: Accu: store for the operands and results of instructions.

Another view is the *dynamic model* [144]. This model gives a state-oriented view on the behaviour of objects. The basic concepts of state-oriented views were introduced in Section 3.1.3. The behaviour of the objects is modelled in *state diagrams*. The notation to describe state diagrams is a variant of StateCharts which were discussed in Section 4.1.1. An example of a state diagram can be seen in Figure 7. It describes the life of the object memory of the simple processor example. The states of the object are the nodes, and the transitions are the directed arcs. The object starts its life by entering the state idle. If a read event is caused by another object in the model with the parameter read address, then memory changes its state to read. The event is annotated to the transition arc. When changing the state, an operation decode address is executed. In the new state read an *activity* datum := mem(address) is started. An activity is an operation within a state which consumes time. When the activity in the state read has finished, the object memory changes its state back to idle. In the latter transition, an event is sent to another object in the model. The target is the sender of the read event, and the event is return datum with the parameter datum.

An *event flow diagram* is used to describe which objects are interacting by sending each other events. Figure 8 shows an excerpt of such a diagram which describes the interaction of the object memory. For example, the object store can send an event read with parameter address or an event write with parameters address and datum to the object memory. Memory in turn can send an event return datum with parameter datum to the object store. As can be seen from the small example, this view is useful to describe the control flow in a system.

The operations which alter the entities of the system are described in a third view called the *functional model* [145]. Different techniques are used to describe the operations and their effects on objects in a system. One technique is to describe the operation as part of a client server contract. Such a technique uses preconditions and postconditions to define the contract as it was explained in Section 3.1.2. In the simple processor, the operation incre-

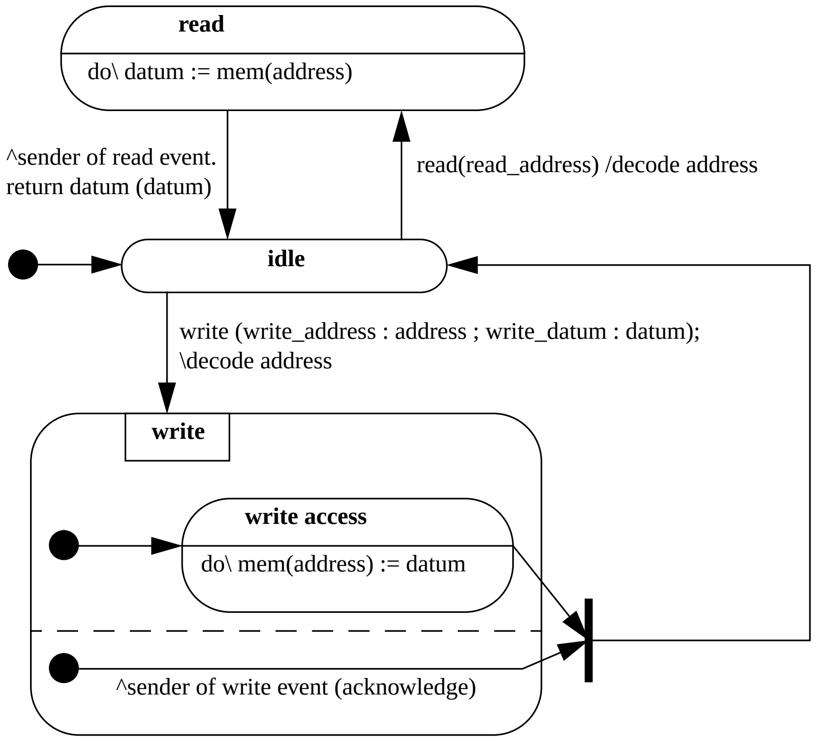

read
do\ datum := mem(address)
^sender of read event.
return datum (datum)
read(read_address) /decode address
idle
write (write_address : address ; write_datum : datum);
\decode address
write
write access
do\ mem(address) := datum
^sender of write event (acknowledge)

Fig. 7 State diagram of memory

ment of the class address, for example, is specified by a precondition which defines the value of the attribute address value as $x \in A$ and a postcondition which defines this value as $x + 1 \in A$ being A the address space of the processor system. The operation is not defined for $x + 1 \notin A$ (see Figure 9). To explain how an operation affects the objects and their attributes in a particular state of the system, an *object-oriented data flow diagram* (OODFD) can be used. An example is given in Figure 10. The example shows the data flow for the execute instruction operation of the class processor system. It describes the data flow between the objects and the transformation of the attribute values in that flow by the operations regarded as functions. The value of the attribute address value of the object address register, which is an instantiation of class address, is an input of the operation read memory of the object main memory. The result of the operation is an input for the operation

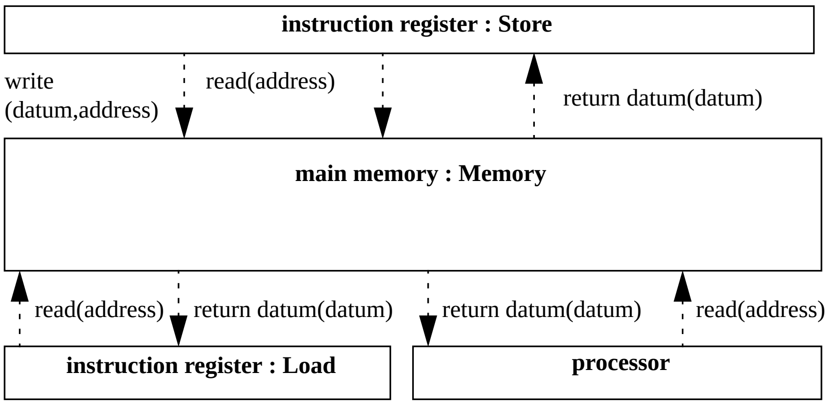


Fig. 8 Event flow diagram of memory

Operation:	increment
Responsibilities:	increments the address value of an address
Inputs:	address value
Returns:	none
Modified objects:	address values
Preconditions:	address value is a valid address $x \in A$, A is the address space
Postconditions:	address value is a valid address $x + 1 \in A$

Fig. 9 Operation specification

execute instruction of the object instruction register. The further data flow depends on the instruction performed by the system. The shown OODFD describes a store operation. The address value of the address register is incremented. The result is a new address value. It is marked by a tick. The new value replaces the old value of the attribute. Finally, the value of the accu is written into the memory.

A second technique is the procedural description of the operations. This means that the operations are described by invocations of other operations in various objects of the system. To specify the sequence of invocations and the corresponding messages between them, an *object interaction diagram* (OID)

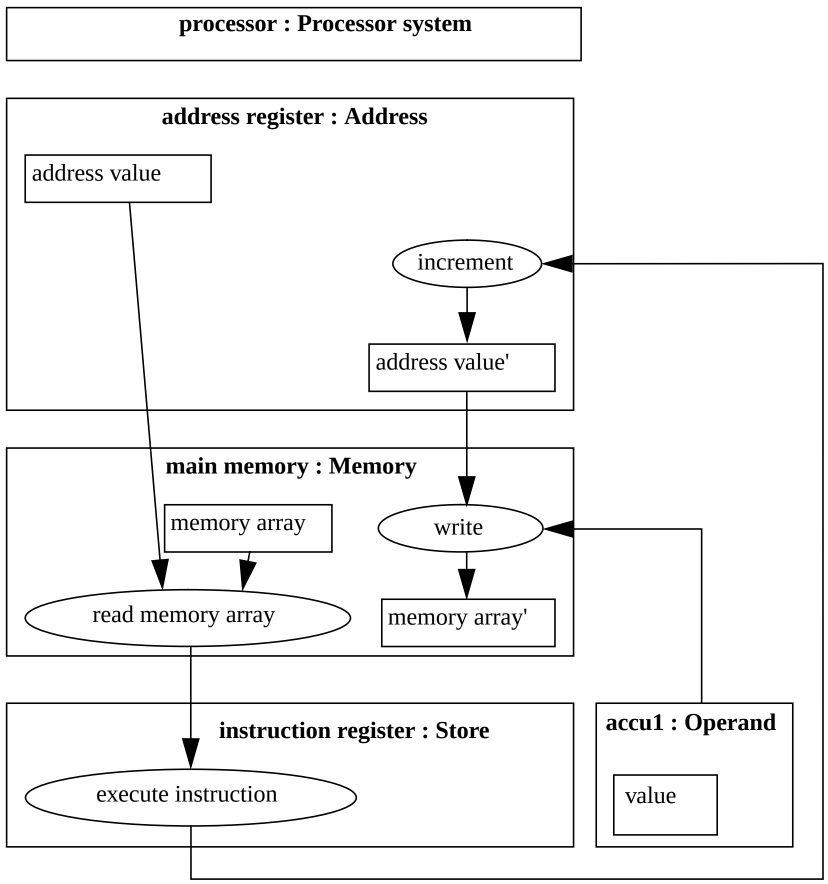


Fig. 10 Object-oriented data flow diagram for operation execute instruction of object processor

as can be seen in Figure 11 is used. The diagram characterizes the same situation as the OODFD in Figure 10, but from a procedural point of view. It shows that the processor system sends a message to the object of class address to get an address value. This value is used to send a message to the memory to read the memory. The numbers define the sequence and the hierarchy of the operation invocations. The non-permanent associations which are established by sending a message are drawn as dotted lines. The latter technique provides models which are closer to an implementation than the

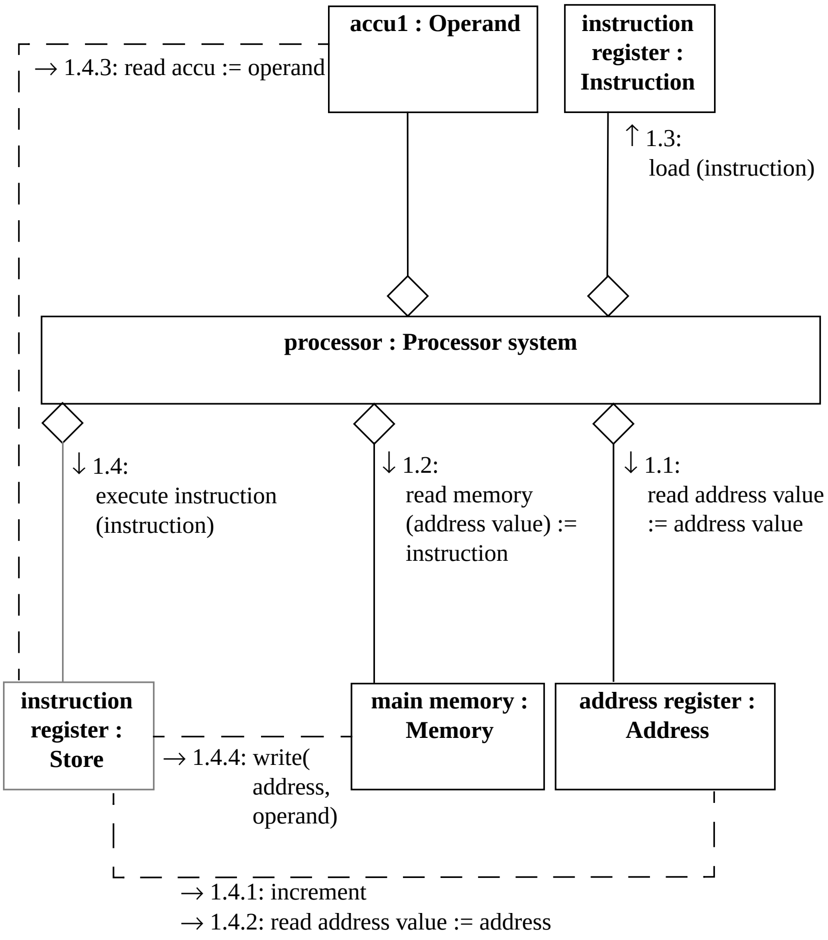

accu1 : Operand
instruction register : Instruction
→ 1.4.3: read accu := operand
↑ 1.3: load (instruction)
processor : Processor system
↓ 1.4: execute instruction (instruction)
↓ 1.2: read memory (address value) := instruction
↓ 1.1: read address value := address value
instruction register : Store
main memory : Memory
address register : Address
→ 1.4.4: write(address, operand)
→ 1.4.1: increment
→ 1.4.2: read address value := address

Fig. 11 Object interaction diagram for operation execute instruction of object processor

functional views. This means that a first analysis model probably contains OODFD, which are supplemented during the further refinement steps by OID. The specification of operations in a very abstract functional view is transformed in further design steps to a procedural implementation, which is a sort of algorithmic description and therefore close to an implementation in a procedural language. The following section sketches how to perform this implementation step for operations together with the modelling of the structure described by object models. The object-oriented language extension to VHDL is used as the target language in the resulting implementation.

7.2 The Implementation

In this section we illustrate the mapping of an OMT specification to an implementation in the language extension to VHDL. The translation of OMT models is a refinement step in the object-oriented design flow in which details about the desired behaviour and functionality is added to the system description.

7.2.1 Mapping the OMT-models

Although the different OMT models cannot be translated independently, it is quite useful to start with one model and then supplement the description when translating the other models. It is practical to start with the object model as it is the core of the OMT. This model can be translated to some extent straightforwardly into a data structure. Each class becomes a tagged record type with the attributes as record elements. An example of such a mapping is translation of the class datum. The type datum is an empty tagged record. As datum is an abstract class, that means, it is only used as a parent class without a concrete behaviour it is modelled as an empty tagged record.

```
type datum is tagged record
    null;
end record;
```

In the OMT model the class address is a subclass of datum and contains the attribute address_value. Such a relationship is modelled in the language extension as a child which is derived from a parent type. Often the rolename can be used as the element’s name. In the object model the rolename of the address is value. Thus value is a part of the attribute’s name.

```
type address is new datum with
private
    address_value : integer;
end record;
```

The example shows how generalisation in the object model is turned into a corresponding inheritance hierarchy of tagged records.

The operations of a class can be mapped to primitive operations. An example is the class instruction which has an operation execute_instruction. The class instruction is a subclass of the class data. Thus, it can be modelled as follows:

```
type instruction is new datum with record
    null;
end record;
procedure execute_instruction (object : in instruction;
                                processor : inout processorsystem);
```

Please note that we have to add some information to the interface of the procedure to make it complete. This information includes the parameter's name and its mode. Adding such information is a typical task in that implementation step.

A subclass may provide its own implementation of an inherited operation. In such a case the inherited operation has to be re-defined by the child. An example is the class load which is a subclass of instruction. The subclass inherits the operation execute_instruction. It re-defines and thus overrides the inherited behaviour.

```
type load is new instruction with record
    null;
end record;
procedure execute_instruction( object : in load;
                                processor : inout processorsystem);
```

A special association which is used to describe relations between objects in OMT is the aggregation. In the implementation step they can be transformed into record elements of the corresponding tagged record type. In OMT a subclass inherits all the associations of its parent's class. In such situations it is often useful to model such an aggregation with a parent's class by record elements of a class-wide record. If multiplicity is involved this has to be modelled by an appropriate number of instantiations.

An example of such an aggregation is the processor system. The processor system with its four aggregation associations is described as a tagged record with five record elements. The elements are all of class-wide type, so that the association exists for derived classes, too.

```
type processorsystem is tagged record
    main_memory : memory'CLASS;
    instruction_register : instruction'CLASS;
    address_register : address'CLASS;
    accu : operand'CLASS;
    accu2 : operand'CLASS;
end record;
```

The class-wide type instruction‘CLASS of the record element instruction_register realizes the association for the type instruction as well as for the type load and any other derived type. The multiplicity two of the role accu as part of the association between the class processor system and operand is described by two record elements.

The processor example illustrates how to map an aggregation to an implementation. There are, however, some aggregations which are difficult to translate. For example, if the memory is modelled as aggregation of memory cells with a multiplicity of many without an exact range specification, the translation causes problems. A possible solution would be to constrain the multiplicity in the model by a fixed range and to describe the aggregation with an array. However, this solution does not always work. If dynamic behaviour was intended by such a description containing multiplicity of many, the translation does not meet the original intention of the specification.

A similar problem with dynamic behaviour arises when recursive structures are used in a model. As we have seen in Section 6.2.4 it is not possible to model a tagged record that recursively contains a record element that is of the tagged record. A solution would be the use of incomplete type declaration and access types. Its disadvantage for hardware systems is that access types can not efficiently be synthesized. If possible, the recursive structures have to be replaced by equivalent non-recursive specifications. Generally, objects which are to be implemented as hardware components cannot be used in the specification with a dynamic behaviour. Dynamic in this context means the creation and destruction of an arbitrary number of objects, which is not exactly defined at specification time, during run-time.

A structure of the object model which has no equivalent in the implementation language is the parallel generalization. It would require a mechanism for multiple inheritance in the language. Such a mechanism would allow the inheritance of record elements and procedures from two or more tagged records at once. However, as explained in Section 6.3 and Section 6.4 multiple inheritance is not supported by the language extension to VHDL.

There is a variety of different translation possibilities for the implementation of the dynamic model. One possibility is to use modelling techniques known from VHDL. The states are encoded by an enumeration type. A next state function uses a case statement to distinguish between different current states and inputs and determines the next state. The state variables which are attributes of the classes are modelled by record elements and the next state functions by methods. Another possibility would be to model each state in a separate class, derived from a common parent's class. Each class has its own next state function as a polymorphic operation. Such a state can contain a sub-state-machine as an attribute. This machine could be started when the state-machine enters the composite state, consisting of a state machine. The state machine is then implemented by an instantiation of an object of class-wide type, which contains the actual state. Especially for state machines with operation calls as events and activities, dynamic models can be implemented by a procedural description of the activities. The different states belong to different places in the procedural description.

As an example, we could look at a possible implementation of the dynamic model of the memory which is described in Figure 7. The class memory is modelled as a tagged type memory with primitive operations read_memory and write_memory. In the corresponding package body the operations are implemented.

```
package body memory_package is
    procedure write_memory (  mem : inout memory;
                              write_address : in address'CLASS;
                               write_datum : in datum'CLASS) is
        variable tmpinteger : integer;
    begin
        address2integer(write_address,tmpinteger);
        mem.memory_array(tmpinteger) := write_datum;
    end;
    procedure read_memory(    mem : in memory;
                              write_address : in address'CLASS;
                              read_datum : out datum'CLASS) is
        variable tmpinteger : integer;
    begin
        address2integer(write_address,tmpinteger);
        read_datum := mem.memory_array(tmpinteger);
    end;
end memory_package;
```

A call of the operation write as an event causes a state transition. The generated event is implemented as a method call of the target object of class address. The activity of the state write is modelled as an access to a record element. In the example, several events are generated. The acknowledgement that a write operation is performed is implicitly included in an implementation of a message as a sequential procedure call. The calling procedure is blocked until the write operation is performed.

As can be seen from the example, the OMT model assumes that the targets of the events automatically can be made visible at the place where the events are generated. There is no special order of analysis in which classes from the OMT model become visible. It is no problem to define mutual dependent classes. This situation is different in the implementation language. The distribution boundaries and the abstraction boundaries of the language extension characterize the visibility in the implementation and the order of analysis and elaboration defines which type can be used by another. If such mutual dependencies between classes occur in an OMT model they require a special treatment in the translation process.

An example of such a dependency can be seen in Figure 6. The instruction is part of the processor system. At the same time instruction uses a parameter of class processor system in its operation. In a translation the corresponding tagged types mutually depend on each other. A nearly identical situation was described in Section 6.3.3. The solution was to use a postponed use clause to break the circular dependencies between the types. The same approach can be used in the example of the processor system and the instruction.

```
use work.datum_package.all;
postponed use work.processorsystem_package.all;
package instruction_package is
    type instruction is new datum with record
        null;
    end record;
    procedure execute_instruction (object : in instruction;
                                   processor : inout processorsystem);
end instruction_package;

use work.instruction_package.all;
package processorsystem_package is
    type processorsystem is tagged record
    private
        instruction_register : instruction'CLASS;
```

```
            …
        end record;
        …
    end processorsystem_package;
```

The tagged type processorsystem is made visible by a postponed use clause in the package containing the tagged type instruction.

As the encapsulation concept of VHDL is very strict, it is often necessary to make the target of the events visible within the scope of the triggering object by passing the objects as parameters. This parameter passing can become very cumbersome. To determine the required visibility of objects within scopes, an event flow diagram can be quite useful. It shows the objects and the events sent to each other. The example in Figure 8 shows that the object main memory has to be visible in the operation execute_instruction of class load. This is realized by passing a parameter of type processorsystem, which contains the object main_memory as record element and which is passed to the operation. The resulting declaration of type load was shown before.

As in the object model, there are some constructs in the dynamic model which cannot be easily mapped in an implementation. There exists no simple equivalent in the implementation for the specification of a composite state containing several parallel working state machines.

The implementation of the operations described in the functional model consists of transforming the non-procedural high-level specification into a procedural description. Transformations from the non-procedural view, describing the state before and after an operation, to the procedural view can already be applied in the functional model. The implementation in the language extension to VHDL is the continuation of this action. As an OODFD shows the effect of an operation across many operations, it is useful among other things to determine the visibility between the objects. The techniques to establish the required visibility in the implementation are the same as described for event flow diagrams. If an object interaction diagram exists, then the sequence of messages it describes can be transformed into the corresponding sequence of procedure calls.

It has been outlined and illustrated by an example how an OMT model can be transformed with some limitations into an implementation. The major restrictions are the lack of unconstrained multiplicity, the missing constructs for multiple inheritance, and the difficult implementation of composite states with parallel working sub-state machines. The model consists of objects implemented as tagged records and polymorphic procedures. All messages are realized as sequential procedure calls. Some effort has to be made to make objects visible to each other. This is achieved by passing the concerned objects as parameters into the operations. Re-use of the OMT model by deriving new objects from existing ones is supported by the implementation.

7.2.2 Implementing concurrency

The modelling presented in the previous section is based on a sequential description of the model in the implementation language. Parallelism has to be introduced in further design steps, when the scheduling is done automatically to improve the performance of the system. The idea is that the order of execution of sequential statements between two wait statements is only determined by dependencies caused by variable assignments. The automatically introduced concurrency is fine-grained on small parts of the system. To introduce a coarse-grained concurrency at system level, it has to be explicitly specified in the model. The OMT offers several constructs to express concurrency. For example, it would be possible in Figure 11 to specify a concurrent sending of messages 1.4.2 and 1.4.3 by re-indexing both messages as 1.4.2. The implementation language offers two notions of concurrency which are strongly correlated. For simulation, both are based on a discrete simulation time. One concurrency concept is based on parallel running processes which are synchronized by wait statements. A second concept can be found in the parallel update of signals. This allows to express concurrency within one process. In other words, sequential signal assignments can be used to model a quasi-concurrent process.

For example, if in Figure 11 the messages 1.4.2 and 1.4.3 are specified to be sent in parallel, as mentioned above, then the implementation of the address value and the operand as signals together with a wait statement allows a parallel updating of the values. As can be seen from the example, this kind of implementation for concurrency is limited to particular cases. It makes sense if the effect of the messages has to be a parallel update of attributes of objects. For specification concurrency at system level, however, the concept of parallel running processes is appropriate in many cases.

In the specification of synchronous hardware systems, control messages are often based on a clocking schema. The processing time of the hardware components executing the operations invoked by the control messages has to be considered by the control units as senders of the messages. The problem is that the information about the processing time of complex operations is not known until the model is synthesized at gate-level. It is possible to annotate processing times as constraints in a specification, they become part of the protocol, but there is not guarantee that they are met by an implementation. As most operations are specified in a non-procedural view at system level, it is not possible to guess realistic processing times. Even during further refinement steps, it is difficult to predict processing times for operations which are performed by sending messages to other objects, which themselves send messages. If the sequence of control messages sent to other objects is based on a fixed clocking schema, then the invocation of polymorphic operations is only possible if they serve the same protocol. The constraints as part of the protocol specification are inherited by derived classes even if the operations are re-defined. However, it is not guaranteed that a re-defined implementation preserves its compatibility to the protocol. These kinds of problems have been discussed in detail in Section 3.2.

For example, if in Figure 6 the operation execute instruction of the processor system is implemented based on a fixed clocking schema which expects the operation execute instruction of the object instruction register to be performed within ten clock cycles, then all derived instructions have to implement the operation with a processing time of ten clock cycles.

A quantitative specification that counts clock cycles to define the time when a message has to be sent is in most cases not appropriate for re-usable operations at that level of abstraction. The synchronisation should be specified in a more abstract way by describing the timing dependencies between the operations relative to each other. Parallel running objects can be implemented by the declaration of objects as variables in different VHDL processes. The processes may be declared in the same or different architectures of different entities. The declaration of the classes as tagged record types in packages can be used in several entities. As objects can be instantiated in different processes or entities, aggregated objects can be modelled by a connection between the processes or entities, instead of tagged records containing other tagged records as record elements. Depending on the specification, a mechanism has to be implemented for the synchronisation of the objects. A similar concept can be applied in some cases to implement a specification which allows parallel access to an object and its methods. It can be modelled as an object consisting of an aggregation. The attributes are modelled as separate objects in different processes. These objects contain operations to solve access conflicts.

The class memory of the example can be used to illustrate the concepts for objects in parallel running processes. If the instantiation of memory and the rest of the processor system are modelled as two objects running in parallel, then the implementation consists of two processes. In the implementation, a link in the form of signals has to be defined between the processes which enables the processor system to invoke the operations of the memory and enables the memory to send back the results of the operation. We illustrate the situation by listing the implementation of the operation execute instruction in the sequential model.

```
use work.address_package.all;
use work.processorsystem_package.all;
use work.memory_package.all;
use work.datum_package.all;
package body load_package is
     procedure execute_instruction (
                                        object : in load;
                                        processor : inout processorsystem) is
          variable tmpdatum : datum'CLASS;
          variable tmpaddress : address'CLASS;
     begin
          increment (processor.address_register);
          read_memory (processor.main_memory,
                          processor.address_register,tmpdatum);
          tmpaddress := tmpdatum;
          read_memory(processor.main_memory,tmpaddress,tmpdatum);
          processor.accu := tmpdatum;
          increment(processor.address_register);
     end;
end load_package;
```

In a parallel design, the object memory can be modelled together with a new implementation of the operation read memory in a process running in parallel to another process, which implements the rest of the processor system. The procedure call read_memory in the operation execute instruction then has to be replaced by an invocation mechanism for a new implementation of the operation read memory of the object memory, which is now implemented in a different process. The procedure execute_instruction which invokes read_memory has to be blocked until the new implementation of the operation which consumes time has finished and a result is returned. But there is no advantage of such a mechanism based on blocking the caller of an operation over the sequential implementation of all objects in one process. To enhance the model, an asynchronous message passing could be introduced. The increment operation is invoked after sending the message to start the read operation. The read operation is divided into two parts: one starting the read operation and one for the reply scheduling. The reply scheduling may consist of receiving several results at different times.

In the example, the reply scheduling requires that the result has to be received before it is written to the accu. As we have cited in Section 3.3.9 this comes at the expense of abstraction. It is just this abstraction that is missing when modelling a common interface for polymorphic objects. The synchronisation with other objects as part of the reply scheduling may differ in the various polymorphic objects. The synchronisation should be part of the protocol, that means it should be part of the abstraction of an object, however, this level of abstraction is just missing due to the reply scheduling. Essentially this is a problem how to preserve behavioural compatibility between polymorphic objects without having an appropriate abstraction of the objects' behaviour.

A pragmatic solution would be to use a fixed protocol that divides each operation into two parts, one to send parameters and one to receive all results at once. For the invocation of the first part, the earliest possible time is chosen and for the second part, the latest. The presented problem is independent of the chosen implementation language. As a consequence, the best thing is to choose a sequential implementation for those objects which are not explicitly specified to run in parallel and to perform the transformation into parallel components automatically during resource allocation in a further design step.

From the discussion about various proposals how to extend VHDL for modelling at system level in Section 5.2 we know that the modelling of message passing across process scopes is a difficult issue. The strong encapsulation and weak abstraction of behaviour in the language constructs that are used for modelling concurrency were identified as reasons. Some ideas how to solve these modelling problems in an implementation are given in the next section.

7.3 A Channel-Based Communication Mechanism

The introduced implementation concept for operations is based on procedures. An operation is implemented as primitive operation, and a message to an object is performed by a call on a primitive operation with the target of the message as parameter. Such an implementation is only feasible if the target object is visible in the primitive operation sending the message. In a sequential model consisting of only one process, this can be achieved by parameter passing of the affected objects. The situation is different in a concurrent model, when the sender and the receiver of the message are instantiated in different processes. As VHDL does not provide communication transparency for distributed objects a dedicated high-level synchronisation and communication concept is required to perform the communication between the objects. As the language extension to VHDL in this thesis does not provide such a mechanism as part of the language extension it has to be modelled using VHDL signals. The signals are used as building blocks for a high-level mechanism.

As we know from the discussions about communication and synchronisation mechanisms at various points in this thesis any attempt to integrate communication transparency in language extensions to VHDL failed. On the other hand, the channel concept integrates best in the abstraction and encapsulation concepts of VHDL. Therefore channels modelled by signals are chosen as high-level communication constructs.

To make the channels as re-usable as possible the idea is to use object-oriented concepts for the modelling. The object-oriented modelling starts with the specification of an abstract class channel which can perform operations needed for communication. The instantiations of the class are signals performing the data exchange. Such a class could be specified, for example, as shown in Figure 12. The class channel has attributes for synchronisation

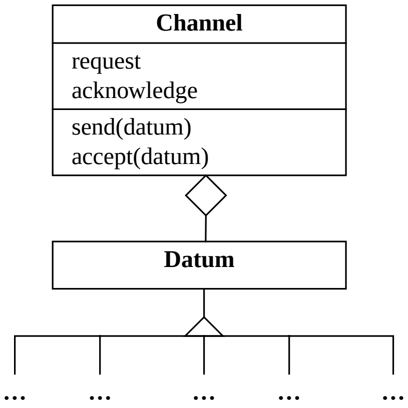


Fig. 12 Object model of channel

purposes between the sender and receiver. It also has an attribute of class datum which models the message to be sent. This class is used as a superclass in a generalization consisting of all types of messages which can be sent on the channel. This makes the channel re-usable for different kinds of messages. Two operations can be invoked by other objects: a send operation and an accept operation. This is illustrated in Figure 13. For simplification,

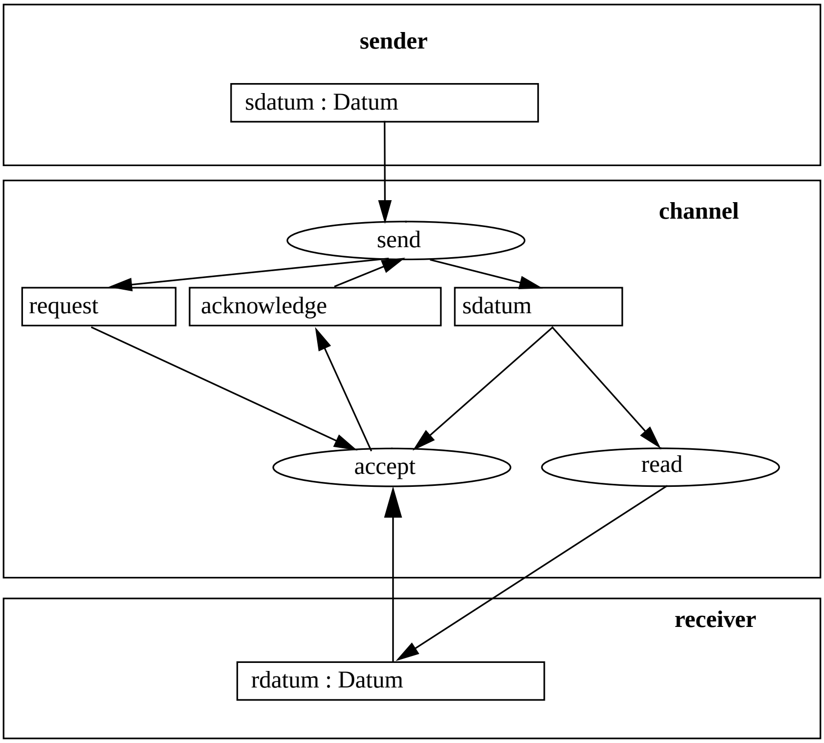


Fig. 13 OODFD of channel operations

datum is considered as an attribute of channel. Such a channel structure could be used to model both synchronous and asynchronous communication. In case of synchronous communication, the sender sends a datum into the channel and waits until the receiver invokes the operation accept. Together with a datum from the receiver as input, another operation accept is performed. The results of this operation are sent back to the sender and the receiver. Then both objects continue their operations. An object invoking an accept operation before a send operation is performed is blocked until the send operation is invoked. In case of an asynchronous communication the sender may continue its execution after a send operation. However, it must not send another message until a receiver has received the message. If multiple sends were allowed that immediately follow each other from the same sender this would require a buffer as an attribute of channel. It would be a buffer similar to those presented in various examples in this thesis. The declaration of such a channel may look like this:

```
type channel is tagged record
private
    request : token;
    acknowledge : token;
    message : datum'Class;
end record;
procedure send (signal channel_from_receiver: channel;
                signal channel_to_receiver : inout channel;
                message : datum'Class);
procedure accept (  signal channel_from_sender: channel;
                    signal channel_to_sender : inout channel;
                    message : out datum'Class);
procedure read (    signal channel_from_sender: channel;
                    message : out datum'Class);
```

The primitive operations have two controlling operands of type channel to avoid resolved signals in the bi-directional communication between processes. Each channel is used for communication in one direction. The mode inout of the channel parameters is chosen to allow the reading of the actual states of both channels in the primitive operations.

Declaring the parameter message to have a class-wide type datum'Class allows the re-use of the channels in various contexts where the messages are derived from type datum. Another kind of re-use is possible by allowing a new channel to be derived from an existing one.

We want to illustrate this with a derived type of channel. It could be used to transport messages that contain instructions. The instructions are derived from the tagged type datum. The new type inherits the primitive operations of channel. Now it would be possible for example to add a new accept operation to the derived type which accepts a message on the channel and which then executes the instruction sent by the message.

```
type channel_for_instruction is new channel with record
    null;
    end record;
    procedure accept (

            signal channel_from_sender: channel_for_instruction;
            signal channel_to_sender : inout channel_for_instruction;
            target : inout processor_system);
```

The inherited access operation is overloaded by the new accept operation. The operation is not overridden due to the different signatures of the operations. Thus, it would be possible in an implementation to call the inherited operation and then to execute the instruction which is received in a message.

```
    procedure accept (
            signal channel_from_sender : channel_for_instruction;
            signal channel_to_sender : inout  channel_for_instruction;
            target : inout processor_system) is
        variable message : datum'Class;
    begin
        -- asynchronously accept message:
        accept (    channel_from_sender,
                    channel_to_sender,
                    message);
        execute_instruction (
                    object => instruction'Class(message),
                    channel_in => channel_from_sender,
                    channel_out => channel_to_sender,
                    target => target);
    end;
```

The instruction is executed by a call on the primitive operation execute_instruction with a controlling operand of the class-wide type instruction. The controlling operand becomes a class-wide type by applying a type conversion in the association list of the call. The target type of the conversion is the class-wide type instruction'Class. The call automatically dispatches to the actual operation of the instruction which was received by the message. The result of executing the instruction on a target could be sent back to the sender by calling the send operation in the primitive operation execute_instruction. The resulting mutual dependency between the type channel_for_instruction and the types modelling the instructions has to be resolved by using a postponed use clause.

The example describes the communication between server and client both running in parallel and sending each other messages via two channels. It is a difficult task to generalize a concept for the modelling of communication and synchronisation between an arbitrary number of distributed objects from the concept of the example.

Basically such a concept requires two channels for each use-relation between two distributed objects. The channels have to be implemented as signals of an appropriate channel type. A class uses the channels as shared resources for communication which are abstracted by interaction points as it was described in Section 3.3.19. Due to the abstraction and distribution boundaries of the language extension to VHDL the interaction points are implemented as signal parameters in the parameter lists of the primitive operations. In cases where it is not necessary to concurrently send particular messages it is possible to multiplex them on one channel and thus to reduce the number of required interaction points. A characteristic of the presented channel is that it already multiplexes different messages. If the messages are sent to different target objects it is necessary to indicate their targets as part of an extended message. In the terminology of protocol design [83] we could say that the extended message is a protocol data unit which contains the target indication as protocol information and the original message as a service data unit. A channel to transport such a protocol data unit could be derived from the tagged type channel.

```
type d_channel is new channel with record
private
    target : object_identity_type;
end record;
```

For such a target indication each object requires a unique identity which allows to distinguish it from other targets. Unfortunately, VHDL does not support the modelling of such an identity very well. A possible approach would be to use the predefined attribute 'Instance_Name for such a purpose. The new primitive operation accept could be specified for the derived type which filters the messages for a particular target. The implementation may look like this:

```
procedure accept (
        signal channel_from_sender : d_channel;
        signal channel_to_sender : inout  d_channel ;
        message : out datum'Class;
        target : in object_identity_type) is
begin
    wait until channel_from_sender.target = target;
    accept (    channel_from_sender,
                        channel_to_sender,
                        message);
    end;
```

Several receivers are connected to the same channel which transports the multiplexed messages for the receivers. The new accept operation can be called by each receiver. The accept operation waits until it receives a message for its target.

So far the extended concept describes how to multiplex messages from one sender. It is an open question how to multiplex messages from different senders, that means, how to multiplex the channels. A solution could be to model such a multiplexing by a new primitive operation of a type derived from channel. It requires the channels to be multiplexed as input parameters and the multiplexed channel as output. The multiplex operation is the central part of the extended channel concept. The entire synchronisation concept depends on how we model the channel as a shared resource and how we implement the mutual exclusion synchronisation for accessing the shared resource. We have to consider in such an implementation all the details about mutual exclusion synchronisation that have been discussed in the previous chapters.

Although finally we have to use the low-level communication mechanisms of VHDL to implement the synchronisation the advantage of using the language extension very often is the possibility to model a new synchronisation concept by modifying an existing one. A typical situation is that the communication mechanism has to be extended by new message or channel. The new message or channel requires its particular synchronisation condition. At the same time the extended communication concept should be compatible to the original one with respect to its original channels and messages. The language extension allows to derive a new type of channel which considers the new synchronisation conditions of the message or channel in its implementation of the multiplex operation. If the new synchronisation conditions are met it executes the new synchronisation otherwise it executes the inherited version of the multiplex operation and thus preserves its compatibility to the original communication concept.
Consider a multiplex operation which is extended by a new channel.

```
    procedure multiplex (
                signal new_channel : in derived_cannel;
                signal original_channel : in derived_channel;
                signal another_original_channel : in derived_channel;
                signal multiplexed_channel : inout derived_channel) is
```

```
begin
    if synchronisation condition is met for new channel then
            perform new multiplex operation
    end if;
    multiplex ( original_channel,
                another_original_channel,
                multiplexed_channel);
end;
```

The implementation checks the new synchronisation condition. If it is met it performs the required synchronisation operation. Then it performs a call on the inherited multiplex operation which also checks synchronisation conditions and if required performs some synchronisation operations. This call on the inherited operation can be viewed as a special kind of delegation to preserve the compatibility. We shall investigate this technique to preserve compatibility in more detail in the next chapter.

This seems a good moment to remember the effects of replacing the original synchronisation points by channel operations that have been discussed in Chapter 5. If we are implementing synchronous or blocking send operations and accept operations we have to make the corresponding considerations.

The presented discussion on designing a channel concept with the language extension was a rather rough sketch how to model communication. As we know from Section 3.3 there are many variants with many details on how to model communication and synchronisation. An appropriate communication concept has to be selected and implemented for each concrete model individually. The key consideration in the modelling with the language extension is not to replace the selection by providing one predefined communication mechanism but to support the implementation of an individually selected communication model.

7.4 A Case Study

In the previous sections of the chapter we illustrated how to use the language extension to VHDL for object-oriented hardware modelling by sketching parts of a small example. Such a small example is not sufficient for an evaluation of the new design concept. To analyse the feasibility of the proposed object-oriented design technique based on the language extension to VHDL a case study was made [153,175]. In this case study a simplified model of a MC68000 microprocessor was developed. The design process started with an OMT model of the processor. The target language for the implementation was a previous version of the language extension to VHDL. To analyse the translation concepts parts of the model were manually translated into a model using standard VHDL.

The results of the study showed a general feasibility of the concept. It was possible to start the specification at a very abstract level without looking too much into the insignificant details. After carrying out several refinement steps on the different views an object-oriented model could be designed which was well suited for an implementation in the object-oriented extension to VHDL. It was illustrated in the case study how to apply the language features for re-use and extension by adding new instruction and addressing modes to the processor model. While the case study demonstrated a principal practicability of the ideas it at the same time helped to uncover and remove some weaknesses in the earlier version of the language extension to VHDL.

For example, it turned out that it might be useful to have a concept for solving circular dependencies between tagged types. This was achieved by adding the postponed use clause to the language. Another example concerns the rules for implicit type conversions which were stated more precisely to remove uncertainties.

Only restricted capabilities for the modelling of concurrency were detected as a major limitation in the modelling of the processor example. The behaviour of the processor executes almost sequential due to a synchronous protocol that was used to perform the communication and synchronisation between all objects in the processor. This led to a closer investigation of the requirements for communication and synchronisation modelling. Especially, the experience from the case study led to some detailed considerations on how to model communication and synchronisation in the context of extendable objects without violating object-oriented modelling principles. We present the results of these considerations in the next chapter.

Chapter 8

Inheritance Anomaly

We have now introduced the features of the language extension to VHDL and we have illustrated by examples how to use them for object-oriented hardware design. Two major topics which have arisen are modelling by extension and modelling of concurrency. Modelling by extension was introduced by the inheritance mechanism of tagged types which is part of the language extension. Modelling of concurrency was described by using processes which are provided by standard VHDL for the modelling of parallel behaviour. The object-oriented constructs are orthogonal to the concurrency constructs (Compare Section 3.3.4). The integration of both concepts for modelling of objects executing in parallel is not provided by the language but is rather a modelling issue. The construction of a high-level communication mechanism between distributed objects using an object-oriented modelling style is an integration example that was presented in the previous chapter.

In this chapter we come to a more detailed discussion of modelling approaches to integrate inheritance concepts and concurrency. We develop a modelling style which provides the required interaction between inheritance and processes, object-oriented programming and concurrency.

In the previous chapters we saw that compatibility between inherited properties of objects which are modified and the original parents' properties are a key to re-use. The modelling style has to consider these compatibility issues. We present such a style as a variant of a delegation mechanism.

As far as concurrency of objects is concerned the modelling style is similar to the constraint oriented specification which can be used in LOTOS (Compare Section 4.4.1). We develop a variant of the constraint oriented specification which models additional constraints by adding quasi-concurrent processes to a model. The special issue how to avoid service interference or service interaction (Compare Section 4.1.4) which may result from composition techniques like the constraint oriented specification is discussed in detail.

Missing concepts to model condition synchronisation have been identified in the survey of Section 5.2 as a weakness of most of the proposals to extend VHDL. To avoid this weakness the modelling style emulates a guard concept which is comparable to the one of Occam (compare Section 4.4.2). Like in Occam the interface of a subprogram–in this case a primitive operation–is abstraction and distribution boundary at the same time. Accordingly, abstraction concepts for communication similar to Occam are proposed for the modelling style.

In earlier chapters we have seen that there are two forms of objects, active and passive ones. Although we deal with both forms in this chapter we put emphasis on the modelling of passive objects and show how they can be modelled using tagged types [155].

8.1 Inheritance and Synchronisation

In an earlier chapter we explained how the synchronisation is modelled in objects by the object's interface control. The interface control implements the synchronisation constraints of the object and thus generates the interface control space of the object. The possible state transitions in the interface control space form the protocol of the object. The interface control is described by synchronisation code. In other words, the synchronisation code implements the synchronisation constraints and thus the protocol which an object must meet to perform communication.

In Section 4.1.4 we saw that incremental modification of protocol specification by adding new transitions to a protocol may cause unintended feature interaction or service interference. The phenomenon is not limited to protocols specified in Estelle but occurs also in other languages like for example in LOTOS (Compare Section 4.4.1). It is a general question how to extend and modify synchronisation code so that it still conforms to the original inherited protocol.

If an object inherits synchronisation code it is difficult to add new synchronisation code to it without re-analysing or re-implementing the inherited one. However, re-analysing or re-implementing means to break encapsulation which goes directly against the object-oriented paradigm. This problem is discussed in several publications for example in [3,17,34,111] and there is still ongoing research to find appropriate solutions, for example [2,41,61,62,112,115]. The problem is known as the *inheritance anomaly.*

Before we propose a solution to the anomaly we want to illustrate some aspects of the anomaly and classify different categories. As inheritance and synchronisation constraints interfere with each other, the way the anomaly appears depends on the approach that is chosen for modelling the synchronisation constraints. We presented different approaches to model synchronisation constraints in the discussion about how to specify and abstract the interface control space in Section 3.3.12. We now consider how they may cause different categories of inheritance anomalies. For the categorization we follow the categories proposed in [111].

8.1.1 Body anomaly

A straightforward way to model synchronisation code is to design a sequential object that uses explicit acceptance of messages as it was described in Section 3.3.16. The synchronisation code is implemented in the object's body. In the language extension to VHDL we could use a channel concept like the one described in the previous chapter where messages could be received and sent at the synchronisation points.

We may illustrate this by the bounded buffer example which was introduced in earlier chapters:

```
package bounded_buffer_package is
    type bounded_buffer_array is array(positive range <>) of item_type;
    type bbuffer is tagged record
        buf : bounded_buffer_array (1 to buffersize);
        buf_in : integer;
        buf_out : integer;
    end record;
    procedure get (   signal object : inout bbuffer;
                      item : out item_type)
    procedure put (   signal object : inout bbuffer;
                      item : item_type);
end bounded_buffer_package;
```

The example is slightly modified compared to the previous versions of bbuffer so that the bounded buffer can be instantiated as signal and used in a concurrent environment. The channel concept is added to perform the communication between distributed objects.

For the modelling of the synchronisation code we can identify three states of the interface control space. The buffer might be empty, it might be full, or it might contain a number of items in between these two states. Accordingly, we may name the synchronisation states buffer_empty, buffer_full, and buffer_partial. A get message is only accepted for execution if the buffer is in the state buffer_full or buffer partial. A put operation is only accepted if the buffer is in the state buffer_empty or buffer_partial.
In a sequential object the synchronisation code may look as follows:

```
read (from_client, message);
-- Note: read does not remove message from channel
if   decode (message) = instruction_get'Tag
     and number_of_items_in_buffer > 0 then
          -- buffer_full and buffer_partial
          accept (from_client, to_client, message);
          get (object, item);
          number_of_items_in_buffer := number_of_items_in_buffer - 1;
          message := item;
          send (from_client, to_client,message);
end if;
if   decode (message) = instruction_put'Tag
     and number_of_items_in_buffer < buffersize then
          -- buffer_empty and buffer_partial
          accept (from_client, to_client, message);
          instr := message; -- implicit type conversion to instruction'Class
          item := operand (instr); -- Read operand of instruction
          put (object, item);
          number_of_items_in_buffer := number_of_items_in_buffer + 1;
end if;
```

The example follows the model used in Section 6.4.

A counter number_of_items_in_buffer is used to discern the synchronisation state. The synchronisation code models a conditional acceptance where the object explicitly accepts the messages if it is in an appropriate state.

If we use inheritance to derive a new type with additional or modified properties the body[1] of the original type has to be analysed and the synchronisation code of the derived type has to be re-implemented. As we know from Section 3.3.3, the conditional acceptance of messages makes the object's state of execution observable from outside the object. The observable states may become synchronisation states. Therefore an analysis of the

1. The expression "body of a type" is synonymous with "body of a class".

body is required to preserve compatibility by conformance. This breaks encapsulation and thus causes the inheritance anomaly. We call this kind of anomaly body anomaly as it occurs in objects that have a body to model the synchronisation code.

The same anomaly occurs if synchronisation code is modelled as part of an operation that contains conditional acceptance. Instead of analysing the body an analysis of the operation is required to derive a new type.

Using path expressions to model synchronisation constraints is an approach that shows an anomaly similar to the body anomaly. We could interpret path expressions as a special kind of grammar based specification of extended automata. Such extended automata were discussed in Section 4.1.2 where we already stated a missing robustness of such automata against modifications. If we compare the approach to the modelling of synchronisation code in an object's body the extended automata specified by the expressions correspond to the body and thus suffer from the same anomaly.

8.1.2 Partitioning of acceptable states

Enable sets were introduced as a means for describing the interface control space (Compare Section 3.3.13). An enabled set is associated to each synchronisation state. It represents the synchronisation state in the synchronisation code. The set contains the message keys that are accepted in the associated state. The state transitions in the interface control space caused by the execution of the operations are emulated by state transitions on the enabled sets.

It was stated that enabled sets are objects which are used as attributes to reflectively model the synchronisation. Enabled sets which are part of an object become visible outside the object.

To illustrate the concept we sketch the bounded buffer example adapting a simplified enabled set approach.

```
constant empty : enabled_set_type := def_set (instruction_put'Tag);
constant partial : enabled_set_type := def_set (instruction_put'Tag,
                                                 instruction_get'Tag);
constant full : enabled_set_type := def_set (instruction_get'Tag);
type bbuffer is tagged record
    enabled_set : enabled_set_type; -- has to be initially empty
private
    buf : bounded_buffer_array (1 to buffersize);
    buf_in : integer;
```

```
        buf_out : integer;
    end record;
```

The primitive operations of the buffer are modelled for execution in a concurrent environment. They use signal parameters of type channel to communicate with the object's client. For example, the get operation could be implemented as follows:

```
    procedure get (  signal object : inout bbuffer;
                     signal from_client : in channel;
                     signal to_client : inout channel) is
    begin
        … -- accept get operation, perform get operation, send result to client
        if object.buf_in = object.buf_out then
            -- buffer empty:
            become (object.enabled_set, empty);
        else
            -- buffer not empty:
            become (object.enabled_set, partial);
        end if;
    end;
```

The operation performs its intended behaviour and then emulates its state transition on the enabled sets by calling an operation become which takes the next state as a parameter.

The instructions which are used to define the message keys are modelled as tagged types with an operation execute_instruction. The operation accepts the message key and invokes its corresponding operation of the bounded buffer.

```
    procedure execute_instruction (object : instruction_get;
                                   signal channel_in : in channel;
                                   signal channel_out : inout channel;
                                   signal target : inout bbuffer) is
    begin
        get (target, channel_in, channel_out);
    end;
```

The synchronisation code could then be modelled like this:

```
    read (from_client, message);
    instr := message; -- implicit conversion to instruction'Class
    if message_key_in_enable_set (instr, target.enabled_set) then
        -- dispatching call to execute operation
        execute_instruction (instr, from_client, to_client, target);
    end if;
```

The synchronisation code checks if the object of type bbuffer (target) is in a state that allows to accept the instruction requested by the message. Only if the check does not fail the corresponding operation is executed.

The idea of the synchronisation code is that if we derive a new type from an existing one it is not necessary to modify the code.

Consider the buffer x_buf which was described in Section 5.2.1 as an example. It inherits the operation put and get of the bounded buffer. It additionally has an operation last which removes the last item which was put into the buffer.

We easily can derive a new buffer x_buf from bbuffer. It is not necessary to re-write the synchronisation code it is sufficient to re-define the enabled sets:

```
constant empty : enabled_set_type := def_set (instruction_put'Tag);
constant partial : enabled_set_type := def_set (instruction_put'Tag,
                                                 instruction_get'Tag,
                                                 instruction_last'Tag);
constant full : enabled_set_type := def_set (instruction_get'Tag,
                                              instruction_last'Tag);
type x_buf is new bbuffer with record
    null;
end record;
```

The implementation of the new operation last is similar to the implementation of get.

```
procedure last (  signal object : inout bbuffer;
                  signal from_client : in channel;
                  signal to_client : inout channel) is
begin
    … -- accept last operation, perform operation, send result to client
    if object.buf_in = object.buf_out then
        -- buffer empty:
        become (enabled_set, empty);
    else
        -- buffer not empty:
        become (enabled_set, partial);
    end if;
end;
```

The difficulties with this concept to model condition synchronisation occur if it is necessary to partition a synchronisation state due to some new synchronisation constraints in a derived type. In such a case a new enabled set is required which has to be considered in the emulation of the state transitions in the operations. This means it is necessary to re-implement the operations.

We want to illustrate this by the example of the buffer x_get2. As explained in earlier examples it is derived from bbuffer and has an additional operation get2 which takes two items from the buffer. This new operation introduces new synchronisation states into the model. The operation get2 only can be accepted if the buffer contains more than one item. The derived type partitions the state partial into two states: partial_more_than_one and partial_one. The new states have to be modelled by enabled sets.

```
constant partial_ one : enabled_set_type := def_set (
                                              instruction_put'Tag,
                                              instruction_get'Tag);
constant partial_more_than_one : enabled_set_type := def_set (
                                              instruction_put'Tag,
                                              instruction_get'Tag,
                                              instruction_get2'Tag);
```

The problem is that the inherited operations do not consider the new enabled sets in the emulation of the state transition. If we look at the get operation the state transition must be re-defined and thus the complete operation:

```
procedure get (  signal object : inout x_buf2;
                 signal from_client : in channel;
                 signal to_client : inout channel) is
begin
   … -- accept get operation, perform get operation, send result to client
   if object.buf_in = object.buf_out then
      -- buffer empty:
      become (enabled_set, empty);
   elsif object_buf_in = (object.buf_out + 1) mod (buffersize+1) then
      -- buffer contains exactly one item:
      become (enabled_set, partial_one);
   else
      buffer contains more than one item:
      become (enabled_set, partial_more_than_one);
   end if;
end;
```

Requiring the re-implementation due to the partitioning of the synchronisation state is another form of inheritance anomaly.

A synchronisation technique which allows to partition acceptable states without suffering from the inheritance anomaly is to use guards[2]. As explained in Section 3.3.13, a guard is a boolean expression that is assigned to an operation. If the guard is true then the corresponding operation which is requested by a message or queued in the message queue may be executed. Guards are appropriate to solve the anomaly because they are allowed to refer to the object's state[3]. Different to the enabled set approach the synchronisation modelling with guards does not introduce any redundancy by emulating states which must be modified in case of partitioning a synchronisation state.

It is a special characteristic of VHDL due to the nature of its simulation cycle that it is possible to model a guard of an operation with a simple if statement that is supplemented by some wait statements. The guard is modelled by the condition in the if statement. If the evaluation of a guard fails the operation must not be executed. Then the wait statement notices events when it might be useful to re-evaluate the guard.

With this approach to integrate guards into VHDL and thus into the language extension it is possible to illustrate how guards avoids the anomaly caused by the partitioning of states in the example of x_buf2.

To provide a clear structuring of the behaviour the synchronisation code containing the guard is separated from the functionality. Synchronisation code and functionality are modelled in different operations.

In the example the implementation of the guard operation of the operation get may look as follows:

```
procedure guarded_get (signal object : inout bbuffer;
                       signal from_client : in channel;
                       signal to_client : inout channel) is
begin
    if object.buf_in /= object.buf_out then -- guard checks state empty
        get ( object, from_client, to_client);
    end if;
end;
```

2. This kind of guard is not to confused with the guards of VHDL which are used to guard signals.

3. The details were discussed in Section 3.3.13.

The guard operation only calls the operation get which contains the functionality of get if the guard is evaluated true i.e., if the buffer is not empty. The wait statements required with the guard operations to model the re-evaluation of the guard are sensitive to the signals object and from_client.

If the get operation and the guard operation are inherited by the derived type x_buf2 the partitioning of the accepting states[4] by the new operation get2 does not affect the accepting state of get. The guard which distinguishes for the operation get if the object is in an acceptable state or not is still the same. No analysis or re-implementation of the guard operation is required. The anomaly does not occur.

Unfortunately this is different in situations where acceptable states are modified in an inherited type. Then the guards of inherited operations also require a modification.

8.1.3 Modification of acceptable states

An example where a derived type modifies the acceptable states is the bounded buffer lb_buf which was introduced in Section 5.2.5 and further discussed in Section 6.4.3. The buffer is derived from bbuffer. It has two additional operations lock and unlock. After the execution of the operation lock no operation is accepted for execution until the operation unlock is executed. The new operations modify the acceptable states of the inherited operations by imposing additional synchronisation constraints on the operations.

For example, the operation get of bbuffer has the acceptable states partial and full. These states are not any longer the acceptable state for the operation get in the derived type. The acceptable states of the operation are modified. The modified states require the number of lock operations that are executed to be equal to the number of unlock operations that are executed. We may denote the new acceptable states of the inherited operation get partial_and_unlocked and full_and_unlocked. The inherited guard of operation get that checks that the object is either in state partial or full is not any longer appropriate to distinguish the new acceptable states from the other states. We have to re-implement the guard. The anomaly again has occurred.

To overcome the anomaly the extension to the guard mechanism could be used that was described in Section 3.3.13 and that allowed to add new guards to an inherited operation. The actual guard of an operation is determined by

4. The partitioning of synchronisation states by guards was described in Section 3.3.13 in detail.

accumulating all guards of an operation along the inheritance path in the type or class tree. As described in Section 3.3.13, the actual guard is a refinement of the inherited. That means, if we add a guard that describes the additional synchronisation constraint the refined actual guard distinguishes just the new modified acceptable states from the other states.

In the example of lb_buf we could add an attribute to the class that traces the invocation history of the operations lock and unlock.

```
type lb_buf is new bbuffer with record
private
    lock_on_object: object_locked_or_unlocked;
end record;
```

The new guard which is added to the inherited operation get tests if the object has a lock:

```
object.lock_on_object = unlocked
```

The accumulation of the guards results in the actual refined guard:

```
(object.buf_in /= object.buf_out) and (object.lock_on_object = unlocked)
```

The refined guard just distinguishes the new acceptable states partial_and_unlocked and full_and_unlocked. If we can add the new guard to the operation without re-implementing the existing implementation of the operation then the anomaly caused by modified acceptable states is solved.

8.1.4 History-only sensitiveness of acceptable states

Another category of anomaly occurs if acceptable states of a derived type depend on the invocation history of certain operations and if the acceptable states of the parent's operations do not depend on that history. In Section 6.2.4 we discussed the bounded buffer gb_buf which could serve as an example of a class which has acceptable states that are history-only sensitive. The buffer is a child of bbuffer. It has an additional primitive operation gget. The new operation has the functionality of the get operation. The difference between the get operation and gget operation is that a gget operation only can be executed immediately after a get or a gget operation. The acceptable state only depends on the invocation history of the operation put in relation to the other operations. The original operations put and get do not consider the tracing of this history. This tracing has to be added as new synchronisation code to the inherited operations. It is quite obvious that the tracing cannot be achieved by guards which only refer to an object's state. Adding new code to an inherited operation typically means to re-implement the operation. Requiring this re-implementation means to suffer from the anomaly.

8.2 How to Solve the Inheritance Anomaly

After this categorization of the inheritance anomaly we present a solution for the language extension to VHDL. The proposals to solve the anomaly that can be found in literature are based on special language constructs for synchronisation that are added to the language to avoid the anomaly. This is illustrated by the representatives of the main proposals that were referenced as examples in the previous sections. We saw that none of the proposals was able to avoid all kinds of anomaly.

Therefore a new approach was chosen for the language extension to VHDL. It does not follow the proposals which add new synchronisation constructs to the language by inventing just another new construct. The approach is rather similar to the ideas presented in Section 4.1.4 which use a modelling style that preserves compatibility between protocols to avoid feature interaction.

The solution to avoid all kinds of inheritance anomaly in the language extension to VHDL is a modelling style that preserves compatibility. In Section 3.3.14 we introduced the requirements for modelling a concept that preserves compatibility. The concept uses guards to model the interface control space. The concept allows that new guards are added to inherited operations. The resulting actual guard is determined by accumulating all guards of an operation along the inheritance path in the type tree. It is required for the concept that such new guards of polymorphic operations must model a compatibility mode. The central requirement is that the concept integrates a delegation mechanism into an inheritance mechanism which supports polymorphism. By integrating the delegation mechanism the modelling style overcomes the limitations of the guard mechanism which were discussed in the previous section.

From these requirements we directly develop the modelling style which avoids the inheritance anomaly.

We now come to a detailed description of the modelling style.

8.2.1 Modelling of guarded operations

The most apparent property of the modelling style is that it separates the protocol modelling from the implementation of the functionality of an operation.

An operation is decomposed in a protocol part and a functional part. Both parts are modelled as primitive operations.

```
procedure protocol_part_operation(signal object : inout taggedtype;
                                  signal from_client : in channel;
                                  signal to_client : inout channel);
procedure operation (signal object : inout taggedtype;
                     signal from_client : in channel;
                     signal to_client : inout channel);
```

The operations use signal parameters of a channel type to communicate with other objects.

The body of the protocol operation implements the matching phase and the transition phase of the operation.

```
procedure protocol_part_operation(signal object : inout taggedtype;
                                  signal from_client : in channel;
                                  signal to_client : inout channel) is
    variable message : datum'Class;
begin
    read (from_client , message);
    if decode(message) = op_code then      -- matching phase
        if guard_expression then           -- guard in matching phase
            operation (object, from_client, to_client);   -- transition phase
            additional protocol code;                     -- transition phase
        end if;
    end if;
end protocol_part_operation;
```

The protocol part of the operation implements the matching phase by checking if a message has arrived which requests the execution of the operation and by checking if the object is in an acceptable state. Checking the state is modelled by a guard which is implemented as an if statement. How to model a guard with an if statement was described in Section 8.1.2. If the matching phase decodes the operation and the object is in an acceptable state then the subprogram is invoked which implements the functional part of the operation. The internal call of the subprogram is dynamically bound, that means, for modifying the behaviour it is sufficient to derive a new tagged type which re-implements the subprogram. The inherited protocol then automatically calls the re-implemented subprogram.

It is important to note that the execution of the matching phase as it is implemented in the protocol part does not consume any simulation time. In other words, there is no synchronisation point in form of a wait statement in the matching phase As the time does not proceed in the matching phase neither the message to invoke the operation nor the guard expression can change between the invocation of the operation implementing the protocol and the invocation of the operation implementing the functional part of the operation.

If the matching phase does not decode the operation or the guard expression does not accept the operation then the execution of the operation implementing the protocol part also does not consume any time.

The situation is different if the operation which implements the functionality is executed. This execution consumes time because the operation that implements the functionality contains synchronisation points.

As an example the protocol part of the operation get of the bounded buffer is modelled.

```
procedure protocol_part_get    (signal object : inout bbuffer;
                                signal from_client : in channel;
                                signal to_client : inout channel) is
    variable message : datum'Class;
begin
    read (from_client , message);
    if decode (message) = instruction_get'Tag then
        if object.buf_in /= object.buf_out then -- guard: buffer not empty
            get (object, from_client, to_client);
            -- no additional protocol code required for get;
        end if;
    end if;
end protocol_part_get;
```

Both, protocol_part_get and get are primitive operations implementing the operation of the bounded buffer.

The modelling style described so far explains how to model an operation of a type by separating the operation’s protocol from its functionality. The next step is to show how to introduce delegation as a means for preserving compatibility. If we look at the protocol part the call on the operation implementing the functionality delegates the execution of the functionality to another primitive operation.

To add new synchronisation code in an additional layer between the inherited protocol and the inherited functionality we could re-define the operation implementing the functionality. The re-defined operation contains the new synchronisation code and a call on the overridden inherited operation. Due to the dynamic binding of subprogram calls in primitive operations the call on the operation implementing the functionality is bound to the re-defined operation. Thus the call in the protocol part is bound to the re-defined operation which executes the additional synchronisation code and delegates the execution of the actual functionality to the inherited overridden implementation of the operation.

The additional synchronisation code has to meet the requirements for modelling a concept that preserves compatibility. One of the requirements is that a new guard that is added to a polymorphic operation must model a compatibility mode. According to Section 3.3.13 this is achieved by only allowing to add new guards to a polymorphic operation that are orthogonal to the inherited guards. As explained in Section 3.3.14, orthogonal means that the new guard is not allowed to reference inherited record elements of the derived type as these record elements may already be referenced by the inherited guards.

The re-defined operation may contain among new guards any other kinds of additional synchronisation code. However, to preserve the compatibility this new code must not remove or modify any existing state transitions in the inherited code. To guarantee such a behaviour without the necessity to analyse the inherited code the new synchronisation code must not perform any write access to inherited record elements.

Another requirement to be compatible is that the resulting actual guard is determined by accumulating all guards of an operation along the inheritance path in the type tree. To achieve an atomic evaluation of the actual guard the guards must not be separated by any synchronisation points and the call that delegates the execution to the overridden implementation of the operation always must immediately follow a guard.

Following this modelling style an implementation of a re-defined operation that adds synchronisation code to an inherited operation looks like this:

```
procedure operation (signal object : inout derived_type;
                     signal from_client : in channel;
                     signal to_client : inout channel) is
begin
   if new_guard_expression then -- new orthogonal guard
      operation'Parent (object,from_client, to_client)'Static; --delegation
      additional protocol code; -- no write access to
                                -- inherited record elements
   end if;
end operation;
```

Please note, like the original protocol operation the operation implementing the additional synchronisation constraints does not consume any simulation time if the guard fails, that means, if the operation is not accepted. We have now introduced how to model the operations together with their associated synchronisation code.

8.2.2 How to model a body of an object

In this section we describe the integration of the complete behaviour of an object that has of a set of primitive operations which are called by synchronous message passing. We integrate the behaviour in an object body. It is implemented as a sequential procedure call which calls a primitive subprogram object_body of the tagged type. The call is part of a process that contains the call and a wait statement which is used to supplement the guard modelling in the primitive operations. The wait statement is executed when the object has reached a stable state.

```
process
    variable old_state : tagged_type;
begin
    object_body ( object => actual_object,
                  from_client => actual_from_client,
                  to_client => actual_to_client);
    if old_state /= actual_object then
        old_state := actual_object;
    else
        wait on actual_object, actual_from_client;
    end if;
end process;
```

The implementation of the procedure consists of sequential procedure calls invoking all operations of the tagged type in a sequential order.

```
procedure object_body (  signal object : inout tagged_type;
                         signal from_client : in channel;
                         signal to_client : inout channel) is
begin
    protocol_part_operation (object, from_client, to_client);
    protocol_part_another_operation (object, from_client, to_client);
end;
```

The procedure object_body is executed when the state of the object changes or a message comes in. Then the protocol part of an operation is executed.

The protocol decodes the message and should the situation arise that the corresponding message key of the operation was decoded it evaluates the guards if necessary along an inheritance chain. If the guards are true the corresponding operation performing the functionality is executed. After executing the functionality additional protocol code may be executed. This completes the call in the primitive operation object_body. The protocol part of the next primitive operation is called.

If the decoding fails the corresponding functionality is not executed. The next primitive operation in object_body is called. If the evaluation of a guard fails the corresponding operation is not executed. The operation is re-scheduled for the time when the object's state has changed or any incoming message causes an event. Then it is checked if the message still decodes the corresponding operation and the guard is re-evaluated. Please note, in case of a re-evaluation a guard is evaluated along the complete inheritance chain if necessary. This is illustrated in Fig. 14 which uses some pseudo code to demonstrate the control flow.

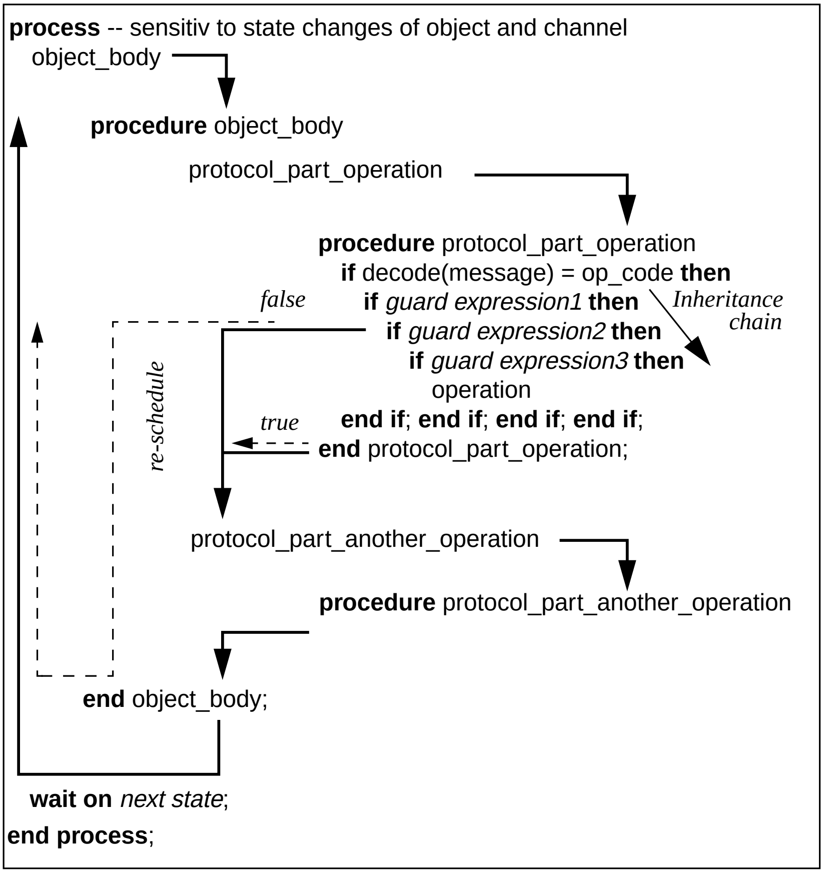
As explained in Section 8.1.1, the conditional acceptance of messages as part of a thread makes the state of execution visible outside the object. The observable states are just the synchronisation states of the object. In a body following the modelling style the only messages that are accepted cause an execution of an operation. The observable state transition is just the expected result of executing the operation. Thus, a body following the modelling style does not suffer from the body anomaly.

To preserve compatibility and to avoid the body anomaly the only allowed extensions that might be added to an implementation of a body are calls on primitive operations. Following the idea to use delegation for preserving compatibility an extended body may look as follows:

```
procedure object_body (   signal object : inout derived_type;
                          signal from_client : in channel;
                          signal to_client : inout channel) is
begin
    protocol_part_new_operation (object, from_client, to_client);
    object_body'Parent (object, from_client, to_client);
end;
```

The call to the protocol part of a new operation is simply added to the sequence of calls on the other operations.

In a similar way it is possible to add new channels to a body in order to allow more than one client to send messages to an object.



```
process -- sensitiv to state changes of object and channel
   object_body
         procedure object_body
               protocol_part_operation
                       procedure protocol_part_operation
                          if decode(message) = op_code then
                             if guard expression1 then
                                if guard expression2 then
                                   if guard expression3 then
                                      operation
                          end if; end if; end if; end if;
                       end protocol_part_operation;
               protocol_part_another_operation
                       procedure protocol_part_another_operation
         end object_body;
   wait on next state;
end process;
```

Fig. 14: Re-scheduling of an operation

```
   procedure object_body (    signal object : inout derived_type;
                              signal from_client : in channel;
                              signal to_client : inout channel;
                              signal from_another_client : in channel;
                              signal to_another_client : inout channel) is
   begin
      object_body ( object => object,
                    from_client => from_client,
                    to_client => to_client);
      object_body ( object => object,
                    from_client => from_another_client,
                    to_client => to_another_client);
    end;
```

The new body which contains additional channels overloads the inherited one. The implementation delegates the execution to the inherited operations by calling the inherited operations in a sequential order. As the decoding in the operations does not consume any simulation time we could say that it is performed quasi-concurrent. The execution of the functionality is performed in a sequential order and thus provides a mutual exclusion synchronisation.

In case of re-scheduling an operation the effective values of the signals referenced by the guard are used for re-evaluation. This may include values of the channels. In other words, it is possible to peek into incoming messages to perform condition synchronisation. This allows to do synchronisation without a re-queuing mechanism which would cause compatibility problems in the modelling of the object body (Compare Section 3.3.15).

We now have presented the complete modelling style which allows to overcome the inheritance anomaly. In the following sections we illustrate the solution by examples. We discuss one example for each category of anomaly.

8.2.3 Partitioning of acceptable state in an example

As it was shown before, the anomaly caused by the partitioning of acceptable states can be solved by the guard mechanism. The new operation which causes the partitioning of the states just has a new guard expression which partitions the state. We already mentioned the buffer x_buf2 as an example where a guard mechanism would be able to avoid the anomaly. It is straightforward to model the new guard for the new operation get2 of the derived type x_buf2.

```
procedure protocol_part_get2 (signal object : inout x_buf2;
                              signal from_client : in channel;
                              signal to_client : inout channel) is
begin
   read (from_client , message);
   if decode (message) = instruction_get2'Tag then
      if (object.buf_in /= object.buf_out) and
        not ((object.buf_out + 1) mod (buffersize + 1) = object.buf_in)
      then -- guard: buffer contains more than one item:
         get2 (object, from_client, to_client);
         -- no additional protocol code required for get2;
        end if;
    end if;
end protocol_part_get2;
```

The guard which checks if there are at least two items in the buffer partitions the state partial. Each of the new resulting states behaves the same as the old state partial with respect to the inherited methods put and get. There is no anomaly.

8.2.4 Modification of acceptable states in an example

The buffer lb_buf was described in Section 8.1.3 as an example of a derived type that requires a modification of acceptable states to model condition synchronisation. The solution that was proposed used a derived type lb_buf with an new record element that traces the invocation history of the new operations lock and unlock.

```
type lb_buf is new bbuffer with record
    lock_on_object: object_locked_or_unlocked;
end record;
```

It was proposed to add new guards to the inherited operations that test if the object is locked. The accumulation of the guards results in an actual refined guard. This is modelled in the modelling style by a re-defined operation that adds the guard to the inherited operations. Consider the operation get:

```
procedure get (  signal object : inout lb_buf;
                 signal from_client : in channel;
                 signal to_client : inout channel) is
begin
    if (object.lock_on_object = unlocked) then -- new guard
            get'Parent (object, from_client, to_client)'Static;
    end if;
end protocol_part_get;
```

The new guard of the re-defined operation is orthogonal to the existing ones as it is required by the modelling style. It only references the new record element lock_on_object of the derived type. The derived type preserves compatibility without analysing or re-implementing existing synchronisation code. The anomaly does not occur.

8.2.5 History-only sensitiveness of acceptable states in an example

As an example of a type that has a history-only sensitiveness of acceptable states the buffer gb_buf was mentioned in Section 8.1.4. It is derived from bbuffer and has an additional operation gget. The operation must not be accepted immediately after a put operation was executed.

The modelling approach following the modelling style derives a type gb_buf that has a new attribute to store the invocation history of the put operation. The guard of the new operation gget references this history. It is necessary to add synchronisation code to the inherited operations that traces the history.

```
type gb_buf is new bbuffer with
    after_put: boolean; -- traces invocation of put
end record;

procedure protocol_part_gget (signal object : inout gb_buf;
                              signal from_client : in channel;
                              signal to_client : inout channel) is
begin
    read (from_client , message);
    if decode (message) = instruction_gget'Tag then
        if not ( object.buf_in = object.buf_out)
        and (object.after_put = false) then
            gget (object, from_client, to_client);
        end if;
    end if;
end;

procedure get (  signal object : inout gb_buf;
                 signal from_client : in channel;
                 signal to_client : inout channel) is
begin
    -- always: guard = true
    get'Parent (object, from_client, to_client)'Static;
    object.after_put <= false;
    wait …      -- add a synchronisation point to the operation so that the
                -- assigned value becomes the current value of the signal
end;

procedure put (  signal object : inout gb_buf;
                 signal from_client : in channel;
```

```
                    signal to_client : inout channel) is
   begin
      -- always: guard = true
      put'Parent (object, from_client, to_client)'Static;
      object.after_put <= true;
      wait …      -- add a synchronisation point to the operation so that the
                  -- assigned value becomes the current value of the signal
   end;
```

The additional synchronisation code that is added to the inherited operations executes an assignment to the new record element to trace the history. Following the coding style, it does not perform a write access to any of the inherited record elements and thus preserves compatibility. A wait statement has to follow the assignment that traces the history so that the assignment becomes effective. According to the modelling style this is allowed as the synchronisation point is reached after the call on the parent’s implementation of the operation.

These examples how we solve the different categories of inheritance anomaly conclude the presentation of the object-oriented language extension to VHDL and the discussion about its application for the design of distributed objects.

Chapter 9

Conclusion

This thesis studies a methodical approach to the object-oriented specification and design of hardware systems with a language extension to VHDL.

We describe the basic design principles for the modelling of hardware at system level. The development of a methodology for the design of robust, technology-independent, and re-usable models is recognized as a major challenge to reduce the design effort of hardware systems.

The results of a survey how to apply system level specification methodologies and languages for hardware design demonstrate that such methodologies and languages provide only limited support for an integration in a hardware design flow. Hardware description languages integrate well in existing design flows but tend to have deficiencies in the design of system level models.

This thesis provides an approach to extend the hardware description language VHDL by introducing object-oriented concepts into the language. It is a typed-based extension that adds encapsulation, inheritance, and polymorphism to the type concept of VHDL. We develop an extension that avoids any incompatibility with the existing language standard. The extension preserves the original philosophy of the language with the strong typing and the deterministic simulation semantics. This is an original contribution to this thesis.

We identify compatibility issues between polymorphic objects as crucial for re-use concepts and maintenance strategies at system level. An analysis of other language extensions to VHDL shows that this aspect is not adequately considered in the corresponding approaches.

In the considerations on compatibility issues we focus on the three key problems of modelling concurrency, synchronisation, and distribution. We diagnose the interference of inheritance and synchronisation that is known as inheritance anomaly to cause the problems.

A key contribution of this thesis is to formulate a modelling style that facilitates the inheritance and modification of synchronisation code while preserving the inherited synchronisation constraints. It is a design strategy that does not only solve a limited category but all kinds of inheritance anomalies. The general ideas and principles behind the strategy are presented and the application of the modelling style is illustrated.

We argue that integrating the language extension in existing design flows plays an important role in the applicability of the design approach. To this end, we develop a translation technique that provides a link to further design steps using standard VHDL.

The idea to define object-oriented VHDL enhancements was taken up by the IEEE VHDL standardization committee which established a study group to discuss proposals for object-oriented extensions to VHDL. From the viewpoint of this thesis many features proposed for a new standard in the study group are compatible to the language extension in this thesis. It is hoped that this thesis fruitfully contributes to the discussion about a future standardization of language extensions to VHDL.

Appendix A: Syntax Summary

This Appendix describes the syntax of the language extension to VHDL. The Appendix does not summarize the complete syntax of the new language it rather lists those syntax rules that are affected by the new language constructs. The syntax rules are described applying the variant of Backus-Naur Form that is already used in the language reference manual of VHDL.

```
composite_type_definition ::=
      array_type_definition
    | record_type_definition
    | tagged_record_type_definition
    | derived_type_definition

tagged_record_type_definition ::=
      tagged record
          record_element_part
      end record
    | null record

record_element_part ::=
      private_element_list
    | element_list

private_element_list ::=
    [ element_list ]
    private element_list

element_list ::=
        element_declaration
      {element_declaration}
    | null;
```

```
derived_type_definition ::=
    **new** identifier **with** record_extension

record_extension ::=
      **record**
          record_element_part
      **end record**
    | **null record**

procedure_call ::=
    *procedure*_name [( actual_parameter_part )]['attribute_designator]

use_clause ::=
    [ **postponed** ] **use** selected_name { , selected_name } ;
```

Appendix B: List of Abbreviations

This Appendix explains abbreviations and acronyms that are used in the thesis.

ADT: Abstract Data Type

ASIC: Application Specific Integrated Circuit

CORBA: Common Object Request Broker Architecture

CSP: Communicating Sequential Processes

DASC: Design Automation Standards Committee

DSP: Digital Signal Processing

HCFSM: Hierarchical Concurrent Finite State Machine

HDL: Hardware Description Language

HDF: Hardware Data Flow

HML: Hardware ML

IEEE: Institute of Electrical and Electronics Engineers

IP: Intellectual Property

ITU: International Telecommunication Union

LISP: List-Processing Language

LOTOS: Language of Temporal Ordering Specifications

LRM: Language Reference Manual (of VHDL)

ML: Meta-Language

OMT: Object Modeling Technique

OODFD: Object-Oriented Data Flow Diagram

OID: Object Interaction Diagram

SDL: Specification and Description Language

RPC: Remote Procedure Call

VHDL: Very High Speed Integrated Circuit Hardware Description Language

Appendix C: Glossary

This glossary contains descriptions of the terms used in this thesis. The terminology that is listed in the glossary may be quite different to the one a reader knows from other object-oriented methodologies and languages. It is a consequence from the fact that in the object-oriented domain every language uses its private terminology. This makes it sometimes difficult to use a common terminology when comparing different languages or language extensions. In such a case we decided to refer to the terminology which is described here and to adapt the terms from the original sources.

Compared to the introduction of the terminology in the text we provide a more specific explanation of some terms in the glossary. Basically, we apply the explanations to classes and objects. Some of the terms are used with different meanings in different contexts. In such a case we list the different meanings as an illustration of overloading.

Abstract Class: A class that does not have an implementation of its operations is called abstract class. Its operations only appear in the interface of the object.

Action: In a state-oriented view certain object states and transitions between them may be explicitly modelled in a meta-model. It may be possible to attach a particular behaviour to such a state transition. This behaviour is called action. The action together with the state transition in the meta-model forms an operation.

Active Object: An object that has one or more activities is called active object.

Activity: In a state-oriented view certain object states and transitions between them may be explicitly modelled in a meta-model. Entering such a state in the meta-model may cause a certain behaviour of the object which is called activity. Activities may be interrupted in their execution.

Activity: An activity is an access to some kind of resource.

Abstract Data Type: An abstract data type is a data type that abstracts its data structure. The data structure is only accessible via the operations which characterize the data type. From an object-oriented view an abstract data type defines a set of objects with a set of operations that characterize the behaviour of those objects.

Abstraction: Abstraction means to omit information that is not relevant for certain views of a system.

Abstraction Boundary: The abstraction boundary is the interface a class or object presents to its clients. It abstracts the resources a client can access. In other words, it hides and encapsulates the information that is inside a class or object.

Aggregation: In OMT an aggregation is an association that expresses a whole-part relationship.

Assertion: A precondition, a postcondition, or an invariant is an assertion in the design by contract approach.

Association: In OMT an association is a relationship containing information that is relevant for a certain time during the existence of the described system.

Asynchronous Message Passing: Asynchronous means that a sender sends a message without blocking until the receiver accepts it.

Attribute: An attribute is a state variable. It establishes a has-a relation by being a container of a sub-object. It stores the object's state. The has-a relation that is defined by the attribute may also be modelled as a reference to another object.

Autonomy: An active object is considered to be autonomous if its state is automatically protected from non synchronized concurrent accesses of its clients.

Behaviour: Behaviour is a property of an object or class. It is formed by the operations of an object or class.

Behavioural Compatibility: If two objects have the same behaviour from a client's point of view they are considered to be behavioural compatible.

Channel: A channel is a (predefined) object that provides an implementation of a communication mechanism between other objects. Its interface abstracts the communication.

Child: A child is a class that is derived from a parent class.

Class: A class is a collection of objects with the same properties. They are all defined by the same (class-) definition which describes the common properties of the objects.

(Class-) Hierarchy: Class-hierarchy is a hierarchy which is based on the derivation of classes.

Class-Wide Type: In a typed class concept a class-wide type is the union of all objects of its associated type and its derived types.

Client: The client object in a clientship is called client.

Clientship: The clientship is a relation between two objects or classes. The relation describes how one object uses the operations of another object. The client object uses the operations of the server object.

Compatibility by Conformance: A class conforms to another class if it can be used in all contexts where the other class is expected. It is able to use the same protocol as the other class. The class is compatible by conformance to the other class.

Component Class: In VHDL++ a component class is a class that uses ports and protocols to describe its interface.

Concurrency: Two objects or processes are concurrent if they have the potential for executing in parallel.

Concurrent Language: In a concurrent language a model may have multiple active threads.

Concurrent Object: An object that may have multiple active threads of control at the same time is called concurrent object.

Condition Synchronisation: To accept a message for execution a process or an object may require that particular synchronisation conditions are fulfilled. This form of synchronisation is called condition synchronisation.

Conditional Critical Region: Mutual exclusion synchronisation can be used in combination with condition synchronisation. The critical region is entered only when the synchronisation condition is fulfilled. Such a critical region is called conditional critical region.

Conformance: A class conforms to another class if it can be used in all contexts where the other class is expected. A more formal definition is: A class B conforms to a class A if B subsumes the operations of A and if the following conditions are met: The invariants of B imply the invariants of A. For each operation of A the precondition of A implies the precondition of B and the postcondition of B implies the postcondition of A.

Contra-Variant Rule: The contra-variant rule requires that each argument of an operation of a parent class conforms to the corresponding argument of the corresponding operation of a child class.

Co-Variant Rule: The co-variant rule requires that each argument of an operation of a child class conforms to the corresponding argument of the corresponding operation of its parent class.

Critical Region: A sequence of activities that performs an access to a critical resource is called critical region.

Critical Resource: A shared resource which requires a protected access is called critical resource.

Critical Section: Critical Region

Deferred Implementation of Classes: If a model references an abstraction of a class before its implementation is modelled the implementation is a deferred implementation.

Delegation: A child implements its inherited properties by instantiating a parent object and by forwarding all messages concerning these properties to the parent.

Derivation: Derivation is the construction of a new class by inheritance.

Design by Contract: Describing a clientship as a contract by using assertions is called design by contract.

Dispatcher: In concurrent programming a dispatcher is a special process that allocates resources to processes or threads. This must not be confused with the term dispatching as it is used in the object-oriented domain.

Dispatching: A call on a primitive operation may have controlling operands of a class-wide type. The tag fields of the controlling operands indicate a particular tagged type. The operation that characterizes this tagged type is executed. In the object-oriented domain such a late binding of an operation is called dispatching.

Distribution Boundary: The distribution boundary is the boundary of names, identifiers, references etc., that are visible when looking outward from within an object or class. The distribution boundary abstracts the environment in which an object may be used.

Dynamic Binding: Late Binding

Dynamic Model: In OMT the dynamic model is a state-oriented view on the behaviour of objects.

Dynamic Polymorphism: Polymorphism

Encapsulation: Relation between a model and its abstraction.

Encapsulation Boundary: Encapsulation boundary is another term for abstraction boundary.

Entry Point: An entry point is an interaction point that abstracts the receiving of messages.

Event Flow Diagram: In OMT an event flow diagram describes objects and the events they send to each other.

Exclusion Synchronisation: Mutual exclusion synchronisation.

Exclusive Sub-Object: Sub-object that can be regarded as a private resource of the class or object containing or referencing the sub-object.

Exploration of Design Space: Collecting information about various alternative implementations.

Feature: At the top level of a protocol hierarchy a service primitive of a protocol is called feature in intelligent network specification.

Feature Interaction: Adding a new feature to a protocol may cause the new feature to interfere with existing ones and thus causing compatibility problems between the original protocol and the extended one. Such an interference is called feature interaction in intelligent network specification.

Functional Model: In OMT the functional model is a view that describes operations and their effects on objects in a system.

Generalization: Generalization is to abstract from certain aspects in a model to obtain a model that can be used in various contexts. In OMT it denotes the relationship that implies generalization.

Guard: A guard is a boolean expression that is used to specify synchronisation constraints for an operation that is requested by a message to be executed . The operation is only executed if the boolean expression is true.

Guarded Operation: An operation that has a guard to specify its synchronisation constraints is called guarded operation.

Has-a Relation: A relation where one class or object contains an object of another class. The object is part of the other class or object.

Has-Parts-Relation: Has-a Relation

Heterogeneous Object Container: Classes derived from each other may have different types in a static typed language. An object that may contain objects of different classes and types all belonging to the same class-hierarchy is called heterogeneous object container.

Implementation: An implementation of an object is the realization of its properties.

Implementation: An implementation of an operation, module, procedure, or package is its body.

Implementation: An implementation is the result of a translation or synthesis step.

Implementation: The system which is ready for application is an implementation.

Inheritance: Relation between two classes in which one class (subclass) takes all the properties from the other class. (One class is derived from the other.) The properties of the subclass can be extended or modified by additional attributes or operations.

Inheritance Anomaly: Inheritance and the modelling of synchronisation constraints may conflict with each other in object-oriented concurrent languages. The conflict typically makes it necessary to re-analyse and re-implement inherited operations and thus negates a main advantages of object-oriented modelling. The conflict is known as the inheritance anomaly.

Inheritance Hierarchy: Hierarchy that is formed by is-a relations between classes.

Instance Variable: An instance variable is an attribute that stores a reference to an object.

Instance Variable: In the LaMI proposal an instance variable is an attribute of an object. The attribute contains values of a give type.

Intellectual Property: Model that contains the essential ideas of a component or system that makes its value.

Interaction Point: An interaction point is an abstraction of an interaction between concurrent processes. In some modelling methodologies an interaction point is denoted as port.

Interface Control Space: The representation of a set of messages that can be accepted by an object at a given moment is called the interface control space.

Invariant: An invariant is a statement about properties of an object or class that always must be satisfied.

Is-a Relation: Relation between classes that is established by inheritance.

Late Binding: An object may belong to different classes during run-time. The implementation of an operation to execute may be determined during run-time according to its present class. This is a late binding of the operation.

Locus of Control: A locus of control is part of a thread control block and points to the activity of a thread to be executed.

Message: Information that is sent from one object to another, typically to request the execution of an operation is called message.

Method: A method is an operation of an object or class.

Mixin Inheritance: Mixin inheritance is the combination of static and dynamic polymorphism to achieve effects similar to multiple inheritance.

Multiple Inheritance: A child is allowed to inherited features from different parents at the same time.

Mutual Exclusion Synchronisation: A synchronisation that allows at most one thread to access a shared resource at a time is called mutual exclusion synchronisation.

Object: An object is a module that has particular properties. It is a combination of a data-structure and the operations which can be applied to that data structure.

Object Interaction Diagram: In OMT an object interaction diagram is a procedural description of the operations. Operations are described by invocations of other operations in various objects in the system.

Object Model: In OMT an object model is the basic view which characterizes the static structure of the system in terms of objects and classes and their relation to each other.

Object Modeling Technique: Object-oriented specification and design methodology from J. Rumbaugh.

Object-Oriented Data Flow Diagram: In OMT an object-oriented data flow diagram describes the data flow between objects and the transformation of the attribute values in that flow by operations of objects regarded as functions.

Operation: Subprogram which implements the behaviour of an object or class. Operations sometimes are called methods.

Overloaded Subprogram: An identifier may denote two or more subprograms at the same time. In that case, the subprogram is called overloaded subprogram.

Overridden Operation: An overridden operation is an inherited operation which is re-defined.

Passive Object: Objects that are not active objects are referred to as passive objects. They do not have their own thread of control.

Pattern: Meta-model which describes particular aspects of a design flow.

Parent: A parent is a class that is used to derive another class from it by inheritance.

Polymorphism: Dynamic polymorphism or simply polymorphism means that an object can belong to different classes during run-time and the classes contain different implementations of operations which have the same name. If a message is sent to an object to invoke an operation the class of the object can be determined only during run-time. After identifying the class the corresponding operation can be executed. If a generic class description is used to describe the class of an object it can belong to different classes at elaboration time. This is called static polymorphism.

Postcondition: Responsibilities of a server are called postconditions in a contract between server and client that describes the clientship.

Precondition: Responsibilities of a client are called preconditions in a contract between server and client that describes the clientship.

Primitive Operation: A primitive operation is an operation of an object or class. In the context of the language extension to VHDL a primitive operation is an operation which belongs to a tagged type. The primitive operations of a tagged type include its predefined operations, its basic operations and they include procedures with a parameter of mode in or inout of the type which are declared together with the tagged type in the same package declaration

Process: A process is an object that consists of one or more threads. It does not have operations it rather uses synchronisation points to synchronize with other processes. The services of a process are abstracted by interaction points.

Property: The property of an object or class is determined by its structure and its operations.

Protocol: A rule that governs the communication between objects is called protocol. The contract between server and client is a part of the protocol.

Proxy: If synchronous and asynchronous message passing concepts are mixed a sender may send its message asynchronously to an object that serves as a proxy for the actual receiver. It performs the actual synchronisation and communication with the receiver.

Quasi-Concurrent Language: In a quasi-concurrent language a model may have multiple independent threads but only one active thread at any moment.

Quasi-Concurrent Object: An object that may have several threads of control with at most one active thread.

Reflection: Reflection is the capability of a system to model and modify its own behaviour. Reflective modelling means the ability to describe and modify a meta-model that is part of the original modelling concept.

Reply Scheduling: In asynchronous message passing a sender has to become a receiver to receive a reply from the original sender. The former sender has to await this reply and accept it. The explicit receiving of such a reply is called reply scheduling.

Re-queuing: A message from the message queue requesting the execution of an operation is accepted for execution by a server. During the execution of the operation a new message is put by the server into the message queue which replaces the original one. The synchronisation conditions for accepting the new message may refer to the new state or history of the server which is caused by the execution. The replacement is called re-queuing.

Role: In OMT a role describes how a member of an association is viewed by the other members.

Rolename: In OMT a rolename is a name that can be used to denote a role.

Scheduler: A scheduler is a special process that allocates resources, especially processor time, to processes or threads.

Sequential Language: In a sequential language a model has only a single thread of control.

Sequential Object: An object that has a single thread of control is a sequential object.

Server: The server object in a clientship is called server.

Service: A service is an operation that a server provides to its clients.

Service Interference: Feature interaction.

Signature: A signature is the syntactic structure of an object's or class's interface

Signature Compatibility: Signatures are compatible if one signature subsumes the other.

Simulation Semantics: A language has a simulation semantics if there is a mapping from a model written in that language to an algorithm that can be executed on a (simulation) machine.

State Diagram: In OMT a state diagram is a diagram that describes the behaviour of an object using a variant of the StateCharts' notation.

State Variable: The variables of an object or class that contain the object's state are called state variables. They characterize the data structure of an object or class.

Static Polymorphism: Polymorphism

Structure: Structure is a property of an object or class. It denotes the data structures that are used to store an object's state. Typically, the structure of an object is characterized by its state variables.

Subclass: In OMT the term subclass is another word for child class.

Sub-Object: An object or class may have an object as an attribute. The attribute is called sub-object. They may also have an attribute that references an object. The object is also called sub-object, particularly, if the class or object encapsulates the access to its sub-object.

Superclass: In OMT the term subclass is another word for parent class.

Synchronisation: Interaction and coordination of activities in different threads is called synchronisation.

Synchronisation Constraint: The acceptance of a message may be constrained by certain conditions that must be met before it is accepted. Such a condition is called synchronisation constraint.

Synchronisation Point: A synchronisation point is an activity which synchronizes the execution of its thread with some events typically caused by other threads.

Synchronisation State: State information of an object that is needed for synchronisation purposes is called synchronisation state.

Synchronous Message Passing: Synchronous message passing means that a sender does not perform any activity until the receiver accepts and executes the message.

Synthesis Semantics: A language has a synthesis semantics if there exists a meta-model how to transform a model written in the language with the synthesis semantics into a new model by adding information to the model

Target Independent: A model is target independent if it generalizes about a technology that is used to implement it.

Thread: A thread of control is a sequential series of activities.

Trigger Event: A trigger event is an event that causes a state transition in an object.

Use-relation: A use-relation is a clientship in which a server is not an exclusive sub-object of a client.

Verification Condition: Assertions can be used to describe proof annotations. Theorems that are generated from the proof annotations are called verification conditions.

View: A modelling technique to show only a particular aspect of a system is called view. A model that shows a particular aspect of a system is also called view.

Whole-Part Relationship: In OMT whole-part relationship is the term for a has-parts-relationship.

References

[1] Agsteiner, K.; Monjau, D.; Schulze, S.: Object-Oriented High-Level Modeling of System Components for the Generation of VHDL Code. Proceedings of the EURO-DAC'95 with EURO-VHDL'95. IEEE Computer Society Press, 1995

[2] Aksit, M.; Bosch, J.; van der Sterren, W.; Bergmans, L.: Real-Time Specification Inheritance Anomalies and Real-Time Filters. in Tokoro, M.; Pareschi, R. (eds): European Conference on Object-Oriented Programming ECOOP '94, Lecture Notes in Computer Science 821, Springer 1994

[3] America, P.: Inheritance and Subtyping in a Parallel Object-Oriented Language. in Bézivin, J.; Hullot, J-M.; Cointe, P; Lieberman, H. (eds.): European Conference on Object-Oriented Programming ECOOP '87, Lecture Notes in Computer Science 276, Springer 1987

[4] Ashenden, P. J.: The Designer's Guide to VHDL. Morgan Kaufmann Publishers, Inc. USA, 1996

[5] Ashenden, P. J.: A Comparison of Alternative Extensions for Data Modeling in VHDL. Technical Report TR-02/97 Department of Computer Scinece, The University of Adelaide, Australia, 1997, Technical Report TR-203/05/97/ECECS Department of Electrical and Computer Engineering and Computer Science, University of Cincinnati, USA, 1997

[6] Ashenden, P. J.: Principles for Language Extension to VHDL to Support High-Level Modeling. Technical Report TR-03/97 Department of Computer Scinece, The University of Adelaide, Australia, 1997, Technical Report TR-204/05/97/ECECS Department of Electrical and Computer Engineering and Computer Science, University of Cincinnati, USA, 1997

[7] Ashenden, P. J.; Wilsey, P., A.: Considerations on Object-Oriented Extensions to VHDL. in VHDL: The Next 10 Years. Proceedings of the VIUF Spring 1997 Conference, 1997

[8] Ashenden, P. J.; Wilsey, P. A.: Abstraction of Concurrency and Communication in VHDL. Second Workshop on System Level Design

Languages, Barga, Italy, July 1997
[9] Ashenden, P. J.; Wilsey, P. A.; Martin, D. E.: SUAVE: A Proposal for Extensions to VHDL for High-Level Modeling. Joint Technical Report, TR-97-07, Dept. Computer Science, University of Adelaide, South Australia, and TR-207/08/97/ECECS, Department of Electrical and Computer Engineering and Computer Science, University of Cincinnati, 1997
[10] Ashenden, P. J.; Wilsey, P. A.; Martin, D. E.: Reuse Through Genericity in SUAVE. Proceedings of VIUF Fall 97 Conference, Arlington, VA, 1997
[11] Ashenden, P. J.; Wilsey, P. A.; Martin, D. E.: SUAVE: Painless Extension for an Object-Oriented VHDL. Proceedings of VIUF Fall 97 Conference, Arlington, VA, 1997
[12] Ashenden, P. J.;Wilsey, P. A.: A Comparison of Alternative Extensions for Data Modeling in VHDL, Proceedings of Hawai'i International Conference On System Sciences, Kona, Hawaii, 1998
[13] Ashenden, P. J.;Wilsey, P. A.: Considerations on System-Level Behavioural and Structural Modeling Extensions to VHDL. Proceedings of VIUF Spring 98 Conference, Santa Clara, California, 1998
[14] Ashenden, P. J.; Wilsey, P. A.; Martin, D. E.: SUAVE: Extending VHDL to Improve Modeling Support. IEEE Design and Test of Computers, 1998
[15] Ashenden, P. J.;Wilsey, P. A.: Extensions to VHDL for Abstraction of Concurrency and Communication. Proceedings of Sixth International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS '98), Montreal, Canada, 1998
[16] Ashenden, P. J.; Wilsey, P. A.; Martin, D. E.: SUAVE: Object-Oriented and Genericity Extensions to VHDL for High-Level Modeling. Proceedings of Forum on Design Languages (FDL '98), Lausanne, Switzerland, 1998
[17] Atkinson, C.: Object-Oriented Resuse, Concurrency and Distribution : an Ada-Based Approach. ACM Press, 1991
[18] Barnes, J.: Introducing Ada 9X. Intermetrics Inc., 1993
[19] Barnes, J.: Programming in Ada 95. Addison-Wesley Publishing Company, 1996
[20] Barnes, J.: High Integrity Ada: The Spark Approach. Addison-Wesley, 1997
[21] Benzakki, J.; Djafri, B.: Object Oriented Extensions to VHDL–The

LaMI proposal. CHDL’97, Toledo, Spain, 1997

[22] Beolet, P.; Bergé, J.; Tagant, A.; Le Maire, C.: Participation in the definition of Needs&Requirements and Analysis of existing proposals in the definition of Object Oriented Extensions to VHDL. Version 1.0

[23] Bergé, J.; Nebel, W.; Putzke, W.: Requirements and Design Objectives for an Object-Oriented Extension of VHDL (OO-VHDL). Design Objectives Document of the DASC Study Group on OO Extensions to VHDL, August 1996

[24] Bergé, J.; Fonkoua, A.; Maginot, S.; Rouillard, J.: VHDL designer’s reference. Kluwer Academic Publishers, 1992

[25] Bergmans, L.; Aksit, M.; Wakita, K.; Yonezawa, A.: An Object-Oriented Model for Extensible Concurrent Systems: The Composition-Filters Approach. Available on the WWW from URL http://wwwtrese.cs.utwente.nl/Docs/Tresepapers/tresepapers.html

[26] Bergmans, L.: Composing Concurrent Objects –Applying Composition Filters for the Development and Reuse of Concurrent Object-Oriented Programs. Ph.D. thesis, University of Twente, 1994

[27] Bloom, T.: Evaluating Synchronization Mechanisms. Seventh International ACM Symposium on Operating Systems Principles, ACM, 1979

[28] Booch, G.: Object oriented design: with applications. Redwood City, Calif., Benjamin/Cummings, 1991

[29] Booch, G.: Object-Oriented Development. IEEE Transactions on Software Engineering, Volume SE-12, Number 2, February 1986, pp. 211-221, 1986

[30] Böttger, J; Ecker, W.: Comparing Ada’95 and VHDL for Behavioural Hardware Description on Causal and Synchronous Level. Proceedings of the SIG-VHDL Spring Working Conference: VHDL User’s Forum in Europe, Publication Service of the University of Cantabria, 1997

[31] Bracha, G.; Cook, W.: Mixin-based Inheritance. in Meyrowitz, N. (ed): OOPSLA ECOOP’90 Conference Proceedings. ACM Sigplan Notices, Vol 25, No 10, 1990

[32] Bredereke, J.; Gotzheim, R.: Specification, Detection and Resolution of IN Feature Interactions with Estelle. 7th International Conference on Formal Description Techniques (FORTE’94), Bern, Switzerland, 1994

[33] Brien, S.; Nicholls, J.: Z Base Standard Version 1.0, 1992

[34] Briot, J-P.; Yonezawa, A.: Inheritance and Synchronization in Concurrent OOP. in Bézivin, J.; Hullot, J-M.; Cointe, P.; Lieberman, H.:

(eds): European Conference on Object-Oriented Programming ECOOP ’87, Lecture Notes in Computer Science 276, Springer 1987

[35] Brunvand, E.: Translating Concurrent Programs into Delay-Insensitive Circuits. in International Conference on Computer-Aided Design (ICCAD), 1989

[36] Burns, A.; Wellings, A.: Concurrency in Ada. Cambridge University Press, 1995

[37] Cabanis, D.: Proposed Object Oriented Extensions to VHDL. Version 1.0, September 1995, Bournemouth University,1995

[38] Cabanis, D.; Medhat, S.: Object-Oriented Extensions to VHDL: The Classification Orientation. Proceedings of the VHDL User Forum Europe 1996, Shaker Verlag 1996

[39] Cabanis, D.: Pre-Processor Considerations. Bournemouth Uni / IBM, Presentation by Benzakki, J.,

[40] Carlson, S.: Modeling Stye Issues for Synthesis. in Harr, R. E.; Stanculescu, A. G. (eds): Applications of VHDL to Circuit Design. Kluwer Academic Publishers, 1991

[41] Caromel, D.: Concurrency and Reusability: From Sequential to Parallel. Journal of Object Oriented Programming, September/October 1990

[42] CCITT Blue book Recommendation Z.100: Functional Specification and Description Language SDL, 1989

[43] Claasen, T. A. C. M.: Design and Test Challenges Behind Systems-on-Silicon. Keynote Addresses Summaries. Proceedings of the Design, Automation and Test in Europe Conference 1998. IEEE Computer Society Press, 1998

[44] Coad, P.; Yourdon, E.: Object-oriented analysis. Yourdon Press, 1991

[45] Covnot, B. M.; Hurst, D. W.; Swamy, S.: OO-VHDL An Object Oriented VHDL. Proceedings of the VHDL International User’s Forum, 1994

[46] Crowl, L. A.: A Uniform Object Model for Parallel Programming. ACM SIGPLAN Notices 24(4), April 1989

[47] Dahl, O. C.; Najm, E.: Specification and Detection of IN Service Interference Using LOTOS. in Tenney, R. L.; Amer, P. D.; Uyar, M. Ü: (eds): Formal Description Techniques, VI, IFIP, Elsevier Science, 1994

[48] Drusinsky, D.; Harel, D.: Using Statecharts for Hardware Description and Synthesis. IEEE Transactions on Computer-Aided Design, Volume 8, Number 7, July 1989

[49] Duke, R.; Rose, G.; Smith, G.: Object-Z: a Specification Language Advocated for the Description of Standards. Technical Report No. 94-45 Software Verification Research Centre Department of Computer Science, The University of Queensland, Australia, 1994

[50] Dunlop, D. D.: Structure Varying Signals and OO Extensions to the VHDL Type System. Revision 1.2, Article for discussion in the Object Oriented VHDL Study Group of the IEEE DASC, 1995

[51] Dunlop, D. D.: Object-Oriented Extensions to VHDL. Proceedings of the VHDL International User‘s Forum, 1994

[52] Ecker, W.: Neue Verfahren für den Entwurf digitaler Systeme mit Hardwarebeschreibungssprachen. PhD thesis (in german), Shaker Verlag, 1996

[53] Ecker, W.; Böttger, J.; Mrva, M.: Klassifizierung von objektorientierten VHDL-Erweiterungen. in Monjau, D. (Hrsg.): Hardwarebeschreibungssprachen und Modellierungsparadigmen: 3. ITG/GI/GMM Workshop, Holzhau, 26.-28. Februar, 1997

[54] Ecker, W.: An Object-Oriented View of Structural VHDL Description. Proceedings of the VIUF Spring 1996 Conference, Santa Clara, USA, 1996

[55] Ecker, W.; Böttger, J.: Evaluation of Ada’95 and VHDL for System Level Modeling. Proceedings of the VIUF Spring 1997 Conference, Santa Clara, USA, 1997

[56] EDA Industry Council: Roadmap Presentation Made at DAC 6/17/98. Available on the WWW from URL http://www.si2.org/ic/roadmap/index.html

[57] Eirund, H.: Objektorientierte Programmierung. Teubner, 1993

[58] Esperan; ICL: VHDL+ and SuperVISE Workshop. Forum on Design Languages (FDL '98), Lausanne, Switzerland, 1998

[59] Faci, M.; Logrippo, L.: Specifying Hardware Systems in LOTOS. Agnew, D.; Claesen, L.; Camposano, R.(eds) : Conference Proceedings of the IFIP Conference on Hardware Description Languages and their Application CHDL’93, Canada, 1993

[60] Færgemand, O.; Olsen, A.: Introduction to SDL-92. Computer Networks and ISDN Systems 26 p1143-1167, 1994

[61] Ferenczi, S.: Guarded Methods vs. Inheritance Anomaly Inheritance Anomaly Solved by Nested Guarded Method Calls. ACM SIGPLAN Notices, Volume 30, Number 2, Februar 1995

[62] Ferenczi, S.: Concurrent Objects with Inherited Synchronization. in Furnari, M., M. (ed): Proceedings of the 2nd International Workshop

on Massive Parallelism: Hardware, Software and Applications. Italy. World Scientific Publishing Co., 1994

[63] Fornaciari, W.; Scutio, D.; Salice, F.: A Two-Level Cosimulation Environment. Computer, Volume 30, Number 6, IEEE, June 1997

[64] Frick, A.; Neumann, R.; Zimmermann, W.: Eine Methode zur Konstruktion robuster Klassenhierarchien. Informatik Forschung und Entwicklung, Band 12, Heft4, Springer Verlag, 1997

[65] Frølund, S.: Inheritance of Synchronization Constraints in Concurrent Object-Oriented Programming Languages. in Lehrmann Madsen, O. (ed.): European Conference on Object-Oriented Programming ECOOP ’92, Lecture Notes in Computer Science 615, Springer 1992

[66] Gajski, D. D.; Vahid, F.; Narayan, S.; Gong, J.: Specification and Design of Embedded Systems. Prentice Hall, 1994

[67] Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Publishing Company, 1994

[68] Gerlach, J.; Eikerling, H-J.; Hardt, W.; Rosenstiel, W.: Von C nach Hardware: ein integratives Entwurfskonzept. in Allgemeine Methodik von Entwurfprozessen: GI/ITG/GMM Workshop, Paderborn, März 1996

[69] Girczyc, E.; Carlson, S.: Increasing Design Quality and Engineering Productivity through Design Re-use. 30th Design Automation Conference, Dallas, 1993

[70] Glunz, W.; Umbreit, G.: VHDL for High-Level Synthesis of Digital Systems. Proceedings of the 1st European Conference on VHDL Methods, 1990

[71] Glunz, W.; Venzl, G: Using SDL for Hardware Design. in SDL'91 Evolving Methods; Proceedings of the Fifth SDL Forum Glasgow, North Holland, 1991

[72] Glunz, W.: Extensions from VHDL to VHDL++. Siemens AG, ZFE BT SE 61 , 1991 , JESSI-AC8/S2-WP1-T2.4-Q3,1991

[73] Glunz, W.; Venzl, G.: Harware-Design using Case Tools. Proceedings of IFIP VLSI’91, 1991

[74] Glunz, W.; Pyttel, A.;Venzl, G.: System-Level Synthesis. in Michel P.; Lauther U.; Duzy P. (eds): The Synthesis Approach to Digital System Design. Kluwer Academic Publishers, p. 221-260 , 1992

[75] Glunz, W.; Kruse, T.; Rössel, T.; Monjau, D.: Integrating SDL and VHDL for System-Level Hardware Design. Agnew, D.; Claesen, L.; Camposano, R.(eds) : Conference Proceedings of the IFIP Conference

on Hardware Description Languages and their Application CHDL'93, Canada, 1993

[76] Glunz, W.: Hardware-Entwurf auf abstrakten Ebenen unter Verwendung von Methoden aus dem Software-Entwurf. PhD thesis. (in german) Paderborn/München, 1994

[77] Harel, D.: Statecharts, A visual formalism for complex systems. Science of Computer Programming 8, 1987

[78] Harper, R.: Introduction to Standard ML. School of Computer Science Carnegie Mellon University Pittsburgh, 1990

[79] Hatley, D. J.; Pirbhai, I. A.: Strategies for Real-Time System Specification. Dorset House Publishing, New York, 1987

[80] Herrtwich, R. G.; Hommel, G.: Nebenläufige Programme, Zweite Auflage, Springer-Verlag, 1994

[81] Hoare, C.A.R.: Towards a Theory of Parallel Programming. in Hoare, C.A.R.; Perrott, R.H. (eds) Operating Systems Techniques, A.P.I.C. Studies in Data Processing No. 9, Academic Press, 1972

[82] Hoare, C. A. R.: Communication sequential processes. in Hoare, C. A. R. Hoare; Jones, C. B. (eds): Essays in Computing Science. Prentice Hall, 1989

[83] Hogrefe, D: Estelle, LOTOS und SDL. (in german) Springer-Verlag, 1989

[84] Holz, E.; Witaszek, D.; Wasowski, M.; Lau, S.; Fischer, J.; Roques, P. ; Cuypers, L.; Mariatos, V.; Kyrloglou, N.: INSYDE Integrated Methods for Evolving System Design ESPRIT Ref: P8641. Technology Assessment. Report. Alcatel Bell Telephone, Dublin City University, Humboldt Universität zu Berlin, Intracom S.A., Verilog S.A., Vrije Universiteit Brussel, 1994

[85] Hopcroft, J. E.; Ullman, J. D.: Introduction to Automata Theory, Languages and Computation. Addison-Wesley, 1979

[86] ICL: VHDL+ Reference Guide, Version 4.0. ICL Design Automation Centre Manchester, UK, 1998

[87] IEEE Standard VHDL Language Reference Manual Std 1076-1993, Revision of IEEE Std 1076-1987, 1994

[88] IEEE Draft Standard VITAL ASIC Modeling Specification. IEEE P1076.4, 1995

[89] IEEE DASC LCS-0046(4): Variant Records, 1992

[90] IEEE DASC PAR 1076A: Shared Variable Language Change Specification, Version 4.0, 1995

[91] IEEE DASC PAR 1076A: Shared Variable Language Change Specifi-

cation, Version 5.7, 1996
[92] INMOS Ltd.: OCCAM Programming Manual, Prentice Hall International, 1984
[93] ISO/IEC JTC1/SC22 WG9 N 193: Programming Language Ada, Language and Standard Libraries, Annotated Draft Version 4.0 IR-MA-1364-3, 1993
[94] ISO/IEC 9074:1989 Estelle, 1989
[95] ISO/IEC 8652:1995(E) Ada Reference Manual, Language and Standard Libraries, Version 6.0, 1994
[96] ISO/IEC 10514-1:1996 Modula-2 (Base Language), 1996
[97] ITU: ITU-T Recommendation Z.100 CCITT Specification and Description Language (SDL), 1993
[98] Jantsch, A.; Kumar, S.; Sander, I.; Svantesson, B.; Öberg, J.; Hemani, A.; Ellervee, P.; O'Nils, M.: Comparison of Six Languages for System Level Descriptions of Telecom Systems. Proceedings of Forum on Design Languages (FDL '98), Lausanne, Switzerland, 1998
[99] Jensen K.; Wirth, N.: Pascal: User Manual and Report. Second Edition, Springer-Verlag, 1975
[100] Jerraya, A. A.; O'Brien, K.; Ben Ismail, T.: Linking system design tools and hardware design tools. Agnew, D.; Claesen, L.; Camposano, R.(eds) : Conference Proceedings of the IFIP Conference on Hardware Description Languages and their Application CHDL'93, Canada, 1993
[101] Jerraya A.A.; O'Brian, K.: SOLAR: An Intermediate Format for System-Level Modeling and Synthesis. in Buchenbrieder, K.; Rozenblit, J. W.: Codesign: Computer-Aided Software/Hardware Engineering, IEEE Press, 1994
[102] Kernighan, B. W.; Ritchie, D. M.: The C Programming Language, Second Edition (ANSI-C) Prentice Hall, 1988
[103] Ku, D. C.; De Micheli, G: HardwareC - A Language for Hardware Design Version 2.0. Technical Report: CSL-TR-90-419 Stanford University, 1990
[104] LaLonde, W. R.; Thomas, D. A.; Pugh, J. R.: An Exemplar Based Smalltalk. OOPSLA'86 Conference Proceedings. ACM Sigplan Notices, Vol 21, No 11, 1986
[105] Leboch, S.; Ryba, M.; Baitinger, U. G.: Wiederverwendung – Kann der Schaltungsentwurf von der Software-Entwicklung lernen, oder umgekehrt? in Kunzmann, A.; Seepold, R.: 1. GI/ITG-Workshop „Wiederverwendung im Schaltungsentwurf“, FZI-Bericht 4/97, Karl-

srube, September, 1997

[106] Lieberman, H.: Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems. OOPSLA'86 Conference Proceedings. ACM Sigplan Notices, Vol 21, No 11, 1986

[107] Lutter, B.; Glunz, W.; Rammig, F. J.: Using VHDL for Simulation of SDL Specifications. Proceedings of the European Design Automation Conference'92 , 1992

[108] Marshall, R. M.: Automatic Generation of Controller Systems from Control Software. in International Conference on Computer-Aided Design (ICCAD), 1986

[109] Marvedel, P.: Synthese und Simulation von VLSI-Systemen : Algorithmen für den rechnerunterstützten Entwurf hochintegrierter Schaltungen, Hanser Verlag, 1993

[110] März, S.: High-Level Synthesis. in in Michel P.; Lauther U.; Duzy P. (eds): The Synthesis Approach to Digital System Design. Kluwer Academic Publishers, 1992

[111] Matsuoka, S.; Yonezawa, A.: Analysis of Inheritance Anomaly in Object-Oriented Concurrent Programming Languages. in Agha, G.; Wegner, P.; Yonezawa, A. (eds.): Research Directions in Concurrent Object-Oriented Programming, MIT Press, 1993

[112] Meseguer, J.: Solving the Inheritance Anomaly in Concurrent Object-Oriented Programming. in Nierstrasz, O. (ed): European Conference on Object-Oriented Programming ECOOP '93, Lecture Notes in Computer Science 707, Springer 1993

[113] Mills, M. T., Lt. Col.: Proposed Object Oriented Programming (OOP) Enhancements to the Very High Speed Integrated Circuits (VHSIC) Hardware Description Language (VHDL). Final Report for 05/04/92-08/04/93 Solid State Electronics Directorate Wright Laboratory Air Force Materiel Command Wright-Patterson Air Force Base, Ohio 45433-7331

[114] Milner, R.; Tofte, M.; Harper, R.: The Definition of standard ML. The MIT Press, 1990

[115] Mitchell, S., E.; Wellings, A. J.: Synchronisation, Concurrent Object-Oriented Programming and the Inheritance Anomaly. Technical Report 234. Department of Computer Science, University of York, UK 1994

[116] Mitchell, S. E.; Burns, A.; Wellings, A. J.: Adaptive Scheduling using Reflection. ECOOP'97 Workshop on Reflective Real-Time Object-Oriented Programming and Systems, Jyväskylä, Finland, June 1997

[117] Nebel, W.; Schumacher, G.: Konzepte objektorientierter Hardware-Modellierung. Invited talk: 2. GI/ITG/GME-Workshop "Hardwarebeschreibungssprachen und Modellierungsparadigmen", Darmstadt, Feb. 15-16, 1996

[118] Nebel, W.; Schumacher, G.: Object-oriented Hardware Modelling – Where to Apply and what are the Objects? Proceedings of the EURO-DAC ’96 with EURO-VHDL ’96. IEEE Computer Society Press, 1996

[119] Neusius, C.: Synchronizing Actions. in America, P. (ed): European Conference on Object-Oriented Programming ECOOP ’91, Lecture Notes in Computer Science 512, Springer 1991

[120] Nierstrasz, O.: Composing Active Objects. in Agha, G.; Wegner, P.; Yonezawa, A. (eds.): Research Directions in Concurrent Object-Oriented Programming, MIT Press, 1993

[121] Öberg, J.; Kumar, A.; Hemani, A.: Scheduling of Outputs in Grammar-based Hardware Synthesis of Data Communication Protocols. Proceedings of the Design, Automation and Test in Europe Conference 1998. IEEE Computer Society Press, 1998

[122] Oczko, A.: Hardware Design with VHDL at a Very High Level of Abstraction. Proceedings of the 1st European Conference on VHDL Methods, 1990

[123] O'Leary, J.; Linderman, M.; Leeser, M.; Aagaard, M.: HML: A Hardware Description Language Based on Standard ML. Agnew, D.; Claesen, L.; Camposano, R.(eds) : Conference Proceedings of the IFIP Conference on Hardware Description Languages and their Application CHDL’93, Canada, 1993

[124] Open Verilog International: Standard Delay Format Specification Version 3.0. May, 1995

[125] Ott, D. E.; Wilderotter, T. J.: A Designer’s Guide to VHDL Synthesis. Kluwer Academic Publishers, 1994

[126] Perry, D.: Applying Object Oriented Techniques to VHDL. Proceedings of the VIUF Spring Conference, p. 217-224, 1992

[127] Potter, B.; Sinclair, J.; Till, D.: An Introduction to Formal Specification an Z. Prentice Hall, 1991

[128] Poulin, J. S.: Measuring Software Reuse: Principles, Practices, and Economic Models. Addison-Wesley Publishing Company, 1997

[129] Pulkkinen, O.; Kronlöf, K.: Integration of SDL and VHDL for High-Level Digital Design. Proceedings of the European Design Automation Conference'92 , 1992

[130] Putzke-Röming, W.; Radetzki, M.; Nebel, W.: Objective VHDL: Hardware Reuse by Means of Object-Orientation. 1st Workshop on Reuse Techniques in VLSI Design, Karlsruhe, Sept. 1997

[131] Putzke-Röming, W.; Radetzki, M.; Nebel, W.: A Flexible Message Passing Mechanism for Objective VHDL. Proc. DATE'98, Paris, France, 1998

[132] Putzke-Röming, W.; Radetzki, M.; Nebel, W.: Modeling Communication with Objective VHDL. Proc. VIUF'98 (Spring Conference), Santa Clara, USA, 1998

[133] Radetzki, M.; Putzke-Röming, W.; Nebel, W.: Language architecture document on Objective VHDL. REQUEST Report D1.2C, ESPRIT Project 20616, OFFIS, LEDA, France Télécom, Italtel, 1996

[134] Radetzki, M.; Putzke-Röming, W.; Nebel, W.; Maginot, S.; Bergé, J.; Tagant, A.; VHDL-language extensions to support abstraction and reuse. Proceedings of the 2nd Workshop on Libraries, Component Modeling and Quality Assurance, Publication Service of the University of Cantabria, 1997

[135] Radetzki, M.; Putzke-Röming, W.; Nebel, W.: Objective VHDL: The Object-Oriented Approach to Hardware Reuse. In: Roger, J.-Y.; Stanford-Smith, B.; Kidd, P.T. (eds.): Advances in Information Technologies: The Business Challenge. IOS Press, Amsterdam, 1998. Presented at EMMSEC'97, Florence, Italy, 1997

[136] Radetzki, M.; Putzke-Röming, W.; Nebel, W.: OO-VHDL: What Is It, and Why Do We Need It? Asia-Pacific Conference on Hardware Description Languages, Hsin-Chu, Taiwan, 1997.

[137] Radetzki, M.; Putzke-Röming, W.; Nebel, W.: A Unified Approach to Object-Oriented VHDL. Journal of Information Science and Engineering 14 (1998), pp. 523-545

[138] Radetzki, M.; Putzke-Röming, W.; Nebel, W.: Objective VHDL: Tools and Applications. Proc. FDL'98, Lausanne, Switzerland, 1998, pp. 191-200

[139] Ramesh, C. R.: Object Orienting VHDL for Component Modeling. VIUF Fall 94 Conference, 1994

[140] Rammig, F. J.: Systematischer Entwurf digitaler Systeme. Teubner, 1989

[141] Rational Software Corporation: Unified Modeling Language: UML Summary, Version 1.0, Santa Clara, 1997, Most recent updates are available on the WWW from URL http://www.rational.com

[142] Rumbaugh, J.; Blaha, M.; Premerlani, W.; Frederick, E.; Lorensen,

W.: Object-oriented modeling and design. Prentices-Hall International, 1991

[143] Rumbaugh, J.: OMT: The object model. in Journal of Object Oriented Programming, January ,1995

[144] Rumbaugh, J.: OMT: The dynamic model. in Journal of Object Oriented Programming, February ,1995

[145] Rumbaugh, J.: OMT: The functional model. in Journal of Object Oriented Programming, March-April, 1995

[146] Rumbaugh, J.: OMT: The development process. in Journal of Object Oriented Programming, May, 1995

[147] Salamunićcar, G.: Case Study of the VHDL Object-Oriented Extension for Data Modeling. Technical Report, University of Zagreb/FER/ZEMRIS/GSCTR1, Department of Electronics, Microelectronics, Computer and Intelligent Systems, Faculty of Electrical Engineering and Computing, University of Zagreb, Croatia, Revision 19971230, 1997

[148] Salamunićcar, G.: A Proposal for Data Modeling Extension to VHDL Using an Object-Oriented Approach. Proceedings of WDTA'98, Dubrovnik, Croatia, 1998

[149] Salamunićcar, G.: A Proposal for Data Modeling Extension to VHDL Using an Object-Oriented Approach. Technical Report, University of Zagreb/FER/ZEMRIS/GSCTR3, Department of Electronics, Microelectronics, Computer and Intelligent Systems, Faculty of Electrical Engineering and Computing, University of Zagreb, Croatia, Revision 19980707, 1998

[150] Salinas, M., H.; Johnson, B., W.; Aylor, J. H.: Implementation-Independent Model of an Instruction Set Architecture in VHDL. Design & Test of Computers, Volume 10, Number 3, IEEE, September 1993

[151] Schumacher, G.; Nebel, W.: Inheritance Concept for Signals in Object-Oriented Extensions to VHDL. Proceedings of the EURO-DAC ’95 with EURO-VHDL ’95. IEEE Computer Society Press, 1995

[152] Schumacher, G.; Nebel, W.: Survey on Languages for Object Oriented Hardware Design Methodologies. Current Issues in Electronic Modeling, Issue 3, Kluwer Academic Press , 1995

[153] Schumacher, G.; Nebel, W.; Putzke, W.; Wilmes, M.: Applying Object-Oriented Techniques to Hardware Modelling – A Case Study. Proceedings of the VHDL User Forum Europe 1996, Shaker Verlag 1996

[154] Schumacher, G.; Nebel, W.: Abstract Hardware Modelling using an Object-Oriented Language Extension to VHDL. Current Issues in Electronic Modeling, Issue 7, Kluwer Academic Press , 1996

[155] Schumacher, G.; Nebel, W.: Object-Oriented Modelling of Parallel Hardware Systems. Proceedings of the Design, Automation and Test in Europe Conference 1998. IEEE Computer Society Press, 1998

[156] Schumacher, G.; Nebel, W.: How to Avoid the Inheritance Anomaly in Ada. in Asplund, L. (ed.): Reliable Software Technologies: Proceedings Ada-Europe '98. Lecture Notes in Computer Science 1411, Springer 1998

[157] Sheraga, R. J.: ANSI C to Behavioral VHDL Translator, Ada to Behavioral VHDL Translator. The RASSP Digest, Vol. 3, September 1996, Available on the WWW from URL http://rassp.scra.org/newsletter/html/96sep/news_11.html

[158] Shlaer, S.; Mellor, J., S.: Object-Oriented System Analysis Modeling the World in Data.Yourdon Press, 1988

[159] Snyder, A. A.; Vetter, B. N.: Eiffel: An Advanced Introduction. Available on the WWW from URL http://www.progsoc.uts.edu.au/~geldridg/eiffel/advance-intro/

[160] Spivey, J. M.: The Z Notation A Reference Manual. Prentice Hall, 1989

[161] Stammermann, A.: Datentypanalyse objektorientierter Hardwarebeschreibungen. (in german) Dissertation for a diploma at Carl von Ossietzky University Oldenburg. Oldenburg, 1998

[162] Swamy, S.; Molin, A.; Covnot B., M.: OO-VHDL Extensions to VHDL. Computer, October 1995 pp. 18–26, IEEE, 1995

[163] Tangemann, B.: Synthese komplexer Datentypen einer object-orientierten Hardwarebeschreibungssprache auf Registertransferebene (in german) Dissertation for a diploma at Carl von Ossietzky University Oldenburg. Oldenburg, 1997

[164] Tanenbaum, A. S.: Operating Systems: Design and Implementation. Prentice-Hall, 1987

[165] Thomas, D. E; Moorby, P.: The Verilog Hardware Description Language, Kluwer Academic Publishers, 1991

[166] Tomlinson, C.; Singh, V.: Inheritance and Synchronization with Enabled-Sets. in Meyrowitz, N. (ed): Object-Oriented Programming: Systems, Languages and Applications. OOPSLA'89 Conference Proceedings. ACM Press, 1989

[167] Vista Technologies, Inc.: OO-VHDL Language Reference. Version

0.3, RASSP Contract No. DAAL01-93-R-3616 Document ID: TR-1.2.11.1.3-01

[168] Vista Technologies, Inc.: Object Oriented VHDL. Webpages http://www.vistatech.com/oovhdl.html, September, 1995

[169] Volan, J.: John Volan's Answers to Frequently Asked Questions about Ada95's "With-ing" Problem. Version 2.05. Webpages http://bluemarble.net/~jvolan/WithingProblem/FAQ.html, June 1997

[170] Wegner, P.: Dimensions of Object-Based Language Design. Proceedings OOPSLA '87, ACM SIGPLAN Notices, vol. 22, no. 12, 1987

[171] Wegner, P.; Zdonik, S. B.: Inheritance as an Incremental Modification Mechanism or What Like Is and Isn't Like. in Gjessing, S.; Nygaard, K. (eds.): European Conference on Object-Oriented Programming ECOOP '88, Lecture Notes in Computer Science 322, Springer 1988

[172] Wegner, P.: Concepts and Paradigms of Object-Oriented Programming. ACM OOPS Messenger, vol. 1, no. 1, 1990

[173] Wegner, P.: Dimensions of Object-Oriented Modeling. Computer, October 1992 pp.12-20, IEEE, 1992

[174] Willis, J. C.; Bailey, S. A.; Newshutz, R.: A Proposal for Minimally Extending VHDL to Achieve Data Encapsulation Late Binding and Multiple Inheritance. Proceedings of the VHDL International User‘s Forum, 1994

[175] Wilmes, M.: Hardware-Spezifikation mit objektorientierten Spracherweiterungen zu VHDL. (in german) Dissertation for a diploma at Carl von Ossietzky University Oldenburg. Oldenburg, 1995

[176] Wytrebowicz, J.: Modeling Shared Variables in VHDL – a Pragmatic Approach. Proceedings of the VHDL-Forum Europe Spring '95 Working Conference, pp. 15-24, 1995

[177] Zippelius, R.; Müller-Glaser, K. D.: An Object-oriented Extension of VHDL. Proceedings of the VHDL-Forum Spring‘92 Meeting, 1992

Curriculum Vitae

1999 – to date	Universität Oldenburg, Assistant Lecturer
1998 – 1999	OFFIS (Oldenburg Research and Development Institute for Informatic Tools and Systems), Scientist
1993 – 1998	Universität Oldenburg, PhD Student
1984 – 1992	Universität Stuttgart, Diploma in Informatics
11. 12. 1963	Born in Ravensburg