CARL VON OSSIETZKY universität OLDENBURG

Fakultät II – Informatik, Wirtschafts- und Rechtswissenschaften
Department für Informatik

State-Based Real-Time Analysis of Synchronous Data-flow (SDF) Applications on MPSoCs with Shared Communication Resources

Dissertation zur Erlangung des Grades eines
Doktors der Ingenieurwissenschaften

vorgelegt von

M.Sc. Maher Fakih

Gutachter:

Prof. Dr. Achim Rettberg
Prof. Dr. Marcio Eduardo Kreutz

Tag der Disputation: 11.07.2016

Abstract

The growing computational demand of real-time applications (in automotive, avionics and multimedia) requires extensions in the traditional design process to support Multi-Processor System-on-Chip (MPSoC) architectures. Due to their significantly increased performance and Space Weight and Power (SWaP) reductions, MPSoCs offer an appealing alternative to traditional single-processor architectures. The timing analysis of hard real-time applications running on MPSoC platforms is much more challenging compared to traditional single processor. This comes from the large number of shared processing, communication and memory resources available in today's MPSoCs. Yet, this is an indispensable challenge for enabling their usage with hard-real time systems in safety critical application domains (e.g. avionics, automotive). In this thesis, a state-based real-time analysis methodology for a subset of data-flow oriented applications using model-checking is proposed. Applications are represented as Synchronous Data Flow (SDF) graphs, the MPSoC is represented as Architecture Resource Graph (ARG) and a mapping relation between these graphs describes the implementation of the application on the MPSoC architecture. This approach utilizes Timed Automata (TA) as a common semantic model to represent execution time boundaries (best-case and worst-case execution times) of SDF actors and communication FIFOs and their mapping as well as their utilization of MPSoC resources, such as scheduling of SDFGs and shared communication resource access protocols for interconnects, local and shared memories. The resulting network of TA is analyzed using the UPPAAL model-checker for obtaining safe timing bounds of the chosen implementation. The proposed methodology is compared with a state-of-the-art analytical method showing a significant precision improvement (up to a percentage improvement of 300%) compared with the worst-case bound calculation based on a pessimistic analytical upper-bound delays for every shared resource access. Furthermore, the analysis feasibility of our approach was demonstrated for small parallel systems. In addition, the limitations of our approach and abstraction methods to improve scalability were explored. We also demonstrate the applicability of our approach to an industrial case-study using a multi-phase electric motor control application (modeled as an SDFG) mapped to a state-of-the-art MPSoC with both the burst and single-beat inter-processor communication styles.

Contents

Chapter 1

Introduction

1.1 Context and Motivation

The last decade witnessed a significant technological revolution of miniaturization technologies of processing devices leading to ubiquitous computing and the wide-spread of embedded systems[1] in our everyday life. For instance, a look at a modern car in the automotive domain, shows that a premium version can have about 70 ECUs (embedded devices) on which hundreds of real-time applications are run [Buttle, 2012] and the trend is going towards a larger number of ECUs with more complexity. Depending on their domain of usage, the timing criticality of applications running on such systems can vary from hard real-time systems (e.g. aircraft control or video-processing applications used in safety-critical automotive systems to detect pedestrians crossing or the street signs) where a violation of the real-time requirement can lead to catastrophic results, to non real-time applications (such as an MP3 player in the infotainment domain) where the harmfulness by a violation is very limited. In order to guarantee the safety of hard-real time systems, a real-time (RT) analysis method is indispensable to validate the fulfillment of their hard real-time requirements. According to safety standards like DO-178B/DO-178C [Aeronautical Radio, 1992], ISO-26262 [ISO26262, 2011], IEC-61508 [IEC, 2010], or CENELEC EN-50128 [EN50128, 2009] the functional safety of the software must be demonstrated with respect to the specified requirements and the absence of critical non-functional hazards (including timing hazards in real-time systems) has to be shown [Kästner Daniel and Christian, 2014].

Because of the growing computational demand of such real-time applications (in automotive, avionics and multimedia), the need for more powerful,

[1] According to [Marwedel, 2010] an embedded system is defined as: "*Embedded systems are information processing systems embedded into enclosing products.*"

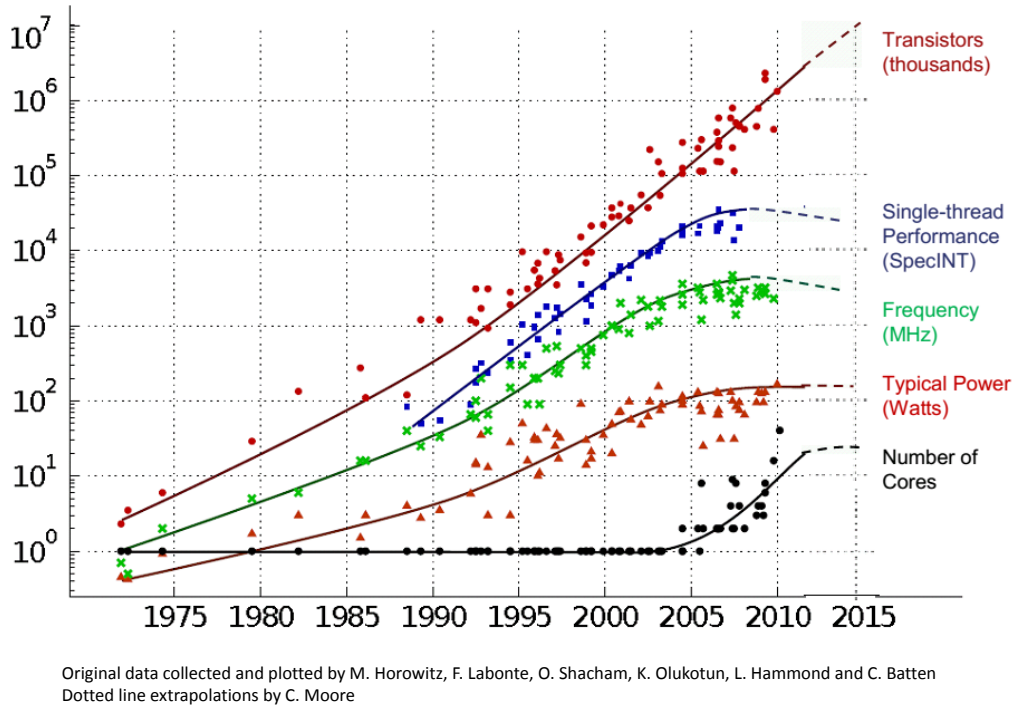


Figure 1.1: Trend towards MPSoCs' design (taken from [Fuller and Lynette I. Millett, 2011])

fast and efficient hardware architectures is emerging. In the last decade, the chip industry was faced with the challenge that the chip clock frequencies (as seen in Fig. 1.1 green, x-curve) couldn't be increased without drastically increasing power consumption (as seen in Fig. 1.1 red, △-curve) and heat wastage. The above phenomena called "clock-racing" (also called frequency scaling) reached its limit at the end of year 2003 (as seen in Fig. 1.1 green, x-curve), while the number of transistors continued to increase (according to Moore's law number of transistors doubles every two years as seen in Fig. 1.1 red, ○-curve). This made the extension of current design process for supporting MPSoC architectures inevitable. Due to their significantly increased performance and their Space Weight and Power (SWaP) reductions (see stable power consumption in Fig. 1.1 red, △-curve, beginning at the end of 2003 as the number of cores was increased), MPSoCs offer an appealing alternative to traditional architectures.

Nevertheless, with MPSoCs emerging, their validation process is becoming a bottleneck. According to [Tang and Wu, 2014], the number of verification engineers needed for nowadays MPSoCs' projects is wide more than the number of design engineers reaching a ratio of 2:1 or even 3:1 the fact which can lead to high costs in the development process. Especially in the real-time (RT) domain, the RT validation of applications running on MPSoCs is indispensable to

guarantee their safe usage. Yet the timing analysis of MPSoC platforms with hard real-time requirements is very challenging making their usage in safety-critical real-time domains difficult. In difference to a single-processor platform an access to a shared resource in an MPSoC can have variable delays depending on the level of interleaving with other processors trying to access the same resource; e.g. if Task A on pe_0 and Task B on pe_1 simultaneously issue an access request on the shared bus, then depending on the arbitration mechanism either pe_0 or pe_1 could win the arbitration and is granted access causing the other processor to delay. In addition to resource sharing, the fact that in industrial MPSoCs' architectures, efficiency (or optimized average-case performance) is still preferred over predictability makes the real-time analysis of such systems even harder. A typical example is the abundant usage of shared caches (with complex replacement strategies) in current MPSoCs which obviously increases the average-case performance of a platform but makes it difficult to perform a RT analysis due to the unpredictable nature of caches (with complex replacement strategies see [Cullmann et al., 2010]). Thus, in a full-featured MPSoC, the contention (as seen in the previous examples) not only can take place at the level of communication resources (bus, interconnects) but also on the level of storage resources (shared memory, caches) which makes the RT analysis of such platforms very challenging.

Due to this fact, adapting traditional static RT analysis methods which are well-established for single-processor platforms for MPSoCs is not an easy task. This in turn, stresses the need for novel RT analysis methods capable of proving the timing predictability of real-time applications running on MPSoCs at an early project stage. To cope with the above challenge, there is a lot of active research on designing predictable[2] MPSoCs [Chattopadhyay and Roychoudhury, 2011, Cullmann et al., 2010, Nelis et al., 2011, Hansson et al., 2009, Metzlaff et al., 2011, Ungerer et al., 2010, Wilhelm and Reineke, 2012, Zamorano and Juan, 2014] on one side (see Sect. 2.3.6) and enhancing the traditional static analysis methods to be able to predict execution times of embedded applications running on MPSoCs on the other side (see Chap. 3).

In [Cullmann et al., 2010], the authors suggested the design of timing predictable MPSoCs to overcome this challenge and gave receipts how to design MPSoCs for predictability. They suggested to support shared communication resources with easy-to-predict arbitration protocols and to use private storage resources (referred to as spacial isolation) to alleviate contention. For Commercial-Off-The-Shelf (COTS) multicore platforms a smart configuration can be done for making them predictable. This configuration discourages the

[2]Predictable MPSoCs are those exhibiting deterministic temporal behavior enabling beforehand to determine whether or not the right outputs happen at the right (predicted) moment.

usage of shared caches in COTS, enables partitioning of memories (if supported by the hardware) to avoid interferences among the cores and utilizes predictable arbitration features of communication resources (as in the MPC8641D avionic processor in [Cullmann et al., 2010]).

An excerpt of first approaches enhancing traditional static analysis towards MPSoCs will be discussed in Chap. 3. There are mainly two real-time (RT) analysis approaches for embedded applications: dynamic and static (formal) methods. In the *dynamic* methods, use-case driven timing measurements of the application are performed either using a virtual-hardware platform simulation model (with variable abstraction levels ranging from untimed to cycle accurate) or by running it on the target hardware employing hardware tracing facilities. This approach is still state-of-the-art in industry since it is capable of handling systems with a huge state space. Yet it is not applicable to applications with hard real-time requirements since even exhaustive simulations provide no guarantee that all interesting corner cases are covered. According to DO-178B/DO-178C, dynamic testing-based real-time analysis methods alone are not enough since testing cannot show the absence of errors [Kästner Daniel and Christian, 2014]. In a *static (formal)* approach, mathematical analysis is performed on a formal representation of both software and hardware. This analysis takes into consideration all possible inputs (use-cases) and combinations of the running applications with all different hardware states of the proposed platform. This makes it possible to identify the worst-case path and to estimate a pessimistic but a safe upper bound on the application execution time. Formal methods guarantee complete coverage of the considered model[3] but suffer from state explosion and scalability issues on one side and of obtaining over-pessimistic timing results (depending on the accuracy of the formal model) on the other side.

As explained above, in order to give safe timing guarantees under all conditions, a formal approach is needed to calculate safe lower/upper bounds based on Worst-Case-Execution Times (WCETs) of the application computation and communication phases depending on the target hardware platform. Since current MPSoCs are composed of concurrent components and their synchronization depends on timing constraints, formal models like timed-automata and model-checkers like UPPAAL [Bengtsson and Yi, 2004] are very suitable to capture and verify their temporal behaviors with rigor. In addition, for unmet timing properties counter examples are provided. Another motivation for using state-based RT analysis methods for analyzing MPSoCs' applications in this thesis, is that they support modularity which makes them easily adaptable to different hardware models. Furthermore, state-based RT analysis methods pos-

[3]Of course this does not imply that the considered model is complete (i.e. represents all relevant corner cases), but formal methods enable complete exploration, independent from the fact whether this model is complete or not.

sess the capability of getting more accurate results [Perathoner et al., 2009] and verifying more complex properties than other formal methods (see Sect. 3.1).

But one of the main drawbacks of recent research using state-based RT analysis methods for analyzing MPSoCs (see Sect. 3.1.2.1) is trying to analyze arbitrarily parallel programs at code-level on MPSoC architectures. Despite the advantage of such an approach being applicable to any code written/generated for any domain, yet the fine granularity of the code-level or instruction-level makes the state-based methods not scalable. In order to circumvent their scalability problem, enabling a composable state-based RT analysis is a prerequisite. This can only be done if we have a task model which exhibits clean semantics that enables distinguishing communication from computation parts in the implemented code which is not the case for generic tasks. With these aspects (communication from computation phases) separated, flexible mapping to different target platforms can be established, and a composable RT analysis method analyzing different mappings is possible.

In this thesis, we aim to develop a state-based RT analysis method to guarantee a timing predictable execution of parallel software on MPSoCs. Our state-based RT analysis method targets, on the one side, the analysis of larger systems (for a chosen use-case in Sect. 7.2.1 up to 96 actors mapped to 4-tiles and up to 320 actors on a 2-tiles platforms) than those analyzable by current state-based approaches and, on the other side, achieving a significant precision improvement (up to a percentage improvement of 300%) compared with a state-of-the-art analytical method. The establishment of our method, would open the way for safety-critical domains, especially in the most conservative domains such as avionics[4], to adapt MPSoCs (for small-scale systems) in their design flow, making it easier to pass the strict certification processes imposed by certification authorities. Clearly, integrating multiple functionalities on a single MPSoC would lead to great saving in terms of the hardware used, making products cheaper and thus more competitive.

1.2 Research Questions and Contributions

1.2.1 Research Questions

As described above, the main concern of a system-level designer is to develop MPSoCs, benefiting from their performance and energy advantages compared to a single-processor platforms, and at the same time guaranteeing that the hard real-time requirements of the applications mapped to them are met. Now the main challenge here is to provide suitable methods to guarantee timing-predictable execution of parallel software on MPSoCs. One of these methods

[4]Currently in the avionics single-core federated architectures are still used.

which we will be using in thesis to achieve this goal is the state-based RT analysis methods (see Sect. 2.5.2.1). As we already explained, for a state-based RT analysis method a formal model of the hardware and the application should be built. The question is now how to build a model of the real hardware/software being of a reasonable size (improving the scalability bottleneck of state-based RT methods with generic task models see Sect. 3.1.2.1) on one hand and of a reasonable granularity on the other hand allowing an accurate real-time analysis of the System Under Analysis (SUA).

To answer the above question, the following research subquestions must be answered:

1. Which constraints should be imposed on the software application in order to be modeled in an abstract but still accurate form?

2. Which constraints should be imposed on the physical platform and which timing properties shall be represented in the formal platform model?

3. How does the formal model scale w.r.t to state-space complexity? How does the approach perform in terms of correctness and accuracy (over-approximation evaluation)?

4. Which kind of properties can be obtained/validated via a state-based RT analysis approach? (end-to-end deadline, WCRT, buffers' sizes etc.)

In order to answer the first question and to circumvent the scalability issues faced by previous state-based RT analysis approaches, we limit applications to the Synchronous Data-flow (SDF) [Lee and Messerschmitt, 1987b] Model of Computation (MoC) (see Chap. 4). In the context of MPSoCs research [Sriram and Bhattacharyya, 2000, Shabbir et al., 2010, Ghamarian, 2008, Kumar, 2009, Moonen, 2009, Stuijk, 2007], the SDF MoC is gaining consideration due to its analyzability features (e.g. deadlocks and bounded buffer properties are decidable for such models [Lee and Messerschmitt, 1987b]). In an SDF specification, parallelism is represented explicitly and static schedules can be obtained. Furthermore, SDF semantics support a clean separation between computation and communication since no communication (resource access) is allowed during the computation phase. This enables a compositional timing analysis where SDF actor execution times can be analyzed independently from communication delays of message passing between SDF actors.

We also constrain our hardware platform to an MPSoC architecture (see Chap. 4) where each processor has its own instruction and data memory, called a "tile". Tiles are connected through one (or more) arbitrated shared interconnect(s) (bus(s), shared DMA(s)). Communication between tiles is realized through FIFO-style message passing on shared memories accessed via shared interconnects.

With these constraints a formal model based on timed-automata semantics can be constructed (see Chap. 5), representing WCETs of SDF actors and access protocol properties (including timing) of shared interconnects, private local and shared memories of the MPSoC platform and questions 3&4 can now be examined (see Chap. 7). With the above knowledge we can now concretely formulate the main research goal of this thesis as follows:

The main goal of this thesis is to examine (according to metrics defined in questions 3&4) a state-based real-time analysis approach to analyze multiple Synchronous Data-Flow (SDF) applications running on MPSoCs with shared communication resources with respect to their hard real-time requirements.

1.2.2 Contributions

We claim the following contributions in this thesis:

C1 We provide a predictable, yet realistic, configuration of MPSoCs (with dynamic arbitration protocols) which enables our state-based RT-analysis method (see Chap. 4).

C2 We enable a state-based real-time analysis of multiple SDF applications mapped to an MPSoC platform (see Chap. 5):

1. Through capturing the delays of SDFGs when run on an MPSoC in the form of timed-automata (TA) templates enabling sensitivity to external events, multiple interconnects, multiple storage resources and different inter-processor communication styles. For this we provide the complete set of timed automata templates capturing the considered system model performance metrics and explaining their implementation and abstraction decisions,
2. Evaluating different methods to improve the scalability of our state-based RT analysis method,
3. Allowing the verification of more complex properties (such as liveness and reachability properties) compared to other analytical methods.

C3 Integrating above state-based RT analysis method into a model-based design-flow which enables functional and temporal analysis of control applications at different abstraction levels (see Chap. 6):

1. Translation concept of Simulink models to SDFGs enabling RT analysis of applications implemented in Simulink (implemented by Warsitz in `SimulinkToSDF` tool [Warsitz, 2015, Warsitz and Fakih, 2016]),

2. Automation concept of our state-based RT analysis (first implemented by Schlaak in `SDF2TA` tool [Schlaak, 2014]),
3. Combining a simulative method[5] with our state-based RT method for functional and accurate temporal Verification and Validation (V&V).

C4 Evaluating the viability of our approach (see Chap. 7):

1. Being applicable to industrial use-cases. For this we show that the timing bounds of different implementations with different communication styles for a motor control use-case are predictable through our framework,
2. Tightening real-time results in comparison to a pessimistic analytical approach from literature [Shabbir et al., 2010],
3. Enabling analysis of larger systems compared to related work [Gustavsson et al., 2010, Lv et al., 2010].

In this thesis, our major contribution is the development of a state-based real-time analysis framework (see C2) which enables (using the UPPAAL model-checker) calculating safe timing bounds of multiple (hard real-time) SDF-based applications running on an (for predictability pre-configured see C1) MPSoC (represented as a network of TA), considering variable access delays due to the contention on shared communication resources. The analysis framework is capable of handling different shared memory architectures, data access granularities and arbitration protocols (such as Round Robin, Fixed Priority and First Come First Serve (FCFS)).

To the best of our knowledge, we pioneered the translation of SDFGs to timed-automata (in [Fakih et al., 2013a]) and we were the first to describe how to use model-checking to analyze real-time properties (e.g. end-to-end deadline) of hard real-time multiple SDF applications mapped to MPSoCs. Our approach has been later taken up by other researchers in [Malik and Gregg, 2013, Ahmad et al., 2014, Zhu et al., 2014, Zhu et al., 2015, Skelin et al., 2015, Thakur and Srikant, 2015] in order to model-check SDFGs/SADGs[6], targeting various objectives (see Sect. 3.1.2.2).

Another major contribution is that we integrated our developed RT method in a model-based design flow (see C3) simplifying the design of MPSoCs applications and their validation. Here, we support Simulink models as entry

[5]The Virtual-Platform-In-the-Loop (VPIL) verification and validation technique was first demonstrated for single-processor platforms in [Fakih et al., 2011, Fakih, 2011] and in the scope of this thesis it was then extended for Verification and Validation (V&V) of MPSoCs and published in [Fakih and Grüttner, 2012].

[6]Scenario-aware Data-flow Graphs (SADGs) are more dynamic SDFGs where according to scenarios, different flavors of the same SDFG are executed.

models and describe how such models can be translated to SDFGs to enable their state-based RT analysis on the one side. On the other side, we introduce a simulation-based RT analysis (VPIL see C3-3) in the design flow enabling functional and temporal validation of embedded Simulink applications on MPSoCs. Simulative approaches are more accurate and can be applied to analyze large-scale applications running on large MPSoCs (for e.g. in the case where the state-based RT analysis fails to analyze the SUA due to the well-known state-space explosion problem).

1.3 Thesis Outline

This thesis is structured as follows. In Chap. 2 we will first discuss the main concepts relevant to this thesis. Afterwards we will briefly discuss the related work in Chap. 3 mainly addressing the RT analysis of SDFGs on MPSoCs. The core of this thesis lies in chapters 4, 5 and 6 where we first introduce and discuss the constraints made on the application and hardware model to enable the applicability of our state-based RT analysis method. Then we illustrate our proposed approach and elaborate on the implementation of our timed-automata templates used to capture the system model. Afterwards, we describe our overall model-based design flow. Chap. 7 presents the experimental evaluation conducted to demonstrate the viability of our state-based RT method. Finally, Chap. 8 summarizes our findings and gives an outlook on open issues and future work.

1.4 Prior Publications

Most of the concepts illustrated in this thesis have been published beforehand in scientific journals, conferences, and workshops by the author (as first author) together with other researchers which contributed mostly through their thoughts in discussions, guidance and feedback to the written publications.

A first proposal answering the four questions (see Sect. 1.2) was published in [Fakih et al., 2013a]. A scalability improvement in terms of the number of applications being analyzable of our approach through enabling spatial and temporal segregation in the MPSoC was published in [Fakih et al., 2013b].

The Virtual-Platform-In-the-Loop (VPIL) verification and validation technique was first demonstrated for single-processor platforms in [Fakih, 2011, Fakih et al., 2011] and in the scope of this thesis it was then extended for V&V of MPSoCs (see Sect. 6.4) and published in [Fakih and Grüttner, 2012].

In [Fakih et al., 2014] our simulative approach (VPIL technique) was combined with our state-based RT analysis approach in a model-based design flow

and the applicability of our approach was demonstrated on an industrial use-case.

In [Fakih et al., 2015] the restrictions made in the previous publications were further relaxed towards enabling sensitivity to external events, multiple interconnects, multiple storage resources and different inter-processor communication styles. In addition, we published in [Fakih et al., 2015] the complete set of used timed automata templates capturing the considered system model performance metrics and explaining their implementation and abstractions' decisions.

It is important to note that the first version of the `SDF2TA` tool (see Sect. 6.3) enabling the automatic configuration of our timed-automata templates was implemented by Schlaak [Schlaak, 2014]. Also the first version of `Simulink-ToSDF` tool which enables translating Simulink models to SDFGs was first developed by Warsitz in [Warsitz, 2015] (based on a major conceptual contribution by the author of this thesis summarized in Sect. 6.2) and then published in [Warsitz and Fakih, 2016]. Both scientific work above were performed under the guidance and support of the author of this thesis.

Chapter 2

Basic Concepts and Background

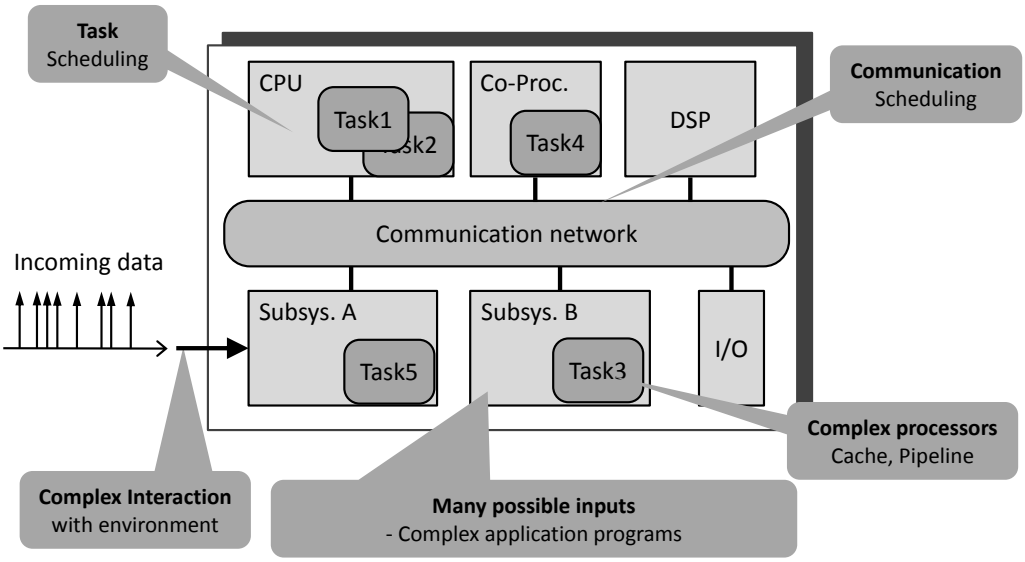


Figure 2.1: Timing issues of MPSoCs' embedded applications (taken from [Roychoudhury, 2009])

This chapter explains the basic terms important for understanding further work in this thesis. Definitions of basic keywords can be found in the glossary (see Glossary B.2.2). Fig. 2.1 shows an overview of different issues which should be taken into consideration when validating the timing properties of embedded applications running on MPSoCs. These factors vary between the influences of the different timing patterns of incoming events (periodic, sporadic, with/without jitter) from the external environment which activate the local tasks, the task model semantics and task scheduling, the application possible

input behavior and the communication access pattern on the communication resources. In addition, the hardware properties of MPSoCs largely influence the timing behavior of the application including the complexity of processors (including cache, pipelines) and the temporal properties of the communication (latency, arbitration complexity) and storage resources.

This chapter will be structured according to these issues depicted in Fig. 2.1. Starting with a short description of current system-level design methodologies, we will then take a look at the task models (with a focus on the synchronous data-flow graphs) considered in this thesis and their scheduling mechanisms in Sect. 2.2.1. Next, the temporal behavior of different MPSoC components with a focus on the communication resources (arbitration policies and timing diagrams) temporal behavior is described in Sect. 2.3. Afterwards, a short description of modeling the interaction with the environment and its timing effect is given. At the end of this chapter, we will take a look at different real-time (RT) timing analysis methods with the focus on formal RT analysis methods, being able of handling the timing issues in Fig. 2.1.

2.1 System Level Design (SLD) Methodologies

One goal of this thesis (see C3) is to implement a suitable design flow to enable timing validation of functional models (see Chap. 6), that is why some concepts and terms of the SLD methodologies are presented in the following (partially taken from the author's work in [Fakih, 2011]).

Basically SLD methodologies aim at introducing "abstraction" as a solution for handling the design complexity of embedded systems. In the 1960s *capture and describe* methodology was used [Gajski et al., 2009]. Software and hardware design were separated by a gap because developers had to wait until gate level design was finished before verifying the system specifications. After that designers began to use the *describe and synthesize* methodology where designers first specified what they wanted in boolean equations or Finite State Machines (FSM) descriptions and synthesis tools were implemented to generate automatically implementations of these descriptions in the form of netlists. But still there was a great gap between higher system level and these low-level specifications. Nowadays the *specify and explore* methodology is the method used to close this gap, the level of abstraction is increased beginning with a functional model implemented in some Model of Computation (MoC) representing an executable specification, then possibilities are explored at different refinement levels before finally the model is refined to be implemented on the target hardware.

In [Gajski et al., 2009], MoCs are defined as follows:

> "*A Model of Computation (MoC) is a generalized way of describing*

> *system behavior in an abstract, conceptual form."* [...] *MoCs are generally based on a decomposition of behavior into pieces and their relationships in the form of well-defined objects and composition rules."* ([Gajski et al., 2009]:50)

MoCs can be classified into *process-based* and *state-based* models. Process-based models are typically used for data-oriented applications and for design modeling at behavioral level. They are represented by a set of concurrent processes, that are untimed and ordering is only limited by the data flow between them. Each process is blocked when trying to read from a channel with insufficient data and it resumes when enough data is available. In a *data-flow* model, which is a special case of process-based models, processes are replaced by atomic blocks of execution, called *actors*. Avoiding the need for context switches in the middle of processes, actors execute according to firing rules depending on the number of tokens that must be available on every input for the actor to fire [Gajski et al., 2009]. *Synchronous data-flow* (SDF) MoC [Lee and Messerschmitt, 1987a] is a data-flow model, in which the number of tokens consumed and produced by an actor per firing is constant and fixed (see Sect. 2.2.1). State-based models on the other hand, focus on explicitly exposing and representing control flow. They are used for control-dominated applications and for modeling of designs at the implementation level (e.g. for capturing cycle-by-cycle hardware behavior). *Process State Machines (PSM)* combines both process-based and state-based concepts in a one MoC [Gajski et al., 2009]. As an instance *SystemC* the well-known standard modeling language [IEEE-1666, 2012] for realizing virtual-hardware-platforms has a generic MoC which only assumes that the system state changes at discrete time points. This means that for example both a PSM [Gerstlauer, 2009] or a Timed Data-Flow (TDF) [Grimm et al., 2009] MoCs can be realized in SystemC.

Fig. 2.2 shows the X-chart [Gerstlauer et al., 2009] which identifies the main tasks in modern Electronic System Level (ESL) design process. All the definitions and terms of the system model used in this thesis (see Chap. 4) are based on the X-Chart defined and described in [Gerstlauer et al., 2009]. The functionality of the system is first captured in a behavioral model which typically represents an executable specification of the system functionality. The expressibility and analyzability of the behavioral model depends on its underlying Model of Computation (MoC). In this thesis, we will mainly use the SDF MoC (see Sect. 2.2.1). Later on, Simulink (see Sect. 2.2.2) will also be supported in our design flow to capture behavioral made. The Model of Architecture (MoA) (see Fig. 2.2) represents a platform model where the architectural template, decisions and constraints are taken into consideration for e.g., available resources, their capabilities and their interconnections [Gerstlauer et al., 2009].

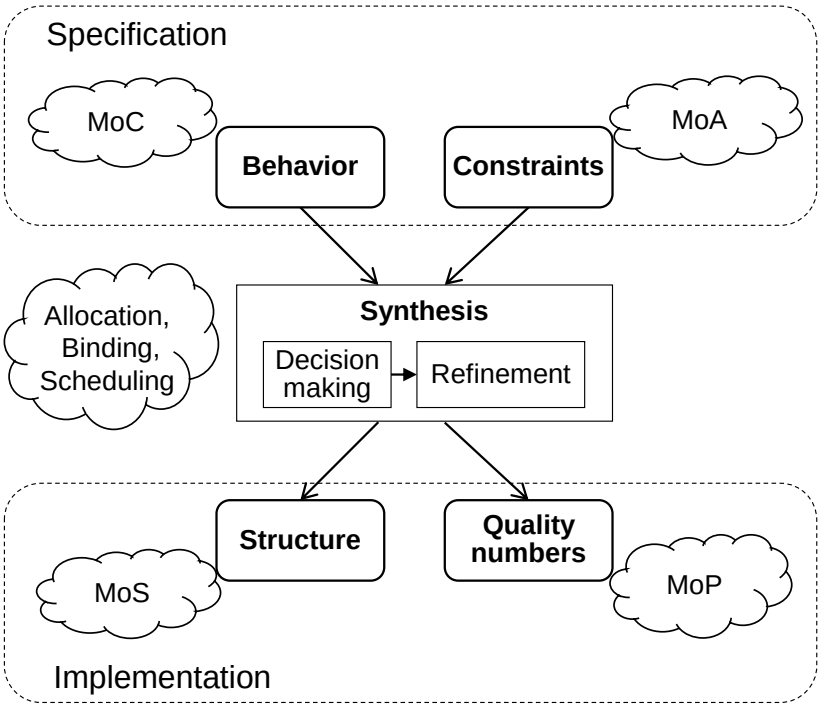


Figure 2.2: X-Chart (taken from [Gerstlauer et al., 2009])

The *synthesis* step includes the processes of *allocating* resources, *binding* and *scheduling* the behavioral model on the defined architecture, and thus transforming a specification into an implementation. An implementation consists of a structural model and quality numbers (in this thesis only timing delays quality numbers are considered). The structural model (MoS: Model of Structure) is a refined model resulting from the behavioral model under the architectural constraints given in the specification after the synthesis decisions above have taken place [Gerstlauer et al., 2009]. Different implementations parameters (for e.g. throughput, response time, latency, area and power) can be estimated for a specific implementation. Instead of implementing each design possibility to obtain above parameters' values, performance models (MoP: Model of Performance) are used. A MoP comprises all individual elements of the MoS contributing to a specific design quality (e.g. worst/average/best case latency). The overall quality estimates can be obtained either through direct measurements, through simulation or through static analysis and highly depends on the abstraction level and granularities in the MoP [Gerstlauer et al., 2009]. In this thesis, the performance values considered in the MoP are merely execution times metrics.

2.2 Task Model (Model of Computation)

Model-based Design (MBD) of embedded systems is nowadays, a standard, easy and efficient way for capturing and verifying embedded software functional requirements. The main idea is to move away from manual coding, and with the help of mathematical models create executable specifications, and then provide automatic code generators which generate consistent imperative code ready to be deployed in real environments. Typically, in MBD abstractions from non-functional issues are made for allowing much faster simulation speeds than other models enriched with hardware issues (e.g. Register Transfer Level (RTL) models). Although this allows the designer to validate requirements at very rapid speed, yet important issues such as timing violations of a safety critical embedded application can't be validated at this abstraction level.

In this thesis, we will mainly use the SDF MoC (see Sect. 2.2.1). Later on, our proposed design flow (see Sect. 6.1) will be extended to enable entry functional models modeled in Matlab/Simulink [MathWorks, Inc., 2015c] (see Sect. 2.2.2).

2.2.1 Synchronous Data-flow Graphs (SDFGs)

A *synchronous (or static) data-flow graph (SDFG)* [Lee and Messerschmitt, 1987b] is a directed graph (see Fig. 2.3) which, similar to general data-flow graphs (DFGs), consists mainly of nodes (called *actors*) modeling atomic functions/-computations and arcs modeling the data flow (called *channels*). In difference to DFGs, SDFGs consume/produce a static number of data samples (*tokens*) each time an actor executes (*fires*). An SDFG suits well for modeling multi-rate streaming applications and DSP algorithms and also allows static scheduling and easy parallelization. An application which is modeled as an SDFG and has a timing requirement will be denoted as a *synchronous (or static) data-flow application (SDFA)* in this thesis. A port *rate* denotes the number of tokens produced or consumed in every activation of an actor. The data flow across a chan-

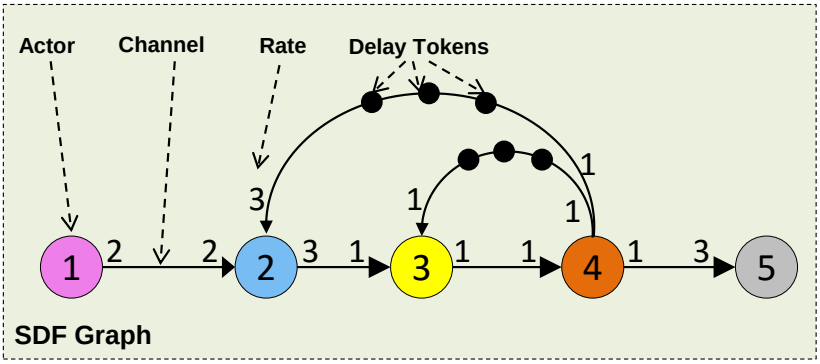


Figure 2.3: Example of an SDFG (based on [Lin et al., 2011])

nel (which represents a FIFO buffer) is done according to a First-In-First-Out (FIFO) fashion. Channels could also store initial tokens (called delays indicated by bullets in the edges see for e.g. Fig. 2.3) in their initial state which help resolving cyclic dependencies (see [Lee and Messerschmitt, 1987b]). An actor in an SDFG can be a consumer (*sink*), a producer (*source*) or a transporter actor. The complete formal definitions of SDFGs will be presented in Sect. 4.2.1.

Synchronous data-flow graphs where the number of tokens which are consumed or produced by all actors when activated, is always equal to 1 are called *homogeneous synchronous data-flow graph* [Lee and Messerschmitt, 1987a]. Below we will give a description of the basic concepts of SDFGs relevant to this thesis including their scheduling decisions, their analyzability (properties of SDF graphs that are analyzable) and expressiveness features compared to more dynamic data-flow models and the clustering technique which will be applied in Sect. 5.4.2.

2.2.1.1 Scheduling

Thanks to the a priori defined rates, a static periodic schedule (at compile time) for connected SDFGs can be easily constructed. Given an SDF specification, a schedule can be constructed by solving a topology matrix representing the SDFG [Lee and Messerschmitt, 1987a]. The number of columns in this matrix is equal to the number of actors. The entries to the matrix are either the number of produced tokens (positive number) or consumed tokens (negative). The SDFG in Fig. 2.3 can be described by following topology matrix:

$$T = \begin{pmatrix} 2 & -2 & 0 & 0 & 0 \\ 0 & 3 & -1 & 0 & 0 \\ 0 & -3 & 0 & 1 & 0 \\ 0 & 0 & 1 & -1 & 0 \\ 0 & 0 & -1 & 1 & 0 \\ 0 & 0 & 0 & 1 & -3 \end{pmatrix}$$

A Periodic Admissible Sequential Schedule (PASS) exists if the rank of the matrix $rank(T) = s - 1$ where s is the number of actors in the graph (c.f. proof in [Lee and Messerschmitt, 1987a]). A topology matrix has a proper rank ($rank(T) = s - 1$) if there is a strictly positive integer vector γ in its right nullspace (according to [Lee and Messerschmitt, 1987b]), meaning that $T\gamma$ is the zero vector:

$$T\gamma = 0$$

where γ is called *repetition vector* and it describes the minimum number of activation of every actor in each scheduling period. An SDFG is said to be *consistent* (see Def. 4.2.4) if and only if a positive integer repetition vector γ

exists. The schedule should be *periodic* because the the SDFG is assumed to a have an infinite stream of input data, *admissible* meaning that all actors are run only when data is available, and *sequential* meaning that the actors are executed sequentially on a single-processor [Lee and Messerschmitt, 1987b].

In our example in Fig. 2.3, the topology matrix T has a rank of 4 which is fulfills the condition that $rank(T) = s - 1$ (where $s = 5$) which implies that a valid PASS schedule exists.

$$\gamma = J \begin{pmatrix} 1 \\ 1 \\ 3 \\ 3 \\ 1 \end{pmatrix}$$

for any positive integer J.

Clearly if a schedule for a single processor (PASS) exists, then there also exists a schedule for multiple processors (PAPS), since in the trivial case all computation can be scheduled on the same processor [Lee and Messerschmitt, 1987a]. Heuristics which help constructing PAPS schedules can be found in [Lee and Messerschmitt, 1987a]. Describing these heuristics would be out of the scope of this work, since we assume an a priori constructed schedule for our real-time analysis method (see Sect. 4.1.1 in Chap. 4).

After describing the basic mathematical method to determine a PASS schedule, we now elaborate on the different existing scheduling methods (suggested in [Stuijk, 2007]) which can be used to realize scheduling of actors within the same SDF application on one side and scheduling between different SDF applications mapped to the same processor on the other side. These scheduling strategies are typically either *compile-time* scheduling (e.g. static-order scheduling) or *run-time* (e.g. round-robin and time division multiplex) strategies [Moonen, 2009]. In general, run-time scheduling requires a run-time supervisor (an operating system) which can lead to severe overheads in terms of performance. On the other hand, this is not the case for compile-time scheduling where (if any) only small run-time overheads are introduced due to the scheduling process (no need for an operating system).

In the following a short description of the scheduling mechanisms used in this thesis. In addition, we will take a look at each scheduling strategy, similar to [Stuijk, 2007], to see whether it is *composable* or *flexible*. According to [Stuijk, 2007], a scheduling strategy is said to be composable if the timing behavior of applications can be analyzed in isolation. Flexibility of a strategy is defined, by the ability to deal with dynamically changing dependencies between actors [Stuijk, 2007].

Static-order Scheduling A static-order schedule for a set of actors (potentially of different SDFGs) where these actors are executed in a cyclic manner according to statically ordered list, as soon as their input data is available [Stuijk, 2007]. This means that a scheduler will wait until the first actor in the list gets ready (as soon as all its input data are available), then executes the ready actor and move to the next actor in the list ready to be executed. Clearly, a static-order scheduling is neither flexible nor composable. It is not flexible since all dependencies between the actors must be fixed and known at compile time [Stuijk, 2007]. The non-composability of static-order schedules is obvious since no actor or set of actors within such a schedule can be analyzed in isolation, as the inter-actor dependencies must be always taken into consideration.

Round-Robin Scheduling *Round-robin* (RR) scheduler can help to achieve more fairness to the execution of ready actors than the static-order schedule. Similar to the static-order scheduler, it gets a list of ordered actors, but with the difference that the RR scheduler checks if the current actor is ready (for e.g. check for input availability or output capability) then it either fires or gives the control back to the scheduler if this is not the case. In both cases of blocking or successful firing, the scheduler switches from the active actor to activate the next actor in the list. In addition to fairness, RR scheduling gives the required flexibility to handle actors for which the order of execution are not known when constructing the schedule [Stuijk, 2007].

Since every actor should wait for all actors to run in the list before it gets to run in the worst case, the worst-case response time of an actor in RR schedule can be calculated as follows:

$$t_{wcrt_j} = \sum_{\forall i \neq j} t_{wcrt_i} \tag{2.1}$$

RR scheduling is not composable since the response time of an actor strongly depends on the execution time of all actors in the schedule [Stuijk, 2007].

Time-Division Multiple-Access (TDMA) Scheduling A TDMA scheduler allows an actor to be executed in only specific time slot and switches to the next slot as soon as the previous slot expires, using the concept of periodically rotating wheel [Stuijk, 2007].

Since we consider, in this thesis, a non-preemptive (for the preemptive one c.f. [Stuijk, 2007]) TDMA scheduler (as assumed also in [Giannopoulou et al., 2012]), we assume that the worst-case execution time of an actor (or cluster of actors, details on this will follow in Sect. 5.4.3) does not

exceed the size of the corresponding slot. The following equation can now be used to calculate the worst-case response time of an actor in TDMA schedule:

$$t_{wcrt} = \sum_{i=0}^{Sl} T_i + (Sl \times \mathbf{s}), \tag{2.2}$$

where T_i is the slot size (in time units) of current slot i, s is the scheduler worst-case delay time needed to switch from one slot to another and Sl is the total number of slots.

It is obvious from Eq. 2.2 that the TDMA scheduling mechanism is composable, due to the fact that the worst-case response time of every actor can be analyzed in isolation from others since it is only affected by the slot length and the number of slots. Moreover, the TDMA scheduling is flexible in the sense that new actors can be added to the TDMA schedule as long as there are unreserved slots available [Stuijk, 2007].

A comparison was made between the above three scheduling methods in [Stuijk, 2007]. The author came to the conclusion that even though the TDMA scheduling is flexible and supports composability, it can (potentially) lead to over-allocation of resources in order to compensate the timing overheads for e.g. in the case where large portions of the slots are unoccupied.

In this work, we assume (see Sect. 4.1) that all scheduling strategies are non-preemptive meaning that actors cannot be preempted by the scheduler and they have to actively hand the control back to the scheduler after finishing or when blocking. While non-preemptive schedulers are easy to implement and overheads of scheduling are easily assessable, yet the non-preemptive scheduling[1] is known to be NP-hard even for single-processor platforms [Jeffay et al., 1991].

The above mentioned worst-case response time of actor (t_{wcrt_i}) is defined in this thesis as follows:

$$t_{wcrt} = t_{wcet} + t_{com} + t_{wait} \tag{2.3}$$

where t_{wcet} is the worst-case execution time of the actor when run on a single target processor (which can be achieved through a static analyzer see Sect. 2.5), t_{com} is the communication time needed by every actor firing to transport a number of tokens over a communication resource and t_{wait} is the waiting time of the actor induced when waiting for other actors to finish communication on communication resources.

2.2.1.2 Timing Properties

When mapped to an MPSoC, SDFGs exhibit interesting timing properties as shown in the example in Fig. 2.4. In this example we have an SDFG of five

[1] Deciding schedulability for a set of concrete periodic tasks is NP-hard in the strong sense [Jeffay et al., 1991].

actors and an MPSoC of three heterogeneous processors: one ARM processor, one Field Programmable Gate Array (FPGA) and one Digital Signal Processor (DSP). The next step would be to obtain a valid periodic static-order schedule (see Sect. 2.2.1.1) represented by the repetition vector as seen in Fig. 2.4 where except actors 3 and 4 (which should be executed three times) all other actors should be executed once. Then, the SDFG is partitioned in three partitions with the first partition consisting of actor 1 and actor 5 which are mapped to the ARM processor, the second partition consisting of actor 2 and actor 3 which are mapped to the FPGA, and the third partition consisting of actor 4 which is mapped to the DSP. These mappings could be reasonable depending on the nature of the actors and which criteria the designer wants to optimize (for e.g. energy, performance or cost purposes).

After being mapped to the MPSoC, actors are run according to the periodic static-order schedule (based on repetition vector in Fig. 2.4) on every processor and their resulting MPSoC schedule is shown in Fig. 2.4. As we can observe the SDF application goes through a *startup phase* (which could comprise several *iterations*) before it reaches the *stable periodic phase*. An iteration is a set of actor firings such that each actor in the SDFG has the same firing number as calculated in the repetition vector [Stuijk, 2007] . If we take a look at our example, then the SDFG needs a *period* of 7 units of time to complete a single iteration

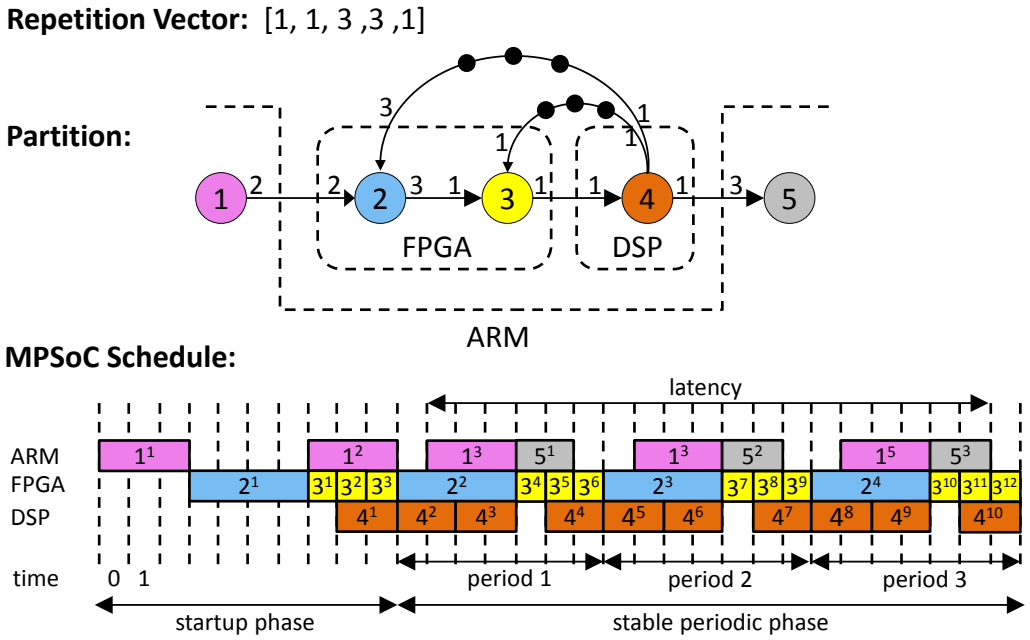


Figure 2.4: Example of an SDFG with its relevant timing properties (taken from [Lin et al., 2011])

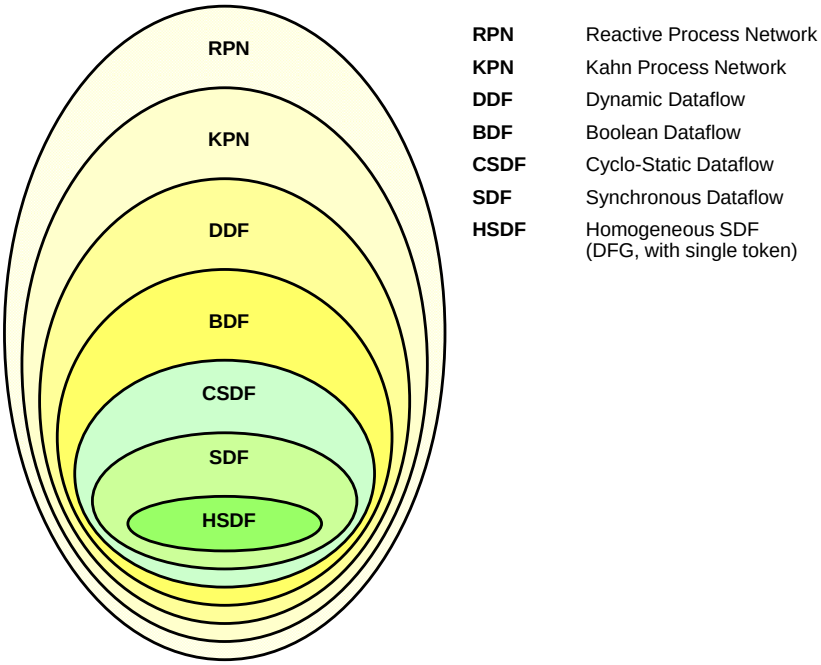


Figure 2.5: Process-based MoCs (taken from [Basten, 2008]), MoCs from BDF and above (highlighted with yellow) are Turing-complete

in the stable periodic phase (thus if one unit of time is equal to one cycle, then every 7 cycles an iteration of the SDFG is completed). For processors having 300 MHz clock frequency, we can reach a maximal *throughput* of about 43 MHz or about 43 Million iterations per second. Another relevant timing metric is the *latency* which is defined as the time duration from the first instance of the source actor of an SDFG to the last instance of the sink actor. In our example the latency is found to be equal to 19 units of time. In this thesis, we will only consider lower and upper bounds of the timing metrics (such as the worst-case period). By doing this, average timing estimations (such as average application throughput estimation) are no longer possible. The formal definitions of relevant properties (period, end-to-end latency, etc.) of SDFGs which are used in thesis are found in Sect. 5.2.8 and Sect. 4.2.1.

2.2.1.3 Expressiveness

According to [Moonen, 2009] homogeneous SDFGs and SDFGs have the same expressiveness as marked graphs [Commoner et al., 1971] and weighted marked graphs, which are a sub-class of timed petri-net theory, respectively. Despite the analyzability advantage of SDFGs (e.g. deadlocks and bounded buffer properties are decidable for such models [Lee and Messerschmitt, 1987b] and with the help of mathematical methods easy-to-analyze compile-time

schedules can be constructed), yet this comes at the cost of expressiveness.

One of the main limitations of SDF MoC is that dynamism cannot be handled for e.g. in the case where depending on the current scenario the application rates changes (this dynamism can be handled by an extension of SDFGs: the so-called Scenario-Aware Data-Flow (SADF) [Theelen et al., 2006] MoC). The fact that SDFGs do not support dynamism, makes SDFGs not adequate for many use-cases. Some of these were stated in [Schaumont, 2013], for e.g. stopping and restarting an SDFG is not possible since an SDFG can have only two states either running or waiting for input. In addition, reconfiguration of an SDFG to be able to (de)activate different parts depending on specific modes is not possible. Moreover, different rates depending on run-time conditions is not supported. Also modeling exceptions which might require deactivating some parts of the graph is not possible.

Another limitation (c.f. [Lee and Messerschmitt, 1987a]) of the SDF MoC is that conditional control flow is only allowed within an actor functionality but not among the actors. However, emulating control flow within the SDFG is possible even though not always efficient (c.f. [Schaumont, 2013]). An additional issue is that the SDF model does not reflect the real-time nature of the connections to the real-time environment.

More expressive data-flow graphs are shown in Fig. 2.5. A short description of these data-flow graphs can be found in [Kumar, 2009, Stuijk, 2007]. It is worth to note that the Boolean Data-flow graph (BDF) MoC which only extends the SDF MoC by enabling conditional and data-dependent execution (by adding *select* and *switch* control actors with boolean control inputs) is Turing-complete (c.f. [Buck, 1993]).

2.2.1.4 Clustering Methods

Multiple actors of an SDFG can be merged (clustered) together into one actor for various optimization purposes. In the following, the formal notation of the clustering method taken from [Bhattacharyya et al., 1997] (which we will use in Sect. 5.4.2) is presented.

Given a connected, consistent (see Def. 4.2.4) SDF graph $G = (\mathcal{A}, \mathcal{D})$ (where $\mathcal{A}$ is the number of actors and $\mathcal{D}$ is the number of channels with a repetition vector γ_G, a subset $\mathcal{Z} \subseteq \mathcal{A}$, and an actor $\Omega \notin \mathcal{A}$. Clustering $\mathcal{Z}$ into Ω means generating the new SDFG $(\mathcal{A}', \mathcal{D}')$ such that: $\mathcal{A}' = \mathcal{A}–\mathcal{Z} + \{\Omega\}$ and $\mathcal{D}' = \mathcal{D}–(\{e \mid (src(e) \in \mathcal{Z})$ or $(dst(e) \in \mathcal{Z})\}) + \mathcal{D}^*$, where $\mathcal{D}^*$ is a "modification" of the set of edges that connect actors in $\mathcal{Z}$ to actors outside of $\mathcal{Z}$.

For each $e \in \mathcal{D}$ such that $src(e) \in \mathcal{Z}$ and $dst(e) \notin \mathcal{Z}$, we define e' by:

$$src(e') = \Omega, dst(e') = dst(e),$$
$$delay(e') = delay(e), cons(e') = cons(e),$$
$$prod(e') = prod(e) \times (\gamma_G(src(e))/\rho_G(\mathcal{Z}))$$

where $\rho_G(\mathcal{Z}) = gcd(\{\gamma_G(A) \mid A \in \mathcal{Z}\}$, prod(e) and cons(e) are production and consumption rates of edge e receptively. Similarly, for each $e \in \mathcal{D}$ such that $dst(e) \in \mathcal{Z}$ and $src(e) \notin \mathcal{Z}$, we define e' by:

$$src(e') = src(e), dst(e') = \Omega$$
$$delay(e') = delay(e), prod(e') = prod(e),$$
$$cons(e') = cons(e) \times (\gamma_G(dst(e))/\rho_G(\mathcal{Z}))$$

and then, we can specify $\mathcal{D}^*$ by:

$$\mathcal{D}^* = \{e'(src(e) \in \mathcal{Z} \textbf{ and } dst(e) \notin \mathcal{Z}) \textbf{ or}$$
$$(dst(e) \in \mathcal{Z} \textbf{ and } src(e) \notin \mathcal{Z})\}$$

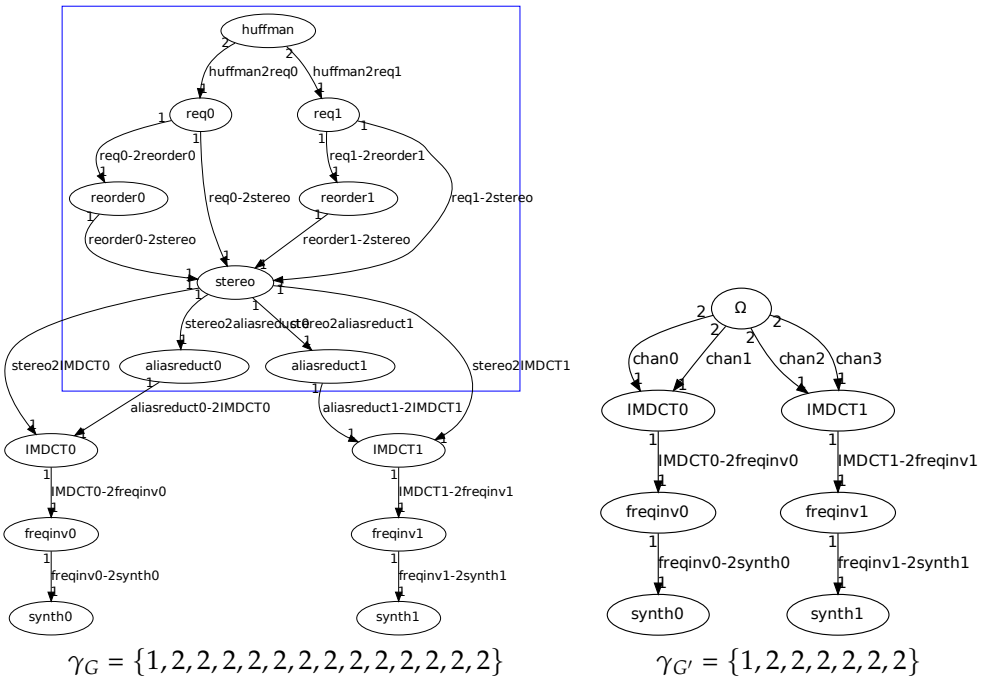
The graph that results from clustering $\mathcal{Z}$ into Ω in G is denoted $cluster_G(\mathcal{Z}, \Omega)$. $\mathcal{Z}$ is clusterable if $cluster_G(\mathcal{Z}, \Omega)$ is consistent and G is acyclic. If $G = (\mathcal{A}, \mathcal{D})$ is a connected, consistent SDF graph, $\mathcal{Z} \subseteq \mathcal{A}$, and $G' = cluster_G(\mathcal{Z}, \Omega)$, then $\gamma_{G'}(\Omega) = \rho_G(\mathcal{Z})$, and for each A $\in (\mathcal{A}–\mathcal{Z})$, $\gamma_{G'}(A) = \gamma_G(A)$.

Fig. 2.6 shows an example of clustering an MP3 decoder application[2] according to the above clustering technique, which we will evaluate later (see Sect. 7.2.4) to show the possible improvements of our state-based RT method when utilizing the clustering mechanism.

2.2.2 Simulink

In the following, a short summary (partially taken from the author's work in [Fakih, 2011]) about Matlab/Simulink main features (including the simulation kernel and the code-generation features) and its MoC is given. Simulink is a software package for modeling of *dynamic systems* and simulating them in virtual time. Modeling of such systems is carried out graphically through Simulink graphical editor consisting mainly of blocks and arrows (*connections*) between them representing signals. Each block has its input, output and optionally state variables. The relationship of the inputs with the old state variables and the outputs update is realized through mathematical functions. Blocks could be linear or nonlinear, discrete or continuous. Discrete blocks are basically either logical *boolean equations* or blocks triggered through events,

[2]MP3 decoder original definitions and timings were taken from [Stuijk et al., 2006]



$\gamma_G = \{1,2,2,2,2,2,2,2,2,2,2,2,2,2\}$ $\gamma_{G'} = \{1,2,2,2,2,2,2\}$

Figure 2.6: MP3 decoder clustering: $cluster_G(\{huffman, req0, req1, reorder0, reorder1, stereo, aliasreduct0, aliasreduct1\}, \Omega)$ with $D^* = \{chan0, chan1, chan2, chan3\}$

while continuous blocks are expressed as *differential equations*. One of the powerful features of Simulink is the ability to combine multiple simulation domains (continuous and discrete). This is very useful for embedded systems, where in general the controller has discrete model and the environment often needs to be modeled as a continuous one. Simulink also supports a state-based MoC the *Stateflow* [MathWorks, Inc., 2015f] which is widely used to model discrete controllers. Simulink allows a fast *Model-in-the-Loop (MIL)* verification, where the functional model (of the controller for example) is simulated and results are documented to be compared with further refinements.

Model of Computation Since Simulink is our basic framework for capturing the functional model of the system in our design flow, it is relevant to know the underlying MoC. Authors in [Gajski et al., 2009] make the following statement:

> *"Dataflow models map well onto concepts of block diagrams with continuous streaming of data from inputs to outputs. As a result, they are widely used in the signal processing domain and as the basis for many commercial tools such as LabView [96] and Simulink [95]." ([Gajski et al., 2009]: 55)*

This means that Simulink supports a data-flow MoC also referred to as the synchronous block diagram (SBD) in [Pouzet and Raymond, 2009]. In

[Lublinerman and Tripakis, 2008] a method was presented to automatically transform SDFGs into SBDs, such that the semantics of SDF are preserved, and it was proven that Simulink can be used to capture and simulate SDF models. As a conclusion from the researches above, we can say that Simulink supports a timed data-flow MoC[3] which could be used to capture SDFGs.

Simulink also offers a control-based MoC represented in the *Stateflow toolbox*. The *Stateflow* language is based on the *statecharts formalism* supporting a combination of *Statecharts*, *Flowcharts* and *Truth tables*. The graphical *Statecharts* language directly realizes a *Hierarchical Concurrent Finite State Machine (HCFSM)* model as stated in [Gajski et al., 2009]. This means that a *Stateflow* diagram extends the classical finite state machine (pure Mealy or Moore automaton semantics) formalism by adding hierarchy and concurrency (parallel states). In this work, the *Stateflow* is only considered at the block level, thus abstracting from the single transitions and activities within this block.

Simulation Kernel Simulink uses an idealized timing model for block execution and communication, with both consuming no simulation time or in other words running infinitely fast. Typically, blocks are evaluated at certain time steps depending on a custom *fixed-step size* (or *sample time*) parameter, which can be set globally or individually for each block. This sample time parameter specifies the period of execution [Lee and Neuendorffer, 2005] of the model (or for each block). As mentioned before, Simulink supports both discrete time and continuous time simulation where the simulation of continuous models is based on differential equations. The values are interpolated using *numerical integration* techniques between the different time points of the fundamental sample time. In Simulink, these techniques are called solvers which are of two types: *variable-step* solver and *fixed-step* solver. In the official Simulink documentation [MathWorks, Inc., 2015c] we read the following:

> *"Both fixed-step and variable-step solvers compute the next simulation time as the sum of the current simulation time and a quantity known as the step size. With a fixed-step solver, the step size remains constant throughout the simulation. In contrast, with a variable-step solver, the step size can vary from step to step, depending on the model dynamics." ([MathWorks, Inc., 2010]: 592)*

In choosing a solver a trade-off between accuracy and performance of the simulation is made. For example the *Runge-kutta-4* solver is more accurate than *Euler* solver but consumes more computational time. Since for models with a variable-step solver code-generation is not possible [MathWorks, Inc., 2015c],

[3]Since Simulink has a notion of time (sample time) and it supports dynamic blocks with variable rates (for e.g. the switch block which can switch between data of variable sizes).

we will only support models with fixed-step solver with a fixed-step size in this thesis.

Because of scheduling issues and in order for a model with multi-rates (different sample times) to be simulated, some essential blocks for rate conversion (so called `Rate-transition` blocks) must be inserted between blocks with different sample times or else the simulation will fail with an output error message.

Code Generation The *Embedded Coder* [MathWorks, Inc., 2015a] takes Simulink models as an input and generates C/C++ source code optimized for embedded systems with configuration/customization options. The generated code can be optimized for different architectures and run on typical microcontrollers (MCU). Embedded Coder makes the *Software-in-the-Loop (SIL)* and *Processor-in-the-Loop (PIL)* verification and validation techniques possible. In the SIL technique the controller model is replaced by the generated code from the embedded coder (usually embedded in a s-function) and the behavior of the code is compared with the reference data achieved from MIL (described above). On the other hand, by the PIL technique the generated code is directly tested on a target processor, the code is compiled with a target compiler and downloaded to an evaluation board with the target controller. The PIL evaluation gives accurate details about the code size, the required RAM/ROM, the stack consumption over time and the execution times.

In our design flow (see Chap. 6), a *Virtual-Platform-in-the-Loop* (VPIL) is enabled in which the generated code after being mapped and run on a virtual-platform target processor (before deploying on the real hardware) is run in the loop with the environmental golden model in Simulink and an evaluation is done regarding functionality and execution times. Also other evaluations are possible with the help of this VPIL such as the code size the required RAM/ROM, stack/heap size analysis, interrupt analysis, tracing etc.

2.3 Timing Issues of MPSoCs

Current architectures e.g. in the automotive domain are witnessing a strong trend of increasing the number of processors in order to achieve increased performance and reduced Space Weight and Power (SWaP). In this thesis, we will consider architectures consisting of various processors (embedded processing elements, FPGAs, DSP) which are connected to storage resources through interconnects. We will use a similar terminology to that defined in [Rochange, 2011] to differentiate in the MPSoC between *storage resources* (e.g. memories, buffers and caches) which keeps information for a while (for several cycles or perma-

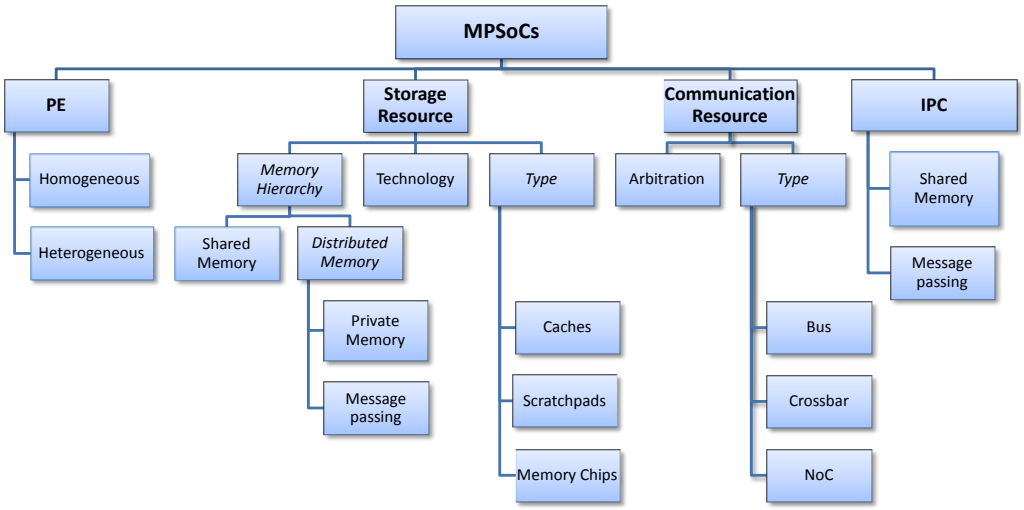


Figure 2.7: Decision tree of an MPSoC design

nently) and *communication resources* (buses, interconnects) where information is transfered from a sender to a receiver at each (number of) cycle(s).

Fig. 2.7 shows the most relevant decisions to be met when developing an MPSoC application and which can influence its timing behavior. The processing elements (PE) could be homogeneous or heterogeneous where in the latter case for e.g. the instructions of some PE are optimized for a certain application domain. In this case, possible conversion[4] of data representation (referred to as endianess) should be performed when different PE are exchanging tokens over a communication resource.

Concerning storage resources, there are different *technology* types: *volatile* (such as SRAM and DRAM) where data are lost after power being unplugged and *non-volatile* (such as ROM or flash memory) where data remains conserved even if the power supply is no more existent [Lee and Seshia, 2012]. Storage resources (typically SRAM) which are used to store temporarily working data are called either *scratchpads* if they have a distinct set of addresses and the program is responsible for moving data into them or out of them to the distant memory [Lee and Seshia, 2012] or are called *caches* if they are able themselves to duplicate data existing in the distant memory in them and to synchronize (according to coherence strategy) or replace them (according to a replacement strategy) when needed [Lee and Seshia, 2012]. In addition, we can, depending on the hierarchy level, differentiate between *shared* memories and *distributed local* memories. A shared memory can be typically accessed by any PE in the

[4]Since every communication partner in a heterogeneous platform can have different data-formating, data-formatting rules should be applied. These rules can be either defined globally or locally for every two communication partners [Gajski et al., 2009].

MPSoC. Whereas local memories can be either *private* memories with exclusive access to one PE or can be utilized to realize a message passing between a subset of existing PEs (typically between two of them).

As to the communication resources in the MPSoC, different *types* of interconnects are available which can support *arbitrations* of various complexities. *Bus*-based communication in MPSoCs is still the state-of-the-art in many industrial domains (especially in the automotive architectures) and has proven to be efficient connecting a small numbers of PEs (see controller area network (CAN) bus [ISO11898-1, 2003]). For more flexibility *crossbars* were introduced (see Aurix platform [Infineon Inc., 2013]) which can improve performance by enabling one-way communication paths (without contention) between masters and slaves on the platform. With increasing the number of PEs in the platform, flexibility is required when connecting several PEs and in order to alleviate the contention bottleneck of traditional communication resources, *Network-On-Chip (NoC)* based interconnects were suggested [Bjerregaard and Mahadevan, 2006].

Several *inter-processor communication (IPC)* types can be applied to achieve data transfer between two different processor elements ranging between *shared-memory based IPC* or direct *message passing* through dedicated hardware components (like FIFOs or mailboxes) which exhibits performance improvements compared to shared-memory IPC.

Since predictable design is becoming inevitable in safety-critical systems, we will take a look in the following sections at the most important timing behavior aspects of the MPSoC components (influencing the predictability) with a focus on the communication resources' properties. Tab. 2.1 summarizes the timing effects at the level of every component in the MPSoC which we will describe in detail in the following sections.

2.3.1 Processor Elements

In nowadays MPSoCs, sharing of logical/execution units among processor elements is possible if *hyperthreading* is enabled [Kotaba et al., 2013]. This contention can lead to timing variation at the instruction level depending on the current number of PEs trying to access these units.

Another major bottleneck for the timing analysis of software applications on PE are the instruction pipeline stages which can also be shared through parallel hyper-threads [Kotaba et al., 2013]. In addition, highly optimized (for improving average performance) pipelines with dynamic branch prediction (based on speculation) makes it even harder to predict the execution time of single instructions [Cullmann et al., 2010].

Since preemptive scheduling is sometimes indispensable to schedule a task set, preemption is also an issue at the PE level. Preemption (interrupts are also considered as preempting tasks [Cullmann et al., 2010]) requires context

Table 2.1: Mechanisms affecting the temporal behavior of an MPSoC (based on [Kotaba et al., 2013])

Class	Shared Resource	Mechanism
PE	Pipeline stages	*Contention* by parallel hyperthreads *Dynamic branch prediction*
	Logical units	*Contention* by parallel applications
Storage resource	Memory (DRAM)	*Interleaved access* by multiple PEs causes address set-up delay Delay by *memory refresh*
	Shared Cache	*Cache line eviction* *Contention* due to concurrent access *Coherency*: - Delay due to contention by coherency mechanism - Read requested by lower level cache - Read delayed due to invalidated entry - Contention by coherency mechanism on this level
	Local Cache	*Coherency*: - Read delayed due to invalidated entry - Contention by coherency mechanism from lower level cache *Cache line eviction*
	TLBs	*Coherency* overhead
Communication resource	Bus	*Contention*: - by multiple PE - by other device: IO, DMA, etc. - by coherency mechanism traffic
	Memory controller	*Contention* due to concurrent access
	Bridge	*Contention* by other connected buses
Addressable device	I/O devices	Overhead of *locking mechanism* accessing the memory I/O Device *state altered* by other thread/application *Interrupt routing* overhead *Contention* on the addressable device - e.g. DMA, Interrupt controller, etc. *Synchronous access of other bus* by the addressable device - e.g. DMA

switching and could cause non-deterministic timing behavior due to eviction of data in the cache or in the resources inside the PE such as the pipeline.

2.3.2 Storage Resources

Contention occurs on a physically shared memory (where instructions and data can be stored and are accessible to all PEs), when PEs perform concurrent accesses. Especially if the path in between the PE and the memory consists itself of other shared resources such as the cache, the interconnect and the memory controller. Typically, in such cases concurrent accesses are serialized (e.g. through arbitration on the interconnect) but still these interleaved accesses of multiple PEs operating on different memory pages to the main memory might cause address set-up variable delays [Kotaba et al., 2013]. In this case, the memory controller must continuously open and close new pages impacting the timing behavior of the MPSoC. Beside above issue and only specific to dynamic RAM, refresh delays due to memory refresh cycles also impact the

overall timings [Kotaba et al., 2013].

Caches exhibit many challenges concerning their timing behavior assessment due to their complex architectures optimized for fast buffering of data. In addition, the multi-layer cache hierarchies ranging from first level (L1) to intermediate level (L2) caches which also may be shared (or processor-local) makes their temporal behavior analysis even harder. Starting with a local (non-shared) cache which requires a coherency strategy that takes care of invalidating outdated cache entries (e.g. by a read access on destination memory through L2 to L1 cache). In addition, coherence strategy could again lead to contention on another cache with different level [Kotaba et al., 2013]. Since caches have a limited size, replacement strategies responsible of evicting data (for e.g. when a new task is accessing the destination memory) also impact the timing behavior (for more information c.f. [Cullmann et al., 2010]). Beside above issues, by a shared cache other issues emerge due to the contention resulting from the concurrent access of the different PEs sharing this cache. For e.g. blocking times are induced when PEs try to concurrently access the shared cache. Moreover, read requests of the lower level cache invoke the coherency mechanism which also induces contention and in turn also impacts the overall timing behavior. Similar coherency overhead exists when using Translation Look-aside Buffer (TLB) [Kotaba et al., 2013].

2.3.3 Communication Resources

On the communication resource level, contention results when multiple PE try to access it concurrently. In addition, contention resulting from the coherence mechanism (for e.g. the case when one PE updates data in the destination memory, this update invokes an invalidation update on all caches which contain these data). Furthermore, contention could be also invoked by devices other than the PEs such as the I/O devices, or DMA controller. Depending on the nature of the communication resource, these contentions can be avoided or minimized e.g. in the case of Network-on-Chip (NoC) having enough channels to serve all connected PEs [Kotaba et al., 2013]. Another major issue, is the kind of arbitration used which decides which PE by a concurrent access should be served first. While TDMA arbitration policy insures determinism since maximum latency can be guaranteed, other policies which allow starvation of PEs (e.g. fixed-priority) or highly depend on the run-time state (e.g. First-Come-First-Serve: FCFS) are more difficult to analyze.

At the level of a *Memory controller*, contention due to concurrent access (depending on the interconnect type) can take place. In this case, the memory controller must continuously open and close new pages leading to timing behavior impacts of the overall MPSoC timing behavior.

Bridges can be used to connect multiple buses and interconnects to each

other. Also at the level of bridges contention occurs when multiple requests from the connected interconnects are issued to the bridge.

Since modeling communication resources (see Sect. 5.2.5) will be a major contribution of this thesis, we will elaborate in the following on their arbitration issues and their timing models.

2.3.3.1 Scheduling (arbitration)

Similar to the scheduling mechanisms presented in Sect. 2.2.1.1, we will describe in the following the arbitration mechanisms of shared communication resources which control concurrent access requests of multiple PEs to a shared storage resource. All arbitration mechanisms used in this thesis are non-preemptive (c.f. [Abel et al., 2013] for a description of preemptive arbitration protocols) meaning that the arbiter grants access to the arriving access only if no other request is currently served.

We will differentiate (Similar to [Abel et al., 2013]) between *Time-driven* arbitration (TDMA) where a predefined schedule assigning fixed time slots to every PE to access the communication resources and *Event-driven* arbitration (First-Come-First-Serve, Round-Robin, Fixed-priority) where at run-time the arbitration mechanism decides which PE should be granted access.

Time-driven Arbitration We have already described the TDMA mechanism in Sect. 2.2.1.1 used for scheduling SDFGs. In the following, the same mechanism will be described but now applied to arbitration of accessors on a shared communication resource. In this case, every PE is allocated a priori to a time slot of a fixed slot length where it can perform its operations (transfers on the shared storage resource).

In order to insure that every transaction of a PE finishes before the slot time expires (since we assume a non-preemptive arbitration), the length of all slots is set to the transaction communication time (including arbitration cycle time and memory access delay) with the maximal delay which can be requested on this communication resource i.e. if t_{max_n} is the maximal communication time of actor n needed to transport a number of tokens among all its ports to a target shared storage resource (including the latency of the storage resource) then the slot size T_{sl} can be calculated as follows:

$$T_{sl} = max\{t_{max_1}, \ldots, t_{max_n}\}$$

Knowing the slot size, we are now able to calculate the WCRT of a PE access to the shared storage resource according to a TDMA arbitration as follows:

$$t_{wcrt_j} = n \times T_{sl} \tag{2.4}$$

whereby n specifies the number of PEs and T_{sl} is the slot size (in time units). The TDMA arbitration is composable and flexible for the same reasons mentioned in Sect. 2.2.1.1.

Event-driven Arbitration Here we will elaborate on the event-driven arbitrations used in this thesis. By a *fixed-priority* arbitration policy, a unique priority is assigned to each PE and if contention occurs on the communication resource, the PE with the highest priority is granted access. Suppose that PE with subscript 0 is the one with the highest priority then the WCRT of PE_0 can be calculated as follows [Pitter and Schoeberl, 2010]:

$$t_{wcrt_0} = \max_{0<i\leq n-1} \{t_{WCCT_i} - 1\} + t_0 \tag{2.5}$$

where i is the identification of all lower priority PEs, n the number of PEs in the system, t_{WCCT_i} represents the maximum duration among all access instances of storage resource accesses of PE_i and t_0 is the time needed to access the storage resource for PE_0. Eq. 2.5 represents the case where another lower priority PE gets the communication resources and only after one cycle $(t_{WCCT_i} - 1)$ PE_0 the one with the highest priority issues a request, which is the worst-case scenario for PE_0. Yet, calculating the WCRT of lower priority PEs accessing storage resources is much more difficult (according to [Pitter and Schoeberl, 2010]) due to the fact that it is strongly dependent on the number of active PEs. For e.g. suppose the higher priority PE prevent the lower priority PE from accessing the storage resource indefinitely, in this case obviously it is not an easy task to bound the WCRT of the lower priority PE.

Fair arbitration can be achieved through *round-robin (RR)* arbitration (similar to RR scheduling Sect. 2.2.1.1). By every arbitration, a counter (typically beginning from 0) indicates the identification of the next PE to be granted the access to the interconnect and which is incremented after every arbitration and thus insuring starvation-freedom between accessors. The WCRT of a PE accessor on a communication resource with a RR arbitration can be calculated as follows [Pitter and Schoeberl, 2010]:

$$t_{wcrt_j} = \sum_{\forall i \neq j} (t_{WCCT_i}) + t_j \tag{2.6}$$

where t_{WCCT_i} represents the maximum duration among all access instances to the storage resource accesses of PE_i (other processors than PE_j), and t_j is the time needed to access the storage resource for PE_j.

In a *First-Come-First-Serve (FCFS)* arbitration a FIFO queue maintains requests from accessors and the oldest request in the queue is granted access to the shared communication resource. According to this scheme and similar to

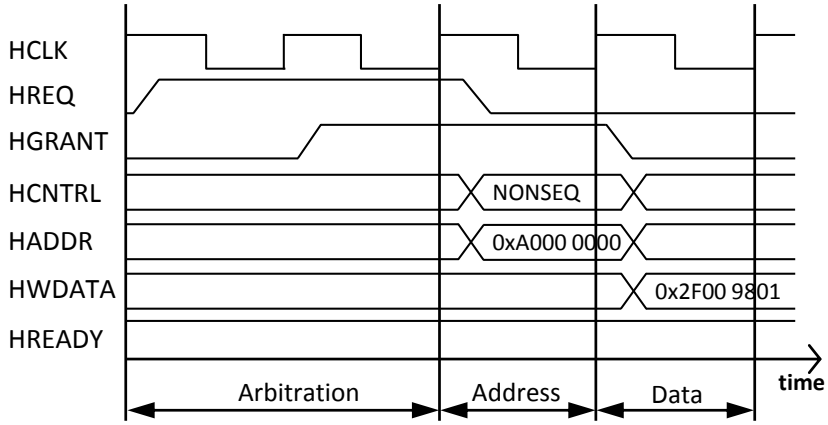


Figure 2.8: Cycle-accurate `Write` single-beat transfer (based on [ARM, 2006, ICVerification , 2015])

RR arbitration, FCFS also insures starvation freedom and fairness among accessors since the PE which has the oldest request is served earlier as the others. Authors in [Shabbir et al., 2010] noted that, in the case of a FCFS arbitration, when a new access request of PE_i arrives on the communication resource, it is assumed that, in the worst-case, it waits for all other PEs which could already have pending requests in the FIFO queue. This WCRT can be calculated similarly to the WCRT of a RR arbitration according to Eq. 2.6.

For real-time applications with hard real-time requirements time-driven arbitration (e.g. TDMA) are superior to event-driven since their composable and predictable behavior makes validating their RT requirements feasible [Marwedel, 2010].

2.3.3.2 Timing models

Modeling communication resources with their timing properties is indispensable for the timing validation of RT requirements of embedded applications running on MPSoCs (see Chap. 5). In this section, we will describe two different timed models (c.f. *bus-functional* models in [Cai and Gajski, 2003]) of the communication resources: *time-accurate* communication model and *cycle-accurate* communication model.

A cycle-accurate model specifies delays (time) in terms of the bus master's clock cycles which can be derived from the communication interconnect protocol. Fig. 2.8 and Fig. 2.9 show how such a cycle-accurate protocol looks like for a `Write` single-beat and a `Write` burst transfer respectively. In the following, we will describe exemplary the main differences between the two transfer styles according to AHB bus protocol [ARM, 2006, ICVerification , 2015]. In both cases, an arbitration phase first takes place, where a bus master requests access to the

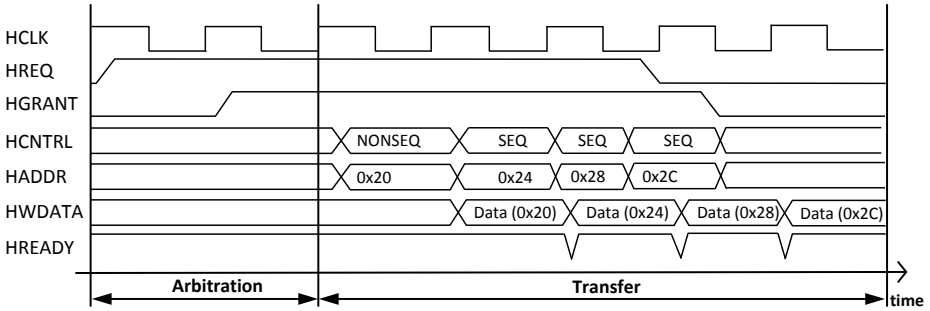


Figure 2.9: Cycle-accurate `Write` burst transfer (4-beats based on [ARM, 2006, ICVerification , 2015])

bus by using `HREQ` signal. If the arbiter decides that this master has the highest priority (according to an arbitration mechanism) then a `HGRANT` is asserted. After some delay the bus master is notified and is set as current master of the bus by setting `HMASTER` signal (not depicted in the figures). Afterwards, address and data phases occur in which some differences can be observed between the single-beat (see Fig. 2.8) and the burst transfer (see Fig. 2.9). After checking if the slave is ready (`HREADY`), the master drives `HADDR` along with other control signals that indicate the type (`Read/Write`), size (byte, half-word, word) and length of the transaction.

In a single-beat transfer (see Fig. 2.8) the length of transaction is set to `single` and the `HTRANS` signal (included in the control signals `HCNTRL`) is set to `NONSEQ`. In addition, arbitration is redone directly after the transaction is finished (signaled by reseting the `HGRANT`), where the master if wanting to continue communication must acquire again the bus (by asserting `HREQ`) and its access would be granted or blocked depending on the arbitration mechanism. In a burst transfer (see Fig. 2.9), however, the length may vary from two to several single-beats (from 4 8, to 16 single-beats per transfer in the AHB). In Fig. 2.9 four sequential single-beats write accesses are depicted. After the arbitration phase, the master indicates a burst in the AHB protocol by using `HTRANS` signal (belonging to the control signals `HCNTRL`). `HTRANS` is set first to `NONSEQ` indicating the first transfer of a new transaction. In the next address phase, `HTRANS` is set to `SEQ` indicating that a sequential transfer of the same transaction follows. During this phase, the address is simply incremented to the next "beat" (incrementing burst: e.g. `0x20`, `0x24`, `0x28` and `0x2C` in Fig. 2.9). Meanwhile, the bus is reserved for the current master (`HGRANT` remains high) until the last "beat" access is done. If any request from other masters is acquired, then it would be blocked until finishing the burst transfer and a new arbitration phase begins.

In difference to a cycle-accurate model, in a time-accurate model lower/upper latency bounds (for e.g. in Fig. 2.10 the time is limited in the range between

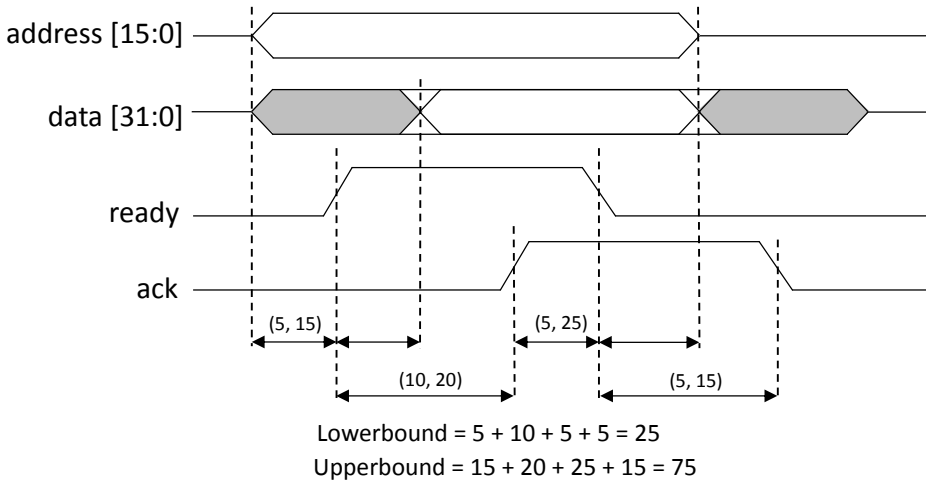


Figure 2.10: Time-accurate bus-functional model (taken from [Cai and Gajski, 2003])

25 and 75) are determined with the help of the time diagram of interconnect's protocol abstracting away from details of the communication protocol. This model would be appropriate in case no accurate (constant) timings can be obtained when transferring data (of specific size) through an interconnect with a specific communication protocol.

2.3.4 Addressable Devices

Typically addressable devices (e.g. DMA controller, Input/Output (I/O) devices or interrupt controller) are memory-mapped and can be accessed by the PEs via the same shared communication resource. For this, there are two options: either these devices are shared by all PEs or only one PE is connected to them and responsible of data exchange [Pitter and Schoeberl, 2010]. In the first case locking mechanism (e.g. *spinlocks*) are needed to ensure exclusive access since truly parallel execution is possible in an MPSoC (in difference to single-processor platform) [Kotaba et al., 2013]. Obviously, the overhead caused by the locking mechanism influences the timing behavior of the MPSoC.

Another issue is the contention caused by the DMA controller access to the shared communication resource which is similar to the contention caused when multiple PEs access the communication resource. In addition, interrupts controllers in MPSoCs usually routing interrupts to multiple PE are typically more powerful than those in a single-processor platforms and can play a role in reducing temporal effects [Kotaba et al., 2013].

2.3.5 Inter-Processor Communication (IPC) Styles

In general, within an inter-processor communication (IPC), intended to exchange data between PEs, mechanisms should be defined such as where to store shared data (tokens), how these shared data are transfered between sender and receiver (including routing along multi-hop paths) and how this transfer is synchronized.

Authors in [Gajski et al., 2009] stated that shared data can be mapped either to the local memory of one of the PEs and accessed through the other PE through the communication resource via a *memory-mapped I/O* fashion, or a local copy of the shared data is created in the local memory of every PE (*distributed storage*) which requires keeping local copies synchronized by sending synchronization messages through the communication resource. Another way is to store the shared data on *shared memory* where data can be accessed through the communication resource from both communication partners.

Two main types are identified of IPC (similar to [Grüttner et al., 2011]) in order to exchange data: *shared-memory* based IPC and *message-passing* based IPC. By the shared storage resource scheme, a memory-mapped interface is used to reserve address space in the shared storage resource where specialized communication mechanisms such as FIFO queues can facilitate the IPC which is issued to the shared storage resource via the communication resource [Schaumont, 2013]. By the message-passing scheme, special communication memory (like hardware FIFOs or mailboxes [Grüttner et al., 2011]) are used which can leverage greater performance, but with the restriction that engaged PEs must support extra ports with predefined instructions to move data through these ports [Schaumont, 2013].

As to synchronization mechanisms which insure that shared data is available, we differentiate between *interrupt-based* synchronization and *polling-based* synchronization. While interrupt-based synchronization alleviates traffic on the communication resource by sending events via dedicated set of wires compared to polling, yet this method can lead to temporal non-deterministic behavior because of interrupt routing and context switches which are usually provoked by interrupts.

Another issue is the nature of an IPC whether its *synchronous* or *asynchronous*. By synchronous IPC both the initiator and the target are participated by the communication and both of them must be ready to complete the transfer or else they are blocked and must wait for the communication partner to get ready. By an asynchronous communication, the communication partners can send messages independent from the state of each other and continue their progress without blocking.

2.3.6 Predicable Design of MPSoCs

After elaborating on the timing issues of different MPSoCs components in the last sections, we will now take a look at an excerpt of recent research and summarize their recommendations for enabling predictable MPSoCs design and RT analysis of these MPSoCs.

For the certification in the avionics domain, which is one of the restricted domains towards usage of MPSoCs, a strict temporal and spacial separation of running tasks on the MPSoC must be guaranteed [Karray et al., 2013]. This requires assuring that any contention at any component-level of MPSoC is avoided (c.f. [Karray et al., 2013] for an example how this can be done) which is indeed a very restrictive approach.

In [Cullmann et al., 2010] the authors gave a classification of architectures w.r.t predictability considerations in the design of multicores: *fully timing compositional*, *compositional with bounded effects* and *non-compositional platforms*. *Fully timing compositional* architectures do not exhibit any *timing anomalies*[5] and the usable hardware platform is constraint to be fully compositional. In this case a local worst-case path can be analyzed safely without considering other paths. *Compositional with bounded effects* architectures exhibit timing anomalies but no *domino* effects[6]. Authors in [Cullmann et al., 2010] argument that it would be best to use/design fully timing compositional architectures, since then the complexity of RT analysis is strongly reduced with the absence of timing anomalies and domino effects.

Authors in [Metzlaff et al., 2011] recommend the usage of small and simple PEs as the best building blocks for a predictable MPSoC. They argument that while complex PEs with out-of-order execution and sophisticated branch predictions can perform better, yet they are costly in terms of power, area and lead to significant increase of RT analysis complexity. Both [Cullmann et al., 2010, Metzlaff et al., 2011] suggested to support shared communication resources with easy-to-predict arbitration protocols (such as TDMA) and to use private (distributed) storage resources (referred to as spacial isolation). Furthermore, [Cullmann et al., 2010] suggests the usage of private caches with easy-to-predict Least Recently Used (LRU) replacement strategies instead of shared caches to alleviate contention. The usage of TDMA arbitration is adequate since time slots are assigned to tasks statically and it can be guaranteed that no unexpected run-time interferences can affect the timing behavior.

[5] *Timing anomaly* is referred to "*the situation where a local worst-case doesn't contribute to the global worst-case*" in [Cullmann et al., 2010] e.g. shorter execution time of an actor can lead to a larger response time of the application.

[6] *Domino effect* occurs when the execution time difference is arbitrary high between two states of the same hardware [Cullmann et al., 2010].

In [Metzlaff et al., 2011], authors argument that timing uncertainty of a task running on a PE in an MPSoC comes from non-local memory accesses (e.g. by IPC or when sharing resources), that is why they recommend to reduce RT analysis complexity by supporting single task analysis per PE (which is already supported by static analysis see Sect. 2.5) and independent from the analysis of the network communication.

For Commercial-Off-The-Shelf (COTS) MPSoCs a smart configuration can be done for making them predictable. This configuration discourages the usage of shared caches in COTS, enables partitioning of memories (if supported by the hardware) to avoid interferences among PEs and utilizes predictable arbitration features of communication resources (as in the MPC8641D avionic processor in [Cullmann et al., 2010]).

2.4 Interaction with the Environment

As seen in Fig. 2.1, the interaction with the environment is one of the issues influencing the timing behavior of an embedded application running on an MPSoC. In order to circumvent the lackness of real-time connections of the SDF MoC to the outside world [Lee and Messerschmitt, 1987b], we will support in our system model, in this thesis, event arrival models to represent incoming events (if existent) from the system environment. There are different arrival models discussed in the literature [Hendriks and Verhoef, 2006]: periodic, periodic with jitter, sporadic and bursty event streams.

In this work, only independent source actors of single SDFGs can be triggered by an event of an event trigger. In addition, we consider periodic with jitter event triggers which can be implemented as timed automata as done in [Hendriks and Verhoef, 2006] (see Sect. 5.2.1). Formally, an event trigger is characterized by a period p and a jitter j, where $p, j \in \mathbb{N}_{\geq 0}$ and $j \leq p$ [Hendriks and Verhoef, 2006]. In addition, we assume similar to [Hendriks and Verhoef, 2006] that the jitter must be smaller than or equal to the period and thus avoiding bursty events with overlapping subsequent intervals (c.f. [Hendriks and Verhoef, 2006]).

2.5 Real-time Analysis Methods

After describing the task model used in this thesis and the different components of the MPSoC influencing its timing behavior, we will now take a look at the different RT analysis methods which can be utilized to analyze the temporal behavior of MPSoCs. A RT analysis method is indispensable for a real-time system where the correctness not only depend on the correct functionality but also on the fact that their services (results) should be delivered in time. De-

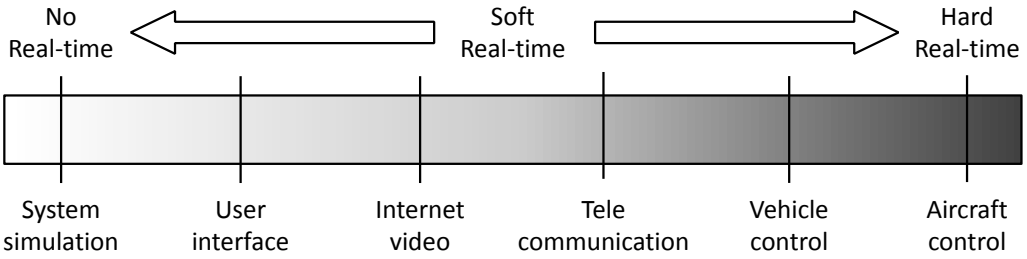


Figure 2.11: Timing criticality (taken from [Ermedahl and Engblom, 2007])

pending on the application domain (see Fig. 2.11), the timing criticality of an application can vary from hard real-time systems (e.g. aircraft control) where a violation of the real-time requirement can lead to catastrophic results, to non real-time application (such as a GUI) where the harmfulness by a violation is very limited [Ermedahl and Engblom, 2007].

Typically, in order to insure that the real-time system has a correct timing behavior, upper/lower bounds on the execution time of the software running on the target hardware must be obtained i.e. it is necessary to determine the *worst-case execution time (WCET) upper bound* and *best-case execution time (BCET) lower bound* respectively. There are two different approaches to achieve above goal: the *static* analysis approach and the *dynamic (measurement-based)* approach.

In a *static (formal)* approach mathematical analysis is performed on a formal representation of both software and hardware with the help of automated tools. The WCET analysis depends on the state space of the input model including the timing properties[7] of the hardware and the logic of the program code. This approach indeed guarantees safe upper/lower bounds for all executions of the functional model, as seen in Fig. 2.12. Nevertheless, the main challenge is to convert all above mentioned system properties to an abstract mathematical model which would normally require huge time overheads even for small systems. Another disadvantage of this approach lies in the obtained pessimistic upper/lower bounds that overestimate the real WCET. These pessimistic results can lead to resources' wastage in the final implementation which can be unaffordable.

In *dynamic methods*, use-case driven timing measurements of less critical applications (see Fig. 2.11) are performed either using a virtual-hardware platform simulation model (with variable abstraction levels ranging from *untimed* to *cycle-accurate* abstraction level see Sect. 2.5.1) or by running it on the target hardware employing hardware tracing facilities. In this way, the designer can test the functionality and through guided simulation with clever test-cases

[7]The hardware model for a single-processor typically includes the time models of the cache, the PE and its pipelining properties.

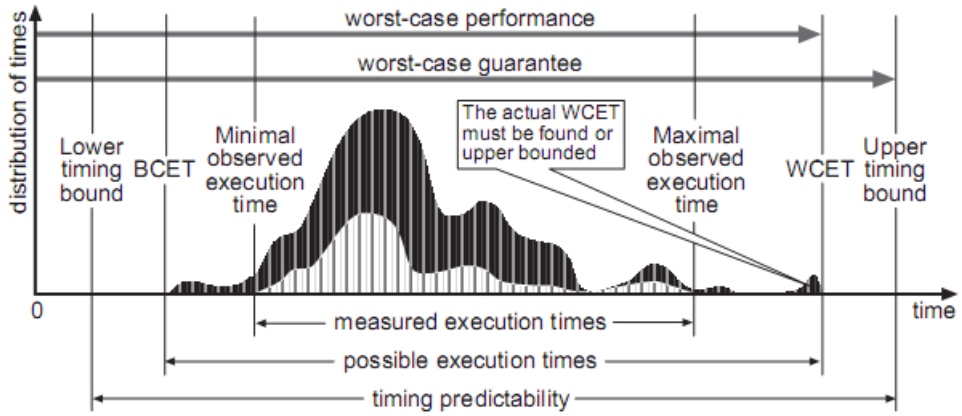


Figure 2.12: Basic notions concerning timing analysis of systems (taken from [Wilhelm et al., 2008])

can verify critical corner issues. Due to the complexity of the system, a full-coverage checking is not possible or it would be at least so much expensive as the formal verification. This approach is still state-of-the-art in industry but not applicable to applications with hard real-time requirements since there is no guarantee that the worst/best corner cases are measured (see the bright curve in Fig. 2.12). Another typical disadvantage of this approach is that the measuring artifacts could influence the timing behavior of the MPSoC also referred to as *invasive* timing measurement. This approach can also be combined with *automatic test-case generation* trying to reach a full coverage of all possible paths through the execution of a software task under analysis which in turn increases the confidence of the estimated execution times.

2.5.1 Dynamic Real-time Methods

Measurement-based techniques are still state of the art to provide an average case timing behavior of embedded applications executed on MPSoCs. An excerpt of dynamic RT methods ranging from *oscilloscopes, hardware traces, high-resolution timers, performance counters, profilers, operating system facilities, emulator* and *simulators* can be found in [Ermedahl and Engblom, 2007]. In the following, we will elaborate on the used simulative based method in this thesis which is based on virtual-hardware platforms.

Typically, implementation models (see Fig. 2.2) are captured in so-called virtual-platforms [IEEE-1666, 2012, Popovici and Jerraya, 2010]. Virtual platforms (VP) are abstract hardware models implemented via programming languages for the sake of simulating the behavior of the hardware. VPs adapt instruction-set simulators for modeling processors, and provide cross-compilers to load developed software as binaries to the target processor mem-

ory where it can be executed without any modification. As a result, a fast and sufficiently accurate (depending on the abstraction level see below) host-based simulation [IEEE-1666, 2012] of the implementation model is enabled. A host-based simulation enables execution and verification of the embedded application (running on the virtual-hardware platform) natively on the designer's host machine. VPs were proved to be efficient in covering the gap between the software and hardware in SLD allowing a faster and early simulation and validation of the developed embedded applications. In addition, VPs take less effort and time to develop than the costly development of prototyping boards. Moreover, VPs are accessible i.e. unlike physical hardware they can be inspected, debugged and observed, and when needed flexibly changed, or duplicated.

Transaction-level /Cycle-approximate Models In the system-level design process, refining the implementation model is done by the step-by-step integration of hardware (HW) issues in the model and taking care of the SW/HW interface synthesis [Gajski et al., 2009]. Typically a designer begins with an untimed functional model as seen in Fig. 2.13. This model is then refined to a *Transactional Level Model (TLM)* where the communication between the various components of the model at this level is realized through function calls and is independent of the later-used bus interfaces. At this level, computation and communication timing overheads are modeled as annotated delay functions which are obtained through real-measurements or estimations (e.g. via datasheet documentation). TLM models run at fast simulation speeds and give system designers the ability to estimate design performance metrics, with the disadvantage of inaccuracy, which might not be acceptable as in the case of validating timing requirements of hard-real time applications. The next step is then to refine the TLM to a *Cycle-accurate Model (CAM)* (see Fig. 2.13) where both communication and computation models are extended with more hardware details which make them cycle-timed models. CAM is used by HW designers to verify the correctness of the generated system HW components and by SW designers to verify the system software providing very accurate timing statements, but with the disadvantage of slow simulation speeds compared to simulation speeds of TLM models.

2.5.2 Static (Formal) Real-time Methods

Static real-time analysis methods can be either: *analytical* or *state-based* methods. Most of the available static RT methods in the literature (see Sect. 3.1.1) are of analytical nature (c.f. [Perathoner et al., 2009] for an overview) since these depend on solving closed-form equations which gives them the advantage of being scalable to analyze large-scale systems. A well-known analytical (stateless) method is the *Integer Linear Programming (ILP)*

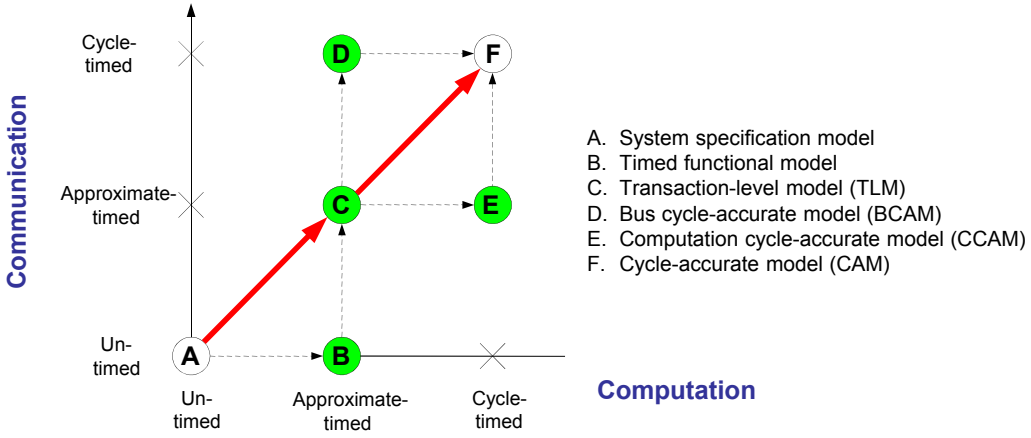


Figure 2.13: TLM and CAM models (taken from [Gerstlauer, 2009])

[Ferdinand and Heckmann, 2004, Li et al., 2007, Li et al., 1997] used to search for the longest execution time path of a *Control Flow Graph (CFG)* running on some specific hardware (ILP is typically utilized in a WCET analyzers for single-processor platform c.f. [Wilhelm et al., 2008]). Real-Time Calculus (RTC) [Thiele et al., 2000] is another analytical method which won attention in the last decade in the real-time domain. Despite their advantages of being scalable, analytical methods abstract from state-based modus operandi of the system under analysis (such as complex state-based arbitration protocols or inter-processor communication task dependencies) which leads to pessimistic over-approximated results compared to state-based RT methods [Huber and Schoeberl, 2009, Perathoner et al., 2009]. In addition, complex properties such as reachability of certain states cannot be verified by such methods.

State-based RT methods are based on the fact of representing the System Under Analysis (SUA) as a transition system (states and transitions) and since they reflect the real operation states of the actual system behavior, tighter results can be obtained compared to analytical methods. Another advantage over analytical approaches is the ability to verify complex properties on the system and to obtain counterexamples in case a property is not satisfied. In addition, integration of complex hardware models is done easily due to the modular nature of these methods compared for instance to ILP methods [Huber and Schoeberl, 2009]. Nevertheless, state-based RT methods are even for small designs not intuitive, they induce much complexity and can have great computation time overheads. Especially for large industrial systems these methods are very hard to be applied or causing high setup costs while risking a state-space explosion of the SUA. In the following, we will elaborate on the state-based RT analysis methods, as these would provide the basic foundation

of our proposed approach in this thesis.

2.5.2.1 State-based RT Analysis Methods

State-based RT analysis methods are based on model-checking methods which were developed in early 1980's [Clarke and Emerson, 1982, Queille and Sifakis, 1982] for the purpose of automatic verification of finite state-based systems. Model-checking methods are very reliable verification methods that enable complete coverage checking of certain specified properties based on an intelligent exhaustive search of the state space of a certain model (where states are modeled explicitly) i.e. for certain properties it is verifiable whether they are satisfied by the model implementation or not [Clarke and Emerson, 1982]. Specification (correctness) properties to be checked are formalized through propositional temporal logic [Pnueli, 1977]. Thus, as seen in Fig. 2.14, the main inputs to a model-checking based method is a transition system representation (states, transitions and labels) of the system model and the temporal properties (causal or temporal) to be proved on this transition system representation [Gajski et al., 2009]. The main idea of model-checking (see [Clarke and Emerson, 1982]) is to unroll the transition system to an infinite computation tree [Gajski et al., 2009] and to utilize a search algorithm which tries, starting by an initial state, to find a state trace which fulfills or violates the properties to be checked. The output of a model-checker would be whether or not the given property holds and in the case the property does not hold, to provide a counter-example.

Our approach in this thesis (see Chap. 5), utilizes Timed Automata (TA) [Alur and Dill, 1990] as a common semantic model to represent execution time boundaries (best-case and worst-case execution times) of SDF actors and communication FIFOs, as well as their mapping and utilization of MPSoC resources such as scheduling of SDFGs, shared communication resource access protocols for interconnects, local and shared memories. The resulting network of TA is

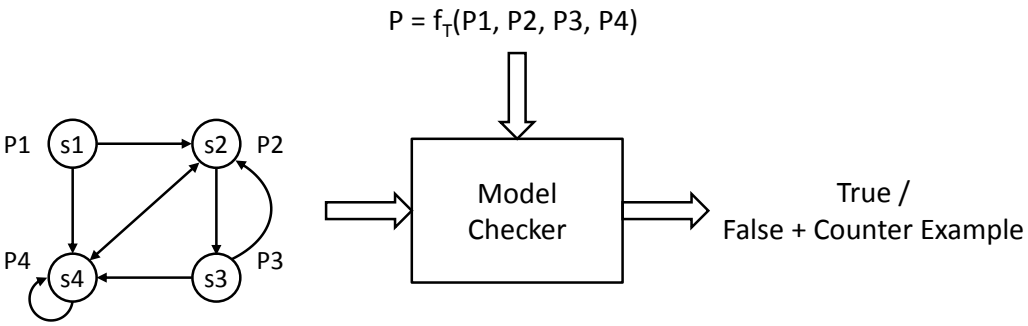


Figure 2.14: Model-checking Approach (taken from [Gajski et al., 2009])

analyzed using the UPPAAL [Bengtsson and Yi, 2004] model-checker for providing safe timing bounds of the implementation. In the following, we will elaborate on UPPAAL TA defining them to the extent needed by this thesis. Afterwards, we reason on the decidability properties of TA and finally we describe the temporal logic (TCTL) language used for capturing properties specification.

UPPAAL Timed Automata Finite State Machines (FSM) which are essentially graphs with states as nodes and transitions as edges [Gajski et al., 2009] are one of the basic foundations of computer science which are suitable to model the behavior of a system. But when it comes to a real-time domain, reasoning on the temporal system interaction with the physical environment is indispensable which disfavors the usage of FSM since they abstract away from time. Timed automata [Alur and Dill, 1990] are finite automata extended by a finite set of real-valued variables called *clocks* (evolving at the same rate) used to model real-time systems and to circumvent above issue. The main elements of a timed automaton consists of a number of *locations* each denoting the state at which an automaton can be active, *edges* representing the possible transitions from one state to another, *invariants* and *guards* on edges.

A location in the automaton can have an invariant associated with it. An invariant is a clock constraint which allows a location to be active only for a given amount of time. The set of constraints denoted by $\Phi(X)$ on the set of clocks X is defined inductively as follows (c.f. [Olderog and Dierks, 2008]):

$$\varphi ::= x \sim c \mid x - y \sim c \mid (\varphi_1 \wedge \varphi_2) \tag{2.7}$$

where

- $\sim \in \{<, >, \leq, \geq\}$, $x, y \in X$ and $c \in \mathbb{Q}^+$

Before defining timed automata formally, we first define some necessary basic notations. A *valuation* v of a clock is a function which assigns each clock a non-negative real number. In addition, the notation $v \models \varphi$ is used to express the fact that a clock constraint φ evaluates to be true under clock valuation v.

The following definitions were taken from [Olderog and Dierks, 2008] (with minor clarification additions adopted from lecture notes in [Westphal, 2012]) to give an insight to the basic semantics of timed automata.

Definition 2.5.1 (*Pure Timed Automata*)**.** A *timed automaton* [Olderog and Dierks, 2008] is a structure

$$\mathcal{A} = (L, B, X, I, E, \ell_{ini}),$$

where

- L is a finite set of locations,

- $B \subseteq Chan$, where a *Chan* is a set of channel names and for each $a \in Chan$ two visible *actions* are observable:
  - $a?$ and $a!$ denote input and output on the channel respectively
- X is a finite set of clocks,
- $I : L \to \Phi(X)$ is a mapping that assigns to each location a clock constraint, its invariant,
- $E \subseteq L \times B_{?!} \times \Phi(X) \times \mathcal{P}(X) \times L$ is a set of directed edges with guards, channels and set of clocks to reset. An element $(l, \alpha, \varphi, Y, \ell') \in E$ describes an edge from location ℓ to ℓ' labeled with an *action* α, a *guard* φ, and a set Y of clocks that will be reset.
- $\ell_{ini} \in L$ is an initial location.

Definition 2.5.2 (*Operational Semantics of TA*)**.** The operational semantics of a timed automaton $\mathcal{A} = (L, B, X, I, E, \ell_{ini})$ is defined (according to [Olderog and Dierks, 2008]) by the labeled transition system:

$$\mathcal{T}(\mathcal{A}) = (Conf(\mathcal{A}), Time \cup B_{?!}, \{\xrightarrow{\lambda} | \lambda \in Time \cup B_{?!}\}, C_{ini})$$

where

- $Conf(\mathcal{A}) = \{\langle \ell, v \rangle | \ell \in L \wedge v : X \to Time \wedge v \models I(\ell)\}$ is the set of configurations of $\mathcal{A}$ and v is a valuation of clocks in X assigning each clock the current time,
- The set $Time \cup B_{?!}$ contains all labels that may appear at transitions,
- For each $\lambda \in$ Time $\cup B_{?!}$ the transition relation $\xrightarrow{\lambda} \subseteq Conf(\mathcal{A}) \times Conf(\mathcal{A})$ has one of the following two types:
  - *Delay transition* relation where some time $t \in Time$ elapses but location is left unchanged. Formally:

    $$\langle \ell, v \rangle \xrightarrow{t} \langle \ell, v + t \rangle$$

    iff $v + t' \models I(\ell)$ holds for all $t' \in [0, t]$.
  - *Action transition* relation where an action $\alpha \in B_{?!}$ occurs and some clocks may be reset, but time does not advance. Formally:

    $$\langle \ell, v \rangle \xrightarrow{\alpha} \langle \ell', v' \rangle$$

    iff there exists an edge $(\ell, \alpha, \varphi, Y, \ell') \in E$ with $v \models \varphi$ and $v' = v[Y := 0]$ and $v' \models I(\ell')$

- $C_{ini} = \{\langle \ell_{ini}, v_{ini} \rangle\} \cap Conf(\mathcal{A})$ with $v_{ini}(x) = 0$ for all clocks $x \in X$ is the set of initial configurations.

Several timed automata can be assembled to form a network of TA: $\mathcal{A}_1||\cdots||\mathcal{A}_n$ which can synchronize between each other through channels where $c!$ and $c?$ denote sending and receiving an event respectively. The parallel composition of two timed automata is defined according to [Olderog and Dierks, 2008] as follows:

Definition 2.5.3 (*Parallel Composition of Timed Automata*)**.** The parallel composition of $\mathcal{A}_1||\mathcal{A}_2$ of two timed automata $\mathcal{A}_i = (L_i, B_i, X_i, I_i, E_i, \ell_{ini,i})$, i= 1,2, with disjoint sets of clocks X_1 and X_2 yields the timed automaton

$$\mathcal{A}_1||\mathcal{A}_2 \stackrel{\text{def}}{=} (L_1 \times L_2, B_1 \cup B_2, X_1 \cup X_2, I, E, (\ell_{ini,1}, \ell_{ini,2}))$$

where the following hold:

- Conjunction of location invariants: $I(\ell_1, \ell_2) \Leftrightarrow I_1(\ell_1) \wedge I_2(\ell_2)$,
- The transition relation E (c.f. [Olderog and Dierks, 2008] for exact formal definition of E) is constructed by the following rules:
  - *Handshake* communication: synchronizing $a!$ with $a?$ yields τ (internal action) i.e. if $(\ell_1, \alpha, \varphi_1, Y_1, \ell_1') \in E_1$ and $(\ell_2, \bar{\alpha}, \varphi_2, Y_2, \ell_2') \in E_2$ with $\{a!, a?\} = \{\alpha, \bar{\alpha}\}$ then also
    $$((\ell_1, \ell_2), \tau, \varphi_1 \wedge \varphi_2, Y_1 \cup Y_2, (\ell_1', \ell_2')) \in E$$
  - *Asynchrony*: if $(\ell_1, \alpha, \varphi_1, Y_1, \ell_1') \in E_1$ then for all $\ell_2 \in L_2$ also
    $$((\ell_1, \ell_2), \alpha, \varphi_1, Y_1, (\ell_1', \ell_2)) \in E$$
    and conversely, if $(\ell_2, \alpha, \varphi_2, Y_2, \ell_2') \in E_2$ then for all $\ell_1 \in L_1$ also
    $$((\ell_1, \ell_2), \alpha, \varphi_2, Y_2, (\ell_1, \ell_2')) \in E$$

After defining the parallel composition of TA, the network of TA definition is described as follows (according to [Olderog and Dierks, 2008]):

Definition 2.5.4 (*Network of Timed Automata*)**.** A timed automaton $\mathcal{N}$ is called network of timed automata if and only if it is obtained as:

$$\text{chan } b_1 \cdots b_m \bullet (A_1||\cdots||A_n)$$

where a *local channel* b is introduced by the restriction operator ($\bullet$) which for a timed automaton $\mathcal{A} = (L, B, X, I, E, \ell_{ini})$ yields:

$$\text{chan } b \bullet \mathcal{A} := (L, B \backslash \{b\}, X, I, E', \ell_{ini})$$

where

- $(\ell, \alpha, \varphi, Y, \ell') \in E'$ if and only if $(\ell, \alpha, \varphi, Y, \ell') \in E$ and $\alpha \notin \{b!, b?\}$.

The state of a network of TA represents a vector of current locations of all TA including all clocks' valuations and synchronizations between the automata [Herber, 2010]. With the help of above definition we are now able to define the operational semantics of networks of TA as follows (according to [Olderog and Dierks, 2008]):

Definition 2.5.5 (*Operational Semantics of Networks of TA*)**.** Let $\mathcal{A}_i = (L_i, B_i, X_i, I_i, E_i, \ell_{ini,i})$ with $i = 1, \cdots, n$ be a set of timed automata with disjoint clocks. Then the operational semantics of the network

$$\mathcal{N} = \text{chan } b_1 \cdots b_m \bullet (A_1 || \cdots || A_n)$$

yields the labeled transition system

$$\mathcal{T}(\mathcal{N}) = (Conf(\mathcal{N}), Time \cup B_{?!}, \{\xrightarrow{\lambda} | \lambda \in Time \cup B_{?!}\}, C_{ini})$$

with

- $X = \bigcup_{i=1}^{n} X_i$ and $B = \bigcup_{i=1}^{n} B_i \backslash \{b_1, \cdots, b_m\}$,
- $Conf(\mathcal{N}) = \{\langle \vec{\ell}, v \rangle | \vec{\ell} \in L_1 \times \cdots \times L_n \wedge v : X \to Time \wedge v \models \bigwedge_{k=1}^{n} I_k(\ell_k)\}$,
- $C_{ini} = \{\langle (\ell_{ini,1}, \cdots, \ell_{ini,n}), v_{ini} \rangle\} \cap Conf(\mathcal{N})$ where $v_{ini}(x) = 0$ for all clocks $x \in X$,
- For each $\lambda \in$ Time $\cup B_{?!}$ the transition relation $\xrightarrow{\lambda} \subseteq Conf(\mathcal{N}) \times Conf(\mathcal{N})$ has one of the three following types:
  1. *Local transition* $(\vec{\ell}, v) \xrightarrow{\alpha} (\vec{\ell}', v')$ occurs if for some $i \in \{1, \cdots, n\}$ there is an edge $(\ell_i, \alpha, \varphi, Y, \ell_i') \in E_i$, $\alpha \in B_{?!}$ in the i-th automaton such that
     - $v \models \varphi$ (guard is satisfied)
     - $\vec{\ell}' = \vec{\ell}[\ell_i := \ell_i']$ (only i-th location changes),
     - $v' = v[Y := 0]$ (A_i 's clocks are reset),
     - $v' \models I_i(\ell_i')$ (destination invariant holds).
  2. *Synchronization transition* relation $(\vec{\ell}, v) \xrightarrow{\tau} (\vec{\ell}', v')$ occurs if there are $i, j \in \{1, \cdots, n\}$ with $i \neq j$, and some channel $b \in B_i \cap B_j$, there are some edges $(\ell_i, b!, \varphi_i, Y_i, \ell_i') \in E_i$ and $(\ell_j, b?, \varphi_j, Y_j, \ell_j') \in E_j$, i.e. the ith and the jth automaton can synchronize their output and input on the channel b, such that
     - $v \models \varphi_i \wedge \varphi_j$, i.e. both guards are satisfied,
     - $\vec{\ell}' = \vec{\ell}[\ell_i := \ell_i'][\ell_j := \ell_j']$,

– $v' = v[Y_i \cup Y_j := 0]$ and $v' \models I_i(\ell_i') \wedge I_j(\ell_j')$.

3. *Delay transition* relation $(\vec{\ell}, v) \xrightarrow{t} (\vec{\ell'}, v + t)$ occurs if for all $t' \in [0, t]$:

$$v + t' \models \bigwedge_{k=1}^{n} I_k(\ell_k).$$

i.e. all invariants are satisfied during the passage of time.

UPPAAL [Bengtsson and Yi, 2004] is a tool used to model, simulate and verify networks of parameterized extended timed automata. A timed automaton in UPPAAL is defined by a so-called *template*. These templates make it possible (similar to object-oriented *classes* concept) that a timed automaton once defined and implemented, can be instantiated to multiple TA each having different parameters. A system in UPPAAL consists of a finite set of these template instances. In addition to the timed automaton primitives, in UPPAAL the *pure* timed automaton properties are extended by the ability of declaring functions, bounded integer variables, binary uni-cast/multi-cast (broadcast) channels and urgent/committed locations [Herber, 2010]. Synchronization between timed automata is done through binary channels and in the case multiple synchronization is possible, one of them is chosen non-deterministically [Herber, 2010]. While broadcast channels never block, synchronizing sender and receiver do block on an uni-cast channel if the corresponding communication partner is not ready. Furthermore, all clocks in UPPAAL are initialized to zero and then increase with the same rate [Gustavsson et al., 2010]. If a location is *urgent*, this means that no time is allowed to pass while the template instance remains in this location. A *committed* location is more strict than the urgent one in the sense that additional to the fact that no time is allowed to pass, the automaton must leave the committed location in the next transition. In order to define a TA network model, UPPAAL offers three parts of declarations [Herber, 2010]:

1. *Global declarations* part where global variables, channels and clocks are declared.

2. *Parameterized timed automata* part where the TA are implemented with the help of a graphical editor and their parameters are defined and their local functions variables are declared.

3. *System declarations* part where the templates are instantiated and their network is declared.

Fig. 2.15 shows an example of a light switch, modeled as a system of two parallel TA. At the top of Fig. 2.15, we can see the system definition consisting of two template instances: `Lamp1` is an instance of template `Lamp` and `User1` is an instance of template `User`. The automata can synchronize through events

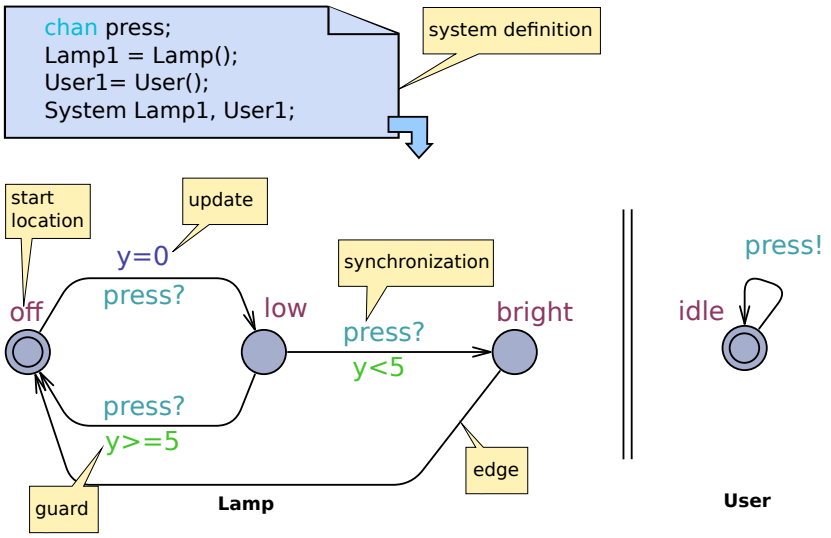


Figure 2.15: An example of a light switch modeled as a system of Timed Automata in UPPAAL (based on [Greenyer, 2010])

on *channels*. The labels `press!` and `press?` in Fig. 2.15 are examples for such channel events. In this case `press` is an uni-cast channel, which means that when the sending edge (labeled `press!`) fires, a currently enabled receiving edge (labeled `press?`) must fire synchronously. If more than one receiving edge is enabled, only one of these enabled edges is chosen non-deterministically for synchronization. If no receiving edge is enabled, the sending edge cannot fire. The latter two cases will, however, never occur in the lamp switch example.

UPPAAL extends the timed automata with integer bounded variables, data variables, urgent and committed channels and their networks which are used in UPPAAL can be defined as the formal definitions above (c.f. [Olderog and Dierks, 2008] for the complete formal description of extended timed automata).

Reachability Analysis and State Explosion Problem From the formal definitions of the timed automata networks, we notice that the set of configurations of the transition system is infinite due to the real-valued clock variables which makes model-checking these networks very difficult. In order to circumvent this, symbolic semantics were suggested by [Alur and Dill, 1994, Bengtsson and Yi, 2004] which were inspired from the idea of symbolic model-checking for untimed systems [Bengtsson and Yi, 2004]. In [Bengtsson and Yi, 2004] we read:

> *"It adopts the idea from symbolic model checking for untimed systems, which uses boolean formulas to represent sets of states*

> *and operations on formulas to represent sets of states transitions."* ([Bengtsson and Yi, 2004]: 92)

In this case, the infinite state space of TA can be finitely represented by symbolic states. In a first step towards enabling symbolic semantics of TA, authors in [Alur and Dill, 1994] suggested a finite representation of TA networks' state space in finite regions (equivalence classes using clock constraints) which is called *region graph*. A more compact and efficient representation of region automata called *zone graph* was presented in [Bengtsson and Yi, 2004]. Zone graphs on the other hand can be again compactly represented through *Difference Bound Matrices (DBMs)* [Bengtsson and Yi, 2004]. More details and formal definitions of the symbolic semantics (including DBMs, region and zone graphs) of timed automata are given in [Bengtsson and Yi, 2004]. Besides other optimizations, all above techniques are implemented in the UPPAAL framework [Bengtsson and Yi, 2004].

Despite the fact that the *location reachability* (reaching a given final state or a set of final states) is decidable for timed automata [Bengtsson and Yi, 2004], model-checking of TA can still suffer from the so-called *state-explosion problem*. Since the number of global states increases exponentially with the number of parallel TA and the number of components per TA [Clarke et al., 2012], a huge state space can be easily reached even for small models which could drive the model-checker to its limits[8]. The following example (taken from [Fränzle, 2012]) stresses the fact how enormous the state space could grow even for a small example. If we consider 11 components each with 8 states, this would yield $8^{11} \approx 9 \times 10^9$ nodes in the transition graph. At the other side, the explicit representation of a transition graph of above nodes (assuming no optimization) would require about 90 GByte of memory (assuming only 10 bytes memory per node). In general, the global state space of a network of TA grows exponentially with the number of concurrent components, number of global and local variables needed for the TA and number of synchronization channels [Clarke et al., 2012, Giannopoulou et al., 2012]. Another major aspect which could lead rapidly to a huge state-space, is the level of *non-determinism* represented in the considered TA templates.

Temporal Logic for Model-checking of Timed Automata Verification of system properties (requirements) formulated as queries is performed by the UPPAAL verifier. UPPAAL queries are a subset of the TCTL (Timed Computation Tree Logic) [Alur et al., 1990] specification language which is an extension of traditional CTL (Computation Tree Logic) adding temporal con-

[8] By overrunning the run-time and memory capacities of current computers on which model-checkers run.

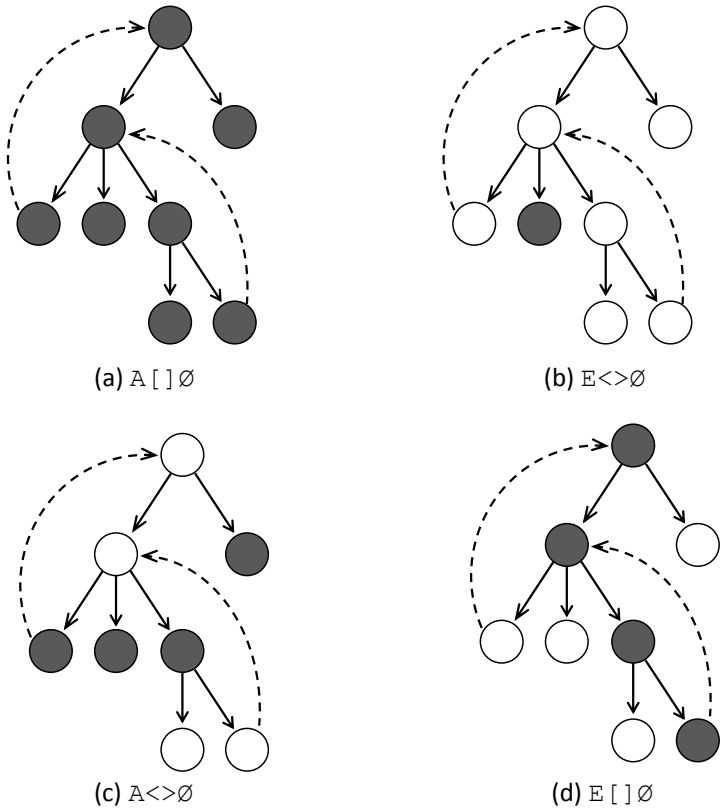


Figure 2.16: TCTL-formula (taken from [Bengtsson and Yi, 2004])

straints. The TCTL formulas which can be checked are (according to [Bengtsson and Yi, 2004], also see Fig. 2.16 for examples):

- `A[]` ϕ — Invariantly
- `E<>` ϕ — Possibly
- `A<>` ϕ — Always Eventually
- `E[]` ϕ — Potentially Always
- ϕ `-->` ψ — ϕ always lead to ψ which is equivalent to:
  `A []` (ϕ `imply A<>` ψ)`.`

where ϕ and ψ are logical properties to be checked locally on a state [Bengtsson and Yi, 2004]. `A` (always) and `E` (exists) are used to quantify over paths. Whereas `A` states that a specific property should hold in all paths, `E` states that at least one path exists where the property holds. On the other hand, symbols `[]` and `<>` are used to quantify over states in a path. While the symbol `[]` indicates that the property should hold for all states, `<>` indicates that at least one state should satisfy the property [Bengtsson and Yi, 2004].

TCTL allows to specify constraints over states (expressions φ over locations, variables and clocks) and *safety properties* (of the form A`[]` φ or E`[]` φ) meaning that a certain property holds in all states of the model, *reachability* properties (of the form E<> φ) meaning that a certain property holds in some state of the model, and *liveness* properties (of the form A<> φ) over infinite paths (see [Bengtsson and Yi, 2004] for more details) meaning that a certain property holds eventually. Moreover in UPPAAL the statement A`[] not deadlock`, verifies whether our system is *always* deadlock free or not. In addition, the specification supports temporal operators such as `sup` which searches for the *supremum* of a variable or a clock value in the system. Likewise, we could find the *infimum* by utilizing the `inf` operator.

2.6 Summary

In this chapter, we have presented the basic concepts relevant to our approach. After a short introduction about system-level design methodologies and the different MoCs used to capture the behavior of the system, we described the X-chart defining the main tasks in modern ESL design process. Afterwards, we elaborated on the SDF MoC with its special semantics, scheduling, expressibility and clustering possibilities. In addition, we described Simulink basic elements used to capture control applications and to enable model-based design with features such as automatic code generation. Next, we described the main timing issues which should be taken into consideration when analyzing the temporal behavior of embedded applications running on MPSoCs. After that a short overview of the different RT analysis methods was given. Finally, we presented the formal language of UPPAAL timed automata and the main constructs of model-checking which enable a state-based RT analysis of systems in rigor. The main message of this chapter is that timing predictability of full-featured MPSoCs is a very difficult task and that novel RT analysis methods are needed to cope with their complexities. Another important issue is the state-space explosion problem of state-based RT analysis methods, which faces a system designer when utilizing such methods. Thus, the challenge which is faced when developing a state-based RT analysis method, is how to choose the right abstraction level of the input model such that the method scales to be able to analyze systems with adequate sizes and at the same time can still obtain tight timing results.

Chapter 3

Related Work

After the comprehensive understanding of the fundamental timing issues influencing the timing behavior of embedded applications running on MPSoCs and the different kinds of RT analysis methods which can be used to analyze such effects, we will take a look in this chapter to existing work of the scientific community and try to spot light on our novel contributions. We will first consider analytical RT methods used to analyze the performance of *generic*[1] embedded applications on MPSoCs. Afterwards, analytical RT methods used for timing analysis of SDFGs are presented. Next, an excerpt of related work using state-based RT analysis methods to analyze generic applications, followed by an excerpt of research utilizing these state-based methods for analyzing SDFGs mapped to MPSoC architectures is discussed. At the end of this chapter, we will discuss the related work concerning our model-based design flow which will be presented in Chap. 6.

3.1 Formal Real-time Analysis Methods

As already concluded from the last chapter, finding lower and upper bounds of applications execution times (running on a target hardware platform) is crucial for real-time analysis of systems with hard real-time requirements. A short description of the history of formal RT analysis methods was done in [Gustavsson, 2010], in the following a summary is given. Alan Shaw [Shaw, 1989] was the first to introduce rules to construct a CFG (Control Flow Graph) from an executable software for temporal analysis, which

[1]The term *generic* is referred to tasks implemented in imperative or declarative programming languages respecting the von-Neumann model [Marwedel, 2010] where communication and computation cannot be separated in dedicated phases. This means that accesses to shared communication resource and computation can happen at any time and in any order [Schranzhofer, 2011].

built the foundation for later WCET reasoning and research. The work in [Colin and Puaut, 2000, Wilhelm et al., 2008] gives an excellent overview of the recent WCET research till the year of 2008. A well-known analytical method used to solve the WCET problem of embedded applications executing on single-processor platforms is *Integer Linear Programming (ILP)* [Ferdinand and Heckmann, 2004, Li et al., 2007, Li et al., 1997]. It is used to search for the longest execution time path of a *Control Flow Graph (CFG)* running on some specific hardware platform. Two approaches were established for computing the ILP problem [Huber and Schoeberl, 2009]: either graph-based approaches [Puschner and Schedl, 1997] (such as model-checking timed automata in UPPAAL) or IPET (Implicit Path Enumeration) [Li and Malik, 1995] based approaches.

Perathoner et al. [Perathoner et al., 2009] gave an overview of formal RT analysis methods typically used for RT analysis of applications running on distributed systems and MPSoCs, comprising analytical methods (*holistic scheduling*, *compositional analysis*) and state-based RT analysis methods (timed-automata based analysis) and tried to evaluate the tightness of the results obtained through these different methods.

In the following, we will elaborate on these formal methods used for RT analysis of MPSoCs. First we take a look at the analytical methods, and distinguish between methods used for generic tasks and those customized for SDFAs. The same procedure is done later on for state-based RT analysis methods.

3.1.1 Analytical Real-Time Analysis Methods

In the following, we will provide an overview of the state-of-the-art and related work of analytical RT methods used for both generic tasks and SDFAs on MPSoC architectures. It is worth to note, that extending formal approaches (developed for single-processors) towards analyzing applications running on MPSoCs, requires the consideration of shared communication resources delays besides delays of task execution on a processing element.

3.1.1.1 Generic Tasks on MPSoCs

Tendell et al. [Tindell and Clark, 1994] were the first to present a holistic approach[2] which analyzes preemptive fixed-priority scheduling on processing elements with a TDMA scheduling on the interconnects. Afterwards, Yen et al. [Yen and Wolf, 1998] presented a RT approach of a set of tasks executing on a heterogeneous distributed system with data dependencies, but they did not consider interprocessor communication and communication resources delays.

[2] A holistic approach extends traditional scheduling theory to apply on distributed systems combining the analysis of processor with bus scheduling [Perathoner et al., 2009]).

Pop et al. [Pop et al., 2002] presented also a holistic approach which enables the analysis of several input task models running on distributed systems with slot-based communication resources having static and dynamic phases.

Richter et al. presented in [Richter et al., 2003] a new compositional approach for RT analysis of MPSoCs which uses classical scheduling for the local analysis of system components and propagate through well-defined interfaces (event-stream models) the local results to other components. Later, the authors contributed to SymTA/S (Symbolic Timing Analysis for Systems) [Henia et al., 2005] commercial formal RT analysis tool which build upon the same compositional approach presented in [Richter et al., 2003]. Schliecker et al. [Schliecker et al., 2010] presented another extension of above methods, which analyzes the worst-case delay of a task (scheduled according to a fixed priority schedule) when accessing a shared resource with FCFS arbitration in an MPSoC. They assume that the maximum/minimum number of accesses (defined as above as event models) are known a priori in a time window. Similarly, Dasari et al. [Dasari et al., 2011] had similar assumptions (known maximum/minimum number of accesses) and suggested a method to compute more tight bounds of tasks' bus requests to tighten up the pessimistic results from [Schliecker et al., 2010].

Thiele et al. [Thiele et al., 2000] presented another modular approach based on real-time calculus (RTC). In this method, resource accesses can be abstracted by a so-called arrival curves [Perathoner et al., 2009] and analysis is made on the flow of events between different components (communication, computation resources of the system under analysis). Many extensions (e.g. [Schranzhofer et al., 2011, Pellizzoni et al., 2010]) for this approach were presented. In [Schranzhofer et al., 2011, Pellizzoni et al., 2010] it was stated that for generic execution models and generic arbitration policies, the problem of determining upper bounds on the WCRT is hard and the results are in general very pessimistic. In order to circumvent this, they suggested more restrictive models of execution (superblocks) and with the help of real-time calculus (RTC) they were able to perform a timing analysis of the interferences on shared resources (with different arbitration protocols). But still, the estimated bounds compared to a state-based RT analysis are very pessimistic especially in the case of state-dependent arbitration protocols like FCFS and RR since analytical methods don't model the states of resources explicitly. Furthermore, the proposed RTC analysis requires high design-time effort since the designer himself must construct the formal model, whereas in our case the formal model is automatically constructed from pre-designed timed-automata templates (see Sect. 5.1).

Anderson et al. [Andersson et al., 2010] presented a highly pessimistic approach which gave a maximal contention time independent from arbitration

protocol of the communication resource, by assuming that the current requesting task must wait for all other possible concurrent tasks accessing the resource.

3.1.1.2 SDFAs on MPSoCs

Bhattacharyya et al. [Sriram and Bhattacharyya, 2000] proposed to analyze hard real-time behavior of a *single* SDFG mapped to a multi-processor system by decomposing it into a homogeneous SDFG (HSDFG) which could be very time-consuming and could result in an exponential number of actors in the HSDFG compared to the SDFG. This in turn may lead to performance problems for the analysis method.

Poplavko et al. [Poplavko et al., 2003] was, to the best of our knowledge, the first to introduce communication modeling (i.e. with more than one latency actor) in data-flow graphs. Yet they assumed that there is no contention on communication resources.

A similar approach to [Poplavko et al., 2003] for analyzing SDFGs temporal properties was done in [Bekooij et al., 2004] where they took use of a network on chip which provides virtual point-to-point connections with a guaranteed throughput and maximal latency. Arbitration on shared communication resource was fixed to an arbitration protocol very similar to the TDMA protocol, enabling analysis of temporal effects of contention on the resource through computing worst-case-response-time.

Moreira et al. [Moreira et al., 2007] presented an approach to analyze hard real-time properties of multiple SDFGs mapped on MPSoCs with shared resources. They analyzed scheduling strategies on processors and showed that a combination of Time-Division Multiplex (TDM) and static order scheduling which can be modeled as extra nodes in the SDFG enable worst-case temporal analysis. One main decision made is that they assumed an MPSoC architecture designed to facilitate a worst-case analysis by following the rules presented in [Bekooij et al., 2004].

Stuijk et al. [Stuijk, 2007] presented a throughput-constrained multiprocessor resource allocation technique which extends previous work to deal with cyclic and multi-rate SDFGs. Like the two previous approaches, also in this case the communication between different processors is assumed to be a point-to-point communication through a network-on-chip with fixed latency.

Ghamarian [Ghamarian, 2008] presented novel methods to calculate performance metrics for single SDF applications which avoid translating SDFGs to HSDFGs. Nevertheless, architecture properties and resource sharing were not considered.

Moonen [Moonen, 2009] analyzed the mapping of SDFGs on a multiprocessor platform with limited resource sharing. With the help of a network-on-chip

supporting guaranteed communication services, they were able to easily derive conservative estimated bounds on the performance metrics of SDFGs.

In summary, authors in [Ghamarian, 2008] didn't consider shared communication resources, other authors [Poplavko et al., 2003, Bekooij et al., 2004, Stuijk, 2007, Moreira et al., 2007, Moonen, 2009] considered these, but they assumed a network-on-chip that supports point-to-point network connections with guaranteed communication services and latency. While our approach can easily support the analysis of NoCs with fixed latencies, it also goes further and tries to capture state-of-the-art buses' behaviors as found for e.g. in automotive domain, with heavily state-dependent arbitration protocols like FCFS.

Kumar [Kumar, 2009] presented a probabilistic technique to estimate the performance of SDFGs sharing resources on a multi-processor system. Although this analysis was made taking into account the blocking time due to resource sharing, the estimation approach focuses on analyzing soft real-time systems rather than hard real-time systems. Furthermore, the resource analysis was limited to non-preemptive, round robin scheduling.

Schabbir et al. [Shabbir et al., 2010] presented a design flow to generate multiprocessor platforms for multiple applications. They also provided two analysis techniques to predict the performance of the applications: a probabilistic based average-case analysis suitable for soft real-time tasks and a worst-case analysis for hard real-time. The former analysis is based on calculating the worst-case waiting time on resources (with a non-preemptive FCFS arbitration) as the sum of all actors' execution times which can access this resource. This is a safe but obviously a very pessimistic approach.

Recently, the work of Hausman et al. [Hausmans et al., 2013] and Lele et al. [Lele et al., 2014] addressed extending above data-flow based systems to support non-starvation-free scheduling particularly preemptive fixed-priority scheduling.

3.1.1.3 Discussion

Above work takes use of a purely analytical approach to obtain upper/lower timing bounds on the application execution time. While such approaches are fast and able to handle large systems, unfortunately they deliver pessimistic results especially when handling state-dependent arbitration protocols (c.f. [Perathoner et al., 2009]) typically used in MPSoCs. This fact leads to major concerns when trying to apply them in an industrial context. We will show in our experiments how far our state-based RT analysis can tighten the results compared to an exemplarily work [Shabbir et al., 2010] of the analytical approaches. In addition, interesting/complex functional or temporal properties (c.f. [Skelin et al., 2015]) cannot be easily analyzed by purely analytical approaches, for e.g. liveness properties or more general the reachability of a

certain state (see Sect. 2.5.2.1). We will present in Chap. 7 a set of such properties which can be validated through our approach.

3.1.2 State-based Real-time Analysis Methods

The discussion whether model-checking is suitable of WCET analysis for single-processor architectures, was first debated by [Metzner, 2004] and later evaluated in comparison with IPET in [Huber and Schoeberl, 2009] and compared to other RT methods for distributed systems in [Perathoner et al., 2009]. Metzner showed that model-checking is a feasible tool to compute safe WCET for CFG programs on pipelined processors with an instruction cache. In the following sections we will provide an overview of the state-of-the-art and related work in state-based RT analysis methods for both generic tasks and SDFAs when run on MPSoCs.

3.1.2.1 Generic Tasks on MPSoCs

Norstrom et al. [Norstrom et al., 1999] were the first to enable a real-time state-based analysis using timed-automata to represent system resources. They showed that the schedulability problem of their real-time system model, can be transformed to a reachability problem of the corresponding timed-automata representation.

Hendrik et al. [Hendriks and Verhoef, 2006] used timed automata to evaluate a RT analysis of a specific embedded application (in-car navigation system) and compared the results obtained with other analytical methods. They suggested the translation of applications modeled in UML diagrams specifications to timed automata. Yet, their MPSoC model was constructed for the case-study and thus not generic as ours and they considered a very simple communication bus model which lacks the ability to capture arbitration protocols of different complexities.

Lv et al. [Lv et al., 2010] extended the work of Norstrom et al. and presented an approach based on model-checking (UPPAAL) combined with abstract cache interpretation to estimate WCET of non-sharing code programs on a shared-bus multicore platform. Here the C-code of a task is translated into equivalent control flow graphs (CFG). The execution of every block in the CFG graph is classified either as cache miss/hit/unknown according to an abstract cache interpretation method. Next, the classified blocks with their corresponding delays are modeled each as a timed automaton. The task is then modeled as a timed automaton which represents the composition of these single timed automata. A bus with a First-Come-First-Server (FCFS) arbitration policy is also modeled as a timed automaton, and whenever a cache miss occurs, an access on the bus is issued.

Gustavsson et al. [Gustavsson et al., 2010] moved further and tried to extend the former work [Lv et al., 2010] concentrating on modeling code sharing programs and enhancing the hardware architecture with additional data cache but without the consideration of bus contentions. In their work, the whole multicore system is modeled as a set of timed automata including a task model, a core with pipeline model and instructions/data caches' models. In addition, they consider generic tasks modeled at assembly level and analyze these when mapped to an architecture where every core has its private L1 cache and all cores share an L2 cache without sharing a bus. Yet, the instruction level granularity of the modeled tasks leads to scalability problems even with a platform of four cores, on which four (very simple) tasks run and communicate through a shared buffer.

Despite the advantage of the former two approaches being applicable to any code generated/written for any domain, the fine granularity of the code-level or instruction-level impedes the scalability of the model-checking technique. In this work we intended to limit the application to an SDF MoC and limit the hardware architecture by disabling caches, in order to reason about the scalability of a model-checking-based method for performance analysis of SDFGs.

Recently, an approach was presented in [Giannopoulou et al., 2012] which combines model-checking with real-time calculus analysis was presented to extend the scalability of worst-case response time analysis in multicores. Tasks are limited to superblocks representation where resource access phases can be easily identified. In our work, we concentrate on SDF based applications with their specific properties and constraints. We also consider a less abstracted system model as the one considered in [Giannopoulou et al., 2012] where we model blocking at shared FIFO buffers, actors' multi-ports and hierarchical scheduling between different applications. Nevertheless, it is possible to use the abstraction techniques from [Giannopoulou et al., 2012] to analyze SDF applications.

Brekling et al. [Brekling et al., 2008] presented a timed automata-based approach to verify the impact of execution platform and application mapping on the schedulability (meeting hard real-time requirements). The granularity of the application is considered at the task level. With tasks and processors having their own timed automata, the approach scales up to 103 tasks mapped to 3 cores. Yet, the communication model is missing in this approach.

Zhang et al. [Zhang, 2011] proposed a model-checking approach to analyze the effects of L2 cache (without considering the interferences on a shared bus) in order to bound WCET for multicores. Their approach is based on providing a static bound on the number of additional cache misses due to inter-thread instruction interferences.

Büker et al. [Büker, 2013] presented an approach for testing real-time task

networks with functional extensions using model-checking. In their work, they have defined the semantics of function networks for modeling tasks. These function networks are then converted to timed automata according to templates defined for every element in the network. The scheduling is also modeled as a timed automaton. WCRT is then calculated for some test input vector through model-checking the function network timed automata with the chosen scheduling automaton. The main drawback of this approach is that no guarantees are given on the estimated WCRT, since it is test based.

3.1.2.2 SDFAs on MPSoCs

Some previous work [Geilen et al., 2005, Gu et al., 2007] used model-checking to optimize buffer sizes in SDF applications, and others [Liu et al., 2008] used SAT solvers to enable mapping and scheduling of HSDFGs with throughput optimization in mind. Other extensions of the above work with same purpose can be found in literature. In the following, we will only focus on state-based methods targeting the RT analysis of SDFGs.

Yang et al. [Yang et al., 2010] introduced a state-space exploration approach to verify the hard real-time performance of applications modeled with SDFGs that are mapped to a platform with shared resources. Nevertheless, they did not consider shared communication resources in their approach.

Malik et. al. [Malik and Gregg, 2013][3] presented a similar approach to ours using model-checking to statically distribute and schedule SDFGs on heterogeneous MPSoCs through translating them to timed automata, and showed that their method outperforms ILP formalization of the same problem in terms of run-time. No contention on the communication resource is considered and that is why the communication resources are not modeled explicitly.

Ahmad et al. [Ahmad et al., 2014] followed the same path of our work, presenting a translation of single SDFGs to timed-automata templates in order to analyze their behavior using model-checking. In contrast to our work, they focused on finding a maximal throughput on a given number of processors. In addition, communication resources (with contention) were not considered in their system model.

In [Zhu et al., 2014, Zhu et al., 2015] the authors (similar to our work) transform a system model which includes an SDFG and a multiprocessor platform to a (priced) timed automata network of UPPAAL and utilize an extended model-checker (UPPAAL CORA) to obtain optimal schedules combining optimization goals with optimal throughput and energy consumption. In difference to our approach, the authors didn't consider communication resources in their work.

[3] [Malik and Gregg, 2013] was published in September 2013 later than our work in [Fakih et al., 2013a] published in March 2013.

Very recently the authors [Skelin et al., 2015] presented a translation from Finite-State-Machine Scenario-aware-Data-Flow (FSM-SADF) graphs to timed automata. FSM-SADF is a MoC which is more expressive than SDF allowing more dynamism but at the same time entailing limitations on the analyzability. In an FSM-SADF MoC a set of typical scenarios are pre-defined (through a finite state machine) for a specific SDF application where it reacts to every scenario in a different manner leading to more efficiency and better throughput. Similar to our work, the authors used model-checking to analyze FSM-SADF and showed that complex properties which are not supported by traditional tools such as SDF^3 (utilizing analytical RT methods) can be analyzed. In difference to our approach, the authors concentrated in their translation and analysis only on the FSM-SADF MoC and did not consider other aspects as resources' sharing (contention on communication resources) in MPSoCs in their work.

Another recent work of Thakur et al. [Thakur and Srikant, 2015] presented a model-checking based approach (based on timed automata) for statically scheduling stream programs (based on SDFGs) in order to provide optimal mapping (in terms of performance) on heterogeneous architectures. In their work, different to our work they assume no contention on the communication resources and don't model them explicitly.

3.1.2.3 Discussion

As seen in the above sections, when using state-based RT methods for the analysis of generic tasks models on MPSoCs, they usually reach their limits even for very small systems. Yet, a better scalability can be definitely reached, if the task model is restricted to SDFGs (since SDFGs have the nice feature of clean separation of communication phases from computation phases) as we will show in this thesis. To the best of our knowledge, we pioneered the translation of SDFGs to timed-automata (in [Fakih et al., 2013a]) and we were the first to describe how to use model-checking to analyze the real-time properties (e.g. end-to-end deadline) of (multiple) hard real-time SDF applications mapped to MPSoCs. In addition, we evaluated different methods to improve the scalability of our state-based RT analysis method (see Sect. 5.4 and Sect. 7.2).

Our approach has been later taken up by other researchers in [Malik and Gregg, 2013, Ahmad et al., 2014, Zhu et al., 2014, Zhu et al., 2015, Skelin et al., 2015, Thakur and Srikant, 2015] in order to model-check SDFGs/SADGs, targeting various objectives (as described above). It is worth to note, that all these researches, unlike our approach, do not support the modeling of communication resources (with their contention properties). Yet, considering communication resources with their timing properties, as done in this thesis, is indispensable for a RT analysis method to be applicable on a real MPSoC.

3.2 Model-based Design Flow

In Chap. 6, we will present our model-based design flow for RT analysis of RT applications modeled in Simulink through translating them to SDFGs (see C3-1). Afterwards, our tool (first developed in [Schlaak, 2014]) takes the role of automatizing our state-based RT analysis method (see C3-2). And then with the help of a simulative method the *Virtual-Platform-In-the-Loop (VPIL)* simulation extended for MPSoCs (see C3-3), we enable a functional and accurate temporal Verification and Validation (V&V). In the following, we will position our main contributions (mainly C3-1 and C3-3) with respect to existing related work.

3.2.1 Simulink to SDFG Translation

In the last decade, several researches [Caspi et al., 2003, Miller et al., 2005, Zhang et al., 2013, Büker, 2013] have been conducted to enable a translation of Simulink models to other formal models for the purpose of formal analysis. In the following, we merely discuss previous work enabling the translation of Simulink models to SDFGs (taken from our prior work [Warsitz and Fakih, 2016]).

In [Dominik, 2011] a structural translation of Simulink models to homogeneous SDFGs (HSDFGs) was pursued with the objective of analyzing concurrency. HSDFGs are SDFGs with the restriction that the number of consumed and produced tokens of each actor must be equal to one 1 [Lee and Messerschmitt, 1987a]. The translation has been done for a fixed number of functional blocks but important attributes, such as the data type of a connection between blocks, have not been taken into consideration by the the translation.

In [Boström and Wiik, 2015] a translation from Simulink models to SDFGs was described. The aim of this work was to apply a methodology for functional verification of Simulink models based on *Contracts*. *Contracts* define pre- and post conditions to be fulfilled for programs or program fragments. However, authors give no clear classification of critical Simulink functional blocks (e.g the *switch* block see Sect. 6.2) which cannot be supported in the translation. In addition, *Triggered-/Enabled* subsystems and other important attributes such as the data type of a connection are not supported.

In [Li, 2013], only the source code of a so-called *Simulink2SDF* tool was published which enables a very simple translation of Simulink models to SDFGs. In this work all Simulink blocks, without any distinction, were translated to data-flow actors and similarly connections were translated in data-flow channels, the fact which makes the translation incomplete as we will see in Sect. 6.2.

Unlike the above work, we present a general concept in this work categorizing blocks and connections in Simulink models in different classes. Then we

describe how a translation based on the classification of the given block/connection to one/or more class(es) of the above identified is done. Our approach enables the translation of critical blocks (such as *Enabled /Triggered* subsystems) including the enrichment of the translated SDFG with important attributes such as the data types of tokens, tokens' size and sampling rates of actors (in case of multi-rate models). Another issue is that the target of our translation is different from above mentioned related work. In [Dominik, 2011] the translation was used for parallelism analysis of Simulink applications implemented on multicore platform. In [Boström et al., 2010], the purpose was to enable functional safety requirements verification based on contracts. In difference, in our approach Simulink models are translated to SDFGs (with details relevant to timings such as tokens' number and sizes) to enable a RT analysis method of applications which implement these models.

3.2.2 Virtual-Platform-in-the-loop Simulation

Contribution C3-3 adds some novel issues to related work. In order to highlight this, we are going to give an excerpt of the research done, mainly focusing on model-based design flow with support for co-simulation of Simulink and virtual-hardware platform frameworks such as SystemC (partly taken from our prior work [Fakih and Grüttner, 2012]) in the following.

Boland et al. [Boland et al., 2005] and Tomasena et al. [Tomasena et al., 2009] have also achieved (similar to our work) a co-simulation between SystemC and Simulink using Matlab's engine interface functions based on a synchronization at fixed time intervals. In [Boland et al., 2005] the intention was to reuse test-cases and golden models in Simulink to verify refined hardware components in SystemC. The approach in [Tomasena et al., 2009] employs a description of the architecture of the system as a SystemC transaction level model and a description of the algorithm in Matlab. During co-simulation, the SystemC architectural elements use Matlab's engine to execute and synchronize with the Matlab model.

Bouchhima et al. [F. Bouchhima, 2005, Bouchhima et al., 2006] presented a tool (CODIS) which enables the co-simulation of continuous environment models (e.g. in Simulink) with discrete models (e.g. in SystemC). This tool supports a more efficient co-simulation than that of [Boland et al., 2005] since it is based on SystemC's event driven scheduling mechanisms.

While being similar to our bi-simulation concepts, above research [Boland et al., 2005, Tomasena et al., 2009, F. Bouchhima, 2005, Bouchhima et al., 2006] assumes that the virtual-hardware platform model is developed independent from the Simulink model and no code-generation support is done.

[Kai Hylla, 2008, Mendoza et al., 2011] presented another approach as the above work, where Simulink is the master of co-simulation and SystemC modules are embedded into Simulink using S-functions. While these approaches are sufficient for functional verification, they lack the ability of validating non-functional properties such as timing behavior of of the target application on the target hardware.

Huang et al. [Huang et al., 2009] presented a Simulink-based heterogeneous MPSoC design flow for mixed hardware/software refinement and simulation. In their design flow the Simulink model is refined manually to achieve a Simulink CAAM (Combined Algorithm and Architecture Model). This offers advantages in terms of modular code generation and fast simulation of the refined system. In their work, it is not obvious how the partitioning in the CAAM model was done without considering explicit knowledge of the timing properties after mapping it on the MPSoC platform. So our virtual-platform-in-the-loop simulation with the Simulink plant model concept would be complementary for this work towards achieving this.

In [Bartolini et al., 2010] Bartolini et al. presented a co-simulation between SystemC and Simulink with the focus on exploring power, thermal and reliability management control strategies in high-performance multicores. The difference to our approach is that the controller in Simulink has been coupled with a model of the plant (environment) represented in SIMICs (a virtual-hardware platform framework) while we did the opposite by coupling the controller code executed on the virtual-hardware platform with the Simulink plant model.

Cha et al. [Cha and Kim, 2011] proposed an automatic synthesis of real-time multicore systems based on Simulink applications. In difference to our work, no virtual-platform-in-the-loop, but a traditional HIL (Hardware In the Loop) approach directly evaluating the implementation on a hardware platforms has been performed. Moreover, communication and synchronization overheads/times between control blocks mapped on different cores were not considered.

Mühleis et al. [Mühleis et al., 2011] presented a co-simulation between a SystemC and a Simulink model for a control algorithm performance evaluation, which similar to our approach they consider the VP framework as the simulation master and synchronization between Simulink are done at each period. As proposed in our approach, their model is also automatically compiled from Simulink into an executable for the virtual-hardware platform. A more recent publications which is based on the same concept above was published by the same co-authors in [Glass et al., 2012]. Yet in their approach the delay caused by the communication and synchronization between different (dependent) control tasks mapped on different processing elements was not considered. In [Zhang et al., 2013] Zhang et al. built upon the previous two publica-

tions and enabled an automatic conversion of Simulink model to an actor-based executable specification (modeled in SystemMoC [Falk et al., 2005]).

Based on our approach Poppen et al. [Poppen, F. and Grüttner, K., 2012] also presented a co-simulation approach of Simulink and OVP (Open Virtual Platform). While our approach uses a lock-step based bi-simulation, their approach uses a fully parallel execution of both simulators. The advantage of our approach is its full accuracy where changes in one simulation will be available at any time (without any loss) to the other, with some performance degradation compared to a parallel execution scheme of both simulators.

With respect to the above related work our proposed approach (see Sect. 6.4) benefits from the co-simulation of a flexible and well accessible virtual-hardware platform with a generic *non-invasive* timing measurement concept with a timing accuracy up to a cycle-accurate level. Our VPIL approach also enables reuse of test-cases and bi-simulation with golden models for the functional verification and validation of the stepping and timing requirements of the control algorithm implemented on a dedicated MPSoC.

3.2.3 Discussion

In literature there are many design methodologies which tries to combine RT analysis methods, synthesis and automatic code-generation for applications captured in a data-flow MoC in one model-based design flow [Gajski et al., 2009, Gerstlauer et al., 2009]. In [Büker, 2013] authors presented a design space exploration (DSE) design flow, which also takes Simulink as an entry model but different to our work translates the Simulink model to function networks (see Sect. 3.1.2.1), in order to perform a state-based RT analysis. In addition, to the different modeling approach of the state-based RT analysis, in their design flow they didn't consider combining the formal RT analysis with a simulative-based RT analysis as we have done (see Chap. 6). In another recent work Rosvall et al. [Rosvall and Sander, 2014] also presented a DSE design flow, which similar to our approach uses SDFGs as a formal model and tries to perform a DSE of SDFGs mappings to predictable MPSoCs (with guaranteed Quality of Service: QoS) based on a declarative style constraint programming approach. In difference to our work they assume a fully predictable MPSoC (no contention with TDM bus-based MPSoC) with no support for Simulink models in their design flow.

In addition to the above mentioned novel aspects of our Simulink to SDFG translation technique and the VPIL simulation technique compared to related work, to the best of our knowledge, no other work combined the concepts of simulative and formal RT analysis of Simulink models through translation to SDF graphs in a model-based design flow as done in this thesis (see Chap. 6).

3.3 Summary

In this chapter we took a look at an excerpt of the most relevant literature concerning formal RT analysis of RT applications running on MPSoCs.

Concerning analytical methods and despite their advantages of being scalable, these methods abstract from state-based modus operandi of the system under analysis (such as complex state-based arbitration protocols or inter-processor communication task dependencies) which leads to pessimistic over-approximated results compared to state-based RT methods [Huber and Schoeberl, 2009, Perathoner et al., 2009]. In addition, complex properties such as reachability of certain states cannot be verified by such methods. Due to this fact, we will study in this thesis the application of state-based RT analysis methods on SDF applications running on MPSoCs.

We have shown that, to the best of our knowledge, we pioneered the translation of SDFGs to timed-automata (in [Fakih et al., 2013a]) and presented recent researches which built upon our approach. To the best of our knowledge, no other approach uses model-checking (see claim C2 in Chap. 1) for the timing validation of multiple hard real-time SDFGs on a MPSoC platform, considering the contention on shared on-chip communication resources (buses, DMA), while supporting different commonly-used arbitration protocols like First Come First Serve (FCFS), Round-Robin (RR), Fixed-Priority (FP) and Time Division Multiple Access (TDMA).

Finally, an excerpt of related work was given in this chapter to conclude that our model-based design flow is novel in the aspect of combining a simulative and a formal RT analysis of embedded applications modeled in Simulink (with the help of a translation to SDFGs) and deployed on MPSoCs.

Chapter 4

System Model Constraints and Definition

One of the contributions (see C1) claimed in this thesis, is the construction of a (predictable) system model which enables a state-based RT analysis. At this level, two main challenges should be handled. The first challenge is to constrain the system under analysis to make it more predictable i.e. the constraints should contribute to the fact that the state space of the abstracted system model should still be manageable by the suggested RT analysis method. At the same time, these constraints must still be practical and applicable for a real implementation. Of course when taking these decisions, trade-offs between efficiency, scalability, analyzability and expressibility of the system model should be considered. In Chap. 1 we defined the main problem and came to the hypothesis (in Sect. 1.2) that constraining applications to the Synchronous dataflow (SDF) MoC and constraining the hardware platform to an MPSoC architecture with shared communication resources (as we will describe in Sect. 4.1.2) should circumvent the scalability issues faced by previous state-based RT analysis approaches (c.f. Sect. 3.1.2.1). In this chapter, we will first describe and justify the set of constraints made on the considered system model based on the timing issues presented in the Chap. 2, which will enable our state-based RT analysis. Next, a formal notation is given to describe in a precise and unambiguous way, the main modeling primitives of the considered system model in this thesis, beginning with the application model (MoC: Model of Computation), the architectural model (MoA: Model of Architecture), the synthesis decisions and the performance metrics in the model (MoP: Model of Performance). These model elements will be the basic entry to our state-based RT analysis method (described in Chap. 5).

4.1 System Constraints enabling State-based RT Analysis

Uncertainties which make a timing analysis of a system difficult are induced either through the environment, through the application model or through the hardware on which this application is run. In the following, we will describe the constraints imposed on our considered system, which improve its predictability and enable our RT timing analysis.

4.1.1 Task Model and Interaction with Environment

Following assumptions are imposed on the task model in this work:

A1 Applications are restricted to the SDFG Model of Computation (MoC) where each SDFG possesses a unique *source* actor and a unique *sink* actor (relevant for the model-checking some timing properties see Sect. 7.1). If this is not the case, *pseudo* actors representing source/sink actors with zero execution time could be inserted to the SDFG as suggested by [Lin et al., 2011]. Additionally, all running SDFGs are independent and known at design time.

A2 External events if existent (in case an SDFG is sensitive to environment) are considered periodic. If the SDFG is not sensitive to the environment, then an infinite stream of input data is expected (for source actors) such that the SDFG perform infinitely often which is typical for signal processing applications (see [Lee and Messerschmitt, 1987a]).

A3 Within an SDFG a *static-order* schedule is assumed, among several SDFGs *non-preemptive* scheduling algorithms (Static order, Round-Robin, TDMA) can be used.

A4 If an SDFG actor blocks on the FIFO buffer, then a *polling*-based IPC synchronization is used.

As already stated (assumption A1), we consider a more restricted application model than a generic task model, called Synchronous Data Flow (SDF), in order to improve the predictability of our system. The SDF MoC enables a compositional timing analysis, which allows analyzing computation (execution times of actors) independent from the communication timings (inter-actor communication). Furthermore, assuming that all SDFGs are known at design time is typical for safety-critical applications with hard real-time requirements and helps improving predictability. In addition, dependencies between different SDFGs are in general a possible extension to our system model but could lead to more complex system model which could become an issue for our state-based RT analysis method.

Several (*signal processing* and *control*) applications require input data which are provided through an external source like the A/D converter. This interaction with the environment can have an impact on the timing of the application and will be considered in this work assuming that this external source has a periodic behavior (assumption A2).

Concerning scheduling of SDFGs (assumption A3), we only consider non-preemptive schedulers (c.f. Sect. 2.2.1.1) since analyzing preemptive schedulers in our state-based RT analysis will lead to (possibly unmanageable) large state space. We differentiate between scheduling *within* an SDFG and scheduling *among* SDFGs (which will be denoted by hierarchical scheduling). Several scheduling strategies were presented in [Stuijk, 2007](c.f. Sect. 2.2.1.1), among them static-order scheduling which was evaluated to be more appropriate for hard-real time SDFGs. In this method, actors belonging to the same SDFG are executed in a fixed order which causes a small run-time overhead and improves predictability compared to other strategies [Stuijk, 2007]. In this work, we assume non-preemptive static-order scheduling technique for actors belonging to the same SDFG. If one SDFG (according to a hierarchical scheduling method) is activated, the first actor (according to a static-ordered list) is granted the processing resource and starts its execution by checking for firing conditions (if their is enough space/tokens in the buffer). If these conditions are satisfied, the actor immediately fires while other actors in the same SDFG are blocked waiting for their turn. Concerning hierarchical scheduling, different non-preemptive scheduling strategies (static order, round-robin and TDMA) are supported in this work due to their predictability features compared to preemptive ones. In the case of TDMA hierarchical scheduling (also referred to in this thesis by *TDMA clusters' scheduling* see Sect. 5.4.3), we assume a non-preemptive TDMA scheduling mechanism where the execution of SDFG starts only if it can be assured that the execution time of the SDFG doesn't exceed the chosen TDMA slot size.

When an actor blocks while accessing a shared FIFO buffer (assumption A4), an IPC synchronization mechanism (either implemented using *polling* or *interrupts* see Sect. 2.3.5) is needed for synchronization. In this work, we will use a *polling*-based mechanism since we mainly consider shared memories as storage resources in our experiments which typically don't support interrupts. In addition, we assume that the polling waiting time in the experiments can be bounded to a fixed value, even though our method is able to support lower/upper bounds of polling waiting time as in [Nelson et al., 2010].

Note that by a given SDFG, mature static analysis methods (such as the one presented in [Stuijk, 2007]) can be utilized to obtain a consistent repetition vector (see Def. 4.2.4), the number of initial tokens on all edges (see Def. 4.2.3) and the sizes of FIFO buffers which realize the asynchronous communication be-

tween actors. One known method which finds the smallest buffer size without leading the SDFG to deadlock is described in [Geilen et al., 2005]. Alternative methods which try to compute optimal buffers sizes depending on a throughput requirement, such as those implemented in SDF^3 [Stuijk et al., 2006], can also be used. One advantage of the state-based RT method presented in this thesis, is that it enables to check whether or not the considered system runs into deadlock, for e.g. in case of non-valid buffer sizes.

4.1.2 MPSoC Hardware Architecture

The following constraints are imposed on the hardware architecture in order to improve its predictability and in order to enable our state-based RT analysis method:

A5 We consider fully synchronous Multiprocessor System On Chip (MPSoC).

A6 Every processing element PE (in a tile as in Fig. 4.2) has two local disjoint memories for instruction (IM) and data (DM). Actors' private instructions and data are mapped to these private memories. Moreover, if two connected actors are mapped to the same tile then the message passing (FIFO buffer) between the two actors can also be mapped to the private memory. Furthermore, we assume that the PEs do not use (shared) caches.

A7 Actors of SDFGs mapped to different PEs communicate via buffers implemented in the shared memory. Only explicit communication (message passing) between actors mapped to different PEs will be visible on the interconnect and the shared memory.

A8 We mainly use buses as interconnects to connect the tiles to shared storage resources. Bus pipelining and transaction splitting (c.f. [Schaumont, 2013]) are not supported.

A9 It is assumed that the interconnect supports only non-preemptive arbiters. This means that if one PE gets access to the interconnect, it will block the interconnect for all other initiators even if those have higher priorities until the entire packet has been transported.

A10 Interconnect bridges are considered in this thesis to be *one-way* directional component (such as the AHB to APB bridge [ARM, 2006, ICVerification , 2015]) connecting two interconnects where it acts as a slave at the input, and as a master at the output. As a result, no contention is allowed on bridges. Also, no contention is allowed on I/O devices (or addressable devices).

While fully synchronous MPSoCs are still state of the art (A5), other approaches such as GALS (Globally Asynchronous Locally Synchronous) are still not widespread [Tatenguem et al., 2011].

A6 is in compliance with recent research recommendations on how to design predictable MPSoCs (c.f. [Wilhelm et al., 2008, Cullmann et al., 2010, Kotaba et al., 2013]). Here we assume that every processor has its own private instruction memory and data memory, such that no contention is induced on the same memory port when fetching instructions and data. This enables usage of state-of-the-art WCET analysis tools for single processors and improves the predictability of the considered system. Nevertheless, the above constraint leads to a limitation concerning the size of private memories particularly for large applications which we will discuss in Sect. 8.1. In this work, we did not use caches but predictable caches, such as scratchpads were utilized. Yet, local (tilewise) caches can be supported in our approach as long as the WCET analysis tool supports their replacement policies (c.f. [Cullmann et al., 2010]). If caches are supported, it should be guaranteed in the implementation that accesses to the shared resource are not cached while all private memory accesses can be cached. This is important to avoid cache coherence timing penalties which are difficult to assess in our approach.

Assumption A7 was made to enable a composable RT-analysis of considered system, being able to analyze the computation times of actors in isolation from their inter-processor communication time.

We will examine bus-based MPSoCs in the experiments (see Chap. 7) of this thesis (A8). Nevertheless, extending the interconnect model towards other topologies (c.f. [Schaumont, 2013]) such as Network on Chips (NoCs) should be straight forward and should be evaluated in future work since burst-transfer modes are already supported by our approach. Yet, NoCs are only meaningful for large number of tiles which can be connected through it. *Pipelining* and *transaction splitting* (c.f. [Schaumont, 2013]) were developed for high-speed buses in order to improve the speed of bus transfers which are limited by the sequential phases. These procedures are not supported in our bus model, since modeling their functionality will require a detailed knowledge of the bus protocol from which we abstract away in our modeling (see Sect. 4.2.5). In case upper and lower bounds can be derived from the communication protocol when utilizing pipelining and transaction splitting, then communication resources with such optimization techniques can be supported by our approach.

We support synchronous IPC (i.e. synchronous resource accesses realized through non-preemptive arbiters A9) in our approach since the RT analysis of synchronous resource accesses is much more challenging than that of the asynchronous case for which already mature RT analysis methods exist [Giannopoulou et al., 2012]. This is due to the fact that in the synchronous

case, once an access is granted for a requesting master on a shared resource, this access cannot be preempted and other requests are stalled which leads to variable and difficult to predict delays [Giannopoulou et al., 2012]. Another reason for this decision is that analyzing preemptive arbiters in our state-based RT analysis will lead to large state space.

In this thesis, we will explore two kinds of IPC styles: *burst* and *single-beat* IPC. While single-beat IPC, in which arbitration is done after every bus-width size transfer on the interconnect, leads to a complex system state space, supporting burst transfers on the interconnect helps reducing the state space being explored by our model-checking based method and improving its scalability as we will show in Chap. 7. If the used interconnect, does not support a burst mode, a DMA component can be utilized to realize a burst transfer (see Sect. 4.2.2).

For the same reason (keeping the state space of system model manageable) as above, contention is not allowed on bridges connecting two buses (A10) or on I/O devices in our system model. In the case of I/O devices, it is assumed that only one dedicated tile (I/O tile) is allowed to communicate with I/O devices which is a typical decision in real-life implementations.

In [Cullmann et al., 2010] the authors gave a classification of architectures w.r.t predictability considerations in the design of multicores: *fully timing compositional*, *compositional with bounded effects* and *non-compositional platforms* (see Sect. 2.3.6). Our approach can support the analysis of the first two types of architectures, as timing anomalies are considered at the application and the hardware level through considering upper and lower bounds on the actors execution times and on the communication and storage resources latencies (as we will see in Sect. 4.2.5). The third kind of architectures (*non-compositional*) exhibit both timing anomalies and domino effects, which require a very accurate and detailed model and an RT analysis method which has to follow all paths. We assume that the hardware architecture of our SUA doesn't exhibit domino effects.

Since we will use WCET analyzers to obtain lower/upper bounds on the execution times of single tasks running on single-processor platforms in our design flow (see Sect. 4.2.3), the SUA should also adhere to the typical constraints imposed by WCET analyzers. An excerpt of these constraints can be found in [Ferdinand and Heckmann, 2004] for the case of utilizing the aiT WCET analyzer. When analyzing these constraints, we found out that these constraints[1], at least those making the usage of aiT WCET possible, are already covered by our constraints (A1 to A10).

[1]For aiT usage it is assumed that no threads/parallelism or waiting for external events within a task is allowed. Also effects of interrupts, IO or timers are not considered [Ferdinand and Heckmann, 2004].

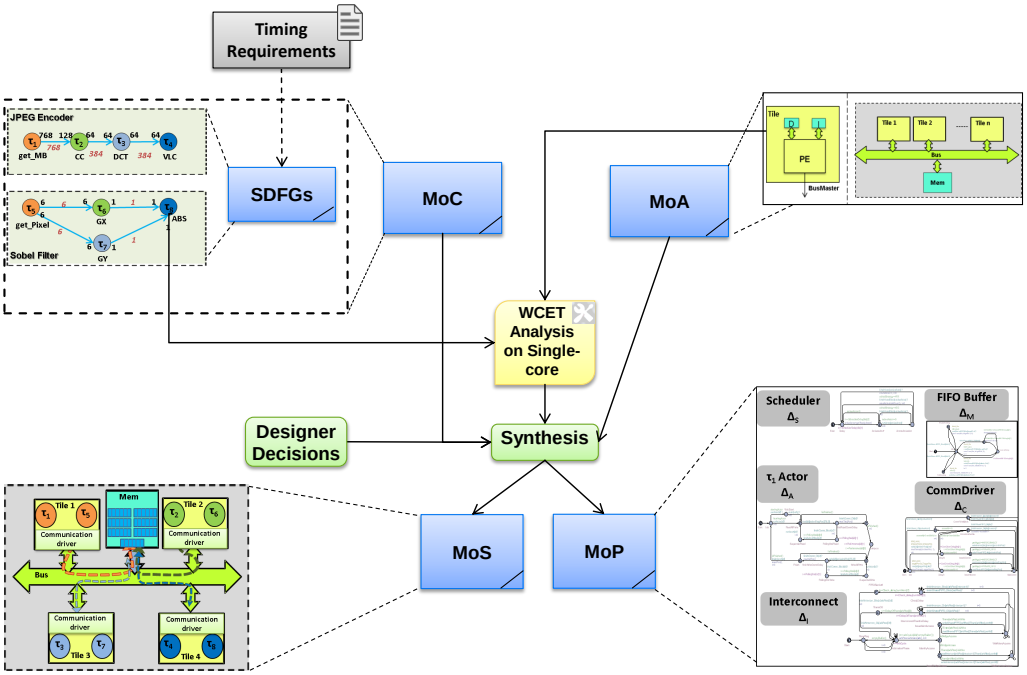


Figure 4.1: Extended X-Chart (based on [Gerstlauer et al., 2009])

4.2 System Model Definition

After elaborating on the constraints made on the system model for enabling our state-based RT analysis method, we will define our conceptual system model in this section based on the X-Chart [Gerstlauer et al., 2009]. Fig. 4.1 shows an extended X-Chart (based on the one in [Gerstlauer et al., 2009], also see Sect. 2.1) which describes the typical synthesis process of an embedded system. The synthesis process takes as a first input a set of behavior models with their specific timing requirements, each implemented in the SDF model of computation (MoC). The second input comprises resource constraints on the target architecture (MoA: Model of Architecture). We extended the X-Chart by adding a third input to the synthesis process, representing the results of the WCET analysis of the computation phase of every actor (see Sect. 4.2.3). The WCET analysis is performed (for every PE in isolation using available WCET analyzer tools) for all combinations of actors and available processing elements of the platform. This extension helps the system designer making binding decisions depending on the worst-case execution times. In addition, the WCET bounds will be needed for constructing the Model of Performance (MoP) (see Sect. 4.2.5). The output of the synthesis process is a structural model (MoS: Model of Structure) and a MoP. The MoS holds information about the realization of design decisions from the synthesis step, whereas the MoP, in addition to that, must

also keep track of all possible timing delays in the system that serves as input for our RT analysis method. In the following, we will use a formal notation (inspired from [Sriram and Bhattacharyya, 2000, Stuijk, 2007]) to describe in a precise and unambiguous way, the main modeling primitives and decisions of the synthesis process in Fig. 4.1. All definitions and terms of the system model are based on the X-Chart based synthesis process (defined and described in [Gerstlauer et al., 2009]).

4.2.1 MoC: Synchronous Data-flow Graphs

The formal semantics (inspired from [Sriram and Bhattacharyya, 2000, Stuijk, 2007]) of an SDFG consisting of a number of actors, edges and ports are defined as follows:

Definition 4.2.1. *(Port)* A Port is a tuple $P = (Dir, Rate)$ where $Dir \in \{I, O\}$ defines whether P is an input or an output port, and $Rate \in \mathbb{N}_{>0}$ which specifies the number of tokens consumed/produced by every port when the corresponding actor fires.

Definition 4.2.2. *(Actor)* An actor is a tuple $A = (\mathcal{P}, F)$ consisting of a finite set $\mathcal{P}$ of ports P, and F a label, representing the functionality of the actor.

Definition 4.2.3. *(SDFG)* An SDFG is a tuple $SDFG = (\mathcal{A}, \mathcal{D})$ consisting of a finite set $\mathcal{A}$ of actors A and a finite set $\mathcal{D}$ of dependency edges D. An edge $D \in \mathcal{D}$ is represented as a triple $D = (Src, Dst, Del)$ where the source (Src) of a dependency edge is an output port of some actor, the destination (Dst) is an input port of some actor, and $Del \in \mathbb{N}_0$ is the number of initial tokens (also called delay) of an edge. Every source and destination ports of all actors are connected to a unique edge, and all edges are connected to ports of some actor.

Definition 4.2.4. *(Repetition vector)* A repetition vector of an SDFG is defined as the vector specifying the activation number of every actor in the SDFG such that the initial state of the graph is obtained. Formally, a repetition vector (see Sect. 2.2.1.1) of an SDFG is a function $\gamma : A \rightarrow \mathbb{N}_0$ so that for every edge $(p, q) \in \mathcal{D}$ from $a \in \mathcal{A}$ to $b \in \mathcal{A}$, $p.Rate \times \gamma(a) = q.Rate \times \gamma(b)$. A repetition vector γ is called non-trivial if and only if for all $a \in \mathcal{A} : \gamma(a) > 0$. If the repetition vector of an SDFG is non-trivial then the SDFG is said to be *consistent*. In this thesis, we use the term *repetition vector* to express the smallest non-trivial repetition vector.

We assume in this thesis that incoming event triggers (if existent) from the system environment respect the event streams semantics [Thiele et al., 2000]. In the case where an SDFG is sensitive to an external event source, this event trigger is defined as follows:

Definition 4.2.5. *(Event trigger)* Formally, an event trigger $E = (p, j)$ is characterized by a period p and a jitter j, where $p, j \in N_{\geq 0}$ and $j \leq p$ [Hendriks and Verhoef, 2006]. Similar to [Hendriks and Verhoef, 2006], the jitter must be smaller than or equal to the period and thus avoiding bursty events with overlapping subsequent intervals (c.f. [Hendriks and Verhoef, 2006]).

4.2.2 Model of Architecture (MoA)

In the following, we describe the formal definitions of the considered MoA consisting of number of tiles, shared interconnects and storage resources:

Definition 4.2.6. *(Tile)* A tile is a tuple $T = (PE, M_p)$ with processing element $PE = (PE_{type}, f)$ where PE_{type} is the type of the processor and f is its clock frequency, and $M_p = (m_i, m_d)$ where $m_i, m_d \in \mathbb{N}_{>0}$ are the instruction and data memory sizes (in bits) respectively.

Definition 4.2.7. *(Execution Platform)* An execution platform is defined as $EP = (\mathcal{T}, \mathcal{I}, \mathcal{I}_{\text{DMA}}, \mathcal{B}_{\text{dg}}, \mathcal{M}_{\mathcal{S}}, \mathbf{C})$ consisting of

1. a finite set $\mathcal{T}$ of tiles T,
2. a finite set $\mathcal{I}$ of shared interconnects $I = (B_i, AP, CS)$ with B_i being the bandwidth in bits/cycle, AP is the arbitration protocol (FCFS, Fixed-Priority, Round-Robin, TDMA) and $CS = \{SingleBeat, Burst\}$ is the communication style supported by the interconnect.
3. a finite set $\mathcal{I}_{\text{DMA}}$ of shared DMA controllers $I_{\text{DMA}} = (B_{DMA}, AP_{DMA})$ with B_{DMA} being the bandwidth in bits/cycle, AP_{DMA} is the arbitration protocol of the DMA,
4. a finite set $\mathcal{B}_{\text{dg}}$ of bridges each having a bandwidth of B_{bridge} in bits/cycle.
5. a finite set $\mathcal{M}_{\mathcal{S}}$ of shared storage resources (such as memories or I/O slaves) $M_S = (B_s, m_s)$, each of them having specific size m_s in bits and a bandwidth B_s in bits/cycle,
6. a configuration $\mathbf{C} = (\zeta_{T,M}, \zeta_{T,\text{DMA}}, \zeta_{\text{DMA}}, \zeta_{bridge})$ with the functions:
   - $\zeta_{T,M}: T \rightarrow I \times M_S$ which maps a tile T to the interconnect I of the shared storage resource M_S,
   - $\zeta_{T,\text{DMA}}: T \rightarrow I \times I_{\text{DMA}}$ which maps a tile T to the interconnect I of a DMA controller I_{DMA} allowing its configuration via this interconnect,
   - $\zeta_{\text{DMA}}: I_{\text{DMA}} \rightarrow T \times I \times M_S$ which maps a DMA controller I_{DMA} to access the private memory of tile T and move data to the shared storage resource M_S via interconnect I,

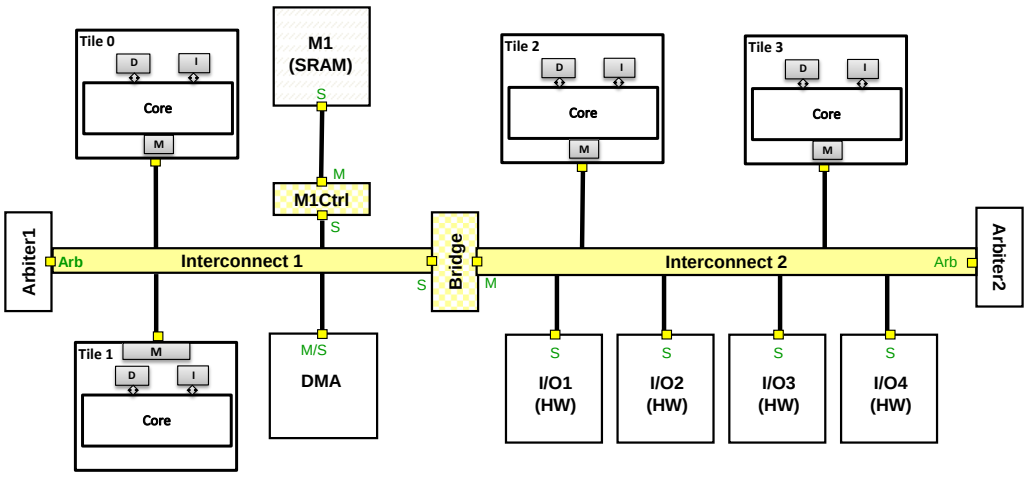


Figure 4.2: Possible analyzable MPSoC architecture in this thesis (based on [Gerstlauer, 2009])

- and a configuration $\zeta_{bridge} = I \rightarrow B_{\text{dg}} \times I$ which (if required) maps interconnect $I_1 \in \mathcal{I}$ to the bridge which connects it to another interconnect $I_2 \in \mathcal{I}$.

Various configurations of the MPSoC can be supported (see definition of **C** in Def. 4.2.7). With the help of configuration $\zeta_{T,M}$, a tile connection to a shared storage resource (such as an I/O device or a memory) through a specific interconnect can be described. In the case a burst transfer is needed and the interconnect does not support a burst mode, a DMA component can be utilized to realize a burst transfer. For this, the initiator tile should first configure the DMA with transfer parameters (which is realized through configuration $\zeta_{T,\text{DMA}}$). After that the DMA controller transfers the requested data (see ζ_{DMA}) without interruption from the private memory of one tile via an interconnect and updating the shared storage resource (where the shared FIFO buffer is mapped). An example configuration for the Aurix MPSoC is found in Sect. 7.4.3.

Fig. 4.2 shows a possible architecture which can be captured by our MoA, consisting of multiple tiles, various storage resources and multiple interconnects each with various arbitration complexities and communication styles.

4.2.3 BCET/WCET Analysis on Single-Processor Platforms

The problem of obtaining lower/upper bounds on the execution times of single tasks running on single-processor platforms (with fairly complex processors) is considered to be solved [Wilhelm et al., 2008]. Due to this fact, we can utilize any of the available WCET analyzer (aiT [Ferdinand and Heckmann, 2004], chronos [Li et al., 2007] or SWEET [Lisper, 2014]) to statically obtain the

BCET/WCET bounds of software actors executed on a specific tile of the MP-SoC platform in isolation from the communication accesses on the communication resources. Fig. 4.3 shows the different execution phases of an SDF actor on a tile. The execution of an actor starts with the *read* phase. All tokens on all input channels required to enable the firing of an actor are read in this phase. Depending on the mapping of the channels to local or shared storage resource different communication driver routines are executed. When all tokens have been copied to the tile's local memory, the *computation* phase of the actor starts. In this phase data from the input tokens are transformed into data of the output tokens. During the computation phase of an actor, only local data is accessed (i.e. no accesses on shared resources occur during computation phase). The final *write* phase copies all output tokens either into local or shared storage resources depending on the channel's mapping. Communication driver routines are executed in the write phase, just as in the read phase. The communication driver is responsible for establishing the FIFO-style message passing between actors using private storage resource, shared storage resource or DMA connecting to a shared storage resource. After actor execution, a scheduler decides which actor is activated next. The scheduler realizes the static-order schedule within an SDFG and hierarchical scheduling among multiple SDFGs (described later in Sect. 4.2.4).

For using WCET analysis tools, binary code for the instruction-set architecture of the specific tile is required. For the compute phase of the actors executable code is needed. In the case the application is available as a Simulink model (as we will demonstrate in Chap. 6), C-code can be generated from it using a code-generator (e.g. via Simulink Coder [MathWorks, Inc., 2015a]). Using a cross-compiler for the tile's processing element, binary code to be used with the WCET tool is generated. In the generated code all loops are statically bound and recursion is avoided to enable the static BCET/WCET analysis. The WCET tool outputs an lower/upper bounds (number of cycles) for the compute phase of an actor. Only if the read and write accesses of an actor are local (not accessing the shared communication resource) the WCET can also give an

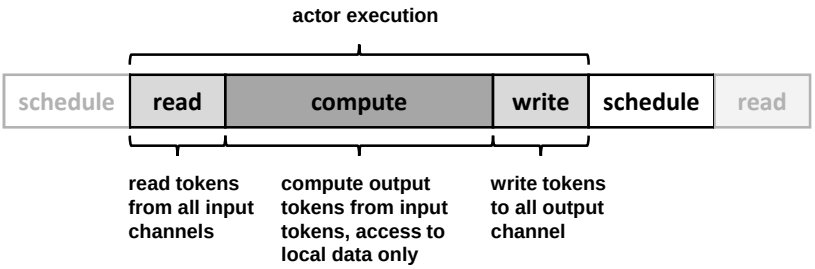


Figure 4.3: Execution phases of an SDF actor

upper/lower bounds for the write/read phases.

Platform dependent communication drivers for the read and write phases, as well as the scheduler are implemented manually. For the communication drivers, we differentiate between the software (communication stack) timing estimation and the communication timing/delay estimation. For the software part, upper and lower bounds on the execution time of the target binary code are estimated with the help of a WCET analysis tool. For the estimation of the variable communication delay, due to contention on a shared communication medium (e.g. a bus), either a clock-cycle accurate (based on data sheet information) analysis model (including the functionality of the arbiter) or a time-accurate model (c.f. Sect. 2.3.3.2) is used.

The estimated execution time lower/upper bounds are then passed over and annotated in the formal MoP representation (see Sect. 4.2.5), which is then used to configure the TA templates of the analysis framework described in Sect. 5.2 in order to validate the complete system against its real-time requirements.

4.2.4 Synthesis

The system synthesis (see Fig. 4.1) includes the processes of binding and scheduling the behavioral model on the defined architecture.

4.2.4.1 Binding Decisions

In the following definition, decisions made when mapping the SDFG(s) on the MoA are described:

Definition 4.2.8. *(Mapping)* If $\mathcal{A}$ is the set of actors of all SDFGs, $\mathcal{D}$ the set of all SDFG edges, $\mathcal{T}$ the set of all tiles of the platform configuration, $\mathcal{I}$ the set of all interconnects, $\mathcal{M}_\mathcal{S}$ the set of all shared storage resources, $\mathcal{M}_\mathcal{P}$ the set of all private memories, then a mapping can be defined as a tuple $M = (\alpha, \beta, \delta, \zeta)$ with

1. the function $\alpha : \mathcal{A} \rightarrow \mathcal{T}$ maps every actor to a unique tile[2] (multiple actors can be assigned to one tile). This function can also be constrained to prohibit the mapping of some actors to some tiles (enabling heterogeneous MPSoCs),

2. the function $\beta : \mathcal{D} \rightarrow (\mathcal{M}_\mathcal{P} \cup (\mathcal{I} \times \mathcal{M}_\mathcal{S}) \cup (\mathcal{I} \times \mathcal{I}_{\text{DMA}} \times \mathcal{I} \times \mathcal{M}_\mathcal{S}))$, where:

   - $\mathcal{M}_\mathcal{P}$: mapping to private memory,

[2]This means that *auto-concurrency* property of SDFGs is not supported. Auto-concurrency for an actor determines the number of multiple instances of that actor which can be executed concurrently on multiple processors.

- $(\mathcal{I}, \mathcal{M}_{\mathcal{S}})$: mapping to shared storage resource which can be accessed using an interconnect ($i \in \mathcal{I}$) which directly connects to it or through a set of transfers on multiple interconnects which lead to the target memory, where our model supports single-beat and burst transfer,
- $(\mathcal{I}, \mathcal{I}_{\text{DMA}}, \mathcal{I}, \mathcal{M}_{\mathcal{S}})$: using DMA as `memcopy` to realize a burst transfer. The DMA is configured through an interconnect ($i \in \mathcal{I}$) to perform a `memcopy` (twice: read/write from/to tile and write/read to/from shared storage resource) via another interconnect ($j \in \mathcal{I}$). E.g. in Sect. 7.4 the DMA configuration in the Aurix platform was done through the System Peripheral Bus (SPB) and the `memcopy` via System Resource Interconnect (SRI).

3. the function $\delta : \mathcal{D} \rightarrow \mathbb{N}_{\geq 0}$ which assigns for every edge ($d \in D$) the maximum number of tokens it can hold (buffer's size),

4. the function $\zeta : \mathcal{D} \rightarrow \mathbb{N}_{\geq 0}$ which assigns for every edge ($d \in D$) the token size attribute T_s (in bits).

The edge mapped to a private or to a shared storage resource represents a consumer-producer FIFO buffer in an actual implementation. In case the interconnect needs extra configuration (e.g. configuration of a DMA burst transfer), we assume that the configuration phase of an interconnect through one tile does not interact with the transfer phase of other tiles (e.g. see realization in Sect. 6.5).

4.2.4.2 Scheduling Decisions

The possible hierarchy of SDFGs and its scheduling configuration in the SUA is depicted in Fig. 4.4. In the first hierarchy level (leafs of the tree in Fig. 4.4), actors belonging to an SDFG are scheduled according to *self-timed (static-order)* schedule (SO), where these are executed in a cyclic manner according to statically ordered list, as soon as their input data is available. In the next hierarchy level, an *SDFG Scheduler* (see Fig. 4.4) defines the order (priority) of execution of multiple SDFGs when executed on the same tile. In a *static-order* SDFG scheduler (SO), all SDFGs assigned to a given tile are executed in a static cyclic order as soon as the input data is available. Yet, the non-preemptive static-order hierarchical schedule has the disadvantage that all actors running on one processor belonging to other graphs may be blocked, waiting for current active actor to fulfill its firing conditions (which could highly undesirable for overall system performance). In this case, a strategy like *round-robin* (RR) scheduler can help to achieve more fairness, by giving the scheduler an option to grant an actor a chance to check for input availability or output capability and then it either fires if these are satisfied or give the control back to the scheduler if this is not

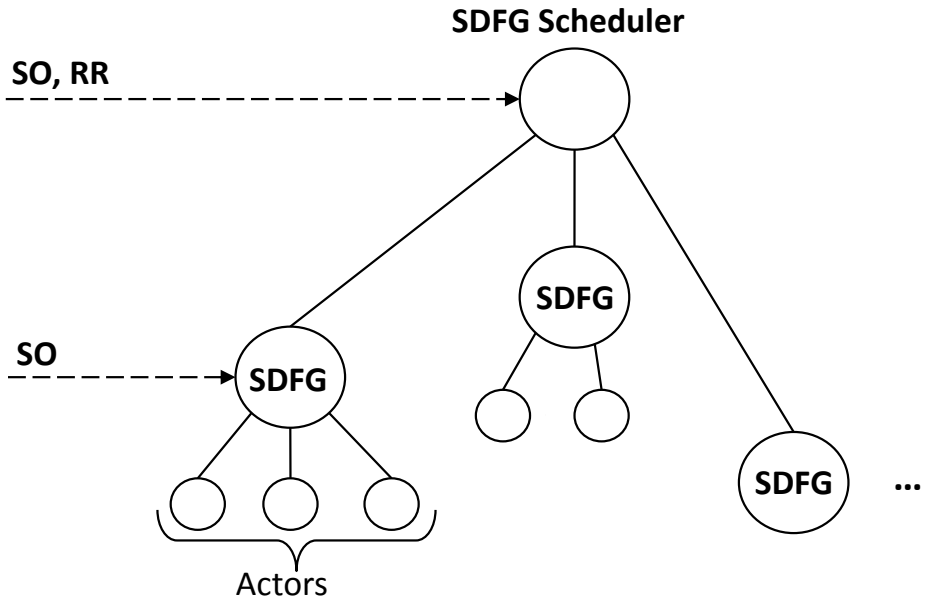


Figure 4.4: Scheduling hierarchy within an SDFG and among SDFGs

the case. In both cases of blocking or successful firing, the scheduler switches from the active actor to activate an actor belonging to other SDFG running on the same processing unit.

The following definitions allow us to express the scheduling behavior of multiple SDFGs mapped to tiles on the platform:

Definition 4.2.9. (*Self-timed (static-order) schedule*) For an SDFG with repetition vector γ, a static-order schedule SO is an ordered list of the actors (to be executed on some tile), where every actor a is included in this list $\gamma(a)$ times. Self-timed means that SDFGs are executed in a static cyclic order as soon as the input data is available.

Definition 4.2.10. *(Scheduling Assignment)* Let $\mathcal{SO}$ be the set of all SO schedules for all SDFGs considered in the system. A scheduling assignment is a function $S : \mathcal{T} \rightarrow \mathbf{so}$, which assigns to every tile $t \in \mathcal{T}$ a subset $\mathbf{so} \subseteq \mathcal{SO}$.

Definition 4.2.11. *(SDFG Scheduler)* An SDFG scheduler is a triple $S = (\mathbf{so}, F, Type)$ where $\mathbf{so} \subseteq \mathcal{SO}$ is the set of different SDFGs schedules assigned to one tile, F represents the functionality (code) of the scheduler and $Type$ is the SDFG scheduling type, defining the order (priority) of execution of independent lists of different SDFGs assigned to one tile according to an arbitration strategy (Static-Order, Round-Robin).

We will denote throughout this thesis, the resulting system with all above decisions made and for which a RT analysis should be made as System Under Analysis (SUA).

4.2.5 Model of Performance (MoP) Extraction

In order to verify that the performance of the SUA stays within the required lower/upper bounds, we must keep track of all possible timing delays of all SDFGs running on the MPSoC platform. To achieve this, a MoP is extracted after the synthesis process which includes all the SW/HW components with their properties influencing the timing in the considered system. This MoP will be the entry model to the state-based RT analysis method (see Chap. 5) and it comprises the following aspects (see Fig. 4.5):

1. The scheduler execution time which activates actors, and the scheduling mechanism utilized.

2. If the SDFG is sensitive to an external event trigger then the period of the trigger would also influence the timing (not depicted in Fig. 4.5),

3. Once activated, the actor undergoes a read phase, after that a compute phase and then a write phase (see execution phases in Fig. 4.3). As already described, for the compute phase an upper and lower bounds can be estimated through a WCET analyzer. In the read and write phase, a communication driver is triggered to communicate with either the shared or the private memory. In both cases, the communication driver execution time influences the overall timing. In addition to functional code, issues such as endianess and different wordlength (between the tile and the interconnect) handling [Schaumont, 2013] with their specific timing influence are captured by the communication driver component. Besides the binding aspects of actors to processor elements, also actor's properties such as number of ports, the ports' rates and the number of channels are relevant.

4. If a communication resource is requested for inter-processor communication between two actors, the communication protocol and the arbitration mechanism would influence the timing.

5. If an actor blocks on a shared FIFO buffer on the shared memory, then the IPC synchronization mechanism also influences the timing. In addition to buffer sizes of the FIFO buffers, the latency delays resulting from accessing both shared and private memories are also relevant. These also include delays resulting from different word-length handling mechanisms (refer to [Schaumont, 2013] for more details) between the interconnect and the storage resources.

We choose a *Bus-Functional-Model* (BFM) [Cai and Gajski, 2003] abstraction level (see Sect. 2.3.3.2) in order to model the interconnect. In this model, the

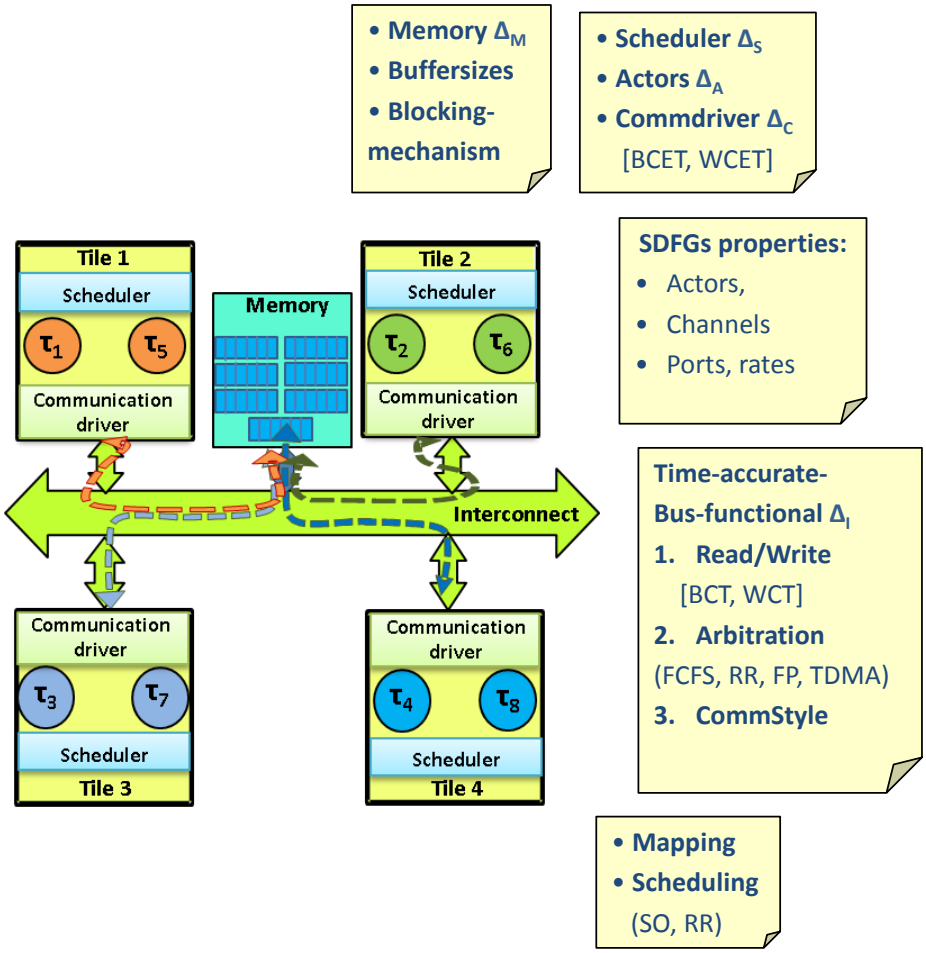


Figure 4.5: Timing issues in an example SUA

application layer issues read/write transactions on the interconnect (with arbitration) and the communication is considered either at a *cycle-accurate* level (as seen in Fig. 2.8, Fig. 2.9) or at a *time-accurate* level (see Fig. 2.10).

Since the time-accurate model (see Sect. 2.3.3.2) is the more abstract/general model, we will use it in this thesis for representing interconnects. However, in the case where we can accurately estimate the duration of a write/read access (at a cycle-accurate level) then adapting this to the generic model ca be easily done through setting the lower and upper bounds to be equal.

After synthesis, the system components *event triggers* (if required), SDFGs with their *actors* and *edges*, *schedulers*, *communication drivers*, *interconnects*, *private* and *shared storage resources* are annotated with execution times/delays according to the following definition:

Definition 4.2.12. (*Delay annotations*) If $\mathcal{E}$ is the set of event triggers (if existing), $\mathcal{SDFG}$ the set of SDFGs, $\mathcal{A}$ the set of actors, $\mathcal{D}$ the set of edges, $\mathcal{S}$ the set of

schedulers, $\mathcal{C}$ the set of communication drivers, $\mathbb{I} = \mathcal{I} \cup \mathcal{I}_{DMA}$ the set of interconnects, $\mathcal{M}_{\mathcal{S}}$ the set of shared storage resources, and $\mathcal{M}_{\mathcal{P}}$ the set of private memories, the following delay functions are defined:

- $\Delta_E : \mathcal{E} \times \mathcal{SDFG} \rightarrow \mathbb{N}_{>0} \times \mathbb{N}_{>0}$ and $\Delta_{E_1}(e) \leq \Delta_{E_2}(e)$ for each $e \in (\mathcal{E} \times \mathcal{SDFG})$ where $\Delta_E(e) = (\Delta_{E_1}(e), \Delta_{E_2}(e))$ which represents the delay interval $[p, p + j]$ for each event trigger invoking an SDFG (see Def. 4.2.5).

- $\Delta_A : \mathcal{A} \times \mathcal{T} \rightarrow \mathbb{N}_{>0} \times \mathbb{N}_{>0}$ and $\Delta_{A_1}(a) \leq \Delta_{A_2}(a)$ for each $a \in (\mathcal{A} \times \mathcal{T})$ where $\Delta_A(a) = (\Delta_{A_1}(a), \Delta_{A_2}(a))$ which provides an execution time interval $[BCET, WCET]$ for each actor representing the cycles needed to execute the actor behavior (compute phase) on the corresponding tile. This delay can be estimated using a static analyzer tool.

- $\Delta_S : \mathcal{S} \times \mathcal{T} \rightarrow \mathbb{N}_{>0} \times \mathbb{N}_{>0}$ and $\Delta_{S_1}(s) \leq \Delta_{S_2}(s)$ for each $s \in (\mathcal{S} \times \mathcal{T})$ where $\Delta_S(s) = (\Delta_{S_1}(s), \Delta_{S_2}(s))$. $\Delta_C : \mathcal{C} \times \mathcal{T} \rightarrow \mathbb{N}_{>0} \times \mathbb{N}_{>0}$ and $\Delta_{C_1}(c) \leq \Delta_{C_2}(c)$ for each $c \in (\mathcal{C} \times \mathcal{T})$ where $\Delta_C(c) = (\Delta_{C_1}(c), \Delta_{C_2}(c))$ represents, in analogy to $\Delta_A(a)$, a delay interval for every scheduler and communication driver which can be estimated same as Δ_A.

- $\Delta_D : \mathcal{D} \times (\mathcal{M}_{\mathcal{P}} \cup (\mathcal{I}, \mathcal{M}_{\mathcal{S}}) \cup (\mathcal{I}, \mathcal{I}_{\text{DMA}}, \mathcal{I}, \mathcal{M}_{\mathcal{S}})) \rightarrow \mathbb{N}_{>0} \times \mathbb{N}_{>0}$ and $\Delta_{D_1}(d) \leq \Delta_{D_2}(d)$ for each $d \in (\mathcal{D} \times (\mathcal{M}_{\mathcal{P}} \cup (\mathcal{I}, \mathcal{M}_{\mathcal{S}}) \cup (\mathcal{I}, \mathcal{I}_{\text{DMA}}, \mathcal{I}, \mathcal{M}_{\mathcal{S}})))$ where $\Delta_D(d) = (\Delta_{D_1}(d), \Delta_{D_2}(d))$. Δ_D assigns a delay interval to each communicating edge $d \in D$, mapped to a communication primitive, which depends on:

  1. the number and size of the tokens being transported,
  2. the type of transaction (read or write),
  3. Δ_I: latency (depending on arbitration, bandwidth of interconnect(s) and bridge(s) if existent) of the communication interconnect to transport current transaction and
  4. Δ_M: latency of the target storage resource ($\Delta_{\mathcal{M}_{\mathcal{S}}}$ or $\Delta_{\mathcal{M}_{\mathcal{P}}}$)

- $\Delta_P : \mathcal{A} \times \mathcal{T} \rightarrow \mathbb{N}_{>0} \times \mathbb{N}_{>0}$ and $\Delta_{P_1}(a) \leq \Delta_{P_2}(a)$ for each $a \in (\mathcal{A} \times \mathcal{T})$ where $\Delta_P(a) = (\Delta_{P_1}(a), \Delta_{P_2}(a))$ which provides a polling delay interval of $[BCPT; WCPT]$ which should be waited by an actor when blocking on a shared storage resource.

Obviously, as can be noted from above delay annotations, the predictability of some timing metric on the defined system model will suffer from the variability of the execution times [Kirner and Puschner, 2010] i.e the larger the interval difference of [BCET, WCET] is, the larger is the number of "guesses" of the execution times that should be explored which could impede the timing analysis.

After defining the possible delay annotations in our system model, we can now abstractly represent every tile by the actors mapped to it, its scheduler, and communication driver. Each of them with their delay annotations as defined above. Each of the tile's private memories and the shared storage resources can be abstracted as a set of (private/shared) FIFO buffers. The sizes of these FIFOs depend on the rates of the mapped edges (each edge is mapped to exactly one FIFO buffer) and the schedule of the involved actors.

4.3 Summary

In this chapter we suggested a system configuration imposing important constraints on the system under analysis (SUA) (e.g. prohibited saving the instruction code of an application in shared memories and allowed only message passing on interconnects). This configuration targeted a predictable system with a reasonable state space of its extracted performance model (which will be evaluated in Sect. 7.2). At the same time, this configuration remains realistic enough to be implemented on current industrial architectures (as we will show in the experiments Sect. 7.4). Moreover, we defined our system model with all decisions which can be made in a synthesis process using an unambiguous mathematical notation, which allowed us to describe the resulting performance model (MoP) where all timing delays in the system are captured. A summary of the decisions made for supported MPSoC components in our system model can be seen in Fig. 4.6 which is based on the timing issues in Fig. 2.7 identified and described in Chap. 2.

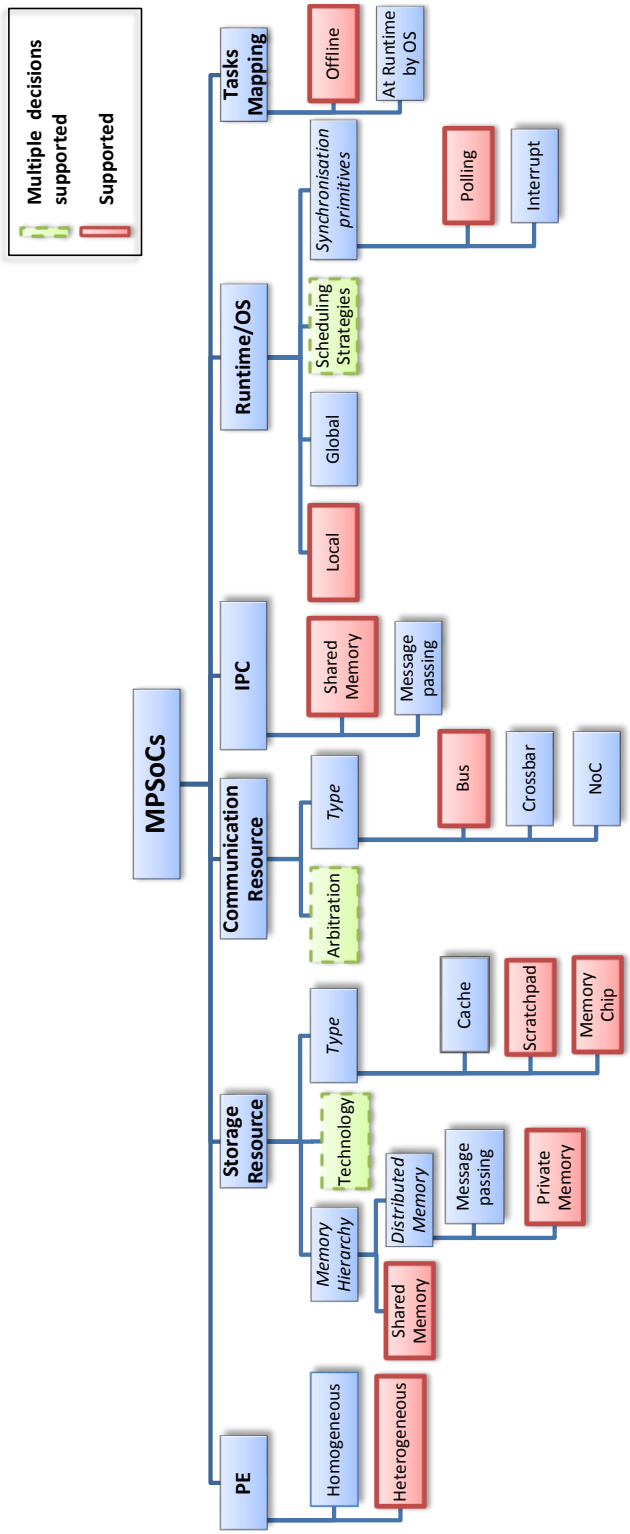


Figure 4.6: MPSoC supported primitives in this thesis

Chapter 5

State-based Real-time Analysis of SDFGs on MPSoCs

Having defined the Model of Performance (MoP) with all the necessary assumptions in the previous chapter, we now need a formalism which comprises the MPSoC components capturing their state-based functional and temporal behavior (including scheduling, arbitration etc.) and at the same time exhibiting parallelism. For this, we find the timed-automata formalism with its related model-checking capabilities which was investigated to be useful to model real-time systems (see Sect. 2.5.2.1), most appropriate. Since UPPAAL [Bengtsson and Yi, 2004] has grown to be one of the well-known, established and well-supported tools offering a highly optimized implementation for model-checking timed automata, we will be using UPPAAL in this work for modeling, simulation and verification the resulting networks of timed automata.

With the help of the MoP delays and abstractions defined in the last chapter, we will show in this chapter how to capture these delays in the form of timed-automata (TA) templates. Furthermore, we will elaborate on the TA templates' implementation and give a relation between every system model component and its relative TA template which captures its delay. Next, auxiliary observers' TA templates, which enables us with the help of UPPAAL model-checker to check timing metrics (such as end-to-end latency or period) are presented. Finally, complexity issues are discussed and methods to improve scalability of our state-based method are presented.

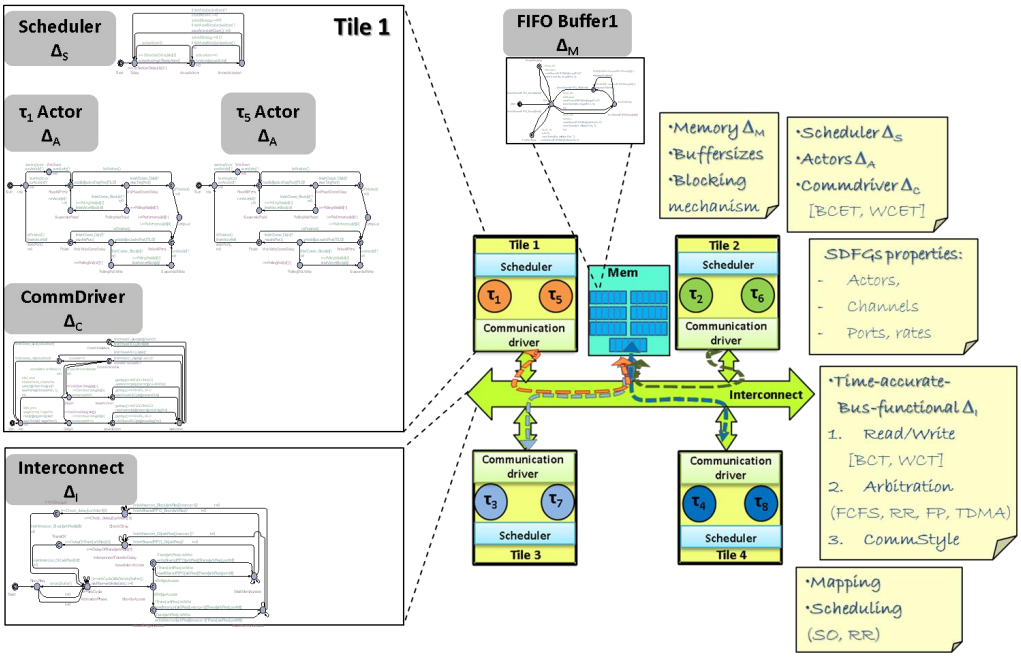


Figure 5.1: Example of capturing MoP of SUA as a network of TAs (c.f. Sect. 5.2)

5.1 Representing Performance Model as Timed Automata

In this section, we will describe how the components of the MoP from Sect. 4.2.5 can be formalized using the timed-automata semantics. We have chosen TA formalism since it is a well-established and intuitive formalism for modeling RT-systems and exhibiting parallel activities occurring within such systems. Moreover, there are also a lot of tools (such as UPPAAL[1]) which support the modeling, verification and simulation of TA.

As concluded in the last chapter, our system consists of an execution platform (MoA), an external event trigger (in case the application is sensitive to an external source) and a number of SDFGs. Every SDFG consists of a number of actors and channels. The execution platform consists of a number of tiles, interconnects and a number of shared FIFO buffers which abstractly represent the shared storage resources in the platform. Every tile can be abstractly represented by the actors mapped to it, its scheduler, its communication driver software components and a number of private FIFO buffers which abstractly represent the tile's private memory. Each of them with their delay annotations. The overall composition of the timed-automata templates representing the system components can be described as follows:

[1]UPPAAL 4.1.19 (rev. 5648), has been used in the experiments.

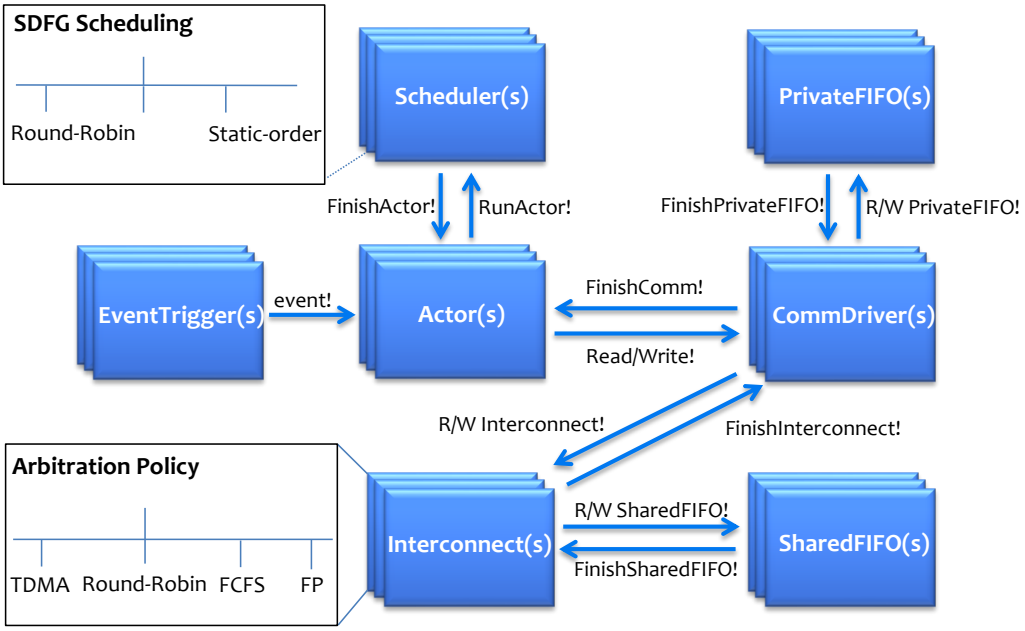


Figure 5.2: TA templates of MoP components with all their interactions

System = ExecutionPlatform$||_{h=1}^{g}$ EventTrigger$_h||_{i=1}^{q}$ SDFG$_i$
SDFG$_i$ = $\|_{j=1}^{r}$ Consumer$_j$ $||$ $\|_{k=1}^{s}$ Producer$_k$ $||$ $\|_{l=1}^{t}$ Transporter$_l$
Platform= $\|_{m=1}^{u}$ Tile$_m$ $||$ $\|_{n=1}^{v}$ Interconnect$_n$ $||$ $\|_{o=1}^{w}$SharedFIFO$_o$
Tile$_i$= Scheduler$_i$ $||$ CommDriver$_i$ $||$ $\|_{p=1}^{x}$ PrivateFIFO$_p$

where $||$ denotes parallel composition of timed automata, $g \geq 0$ is the number of event triggers (where every SDFG can be triggered by at most one event trigger), $q \geq 1$ is the number of SDFGs, r, s, t represent the number of actors distinguished according to their types `Consumer`, `Producer` and `Transporter` (where $r+s+t \geq 1$), $u \geq 1$ is the number of tiles, $v \geq 1$ is the number of shared interconnects (each representing, in this work, either a DMA or a shared bus), $w \geq 1$ is the number of shared FIFO buffers, and $x \geq 0$ is the number of private FIFO buffers. The edges properties (ports' rates, delay tokens, connections etc. c.f. Sect. A.2) of SDFGs, the mapping decisions, and other system configuration parameters (e.g. BCET/WCET of actors and buffer sizes) are implemented as global variables.

Fig. 5.1 shows an example of the MoP representing an (environment non-sensitive) SDFG application mapped to an MPSoC with all properties influencing the performance (mapping, scheduling, delays etc.). As seen every tile is represented by a number of TAs (16 TAs in total for all tiles in the example): one for the scheduler, one for the communication driver and one for every actor mapped on that tile (for *tile*1 in the example one TA for τ_1 and one TA for τ_5).

Furthermore, every FIFO buffer is represented by a TA (7 TAs for the example) and finally for every interconnect one TA is needed (one TA for the example).

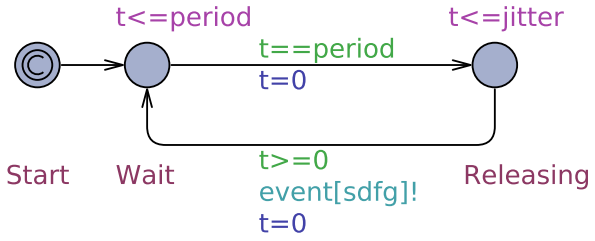
Fig. 5.2 depicts the interactions between the timed automata of different components of the MoP. The scheduler starts and activates the actors on each tile (via `RunActor`) according to its scheduling algorithm. If the actor is a producer (source) and the SDFG is sensitive to an external event source, it needs additionally to wait until it is notified by a periodic event (`event`) generated from an event trigger automaton. If an actor needs to communicate with another actor it issues a `Read/Write` signal to the communication driver which realizes the communication with the interconnects or the private memory (depending on the mapping). The interconnect arbitrates different requests from different tiles according to a specific arbitration mechanism and transports tokens either directly to the specific shared FIFO buffer or to other interconnect(s) (not depicted in Fig. 5.2). When the communication with the shared FIFO buffer is successfully finished, a `FinishSharedFIFO` signal is returned to the interconnect which forwards this notification to the communication driver. The communication driver acknowledges this event by sending a `FinishComm` signal to the actor. If the target buffer is blocked, it issues a `FinishSharedFIFO_Block` signal which is propagated by the interconnect to the communication driver and back to the actor which in turn waits for some time before it retries the communication (polling) (not depicted in Fig. 5.2 for clarity reasons c.f. Sect. 5.2).

5.2 Implementation of the Timed-automata Templates

After identifying the overall composition of timed automata needed to capture the components of the MoP, we will elaborate in the following on their templates' implementation in UPPAAL (for background information refer to Sect. 2.5.2.1).

5.2.1 Event Trigger Template

We introduce explicit event triggers (see Sect. 2.4) in our analysis model for modeling periodically triggered control applications, or signal processing applications which are sensitive to periodic external events. Independent SDFGs (if sensitive to external trigger) can be triggered through events of a corresponding event trigger. Event arrival models, e.g. the periodic arrival of sensor data are characterized by a period p, and a jitter j where $p, j \in N_{\geq 0}$. The implementation of the event trigger automaton is depicted in Fig. 5.3. The states of the event trigger alternate between `Start, Wait` and `Releasing`. After some period, the automaton changes its state from `Wait` to `Releasing`. In this state, it waits for some jitter delay (between `[0, jitter]`) and then activates



Identifier	Type	Comment
t	clock	Used for delay of period
period	int (Parameter)	Period of the event
jitter	int (Parameter)	Non-deterministic selection of a (real-valued) delay in the interval [0; *jitter*]
sdfg	int (Parameter)	Identification of the SDFG to be activated by the trigger
event	broadcast channel array	Global channel array used for signaling the arrival of event

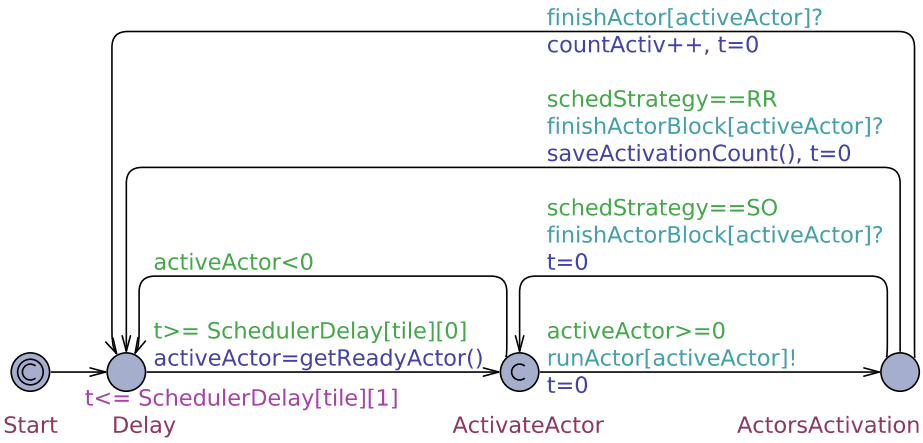
Figure 5.3: Template of periodic event trigger

the corresponding SDFG by releasing an event to its corresponding source actor. Next, the automaton waits for the same period to iterate the whole procedure again (see Δ_E in Def. 4.2.12).

In the case, the SDFGs are not sensitive to external events, no event trigger automaton is instantiated, and the attribute `startingActor` in all actors' implementations (see Sect. 5.2.3) is set to false.

5.2.2 SDFG Scheduler Template

Fig. 5.4 shows how the SDFG scheduler (c.f. Def. 4.2.11) is implemented (refer to a pseudo-code in Sect. 6.5.2) as a timed-automaton template. The states of the scheduler alternate between `Start`, `Delay`, `ActivateActor` and `ActorsActivation`. For every tile, a scheduler automaton must be instantiated which activates the actors mapped to it through an explicit signal (`runActor[id]`). This activation is done according to a specific activation strategy (static-order or round-Robin see Sect. 2.2.1.1). In case of an SO schedule (where the delay is negligible), the scheduler activates the actors according to a static ordered list given by the user, if some actor is blocked on some input/output this will block other actors on the same tile even though other actors of other SDFGs are ready to execute. If RR is chosen then fairness between different SDFGs is realized by allowing to switch between them whenever an actor instance finishes execution (`finishActor`) or blocks (`finishActorBlock`). In the case an actor blocks, then the next actor belonging to the next SDFG is activated. When switching to another SDFGs, current activation count of the current SDFG is saved (`saveActivationCount`). In addition, the SDFG scheduler also registers the finishing of every activated actor and activates the next one according to the chosen scheduling strategy. Every time the scheduler timed automaton is activated a delay time (see Δ_S in Def. 4.2.12).



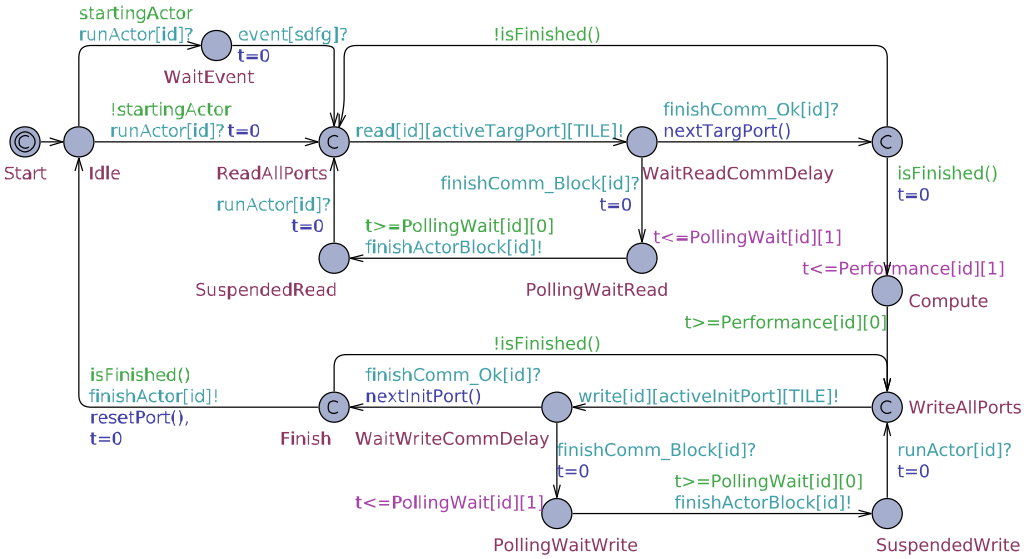
Identifier	Type	Comment
t	clock	Used for delay of computation time of the scheduler
tile	int (Parameter)	Identification of the tile to which this scheduler is mapped
schedStrategy	int (Parameter)	SDFG scheduling strategy (static-order (SO), Round-robin (RR))
SchedulerDelay	array	Non-deterministic selection of a (real-valued) delay in the interval Δ_S which represents delay of scheduler code when executed on tile
countActive	int (local	Counts the activation of current active actor
activeActor	int (local)	Identification of activeActor which should be activated after scheduling
saveActivationCount()	method(local)	Saves the current actor ID and its Activation count of the active SDFG
getReadyActor()	method(local)	implements the scheduling strategy see Sect. 6.5.2
runActor	channel array	Global channel array used for activating the ready actor
finishActor	channel array	Global channel array indicating that the active actor finished execution

Figure 5.4: Template of SDFG scheduler

5.2.3 Actor Templates

As already explained in previous sections, we differentiate between three kinds of actors: `Producer`, `Consumer` and `Transporter`. Fig. 5.5 depicts the transporter actor timed-automaton template. The `Transporter` actor's template is the composition of the consumer and producer TA templates (c.f. Sect. 4.2.3 for single phases of an actor), consisting of consuming states (`ReadAllPorts`, `WaitReadCommDelay`), computing (`Compute`) and producing states (`WriteAllPorts`, `WaitWriteCommDelay`).

One specific attribute (`startingActor`) of the transporter template decides whether this actor reacts to the event trigger (`event[sdfg]`) or not. If the actor is sensitive to an event trigger and got activated by the scheduler (`runActor`) then it changes its state from `idle` to `WaitEvent` state where it waits until a trigger comes from the event trigger before going to `ReadAll-Ports` state. If the actor is not sensitive to an event trigger and got activated by the scheduler then it changes directly to `ReadAllPorts` state. In the `ReadAllPorts` state, the actor reads on all its ports according to their rates, and for every communication issues a `read` signal to the communication driver. When the communication of the last port is finished, a time from the interval Δ_A =`[BCET;WCET]` (see Def. 4.2.12) is passed (`Compute` state). Then



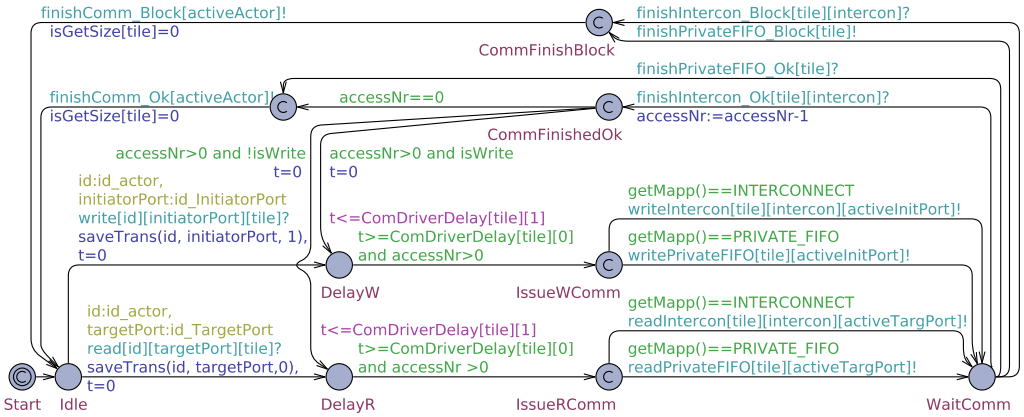
Identifier	Type	Comment
t	clock	Used for delay of computation time of the actor
id	int (Parameter)	Identification of an actor
TargetPort_Nr	int (Parameter)	Nr. of the target ports of the transporter
InitiatorPort_Nr	int (Parameter)	Nr. of the initiator ports of the transporter
Performance	array	Non-deterministic selection of a (real-valued) delay in the interval Δ_A
activeTargPort	int (Local)	A reference to the id of the current active target port
activeInitPort	int (Local)	A reference to the id of the current active initiator port
PollingWait	array	Non-deterministic selection of a (real-valued) polling delay in the interval Δ_P
startingActor	bool(local)	If true this actor is registered to react on event[sdfg]
resetPort()	method	Resets the port id to the first port of current actor
nextTargPort()	method	Returns the identification of next target port of an actor
nextInitPort()	method	Returns the identification of next initiator port of an actor
isFinished()	method	Returns true if actor finished reading/writing all ports
read/write[id][port][TILE]	channel array	Used to initiate read/write communication via the communication driver
finishComm_Block[id]	channel array	Signaling that communication attempt failed due to blocking on FIFO buffer
finishComm_Ok[id]	channel array	Signaling that communication attempt succeeded on FIFO buffer

Figure 5.5: Template of SDF transporter actor

the actor writes to all its ports depending on their rates, and after writing to all ports, the actor sends a `finishActor` signal, indicating a single successful execution.

The `Consumer` actor does not produce any tokens and consists mainly of consuming (the tokens on all ports needed to fire the SDF actor) and computing states (`ReadAllPorts, ReadWriteCommDelay`, the implementation resembles the upper part of transporter actor in Fig. 5.5). A `Producer` actor mainly consists of compute and produce states (`WriteAllPorts, WaitWriteCommDelay`).

Every time, a read or write access blocks on FIFO buffers, a `finishComm_Block` signal is received and the actor's automaton changes to `pollingWait` state. After some `pollingWait` delay (Δ_P see Def. 4.2.12), it goes to a `Suspended` state where it waits to be activated by the scheduler. In the case of a SO hierarchical schedule, this is done directly without any further time delay,



Identifier	Type	Comment
t	clock	Used for track the delay of communication driver every time it runs
tile	int (Parameter)	Identification of tile to which the communication driver is mapped
style	bool (Parameter)	Type of inter-processor communication: burst-transfer or single-beat
intercon	int (local)	Id of the interconnect on which the communication is initiated
accessNr	int (local)	Counter of data (with the same width of the interconnect) to be transported
CommDriverDelay	array (Global)	Non-deterministic selection of a (real-valued) delay in interval Δ_C
isGetSize	array (Global)	Flag: *false* if current access is *getSize* access, *true* if token transfer access (c.f. Sect. 6.5.3)
saveTrans	method (local)	Saves the current transaction properties (id, type and portId) in local variables
getMapp	method (local)	Returns the mapping of the current channel
getNrOfInterconAccesses	method (local)	Returns the number of interconnect accesses to be served for current transaction
write/readInterconnect	channel array	Used for initiating Read/Write communication on the interconnect
write/readPrivateFIFO	channel array	Used for initiating Read/Write communication on the private FIFO buffer
finishInterconnect_Ok/Block[tile]	channel array	Signaling that communication attempt succeeded/blocked on shared FIFO buffer
finishPrivateFIFO_Ok/Block[tile]	channel array	Signaling that communication attempt succeeded on private FIFO buffer

Figure 5.6: Template of communication driver

but in case of RR schedule, the actor would stay suspended while other actors from other SDFGs get the chance to run, before being again activated by the scheduler to retry the communication attempt by issuing a `read/write` signal.

5.2.4 Communication Driver Template

Fig. 5.6 shows the implementation of the communication driver template (c.f. Sect. 6.5.3 for implementation issues of communication drivers). After getting a read/write request from the corresponding actor, the template saves current transaction parameters (`saveTrans`) and switches to a `Delay` state where time (see Δ_C in Def. 4.2.12) is delayed modeling the upper/lower bounds of driver execution. The communication driver is responsible of realizing the communication either with the private FIFO buffer or with the shared FIFO buffer (via interconnect(s)). After the delay phase, the driver issues a read/write channel on the communication resource depending on mapping parameters. After waiting for the communication (`WaitComm`) to finish, the driver either forwards the result (`finishInterconnect_Ok` or `finishInterconnectBlock`) back

to the actor (in case all accesses are served i.e. `accessNr==0`) or it continues accessing the shared FIFO buffer if there are still accesses to be served (`accessNr>0`).

Depending on the chosen inter-processor communication style, the communication driver either issues one burst transfer or in case of single-beat transfer every read/write transaction is partitioned (with the help of `getNrOfInterconAccesses()` method which is executed every time a new transaction is issued within the `saveTrans()` method) to a number (`accessNr`) of atomic read/write transactions depending on number of tokens to be transported and the interconnect width.

For a burst-transfer, the variable `accessNr` is fixed to merely two accesses:

$$accessNr_{burst} = 2$$

where the first access represents a `getSize` access as shown in Fig. 6.17 in which the `size` attribute of the FIFO buffer is first read and the second access represents the uninterruptable burst transfer where the actual tokens are transported.

For the single-beat transfer, we differentiate between write accesses, where the `accessNr` can be calculated according to the following equation:

$$accessNr_{single_W} = 1 + \left\lceil \frac{Rate[activeInitPort] \cdot TokenSize[activeInitPort]}{Width[intercon]} \right\rceil$$

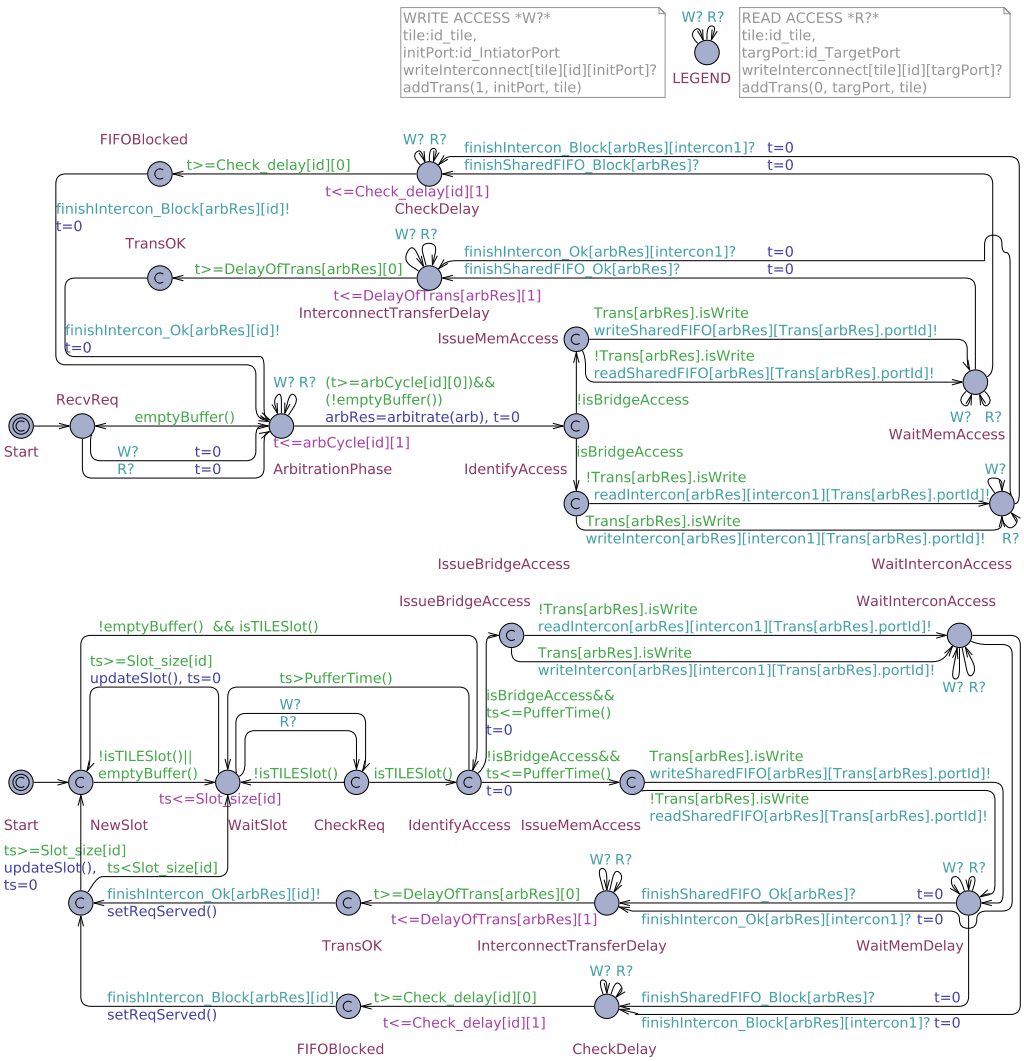
and read accesses:

$$accessNr_{single_R} = 1 + \left\lceil \frac{Rate[activeTargPort] \cdot TokenSize[activeTargPort]}{Width[intercon]} \right\rceil$$

In both cases, one access represents the `getSize` access shown in Fig. 6.17 is added to the equation. The second part of the equations above represents the number of single-beat accesses which depend on the rate of the current active port, the size (in bits) of tokens being transported and the interconnect width.

5.2.5 Shared Interconnect Templates

Fig. 5.7 shows two similar timed-automata templates' implementation of the shared interconnect, both having the same interfaces with the difference that the first template (Fig. 5.7 top) supports round-robin, FCFS, and fixed-priority arbitration policies (see Sect. 2.3.3.1) and the other one (Fig. 5.7 bottom) supports a TDMA arbitration. The timed automaton in Fig. 5.7 (top) alternates between `Start`, `RecvReq`, `ArbitrationPhase`, `IdentifyAccess`, `IssueBridgeAccess` or `IssueMemAccess`, `WaitMemAccess` or `WaitInterconAccess`, `InterconnectTransferDelay` or `CheckDelay`,



Identifier	Type	Comment
t	clock (local)	Used to track the delay of every transaction transported
id	int (Parameter)	Identification of the interconnect
arb	int (Parameter)	Arbitration protocol of the interconnect
style	bool (Parameter)	Type of inter-processor communication: burst-transfer or single-beat
bandwidth	int (Parameter)	Bandwidth of the interconnect
arbRes	int (local)	Arbitration result: Id of the tile which wins the arbitration
isBridgeAccess	bool (local)	True: in case of bridge access to other interconnect, False: in case of an access to other interconnect via a bridge
arbCycle	array (global	Non-deterministic selection of a (real-valued) delay in interval Δ_{arb}
DelayOfTrans	array (local)	Non-deterministic selection of a (real-valued) delay in interval $[bcat; wcat]$ when reading/writing a number of tokens from/to shared FIFO
Check_delay	array (global)	Non-deterministic selection of a (real-valued) delay in interval $[bcat; wcat]$ when reading *size* value of shared FIFO buffer
WriteInterconDelay	array (global)	Non-deterministic selection of a (real-valued) delay in interval $[bcat; wcat]$ time needed to write a token of bus width size on the interconnect
ReadInterconDelay	array (global)	Non-deterministic selection of a (real-valued) delay in interval $[bcat; wcat]$ time needed to read a token of bus width size on the interconnect
emptyBuffer	method (local)	Returns true if the transaction buffer is empty
write/readInterconnect	channel array	Used for initiating read/write communication on another interconnect
write/readSharedFIFO	channel array	Used for initiating read/write communication on the shared FIFO buffer
finishInterconnect_Ok/Block[tile]	channel array	Signaling if communication attempt via other interconnect succeeded or not
finishSharedFIFO_Ok/Block[tile]	channel array	Signaling if communication attempt succeeded on shared FIFO buffer or not
	TDMA Specific	
ts	clock (local)	Used to track the delay of slot in case of TDMA arbitration
Slot_size	array (global)	Maximal slot size length which can be set for every interconnect
isTileSlot	method (local)	Returns true if transaction in buffer and current slot has same id as the tile id
setReqServed	method (local)	Marks the request served after finishing slot time
updateSlot	method (local)	Updates the slot Id to the next one after finish serving the current one
PufferTime	method (local)	Returns the rest time of the slot after finish serving the current transaction

Figure 5.7: Top: TA template of shared interconnect with FCFS/RR/FP arbitration, Bottom: with TDMA arbitration

TransOK or FIFOBlocked states (see timing models of the interconnect in Sect. 2.3.3.2). In RecvReq the interconnect waits for requests either from different tiles or from other interconnects, and when received these transactions are saved (saveTrans()) to a local buffer. For every arbitration protocol there are different access methods to that local buffer. For e.g. in case of Fixed-Priority or Round-Robin arbitration, the transaction is inserted to an array index equivalent to that of the tile (tileid), while in case of FCFS the transactions are enqueued/dequeued according to a typical FIFO manner. Requests are received as long as the arbitration cycle (with a delay interval Δ_{arb} = [$arbCycle$[id][0], $arbCycle$[id][1]]) does not expire. After that, an arbitration phase (arbitrate()) is done according to the configured arbitration protocol to choose the transaction with the highest priority. Next, depending on the kind of access (isBridgeAccess: if *False* then a direct access to shared FIFO is issued else an indirect access of the shared FIFO via other interconnects is realized), a read or write transaction is issued (in IssueInterconAccess or IssueMemAccess states) either to another interconnect (through a bridge[2]) which forwards it to the corresponding shared FIFO buffer or directly to the shared FIFO buffer belonging to a storage resource which is directly connected to the current interconnect. After waiting (WaitInterconAccessDelay or WaitMemAccessDelay state) till the access is finished and a corresponding success signal is received (finishInterconnect_Ok), a delay modeling the transfer of the arbitration-winning transaction in InterconnectTransferDelay state is passed. It is important to note, that in this case, the above delay either represents the time needed to get the size of the buffers (which is the case if getSize flag is false) or the time needed to transport the tokens (when getSize flag is true as described in Sect. 5.2.4). If the access to the FIFO buffer was not successful (finishInterconnect_Block) a delay is passed which models only reading the variable size of the FIFO buffer. The result is then forwarded (in TransOK, FIFOBlocked states) either directly back to the communication driver of the requesting tile or indirectly through the requesting interconnect(s). In states where time elapses (RecvReq, ArbitrationPhase, WaitMemAccess/WaitInterconAccess, InterconnectTransferDelay/CheckDelay), read-/write requests (Ⓦ? or Ⓡ?) are registered and saved to local buffers.

The timed automaton in Fig. 5.7 (bottom) is very similar to the above one except for the arbitration phase (states: NewSlot, WaitSlot, CheckReq), which supports a non-preemptive slot based TDMA arbitration (see Sect. 2.3.3.1). In this arbitration, each tile is mapped to a slot (having the same id of the tile) of a fixed length where it is allowed to perform inter-processor

[2]It is important to note here that the delay bounds of the bridge component is included in the delay of the target interconnect.

communication. In order to insure that every transaction on the interconnect finishes before the slot time expires (since it is non-preemptive), the length of all slots is set to the transaction communication time (including arbitration cycle time) with the maximal delay which can be requested on this interconnect i.e. if t_{max_n} is the maximal communication time of actor n needed to transport a number of tokens among all its ports (if necessary traversing multiple interconnects) to a target shared storage resource then:

$$Slot\_size_I = max\{t_{max_1}, \ldots, t_{max_n}\}$$

where $I \in \mathcal{I}$ and $n \subseteq \mathcal{A}$ is the set of all actors which can access the interconnect I. As the automaton starts, a new slot (in `NewSlot`) is initiated and the clock `ts` measures the elapsed time since the begin of this slot. Then it first waits for a read or write request, if received it checks whether the currently requesting tile is mapped to the current slot, if this is the case then it switches to `IdentifyAccess` state where it proceeds just as the automaton described above. The function `PufferTime()` helps in cases where for e.g. the same tile after being served once in its slot, it requests the interconnect again or in case the slot time was already delayed for some time before the request comes from the tile mapped to it. In above cases, `PufferTime()` function can calculate if their is enough rest time to serve the current request or not.

Every time a read/write request is registered, the method `saveTrans()` is called where the transaction is saved, and an internal `updateDelay()` function is called. This function calculates the delay interval for the current requesting tile needed to transport the requested number of tokens on this interconnect. Depending on what type of transfer, we differentiate between three kinds of delays:

1. The case of only reading the `size` (c.f. Sect. 6.5.3) of the buffer (if it is still not read indicated by a false `isGetSize` flag or if in state `CheckDelay`):

$$DelayOfTrans_{t0} = Check\_Delay_{i0}$$
$$DelayOfTrans_{t1} = Check\_Delay_{i1}$$

2. In case of *single-beat* transfer this delay is calculated according to the following formula for a write access:

$$DelayOfTrans_{t0} = WriteInterconDelay_{i0}$$
$$DelayOfTrans_{t1} = WriteInterconDelay_{i1}$$

and in case of a read access:

$$DelayOfTrans_{t0} = ReadInterconDelay_{i0}$$
$$DelayOfTrans_{t1} = ReadInterconDelay_{i1}$$

3. In case of a *burst-transfer* the delay is calculated according to the following formula for a write access:

$$DelayOfTrans_{t0} = accessNr_W \times WriteInterconDelay_{i0}$$
$$DelayOfTrans_{t1} = accessNr_W \times WriteInterconDelay_{i1}$$

and in case of a read access:

$$DelayOfTrans_{t0} = accessNr_R \times ReadInterconDelay_{i0}$$
$$DelayOfTrans_{t1} = accessNr_R \times ReadInterconDelay_{i1}$$

where $t \in \mathcal{T}$, $i \in \mathcal{I}$, $accessNr_R$ and $accessNr_W$ as defined in Sect. 5.2.4 are the number of accesses needed to transport tokens of specific sizes on an interconnect of specific width, $ReadInterconDelay_{i0}$, $WriteInterconDelay_{i0}$ are best-case time delays for the interconnect to make a read or write access of data with size equal to that of the interconnect width and $ReadInterconDelay_{i1}$, $WriteInterconDelay_{i1}$ are the worst-case time delays for the interconnect to make a read or write access of data with size equal to that of the interconnect width. According to Def. 4.2.12 the delay interval Δ_I of the interconnect can now be calculated:

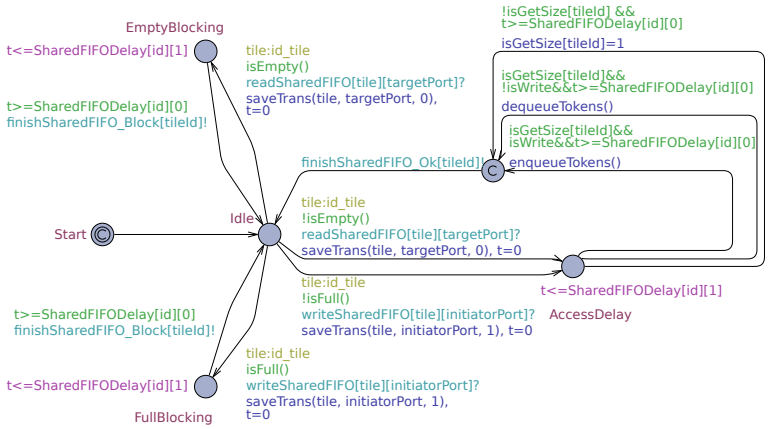
$$\Delta_I = \Delta_{arb} + \Delta_{Trans}$$

where $\Delta_{Trans} = [DelayOfTrans_{t0}, DelayOfTrans_{t1}]$ and as defined above $\Delta_{arb} = [arbCycle[id][0], arbCycle[id][1]]$.

5.2.6 Templates of Shared and Private FIFO Buffers

The FIFO buffer timed-automaton template is straight forward and models a queue with a blocking synchronization behavior (see Fig. 5.8). The timed automaton simply models a delay depending on the transaction type (Read-/Write) and the number of tokens to be consumed or produced of every port. If there are not enough tokens available (`isEmpty()`) by a read transaction or no enough space to store tokens (`isFull()`) by a write transaction, the FIFO buffer blocks and sends a corresponding signal (`finishSharedFIFO_Block`) back to the interconnect. Now if this is not the case and if the current access is not a `GetSize` access (`isGetSize[tile]==1`), the tokens needed are enqueued or dequeued (`enqueueTokens()`, `dequeueTokens()`) depending on the transaction (read or write). Yet, if it is a `GetSize` access (`isGetSize[tile]==0`), then the flag `isGetSize` is set to true and no tokens are enqueued or dequeued (c.f. Fig. 5.6).

In case of a *burst-transfer* inter-processor communication style, the number of tokens (depending on the port's rate of the current communicating actor) is



Identifier	Type	Comment
t	clock (local)	Used for delay of memory access time of transaction
id	int (Parameter)	Identification of the FIFO buffer
depth	int (Parameter)	Depth of FIFO buffer which can be obtained from SDFG schedule
style	bool (Parameter)	Type of inter-processor communication: burst-transfer or single-beat
isWrite	bool (local)	Flag set to true when transaction is a write and to false if it is a read one
SharedFIFODelay	array (local)	Non-deterministic selection of a (real-valued) delay in $[bcat; wcat]$ for access delay
isGetSize	array (Global)	Flag: *false* if current access is *getSize* access, *true* if token transfer access
isFull	method (local)	Depending on *style* returns true if buffer is full
isEmpty	method (local)	Depending on *style* returns true if buffer is empty
dequeueTokens	method (local)	Depending on *style* dequeues the number of tokens from buffer
enqueueTokens	method (local)	Depending on *style* enqueues the number of tokens to buffer

Figure 5.8: Template of (private or shared) FIFO buffer

dequeued/enqueued completely in one access (only in case the buffer is not full or not empty), while in the case of *single-beat* transfer only one token (of size equal to interconnect width) is dequeued/enqueued at every access. If successful, a success signal is sent back to the interconnect (`finishSharedFIFO_Ok`). The `SharedFIFODelay` (see Δ_{M_s} in Def. 4.2.12) can be calculated similar to the calculation of $DelayOfTrans$ (see formula presented in 5.2.5) for both single-beat and burst-transfer styles, only $Write/ReadInterconDelay$ should be replaced with the write/read latency of the shared storage resource.

5.2.7 Extensions for DMA Burst Transfer

In the above templates, it is assumed that the interconnect has built-in capabilities supporting burst transfers without extra synchronization primitives. In the case where a DMA (see Sect. 6.5.3) is used as an interconnect to realize a burst transfer, some minor changes should be applied to the communication driver and interconnect TA templates, which will be described in the following.

As we will show in Sect. 6.5.3, in case of DMA for every read/write access of any actor to the shared storage resource two main accesses are launched (see Fig. 6.19): first one for making a synchronization of the shared buffer with the private one (`syncPrivate()`) and another access to do the opposite by synchronizing the shared buffer with the private one (`syncShared()`).

If we now take a sharp look at the implementation of burst transfer in TA template of communication driver in Fig. 5.6, we can see that these semantics are already supported. By a burst transfer, for every read/write transaction of an actor two accesses are issued: a synchronizing `getSize` access where the shared FIFO buffer `size` is read (without enqueuing/dequeuing tokens) and a tokens' transfer access (where the actual enqueue/dequeue action takes place). If we now consider the first access as the `syncPrivate()` and the second one as the `syncShared()`, only following minor changes concerning the timings of communication driver and interconnect must be applied:

1. For the communication driver timing we differentiate two cases:
   In the case of a `syncPrivate()` access (signalized through a false `isReadSize` flag), Δ_C can be calculated as follows:

   $$\Delta_C = \Delta_{initDma} + \Delta_{configDma}$$

   otherwise if a `syncShared()` access is taking place (signalized through a true `isReadSize` flag), then :

   $$\Delta_C = \Delta_{initDma} + \Delta_{configDma} + \Delta_{CanGet/CanPut} + \Delta_{dequeue/enqueue}$$

   where $\Delta_{configDma}$ is the time interval needed to configure the DMA through the configuration interconnect I_{config}, $\Delta_{initDma}$ is the local initialization phase of the driver software code with no accesses to the interconnect, $\Delta_{CanGet/CanPut}$, $\Delta_{dequeue/enqueue}$ are the BCET/WCET of local methods executed on the local memory (as seen in Fig. 6.19).

2. The interconnect delay resembles that of the generic interconnect TA template with the difference that the specific arbitration of the DMA should be taken into consideration:

   $$\Delta_I = \Delta_{arbDma} + \Delta_{DelayOfTrans}$$

   where $\Delta_{DelayOfTrans}$ is the time needed to transport the whole buffer using the transfer interconnect I_{trans} and Δ_{arbDma} is the time delay of the DMA needed to perform internal arbitration.

5.2.8 Observer TA Templates for Real-time Analysis

Timing properties (see Sect. 2.2.1.2) of timed SDFGs (SDFGs after mapping, binding and scheduling to specific MPSoC) which will be evaluated in this work (see Chap. 7), are defined in the following (based on [Lin et al., 2011]):

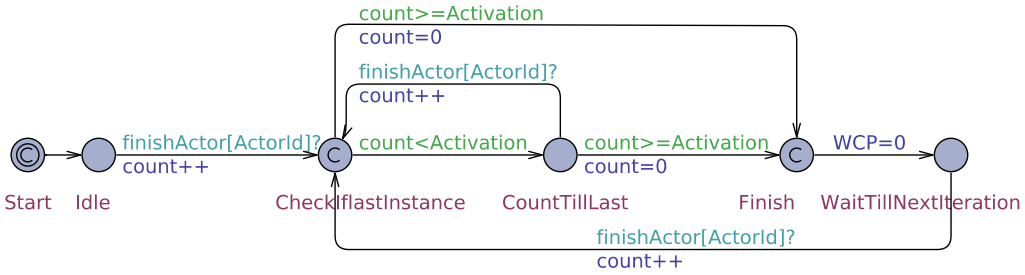


Figure 5.9: Observer TA template of the period of an SDFG

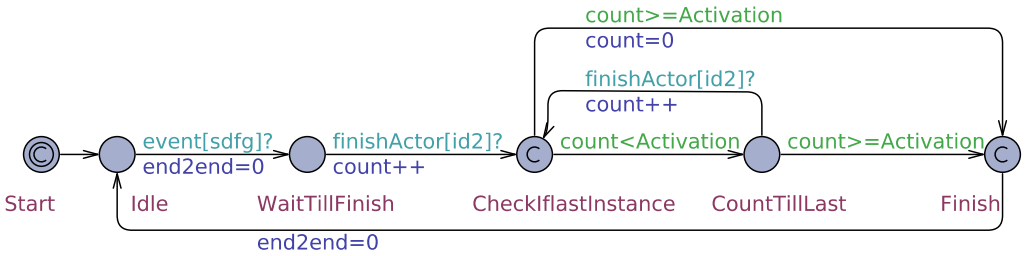


Figure 5.10: Observer TA template of end-to-end latency of an SDFG

Definition 5.2.1. *(Iteration)* Given a timed $SDFG$ with a repetition vector γ, an iteration of an SDFG is defined as the minimum non-zero execution (i.e. at least one actor has executed) such that the initial state of the graph is obtained i.e. according now to Def. 4.2.4 an iteration is a set of actor firings such that for each $a \in SDFG$, the set contains the $\gamma(a)$ firings of a.

Definition 5.2.2. *(Period)* The period of an SDFG is defined as the time an SDFG takes to complete one iteration.

Definition 5.2.3. *(End-to-end latency)* The end-to-end latency is defined as the time starting from activating the first instance of the source actor (upon receiving the first event e.g. reading sensor values), executing the SDFG application till the last instance of the sink actor is finished (e.g. updating actuators).

If an SDFG is sensitive to a periodic event trigger, then its period is equal to that of its event trigger. This is why in this case the end-to-end latency of the SDFG should always be less than or equal the period of the event trigger.

To obtain the period of an SDFG which is not sensitive to an event trigger, we have implemented an observer automaton Fig. 5.9 which traces the finishing time of the last instance of the sink actor of that SDFG (see Fig. 2.4). For this, the `id` of the sink actor and its `Activation` number (activation number in repetition vector see Def. 4.2.4) are given as parameters for this template.

Another important metric which will be also evaluated in the experiments is the end-to-end latency. Fig. 5.10 depicts the timed-automaton template which allows us to trace the end-to-end latency for an SDFG sensitive to an event trigger from the time the event triggers the SDFG till the finishing of the sink actor's last instance. If an SDFG is not sensitive to an external event, the end-to-end latency observer TA traces now the time from the first activation of the source actor in an SDFG (instead of waiting for `event[sdfg]`, it waits for `runActor[id1]` where `id1` is the id of the source actor) till the finishing of the sink actor's last instance. For this, the `id1` of the source actor (first actor to be executed in the SDFG, from which the SDFG id can be obtained), the `id2` of sink actor and the activation number (`Activation`) of the sink actor are needed as parameters for this template.

5.3 Real-time Analysis via Model-checking

As already described in Sect. 2.5.2.1, UPPAAL can verify whether a property holds for a given network of timed automata or not. The verification properties can be formalized in a subset of TCTL (Timed Computation Tree Logic). By checking `A [] not deadlock`, we can verify whether or not our system is deadlock free. We could also take use of the model-checker operator `sup` which searches for the *supremum* of a variable or a clock value in the system. Likewise, we could find the *infimum* by utilizing the `inf` operator.

To obtain the worst/best-case period of an SDFG, we can now utilize the `sup/inf` operator of UPPAAL model-checker to search for the maximum/minimum delay between two consecutive finishing instances of the sink actor which coincides with the (worst/best case) period of the graph as implemented in Fig. 5.9. In addition, we can utilize the same operators to search for the maximum/minimum delay from the time the first event comes till the time the last instance of last actors is finished which coincides with the end-to-end latency as implemented in Fig. 5.10. In order to obtain the Worst-case Period (WCP) and end-to-end latency of an SDFG respectively, the following two TCTL formula should be checked through UPPAAL:

$$sup\{obs.Finish\} : obs.WCP$$
$$sup\{obs.Finish\} : obs.end2end$$

where `obs` represents an object of the corresponding observer template.

Other properties which can be checked by the model-checker are presented and evaluated in Sect. 7.1.

5.4 Methods for Improving Scalability

In Sect. 2.5.2.1 we shortly identified the basic elements of TA which can exponentially blow the state space of a network of timed automata implemented in UPPAAL. In general, the global state space of a network of TA (see Sect. 2.5.2.1) grows exponentially with the number of concurrent components, number of global and local variables needed for the TA and number of synchronization channels. Another major aspect which could lead to a huge state space, is the level of non-determinism represented in the considered TA templates (for e.g. the larger the interval between the expected [BCET,WCET] the larger the non-determinism).

If we now try to identify these parameters for our timed-automata network taking into consideration our system model specific implementation (see Sect. 5.2), we find out that the total number of TAs (TA_{total}) needed to model the System under analysis (SUA) can be calculated as follows:

$$TA_{total} = (2 \times T) + A + C + I + E + 1 \tag{5.1}$$

where T is the number of tiles, since for every tile we have one scheduler and one communication driver, A is the total number of actors running in the SUA, C is the sum of channels of all SDFGs in the SUA where for every channel one TA is needed for representing the corresponding FIFO buffer, I the number of interconnects, E is the number of event trigger TAs (if needed) and since an observer automaton would be needed to validate the real-time metric, one TA is added to the equation above. The total number of clocks would be calculated the same as above equation, since we have by every TA one clock (except in case a TDMA arbitration in the interconnect, where two clocks are needed).

On the other hand, the total number of UPPAAL synchronization channels needed to model the SUA can be calculated as follows:

$$\begin{aligned} Ch_{total} = {} & Ch_{runActor} + Ch_{finishActor\_Ok} + Ch_{finishActor\_Block} + Ch_{finishComm\_Ok} \\ & + Ch_{finishComm\_Block} + Ch_{finishPrivateFIFO\_Ok} + Ch_{finishPrivateFIFO\_Block} \\ & + Ch_{finishSharedFIFO\_Ok} + Ch_{finishSharedFIFO\_Block} + Ch_{read} + Ch_{write} \\ & + Ch_{readPrivateFIFO} + Ch_{writePrivateFIFO} + Ch_{readSharedFIFO} \\ & + Ch_{writeSharedFIFO} + Ch_{readInterconnect} + Ch_{writeInterconnect} \\ & + Ch_{finishInterconnect\_Ok} + Ch_{finishInterconnect\_Block} + Ch_{event} \\ = {} & (P_{target} + P_{intiator})(2A + (A \times T) + (T \times I)) + 2(T \times I) + 7A + 2T + E \end{aligned} \tag{5.2}$$

where P_{target}, $P_{intiator}$ are the total number of target and initiator ports respectively and I is the number of interconnects. From the above equation, we can notice that the number of channels highly depends on the number of ports, actors and tiles. Take as an example the case where we have only 3 actors with 4 ports and 2 tiles, then the total number of channels would grow to be

85 (assuming one interconnect, and no event triggers) according to the above equation. If the number of ports is now changed to 8 then we will have a total number of 141 channels. The reason behind this large number induced by such a small SUA, is that we are modeling the actors at the port level, which requests synchronization channels at the port level, this in turn requires to initialize multi-dimensional channel arrays in UPPAAL (for e.g. 3-dimensional channel arrays in the case of $Ch_{read/write}$ or $Ch_{read/writeInterconnect}$)[3].

In addition, the level of non-determinism in our timed-automata network implementation strongly depends on the difference between the upper and lower bound of delay intervals in these TAs. Also the sequence in which the tiles (represented by scheduler automaton) start execution is non-deterministic.

In the scope of this work, we applied some methods/optimizations to our approach targeting state-space improvement of our templates' implementation. In the following section, we will describe the application of these methods.

5.4.1 Optimizing the Implemented Timed-automata Templates

The timed-automata templates in Sect. 5.1 were the first-shot intuitive implementation capturing the MoP of our considered MPSoC (which were presented and evaluated in [Fakih et al., 2013a]). But if we take a sharp look at these templates following optimizations/abstractions could be made. These optimizations were applied, when possible, in the evaluation chapter (see Chap. 7) and lead to major improvements in terms of state space.

Abstracting Shared/Private FIFO Buffers We have described the TA template of the shared FIFO buffer in Sect. 5.2.6. But according to our system model definitions, no parallel accesses can be issued on the shared FIFO buffers, since the accesses are sequentialized in the interconnect, where the one with the highest priority wins the earliest access. Taking this into consideration, we are able to abstract (not modeling explicitly) the shared FIFO buffers and model them as local queues (with their specific methods) within the interconnect, adding their timing delays to that of the interconnect without distorting the timing semantics.

Similarly to the above abstraction, the private FIFO buffers could also be abstracted. In order to abstract private FIFO buffers, the idea here is to include the time needed to access these FIFO buffers (since these are mapped to local private memories) in the communication driver's BCET/WCET access calculation and model them as local queues (with their specific methods) within the communication driver TA template of the corresponding tile.

[3]Equation 5.2 assumes that UPPAAL doesn't perform further optimization on the initialized multi-dimensional channel arrays.

After applying above abstractions, state-space savings in terms of synchronization channels (since these would be implemented as local methods either in the interconnect or in the communication driver templates) and Eq. 5.2 becomes:

$$\begin{aligned} Ch_{total} &= Ch_{runActor} + Ch_{finishActor\_Ok} + Ch_{finishActor\_Block} + Ch_{finishComm\_Ok} \\ &\quad + Ch_{finishComm\_Block} + Ch_{read} + Ch_{write} + Ch_{readInterconnect} + Ch_{writeInterconnect} \\ &\quad + Ch_{finishInterconnect\_Ok} + Ch_{finishInterconnect\_Block} + Ch_{event} \\ &= (P_{target} + P_{intiator})((A \times T) + (T \times I)) + 2(T \times I) + 5A + E \end{aligned} \tag{5.3}$$

which leads in the example described above (3 actors, 4 ports, and 2 tiles) to a large reduction of total number of channels of 55 (85 in Eq. 5.2) and 87 (141 Eq. 5.2) channels when increasing the number of ports to 8. In addition, the number of timed automata in Eq. 5.1 becomes independent of the number of SDFGs' FIFO buffers:

$$TA_{total} = (2 \times T) + A + I + E + 1 \tag{5.4}$$

and we are able to spare one TA and one clock for every FIFO buffer, which significantly improves the scalability of our method.

Merging Scheduler TA Template and Actors' TA Templates For the case in which we have a static-order SDFG scheduler, additional optimizations can be made. Here, the order of the actors defines the execution priority and no extra scheduling mechanism is needed (for implementation details please refer to [Schaumont, 2013]). Taking this into consideration, we are now able to spare the scheduler timed automaton with its primitives for every tile. Additionally, the following optimization can also be made. Let `so` be an ordered list of actors (see Def. 4.2.10), possibly including actors belonging to many SDFGs, which are executed in a fixed-static order on a specific tile. Now instead of having the cost of one timed automaton for modeling every actor, we utilize only one automaton called *SOonTile* (depicted in Fig. 5.11) to capture the behavior of all actors (belonging to the sorted list `so`) mapped to a tile, and this is done for every tile in the SUA. Obviously, the above optimization leads to significant savings in terms of instantiated number of TAs, clocks and synchronization channels, where Eq. 5.4 becomes:

$$TA_{total} = T + I + E + 1 \tag{5.5}$$

which leads in the example described above (3 actors, 4 ports, and 2 tiles) to a large reduction of total number of TA of 4 (9 in Eq. 5.4), allowing to analyze greater number of actors without drastically increasing the state space. Notice that the number of channels (Eq. 5.3) cannot be reduced since the scheduler

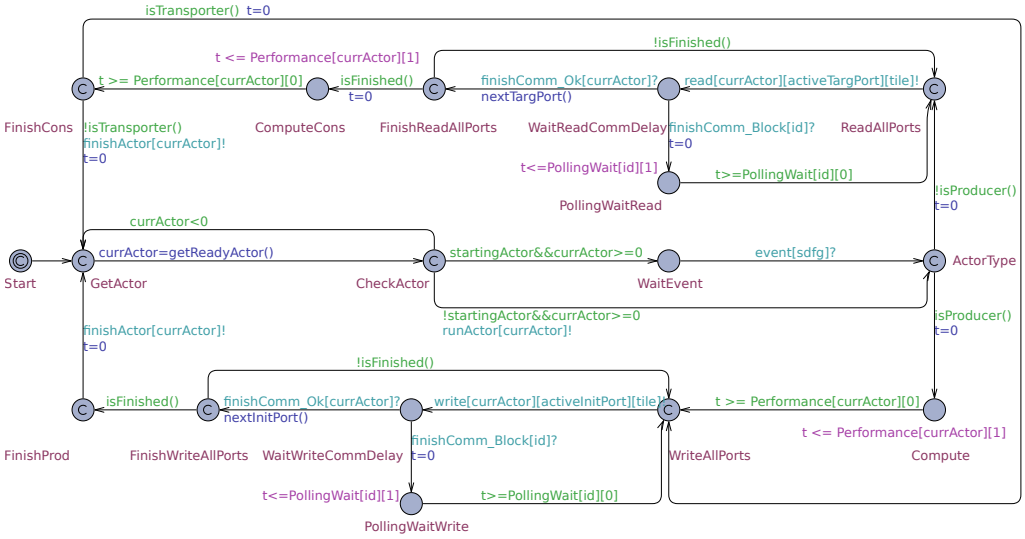


Figure 5.11: Optimization of TA templates in case of SO SDFG scheduler

channels $Ch_{runActor}$ and $Ch_{finishActor}$ are still needed by the observers templates (see Sect. 5.3). Fig. 5.11 shows the optimized TA template of *SOonTile*. The TA starts by choosing the first actor in the ordered list (in state `GetActor`) and then it checks if that actor is sensitive to an event trigger (in state `CheckActor` depending on the flag `startingActor`), where it either proceeds or it waits for an event (in state `WaitEvent`). Now depending on the type of the actor (if a *Producer* or not in state `ActorType`), the TA either begins with consuming tokens on its ports (in case of *Consumer* or *Transporter* actor see above part of Fig. 5.11) or delays Δ_A modeling the computation time of the actor. In the latter case and after producing tokens on all its ports, it finishes (similar to *Transporter* TA implementation in Fig. 5.5). In the first case, after finishing consuming on all ports (in state `FinishCons`), the actor either finishes (if it is a *Consumer* actor) or it continues (if it is a *Transporter* actor) delaying Δ_A and after that producing tokens on its ports to reach at last the finish state (in the state `FinishProd`).

5.4.2 Applying Clustering Method

In Sect. 2.2.1.4, we have described a clustering method for SDFGs known from literature [Bhattacharyya et al., 1997]. This method can obviously improve the scalability of our RT analysis method when analyzing SDFGs with large number of actors. With the help of clustering, the number of actors in an SDFG can be reduced leading to the reduction in the number of TAs which should be explored by the model-checker. In the following, we will describe how this method can be applied taking into consideration our system model properties.

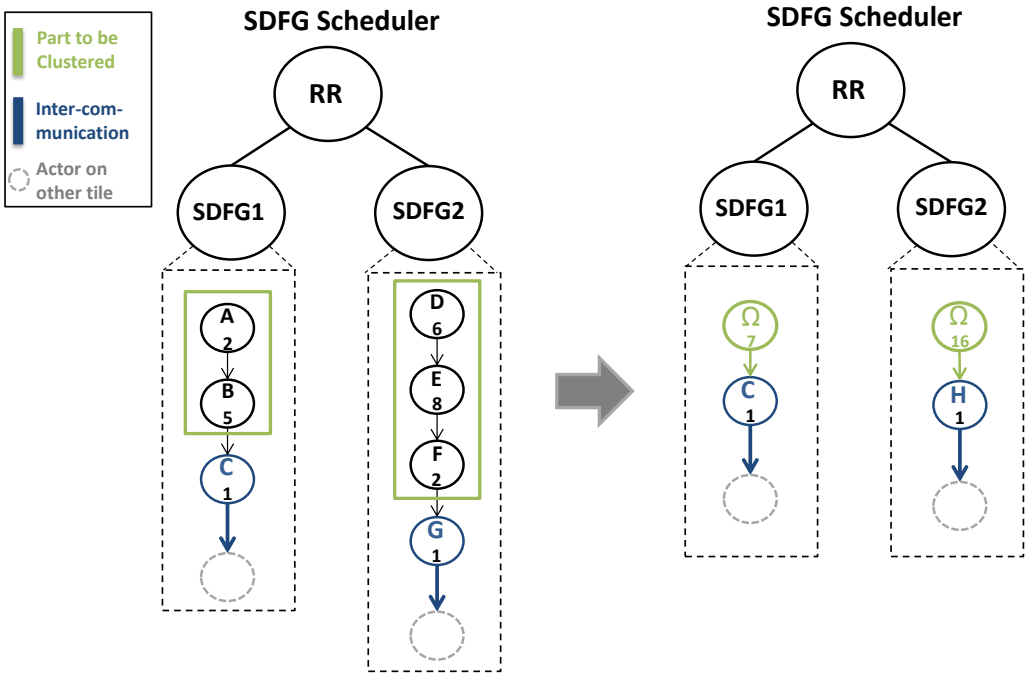


Figure 5.12: Example of clustering timing violation by RR SDFG scheduler

The clustering method as already stated in Sect. 2.2.1.4, assumes connected and consistent SDFGs. Furthermore, the part of the SDFG to be clustered should be acyclic. In order to apply this clustering method on the SDFG(s) in our system model, the following additional conditions should be satisfied:

D1 A static-order SDFG scheduler is assumed.

D2 The mapping of the SDFG(s) actors to the MPSoC must be known since only actors mapped to the same tile and which do not engage in an inter-processor communication can be clustered.

It is obvious that the clustering method can be applied only in the case where a static-order SDFG scheduler is used (D1). For the case, e.g. the round-robin SDFG scheduler is used, the timing semantics could be violated when applying the clustering method in its general form as shown in the example in Fig. 5.12. Here, we assume that both homogeneous SDFGs (*SDFG1* and *SDFG2*) are mapped to the same tile (except the gray shadowed actors are mapped to another tiles) and are scheduled according to RR scheduling (see Sect. 4.2.4.2) with SDFG1 being executed before SDFG2. In addition, we assume that the BCET and the WCET are equivalent for every actor. This execution time is annotated to every single actor as seen in Fig. 5.12. The resulting SDFGs after applying the clustering method are shown in Fig. 5.12 (to the right). Notice that

in the unclustered (to the right of Fig. 5.12) SDFG1 (by RR SDFG scheduler) actor C comes to execution after 21 time units (under the assumption that no blocking on the FIFO buffers occurs) while in the clustered version it will come to execution after 23 time units. This fact obviously proves that the timing semantics can be violated when applying the clustering method, in its general form, to SDFGs scheduled according to Round-Robin.

Mapping information (see D2) are needed, since only actors mapped to same tile and which do not engage in an inter-processor communication can be clustered. The reason behind this is that actors of an SDFG which are engaged in an inter-processor communication, when clustered show different timing semantics (because of possible changes in the rates of the ports in the resulting hierarchical actor). This in turn, could lead to a distorted access pattern on the shared interconnect which could lead to false real-time results.

If above conditions hold, clustering can now be applied. After clustering, one issue remains, which is how to calculate the WCET/BCET of the resulting hierarchical actor Ω. If n denotes the number of actors in $\mathcal{Z}$, $\gamma(a)$ is the repetition vector value of actor a (for notations' details refer to Sect. 2.2.1.4) then the new *wcet* of the hierarchical actor Ω can be calculated as follows:

$$\gamma(\Omega) \times wcet(\Omega) = \sum_{i=1}^{n} (\gamma(a_i) \times wcet(a_i))$$

$$\Leftrightarrow wcet(\Omega) = \frac{\sum_{i=1}^{n} (\gamma(a_i) \times wcet(a_i))}{\gamma(\Omega)}$$

similarly the *bcet* of the hierarchical actor Ω can be calculated as follows:

$$bcet(\Omega) = \frac{\sum_{i=1}^{n} (\gamma(a_i) \times bcet(a_i))}{\gamma(\Omega)}$$

It is important to note that beyond the estimated WCET/BCET of single actors when being executed on a target processor, the clustering technique is fully independent from the target architecture of the MPSoC.

5.4.3 Temporal and Spatial Segregation for a Composable and Scalable RT Analysis

Another way to improve the scalability of our approach is to enable composability by extending the MPSoC with extra hardware components (timers or hypervisors [Aeronautical Radio, 2003]) which guarantee temporal and spatial isolation of clusters (where a cluster is defined as a group (or parts) of SDFGs see Fig. 5.14) mapped to an MPSoC. This allows us to verify these clusters in isolation and afterwards use the composability property to analyze these clusters when they are integrated on one MPSoC (see Fig. 5.15). We will show

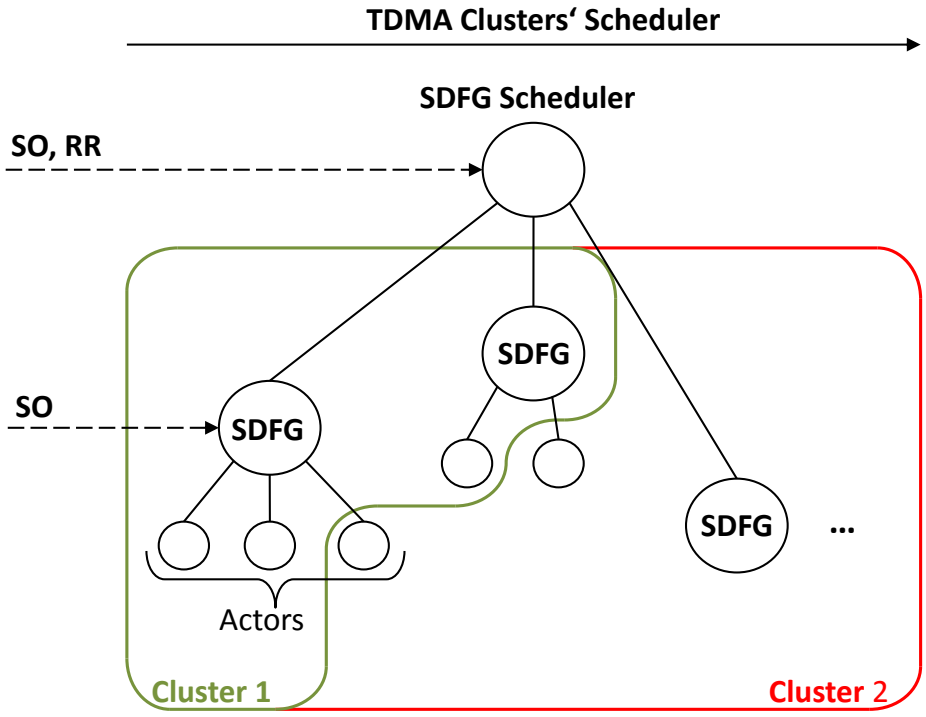


Figure 5.13: Scheduling hierarchy extended with TDMA clusters' scheduler

(in Sect. 7.2.5) that with the help of this extension, we are able to improve the number of actors, being analyzable by our approach, on an MPSoC with a fixed number of tiles.

In the following, we assume that the spatial isolation is already realized (e.g. either through virtualization with the help of a hypervisor [Fakih et al., 2013b] or through static memory allocation) and will describe how such a composable RT analysis can be made based on a TDMA clusters' scheduler and with the help of our state-based RT analysis method. Fig. 5.13 shows the scheduling hierarchy (see Sect. 4.2.4.2) extended with a non-preemptive TDMA clusters' scheduler in the top hierarchy level. This TDMA scheduler allows clusters of actors (still respecting the lower two hierarchy scheduling levels: SO within the SDFG and SO or RR among SDFGs) to be executed in only specific time slots and switches to next slot as soon as the previous expires, and is defined as follows:

Definition 5.4.1. *(TDMA Clusters' Scheduler)* A TDMA scheduler is defined as a tuple $S = (F, \mathcal{SL})$ where F represents the functionality (code) of the scheduler (c.f. pseudo-code in Sect. 6.5.2 which switches between different slots on a tile, $\mathcal{SL}$ is a finite set of slots $Sl = (d, Cl)$ each having a duration d after which the slot expires and a cluster $Cl \subseteq \mathcal{SO}$ of different SDFG schedules to be executed in this slot. Let $\mathcal{T}$ be the number of tiles in the system then every tile $t \in \mathcal{T}$ has

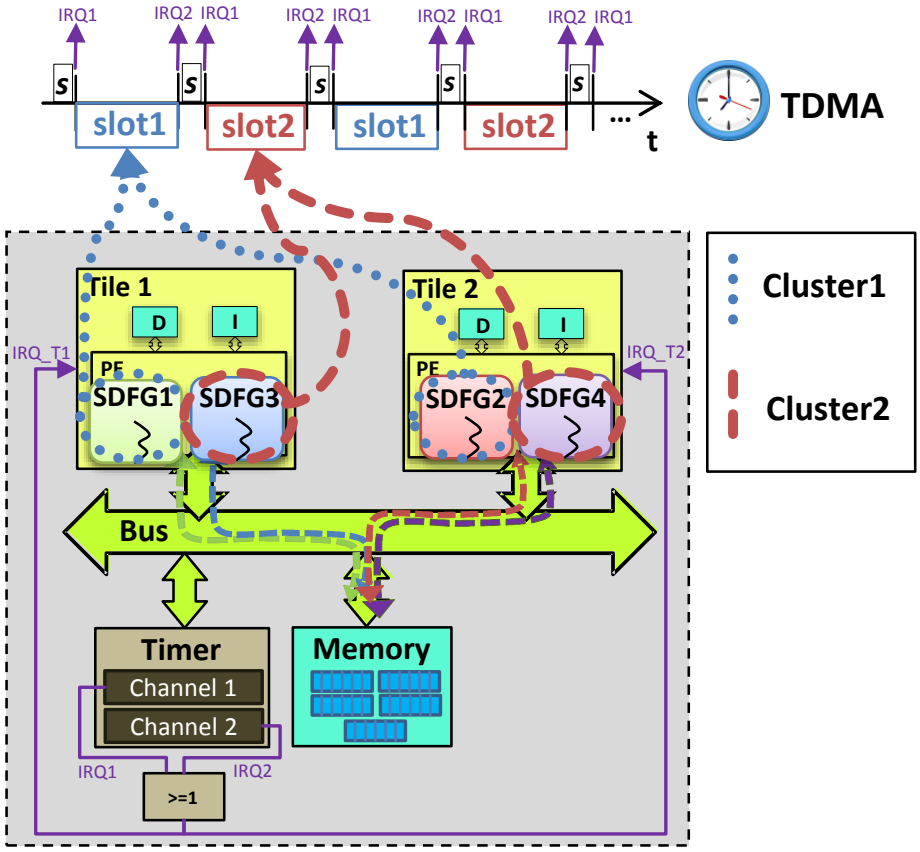


Figure 5.14: Example of TDMA scheduling of clusters of SDFGs

its own $\mathbf{so_t} \subseteq \mathcal{SO}$ (see Def. 4.2.10) and every cluster $Cl = \{sot_0, sot_1, \ldots, sot_t\}$ consists of a set of schedules to be executed on every tile in the current slot where $sot_0 \subseteq \mathbf{so_0}$, $sot_1 \subseteq \mathbf{so_1}$ and $sot_t \subseteq \mathbf{so_t}$ respectively.

In this work, we make a simplification of the general case of Def. 5.4.1 concerning at which granularity we allow to construct the clusters and assume that a cluster can consist of a number of SDFGs and these are independent from other SDFGs mapped to other clusters. Note that the general case, when permitting the clustering at the granularity level of actors (see Fig. 5.13), could easily lead to deadlocks in the SUA if care is not taken, this would not be the case if the clustering is made at the granularity level of SDFGs.

Let us take a look at a concrete example to understand how the TDMA clusters' scheduler works. In a first step, clusters of SDFGs are identified as shown in the simple example in Fig. 5.14, where $SDFG1$ and $SDFG2$ are mapped to $cluster1$ and $SDFG3$ and $SDFG4$ are mapped to $cluster2$. Next, scheduling strategy is chosen for the SDFG scheduler (say for e.g. RR).

Now, the worst-case instance of some timing metric (e.g. period or end-to-

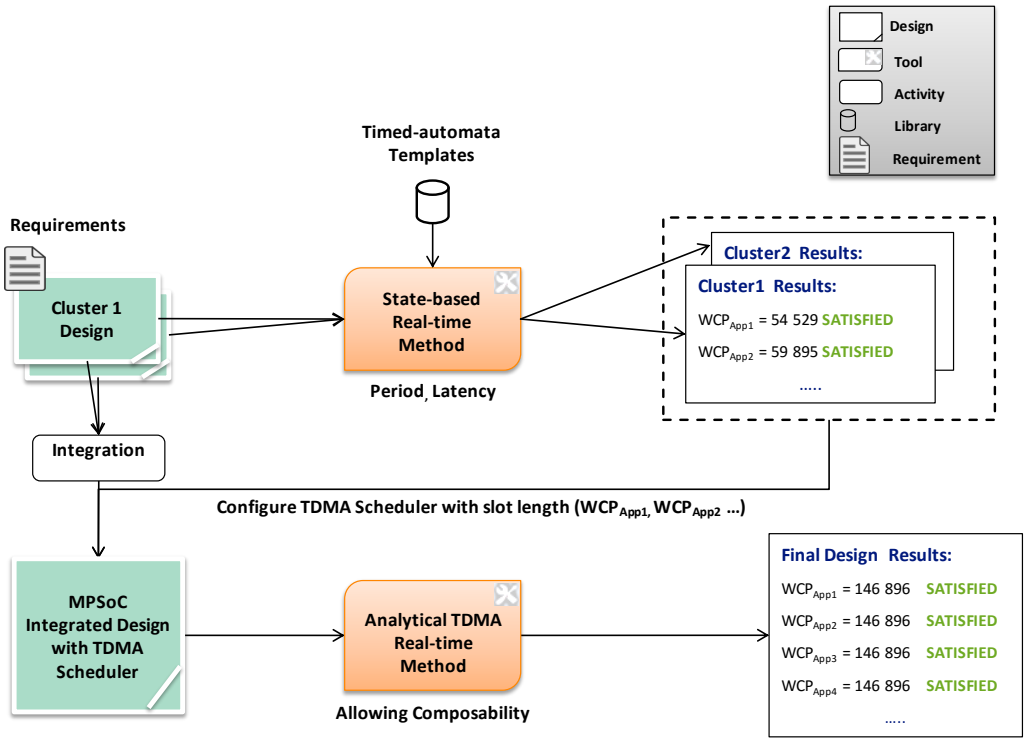


Figure 5.15: Two-Tier RT analysis method through TDMA clusters' scheduler

end latency) for every SDFG in every cluster is obtained in isolation (without considering other clusters see Fig. 5.15) with the help of our state-based RT method (presented in the Sect. 5.3). In our example in Fig. 5.14, we can first analyze the Worst-Case Period (WCP) of $SDFG1$ and $SDFG2$ belonging to $cluster1$ considering all contentions on the shared bus (for different arbitrations protocols) between the two SDFGs but without considering $cluster2$. Then we do the same for $cluster2$ SDFGs without considering $cluster1$ SDFGs. After that, every cluster is mapped to a slot of a fixed size equal to the maximum of obtained worst-case time among all SDFGs (obtained from the state-based RT method) in this cluster so that it is guaranteed that all SDFGs mapped to this slot are already executed when the slot expires without the need for preemption (see Fig. 5.15). The TDMA scheduler has the role to switch between the slots of different clusters when the slot time of every cluster expires.

Assuming that SDFGs running in one slot are independent from those running in other slots, in order to calculate now the worst-case execution (T_{compos} see Fig. 5.15) of single SDFGs when all clusters are integrated and executed on the MPSoC platform, we take advantage of composability property of such a TDMA based scheduling and can calculate it (similar to Eq. 2.2) using the

following formula:

$$T_{compos} = \sum_{i=0}^{Sl} T_{max}(i) + (Sl \times \mathbf{s}), \tag{5.6}$$

where T can either be the worst-case period or the worst-case end-to-end deadline depending on the timing requirement we are interested in, $T_{max}(i)$ is the maximal T among the SDFGs running in slot i, s is the scheduler worst-case delay time when switching from one slot to another and Sl is the total number of slots.

One possible realization of the above TDMA clusters' scheduler with the help of hardware customized timers can be found in Sect. 6.5.2. Another realization, which we described in [Fakih et al., 2013b], is used in the experiment in Sect. 7.2.5 requires the existence of a resource manager (hypervisor) in the MPSoC which takes care of the temporal and spatial segregation.

5.5 Summary

In this chapter, we have presented the set of flexible and parameterizable timed-automata templates (including event trigger, scheduler, actor, communication driver, interconnect and FIFO buffer TA templates) capturing the MoP of SUA (presented in Chap. 4) and explaining their implementation and abstractions' decisions (see claim C2 in Chap. 1). We have also shown how these flexible TA templates support modeling different issues such as sensitivity to external events, single-beat/burst transfers (with and without the DMA hardware component) on the interconnect (with different arbitration protocols), multi-interconnects extension and multiple storage resources (see claim C2-1). Furthermore, we described how these templates enable us to make a RT analysis with the help of UPPAAL model-checker computing the effects of waiting times due to contention on the interconnect(s) (see claim C2-3). Finally, we examined the state space which should be explored when modeling an MPSoC system with the help of the implemented TA templates, and proposed some optimizations on these templates to minimize the state space (see claim C2-2). In addition, techniques from the literature such as clustering and enabling a TDMA-based composable RT analysis were examined to be useful in terms of improving the scalability of our approach and their adaptation to our system model was described.

Chapter 6

Model-based Design Flow for RT-Analysis of Embedded Applications on MPSoCs

In this chapter, we will elaborate on the integration of different concepts such as enabling the RT analysis of applications modeled in Simulink, automation of our proposed state-based RT analysis method (presented in Chap. 5), and a virtual-platform-in-the-loop V&V simulation technique in a single model-based design flow.

Simulink is a wide-spread commercial tool, supporting hierarchy, domain specific building blocks, functional simulation and automatic code-generation which makes it well-suited for embedded systems design. Since SDFGs and timed automata lack above features and especially the ability of high level building blocks [Srba, 2008], we will extend our design flow for enabling entry models designed in Simulink and discuss the underlying concept allowing the translation of these models to SDFGs (implemented in `SimulinkToSDF` tool).

Furthermore, we will integrate our RT analysis method in a design flow and automatize its steps via a maintainable `SDF2TA` tool.

In addition, our design flow is extended with a simulative method in combination with our state-based RT analysis method. While our approach obtains lower/upper timing bounds of multiple Synchronous Data-Flow Applications (SDFAs) running on an MPSoC, the simulative approach is used for the functional validation of the SDFA implementation and its mapping on the targeted hardware platform. Moreover, the simulative approach could be used to give confidence for the timing values obtained via our state-based RT analysis method. In our proposed methodology, we use a binary-compatible and

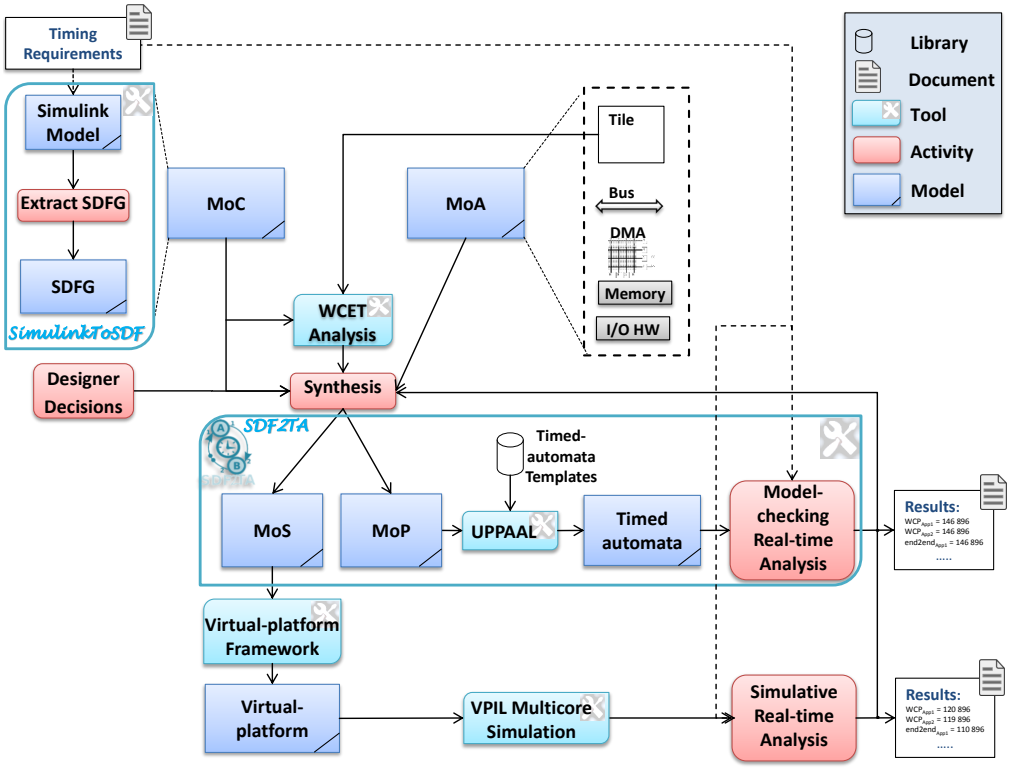


Figure 6.1: Overall model-based design flow

cycle-accurate virtual-hardware platform representation to simulate and map all relevant architectural properties.

6.1 Model-based Design Flow Overview

Fig. 6.1 depicts the overall design flow (extending the X-chart in Fig. 4.1) proposed in this thesis. Typically, the focus of the system designer is on the representation of the application in the proper MoC and mapping it to the available platform resources. The mapping constraints considered in this work depend mainly on the timing requirements. The input of our design flow is an SDF Model of Computation (MoC) and a Model of Architecture (MoA) (defined in Chap. 4). If the functional input model is described in Matlab/Simulink, an SDFG can be obtained from the Matlab/Simulink model using the translation procedure described in Sect. 6.2. For the purpose of automating this step, as highlighted in Fig. 6.1, a `SimulinkToSDF` tool [Warsitz, 2015, Warsitz and Fakih, 2016] was developed. The designer provides the Simulink model (respecting some constraints) and the translation depth to `SimulinkToSDF` which in turn generates automatically an equivalent SDF

graph (in XML format) preserving the structure as well as the precedence/activation relationships of the original model.

The target platform is represented in the MoA by combining tiles (processor with private memories) with other hardware components (DMA, buses, shared storage resources see Fig. 6.1 top right). Afterwards, a WCET analysis (c.f. Sect. 4.2.3) is performed for all combinations of actors and available processing elements of the platform. To enable this timing analysis, C-code implementation of each actor is needed. In the case where the application is available as a Simulink model, C-code can be generated from it using a code-generator (e.g. Simulink Coder). Next, a synthesis activity takes place, which takes mapping and scheduling decisions (manually chosen by the designer) as input, maps all SDF actors to tiles and all SDF edges to communication resources and configures the scheduling/arbitration strategies of resources, resulting into an annotated parallel Hardware/Software model (called Model of Structure (MoS)).

In order to be able to verify that the timing of all mapped SDFGs stay within specified bounds (e.g. WCP: Worst-Case Period), we must keep track of all possible timing delays including delays caused by communication interferences in the MPSoC. To achieve this, a Model of Performance (MoP) is extracted from the synthesis process. The MoP is a network of TA representing all actor WCETs, communication delays, scheduling and communication resource access protocols of the platform (see Chap. 5). Pre-defined TA templates (c.f. Sect. 5.2) are configured and instantiated in the UPPAAL framework, taking into account the mapping, timing and platform configuration. After converting the timing requirements into UPPAAL TCTL queries, performance analysis (e.g. end-to-end deadline) is done using the UPPAAL model-checker. For the purpose of automating these last steps, as highlighted in Fig. 6.1, the *SDF2TA* editor was developed (for details see Sect. 6.3) using the Eclipse Modeling Framework (emf)[1] Ecore model, where the designer can provide all needed parameters (SDFGs, mapping, hardware constraints) and the equivalent UPPAAL system can be generated automatically. If required (e.g. through a failed TCTL query, indicated by a counter example), modifications on the application, the platform or the mapping can be done in order to optimize the performance in case of timing violations. Optimizations could be realized through different approaches e.g. when using Simulink, the generated code can be optimized by replacing default mathematical libraries with optimized and more efficient implementations.

Parallel to the state-based RT analysis of the SUA, a simulative method is approached (see Fig. 6.1). For this, a cycle-accurate virtual MPSoC platform is developed in a certain virtual-platform framework (for e.g. based on SystemC [IEEE-1666, 2012]). After doing that, we are now ready to refine the

[1] http://www.eclipse.org/modeling/emf/

implementation (generated code from Simulink compatible to SDF semantics see Sect. 6.4) into a binary running on the cycle-accurate virtual-hardware platform of the MPSoC to simulate all relevant architectural properties and validate the algorithm implementation and its mapping on the target platform. For this, a virtual-platform-in-the-loop (VPIL) simulation technique (see Sect. 6.4) validates Simulink control applications running on a concrete MPSoC virtual-platform in the loop with the Simulink environmental model preserving the causality of the golden model and enabling a simulation-based validation of their functional requirements. In addition, it supports a non-invasive measurement based method which helps assessing execution times on a cycle-accurate virtual-platform allowing the validation of timing requirements of such applications. These accurate measured values also help bringing certainty to the state-based RT method estimated values. Finally, if the obtained results are satisfying, the developed virtual-platform can be transformed to a real Register-Transfer Level (RTL) model which eases its deployment later on real hardware (not in the scope of this thesis).

In Chap. 4&5 we described the synthesis process and the state-based RT analysis constituting the main part of this design flow. In the following, we will elaborate on the extensions introduced, including the concept behind the translation of Simulink models to SDFGs, automation of our state-based with the help of *SDF2TA* tool and enabling a seamless virtual-platform-in-the-loop (VPIL) simulation in our analysis flow.

6.2 Simulink to SDFGs Translation

Since Simulink (see Sect. 2.2.2) is one of the most wide-spread model-based modeling tool for embedded systems, supporting a data-flow based MoC (c.f. Sect. 2.2.2) and providing many features (such as code-generation, simulation of discrete/continuous systems etc.), we extend our design-flow to support Simulink models as entry models. In order to still enable our state-based real-time analysis of applications modeled in Simulink, a translation from Simulink models to SDFGs is mandatory. In the following, we will present a procedure (mostly taken from our prior work [Warsitz and Fakih, 2016]) which allows us to translate a Simulink model (subjected to some constraints) into an equivalent SDFG.

As already stated (see Sect. 2.2.2), Simulink MoC is much more expressive than the SDFG MoC. Unlike SDFGs, Simulink supports following additional features:

U1 Hierarchy (e.g. *subsystem* blocks): While in Simulink multiple functional blocks can be grouped into a subsystem, in SDFGs each actor is atomic and therefore no hierarchy is supported.

U2 Control-flow logic/Conditional (for e.g. *switch block* or *triggered subsystem* see [MathWorks, Inc., 2015e]): In Simulink control flow is supported on the on block level. This means that depending on the value of a control signal at a block, different data rates could be output by the block. In contrary, in SDFGs data rates at input and output ports of an actor are fixed and control structures are only allowed within the functional code of an actor and can't be represented in an SDFG.

U3 Connections:

1. **Dataflow without connections** (e.g. *Goto/From* blocks): In contrast to Simulink, there is no dataflow without a channel connection in connected and *consistent*[2] SDFGs considered in this thesis.
2. **Grouping of connections** (e.g. *BusCreator* block for *bus* signals): In Simulink, connections with different properties (e.g. different data types) can be grouped into one connection. This is not possible in an SDFG since the tokens transfered among a channel must have the same properties.
3. **Connection style**: While in Simulink the storage of data between blocks has the same behavior as that of a register where data can be overwritten (in case of multi-rate models), the inter-actor communication via channels in SDFGs follows a (data-flow) FIFO buffer fashion, where tokens must be first consumed before being able to buffer new ones.

U4 Sampling rates: In addition to the number of data transported over a connection by every block activation, a periodic sampling rate is assigned to each block in Simulink to mark its periodic activation at this specific frequency. If all blocks exhibit the same sampling periods in a model, then this model is called a *single-rate* model otherwise it is a *multi-rate* model. In SDFGs, however, an actor is only activated based on the availability of inputs. Actors do not have explicit sampling periods and therefore data rates can only be represented by the rates assigned to their (input/output) ports.

Because of the above differences, some constraints must be imposed on the Simulink input model in order to enable its translation to an equivalent SDFG, which we will discuss in the following (mostly taken from our previous work [Warsitz and Fakih, 2016] where a more detailed description can be found).

[2]Inconsistent SDFGs require unlimited storage or lead to deadlocks during execution[Lee and Messerschmitt, 1987a].

6.2.1 Constraints on the Simulink Model

Only Simulink models with fixed-step solver (fixed-step solver is a prerequisite for code-generation) are supported in the translation. In case of multi-rates (which are not in focus of this thesis), rate transitions should be inserted to the Simulink model and the rates should be divisible. These constraints are indispensable to enable code-generation [MathWorks, Inc., 2015d], since we aim with the help of Simulink built-in code-generator to generate SDF compatible executable code for the translated SDFG (see Fig. 6.14). In this thesis, only single-rate Simulink models were evaluated, that is why we constrain the models to be single-rated (where the classes *U3-3* and *U4* of differences between Simulink und SDFGs become no issues anymore). Nevertheless, in [Boström and Wiik, 2015], a procedure based on adding rate-transition blocks to the multi-rate models was described to enabling their translation. Since in our translation we are able to capture these rates and their forward propagation among the blocks, enabling this translation would be a straight-forward extension for our work in the future.

Even though it is possible to translate a Simulink model to multiple SDFGs through our tool, we deal only with one application (implemented in Simulink) at a time in this thesis, which results after translation into one equivalent SDFG. This application is considered to be a control application having the general structure depicted in Fig. 6.14. Moreover, a correct functional simulation of the Simulink model is a prerequisite for the translation in order to get an executable SDFG. In addition to above general prerequisites, the following constraints are imposed on the input Simulink model to enable the translation:

E1 Hierarchy: Hierarchical blocks (e.g. *subsystems*), in which one or more functional blocks of the types described in *U3-1* and *U3-2* exist, are not allowed to be translated to atomic actors. Either these blocks should be removed from the entry Simulink model as they have no functional behavior (for signal forwarding and visualization issues) or the hierarchy level at which these components exist should be dissolved and these blocks should be translated and connected in accordance with the rest of the SDFG. This constraint is mandatory, otherwise if we allow an atomic translation of such hierarchical functional blocks, their contained functional blocks of the form *U3-1* and *U3-2*, which may be connected with functional blocks in different hierarchical levels, would disappear in the target SDFG. A translation of these blocks would thus no longer be possible and would cause a malfunction of the target SDFG (see restriction *E3*).

E2 Control-flow logic/Conditional: Blocks such as *Triggered/Enabled* subsystems can be translated just like the general subsystems. Upon dissolving the hierarchy of such subsystems, the control flow takes place now

within the atomic functionality of the actor without being in contradiction to SDFG semantics (c.f. Sect. 2.2.1.3) . In such a translation, however, additional control channels must be defined (see Sect. 6.2.2). Yet, the case described in *U2* must still be prohibited. In order to do that, there is an option "allowing different data input sizes" in Simulink for such blocks, which when disabled, prohibits outputs of variable sizes of a control block[3]. A special case of these blocks is the powerful stateflow supported by Simulink (see Sect. 2.2.2). In our translation we do not flatten the stateflow block and we always translate it into one atomic actor.

E3 Connections

1. **Dataflow without connections:** For blocks having the same behavior described in *U3-1* (such as From/Goto or DataStoreRead,/DataStoreWrite blocks), we assume that the source block (e.g. DataStoreWrite block), intermediate block (e.g. DataStoreMemory block) and the target block (e.g. DataStoreRead block) which communicate without connections are available in the input Simulink model. This constraint is important as Simulink allows instantiating a source blocks without for instantiating for e.g. the sink block.
2. **Grouping of connections:** In order to support the translation of Simulink models with blocks having the same behavior as those described in *U3-2*[4], two constraints must be imposed. The first one is that every block which groups multiple signals (e.g. BusCreator) into one signal must be directly connected to a block which have the opposite functionality (e.g. BusSelector). The second constraint is imposed on the block (e.g. BusSelector) which takes the grouped signals and splits them again. An "Output as bus" should be prohibited in the options of this block. By doing this, grouping of signals for better visibility in the Simulink model is still with the limitation above allowed, while prohibiting grouping of signals of different parameters in one signal in the target translation.

6.2.2 Translation Procedure

The complete workflow and implementation details of `SimulinkToSDF` tool can be found in [Warsitz, 2015, Warsitz and Fakih, 2016]. In the following, we will roughly describe the procedure implemented (mostly taken from our prior

[3] According to [MathWorks, Inc., 2015e] blocks having this option are: *ActionPort, Stateflow, Enable/Trigger Subsysteme, Switch, Multiport Switch* and *Manual Switch*.

[4] e.g. *BusCreator/BusSelector*, *Bus Assignment* and *Merge* blocks [MathWorks, Inc., 2015e].

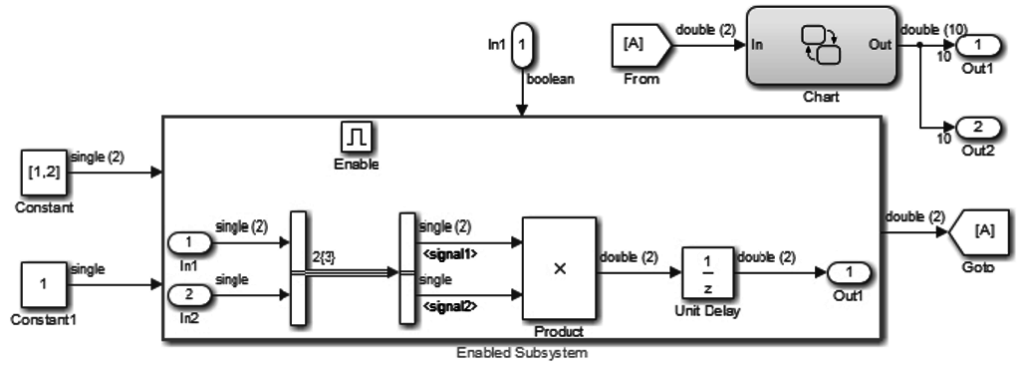


Figure 6.2: Original Simulink model (taken from [Warsitz and Fakih, 2016])

work in [Warsitz and Fakih, 2016]) to extract an SDFG from a Simulink model under the above defined constraints.

1. **Translation of blocks:** If S is the set of all blocks in Simulink model M then each block $s_l \in S$ in M (till the required depth level) is translated into a unique *advanced* actor in the translated SDFG $a_l \in \mathcal{A}^*$ (where $\mathcal{A}^*$ is the set of advanced actors). At this step, an advanced actor (in contrary to the atomic actor known in SDFGs) can still exhibit hierarchy containing other (advanced) actors and channels (see Fig. 6.3).

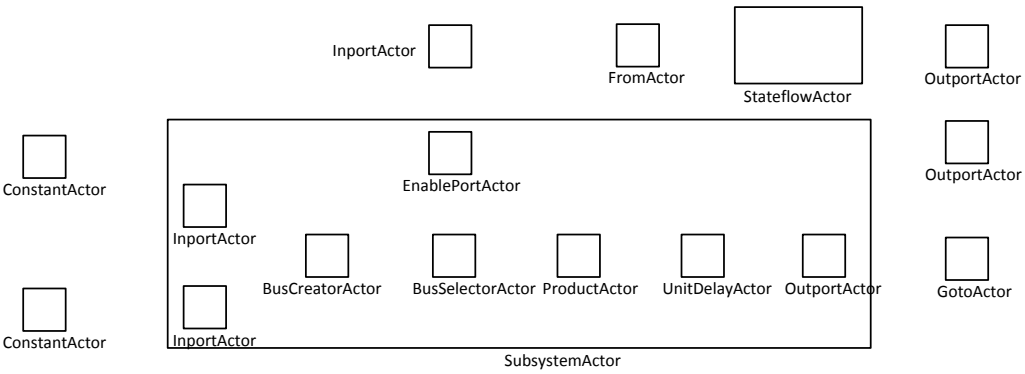


Figure 6.3: Translation of blocks (taken from [Warsitz and Fakih, 2016])

2. **Translation of connections:** Each output port $s_l.o$ is translated into a unique initiator port $a_l.p_i$ and each input port $s_l.i$ is translated into a unique target port $a_l.p_t$. In case multiple connections $t_1, t_2, \cdots, t_n$ going out from an output port p_{o1} in Simulink (which is permitted in Simulink see connections of statechart before translation Fig. 6.2, but not in SDFGs see connections of statechart after translation Fig. 6.4), then for each one

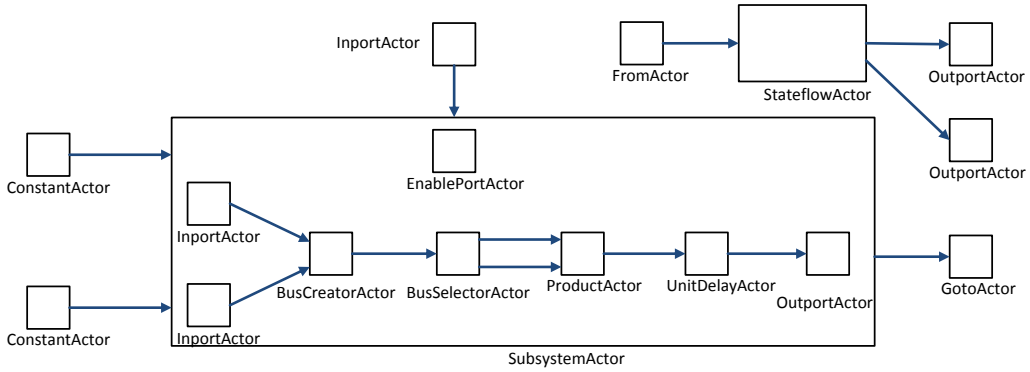


Figure 6.4: Translation of connections (taken from [Warsitz and Fakih, 2016])

of these connections, the output port is replicated $p_{o11}, p_{o12}, \cdots, p_{o1n}$ (each having the same properties) in the resulting SDFG, in order to guarantee that every edge $d \in \mathcal{D}$ has unique target and initiator ports. Now, each connection $t \in M$ in the Simulink model is translated into an edge $d \in \mathcal{D}$ (see Fig. 6.4).

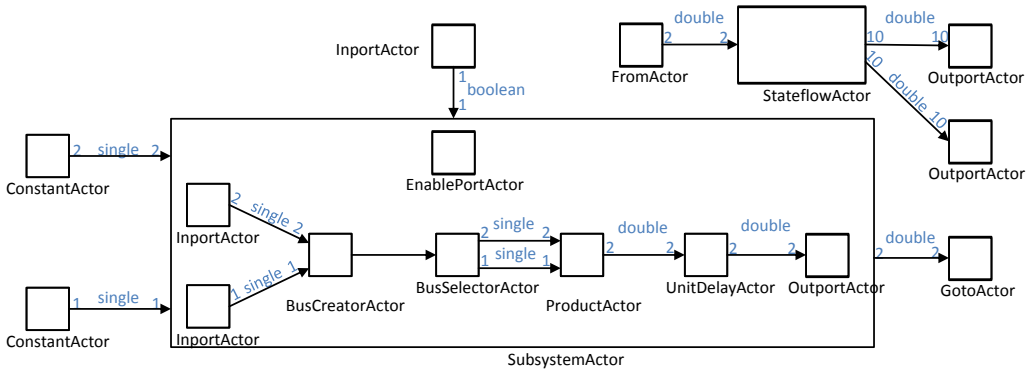


Figure 6.5: Propagation of number of data to be transfered, datatypes and sampling rates (taken from [Warsitz and Fakih, 2016])

3. **Propagation of number of data, datatypes and sampling rates:** In this step, the number of data transfered over a connection, the datatype and the sampling rates are obtained for every block/port/connection (see Fig. 6.5, all blocks having same sample rate of 1, data types are either single or double and number of data transported ranges between 1 to 10). Since we only consider single-rated models in this thesis, the rates of initiator and target ports are always equal. The sampling rates are propagated according to the forward-propagation technique supported by Simulink (for more details refer to [Warsitz, 2015]).

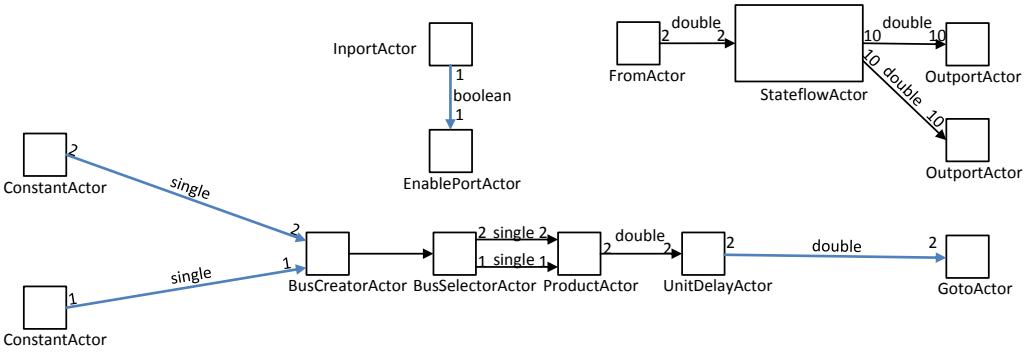


Figure 6.6: Dissolving hierarchy (taken from [Warsitz and Fakih, 2016])

4. **Dissolving hierarchy:** In this step, a top-down flattening of the Simulink model (respecting *E1*), till the required depth level is reached, is done (see Fig. 6.6).

5. **Inserting delay tokens:** In this step, actors representing delay blocks (e.g. *Unit-Delay* block) are removed and an equivalent number of delay tokens are then inserted on the corresponding channels (see Fig. 6.7). Alternatively, these blocks can be transformed to equivalent actors which implement the delay behavior in their internal process (functionality).

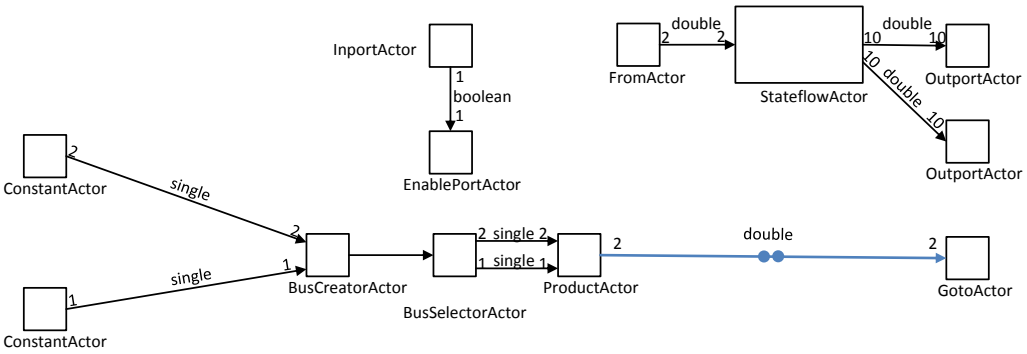


Figure 6.7: Translating delay blocks (taken from [Warsitz and Fakih, 2016])

6. **Removing connecting blocks of type U3-1:** Goto/From, DataStore or other similar blocks are removed in this step. When doing this, the predecessor block of the source block (e.g. DataStoreWrite block) is directly connected either to the intermediate (if existent) block (e.g. DataMemory block) or to the successor block of the target block (e.g. DataStoreRead block) and these connecting blocks (source and target blocks) are removed (see Fig. 6.8 where Goto/From blocks are removed).

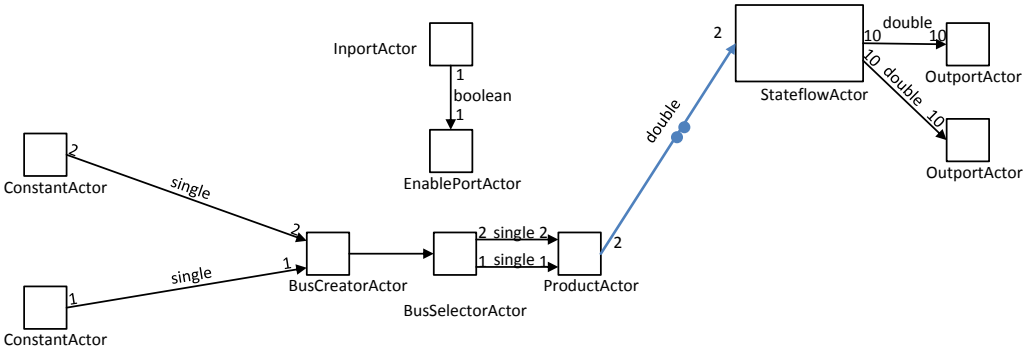


Figure 6.8: Removing data-flow blocks without connection of type *U3-1* (taken from [Warsitz and Fakih, 2016])

7. **Removing connecting blocks of type U3-2:** During the translation process, blocks respecting the *E3-2* constraint are simply removed and the predecessors' blocks are connected with the successors' blocks (see Fig. 6.9 where BusCreator/BusSelector actors are removed).

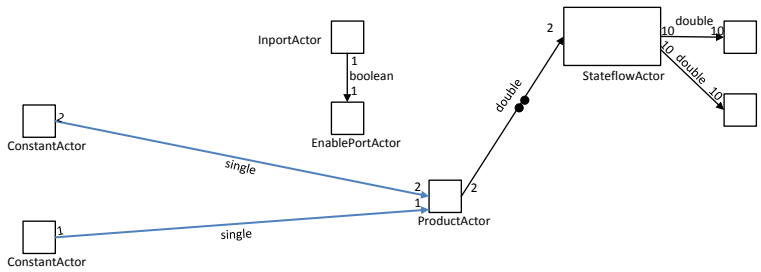


Figure 6.9: Removing blocks of type *U3-2* which group connections (taken from [Warsitz and Fakih, 2016])

8. **Adding event channels:** In this step, channels for handling (enabling/triggering) events are added. These edges are needed when the hierarchy of a enabled/triggered subsystem is dissolved. In this case, each actor, belonging to the *triggered* or *enabled* subsystem has to be sensitive to the (triggering/enabling) event and thus is connected with the event source (see Fig. 6.10).

9. **SDFG representation in XML:** After the above translation steps, we obtain an SDFG representation compatible to Def. 4.2.3. `SimulinkToSDF` outputs the result in XML format which can be imported by `SDF2TA` see Sect. 6.3 and graphically plotted using the `Dot` [Gansner et al., 2015] tool (see Fig. 6.11).

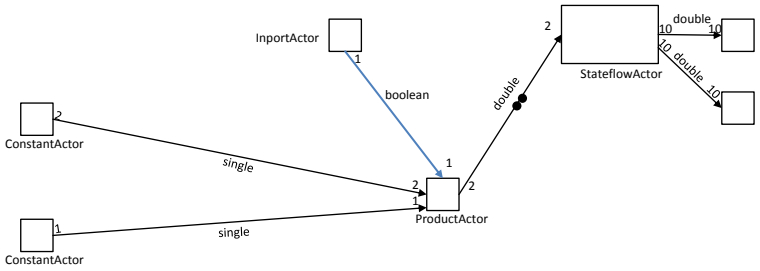


Figure 6.10: Addition of event channels (taken from [Warsitz and Fakih, 2016])

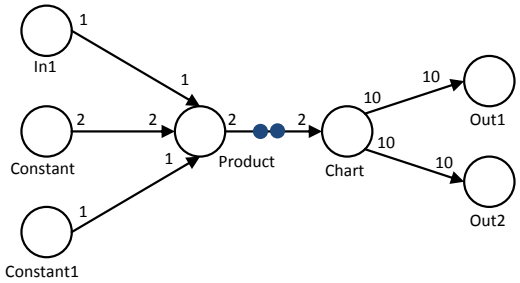


Figure 6.11: Resulting SDFG (taken from [Warsitz and Fakih, 2016])

Finally, the actors in the resulting SDF graph can be statically scheduled to obtain a minimal periodic admissible sequential schedule (see Sect. 2.2.1.1).

After translating the Simulink model to an equivalent SDFG, we are now able to generate C/C++ code (with the help of Simulink Embedded Coder) for every block in the reference Simulink model, corresponding to an actor in the resulting SDFG. The generated code can now be (manually) customized to respect the execution semantics of the output SDFG resulting from the translation (see pseudo-code implementation of SDFGs in Sect. 6.5). With the help of our VPIL verification and validation technique (presented in Sect. 6.4), we can verify whether or not the functional semantics of the executable, SDF compatible code of the translated SDFG performs the same as the reference Simulink model.

6.3 Automation of our State-based RT Approach

For the purpose of automating the highlighted steps in Fig. 6.1, the `SDF2TA` editor (first version developed in [Schlaak, 2014]) was developed using the Eclipse Modeling Framework (EMF)[5] Ecore model, where the designer can provide all necessary parameters (SDFGs, mapping, hardware constraints) and the equivalent UPPAAL system is generated automatically. The Ecore format is an XML-like format which has the ability to automatically generate equivalent XML,

[5]http://www.eclipse.org/modeling/emf/

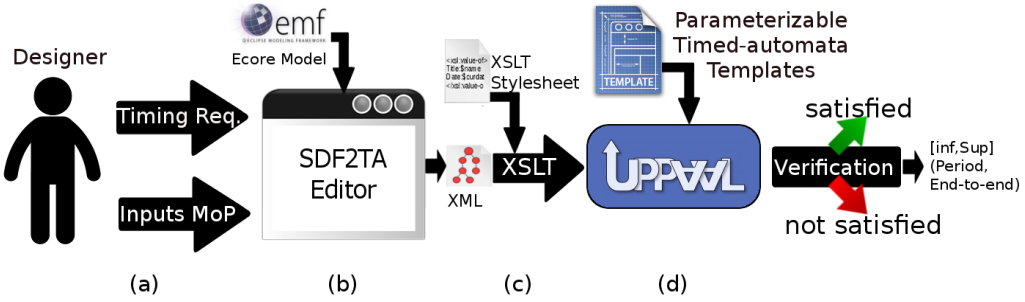


Figure 6.12: Work flow of the SDF2TA tool (based on [Schlaak, 2014])

UML (see the SDF2TA UML diagrams representing the MoP in Appendix A.2) and Java code.

Fig. 6.12 shows the work flow of the SDF2TA tool. First, the designer needs to provide all relevant timing properties of the SUA as an input to the SDF2TA editor. Since the SDF2TA editor is already provided with an Ecore model of the supported MoP semantics, it can validate whether or not the designer input conforms with the model definition (see Appendix A). If the input validation step is successful, the developer can now choose some property to be checked for the given MoP. Properties that can be checked (refer to Sect. 7.1 for more details) are either *timing properties* such as the period, end-to-end deadline or *liveness properties* such as checking whether or not the repetition vector of an SDFG is valid or if specific states can be reached (for e.g. is finishing of sink actors always guaranteed to be after finishing source actors for all possibilities) during execution.

After doing this, the SDF2TA generates an XML equivalent representation of the input model and calls for an XSLT (Extensible Stylesheet Language Transformations) processor which processes it by applying pre-defined XSLT rules and transforming it into an equivalent UPPAAL XML representation. This step enables the parametrization and instantiation of our pre-defined UPPAAL timed-automata templates (see templates' description in Sect. 5.2), which represent different components of the MoP. At the end, the generated UPPAAL model is checked against TCTL queries (which are generated depending on the designer requirement input) using the verifyta[6] tool. Finally, the verification results are provided to the developer through the SDF2TA editor.

SDF2TA editor (see Fig. 6.13) provides a user-friendly GUI, supports tool-tips and validates input models in order to generate a correct timed-automata model. Furthermore, the usage of Ecore as the common modeling language for capturing the MoP makes SDF2TA maintainable. If any extensions or changes

[6]This is the stand-alone UPPAAL timed-automata verification tool.

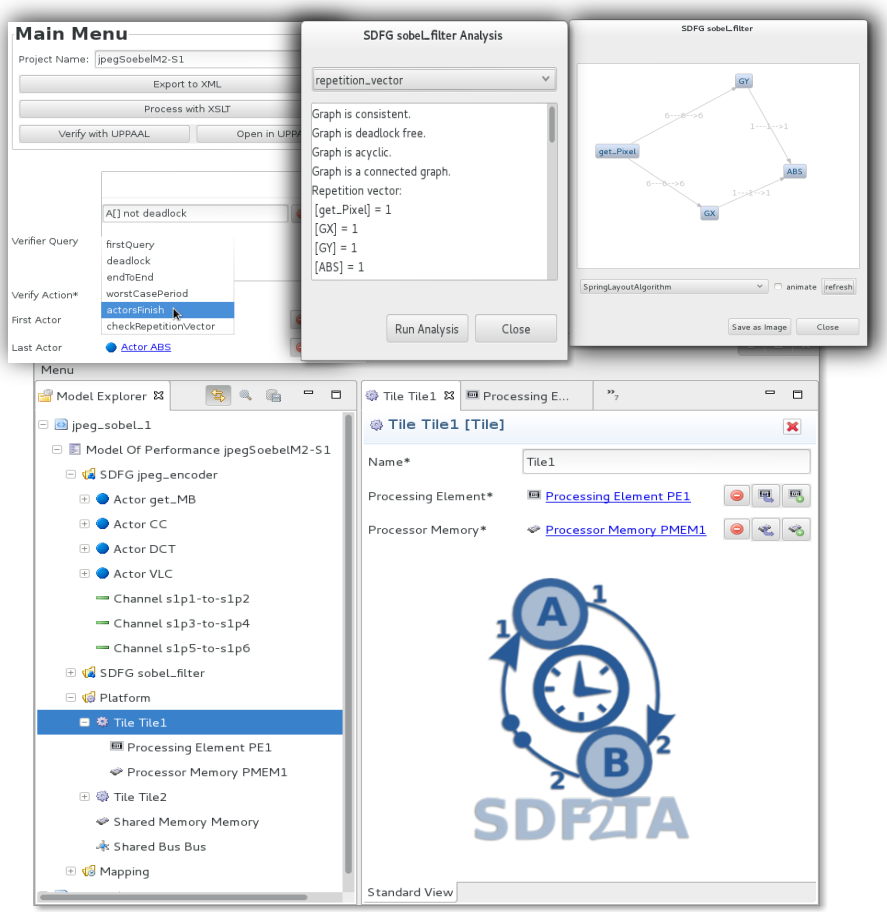


Figure 6.13: SDF2TA GUI

should be done on the MoP, the developer only needs to change the Ecore model and with minimal additional effort, modified the SDF2TA editor is generated. In addition, our SDF2TA editor allows to import SDFGs from the well known SDF^3 tool [Stuijk et al., 2006], and is able to run SDF^3 in the background to perform different analysis of the SDFGs created in SDF2TA, such as finding and adapting values of repetition vector, buffer sizes, etc.

6.4 Virtual-Platform-in-the-Loop Simulation for MPSoCs

6.4.1 Motivation

Although moving to higher level of abstraction, as in the case of MBD in Simulink, allows the designer to easily model the system and verify it's functionality, the abstraction of non-functional aspects like timing behavior can lead to severe problems late in the development process causing for expensive re-

designs. Even worse, such issues could stay undetected and become a safety hazard. Virtual-hardware platforms (see Sect. 2.5.1) propose in this context an efficient way to close the gap between the high level model and the targeted embedded architecture for functional and non-functional V&V. Also exact timing verifications can only be applied if the timing behavior of the hardware can be analyzed (such as bus latencies). For this purpose, we use a cycle-accurate virtual platform (VP) of the MPSoC. Using a virtual-hardware platform (VP) instead of the development board is very beneficial in terms of fast software execution and better debugging capabilities during functional implementation. Especially for new hardware platforms, (like the Aurix TriCore used in experiments Sect. 7.4), instead of waiting for the first engineering samples to be developed, manufactured, and delivered, the VP enables early V&V of the SW implementation together with hardware design.

With the help of Virtual-Platform-in-the-loop (VPIL) simulation, we are able to combine the benefits of the MBD (see Sect. 2.2) and VP (see Sect. 2.5.1). It provides a simple and fast approach to handle the complexity of embedded systems design and allows validating their timing requirements at early design phases enabling targeted design decisions, minimizing costly redesigns, improving development effort, cost and time to market, and thereby helping to design safer systems. It is important to note that the VPIL simulation is an intermediate step towards Hardware-In-the-Loop (HIL) simulation (as we have done in [Walter et al., 2014]). In this thesis, we have also used the VPIL simulation as a mean of validating our model-checking approach.

6.4.2 Bi-simulation Procedure

VPIL simulation is a validation and verification (V&V) technique which we first developed in [Fakih, 2011] for single-processor platforms and then we extended it throughout this work [Fakih and Grüttner, 2012] for supporting SDFGs running on MPSoCs. The VPIL simulation targets data transfer between the VP framework and Simulink allowing following benefits:

1. Build complex test environment in Simulink and reuse of test-cases developed for the implementation model,

2. On-the-fly comparison of the functionality of the reference (golden) controller model and that of the generated controller code running on a virtual-target platform when subjected to the same test-scenarios,

3. Non-invasive timing measurement of the synthesized platform (HW/SW) at a cycle-accurate level for timing requirements validation and the visualization of the results in Simulink.

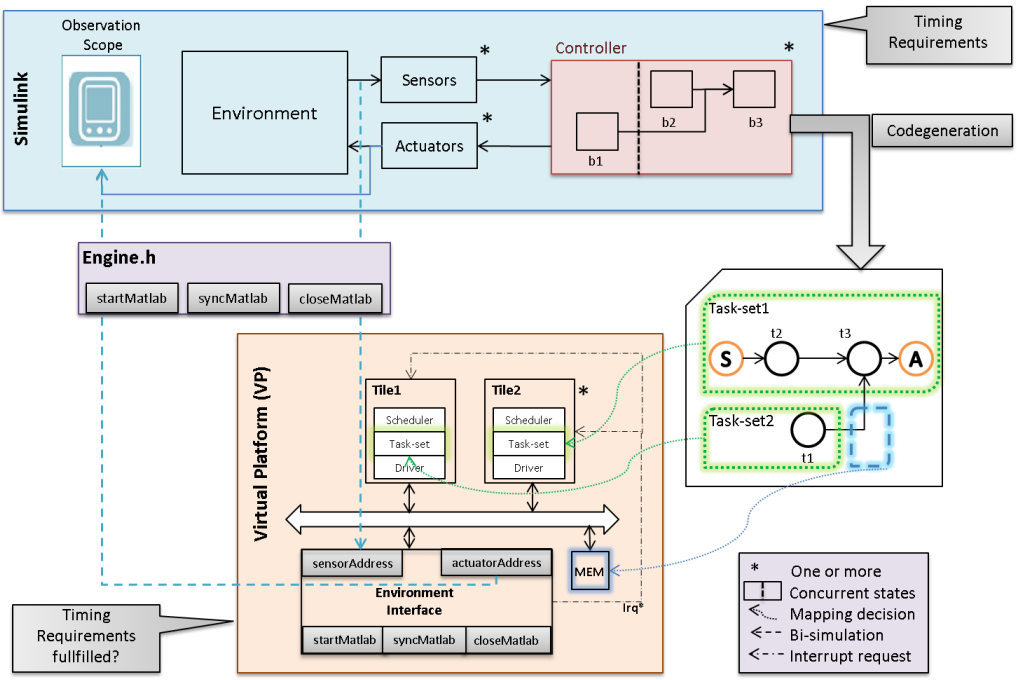


Figure 6.14: VPIL simulation for MPSoCs

Complaint to the constraints made in Chap. 4 on the MPSoC, we made the assumption that only one dedicated tile (I/O tile) is allowed to access I/O devices (in the following denoted by *Environment Interface*[7]). Moreover, we assume that the VP software uses a classic bare-metal approach i.e. only manufacturer-supplied software framework is used which provides basic system management and drivers with no explicit real-time operating system being deployed. In addition, we assume single-rated Simulink models as the use-cases evaluated were single-rated use-cases. Yet extending the VPIL simulation technique for multi-rated Simulink models should be straight forward (see suggestion in [Fakih, 2011]).

Fig. 6.14 shows an overview of our VPIL simulation framework. As it can be seen, the functional model can consist of several sensors and actuators and possibly multiple controllers that control the same environment model (or the process to be controlled). After defining functional and non-functional requirements of the control system to be developed, the next step is to model it in Simulink. The controller(s) can be implemented as a combination of Simulink blocks that must be supported by the code-generator (as already described in Sect. 6.2). At this level, the control algorithm can be partitioned into con-

[7] An interface responsible to connect the controller with the environment modeling sensors' and actuators' interfaces

currently executing blocks in Simulink (see `b1, b2, b3` in Fig. 6.14). After verifying and validating the functionality of the controller model within Matlab/Simulink, we are able to generate target C code from the Simulink control models (with the help of Simulink Coder (R2011-b) [MathWorks, Inc., 2015a]. This code is then manually customized with the help of a lightweight SDF library (see pseudo-code in Algorithm 2) to make it compliant to the SDF semantics of the translated SDFG (translation is done according to procedure described in Sect. 6.2). The result would be the task-sets in Fig. 6.14 which represent the implementation of the *SO* list defined in Def. 4.2.9. The additional sensor and actuator actors (denoted by `S` and `A` in Fig. 6.15) represent the source and the sink actors of the translated SDFG (see Sect. 6.2), which are responsible of communicating with the *Environment Interface* getting sensors and updating actuators. After implementing the SDFG actors and configuring the drivers for a chosen inter-processor communication, the code is cross-compiled and can be executed on the target processors and we are now ready to start the bi-simulation of our control software on the target VP with Simulink. But since we want to validate our control software implementation while interacting with the environment behavior, an environment model is still missing. This is where our Virtual-Platform-In-the-Loop (VPIL) V&V technique comes in, to fill this gap and realize a link to the top-level Simulink environment (golden) model. Fig. 6.14 shows the *Environment Interface* IP component in the virtual-platform we developed to realize the communication with the functional Simulink model allowing the bi-simulation of VP implementation and Simulink, getting sensor values from the environment model (at Simulink-level) and at the same time supporting the non-invasive execution-time measurement of the generated code. Whenever the target processor gets an interrupt[8] signal from the *Environment Interface* (signalizing the availability of sensor data), the corresponding task-set is executed and runs to completion.

Finally the measured execution-time values with the functional outputs of the control software are send back to Simulink were they plotted (together with reference values in an observation scope see Fig. 6.14) and can be analyzed and evaluated w.r.t to the functional and timing requirements. In the case of functional mismatch, the implementation on the virtual-hardware platform must be examined and corrected until the expected behavior is reached. In the case of timing violation detection, the following actions can be performed: modifications in the functional model (e.g. from floating point to fixed-point calculation or other controller algorithm), changing the mapping or changing the platform architecture (e.g. faster cores, faster interconnect, etc.).

[8] Alternatively to interrupt-based synchronization, the sensor actor can poll (as we will see in the experiments) on the specific addresses in the *Environment Interface* to check if new values are available from Simulink environment model. In case of polling, extra traffic is induced on the interconnect which makes the timing measurement (see Fig. 6.16) invasive.

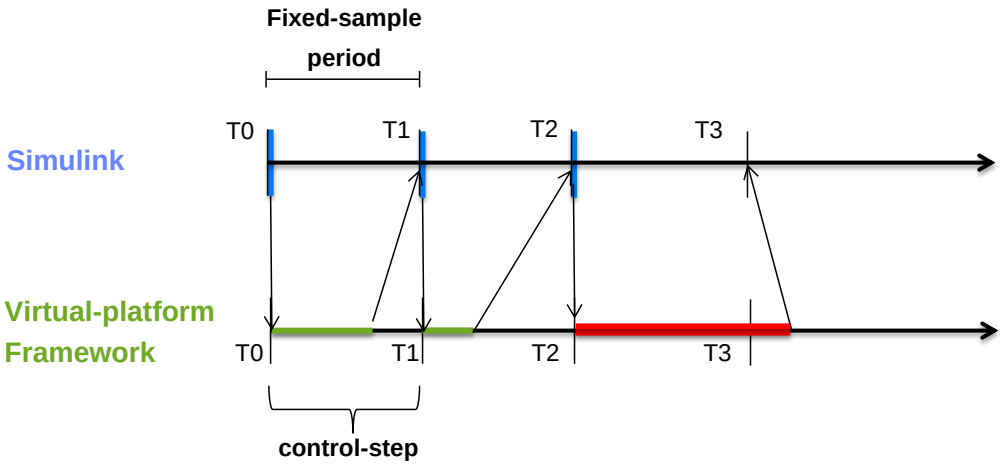


Figure 6.15: Bi-simulation procedure of Simulink and VP Framework (taken from [Fakih, 2011])

The basic idea behind the bi-simulation with the non-invasive timing measurement in our VPIL simulation is shown in Fig. 6.15. In a Simulink simulation with a fixed-step solver type, models are stepped periodically according to a fixed sampling period called *fixed-step-size* (this variable can be specified in the simulation configuration). The simulated time proceeds until the *fixed-step-size* value is reached and updates the model at this moment (takes an input, makes the internal computation and produce an output) according to a specified solver[9]. We define a *control-step* as a one such update of the controller model in a given period which also corresponds to one update of its generated code. In this update one execution of the generated step functions[10] of all task-sets of the partitioned controller including the communication and synchronization between them is done. Our VPIL simulation follows a lock-step based schema (as shown in Fig. 6.15) where one control-step in Simulink is executed and then the same control-step in the VP framework is executed. The virtual-platform framework (for e.g. COMET/Virtualizer [Synopsys Inc., 2015] in experiments in Sect. 7.4) is the master of the bi-simulation. It starts by initializing all virtual-platform components and then launches an instance of the Simulink model (with the help of Mathworks `engine.h` API [MathWorks, Inc., 2015b] used for data exchange). The basic idea of the *non-invasive* timing measurement method, is to measure the execution of every single control-step of needed to finish a control scenario (specified as a *test-case* control scenario). In Fig. 6.15, we see

[9] A solver implements a specific numerical integration technique to interpolate values between two consequent time instants of simulation.

[10] The code-generator generates for every block a step function which should be executed according to the given period. When executed it takes the input values performs an update of the controller (in Simulink) and produces its output values.

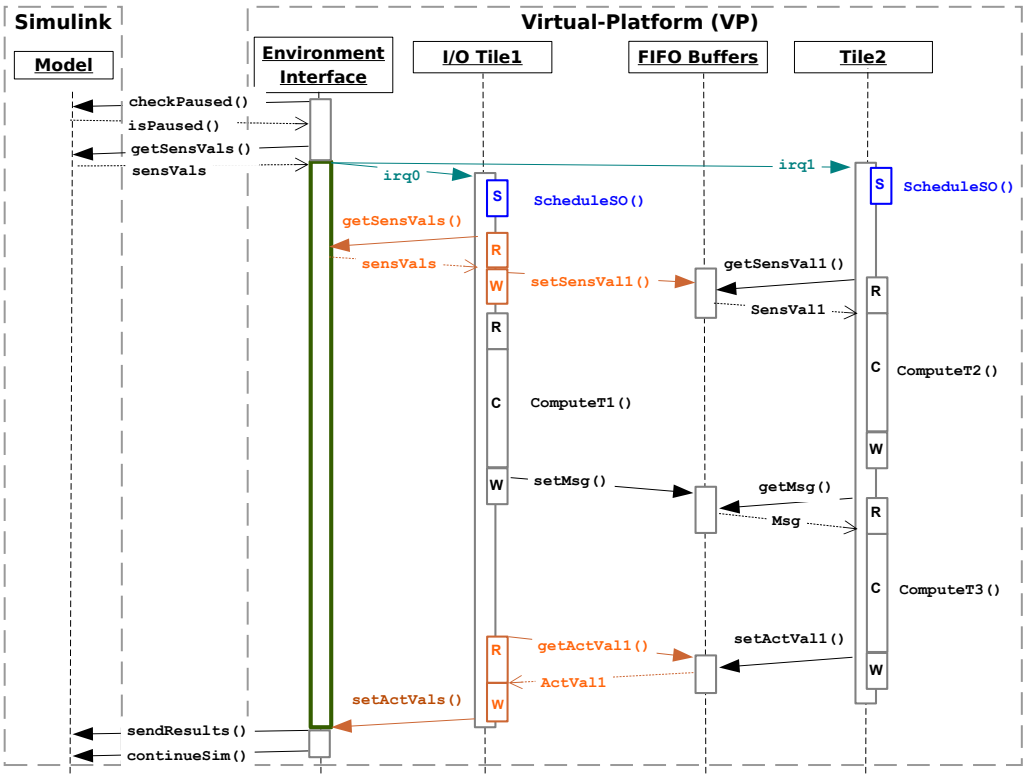


Figure 6.16: Sequence of execution in one control-step within a VPIL simulation

that in every period (*fixed-step size*) a different control-step is executed and for which different number of processor cycles are measured. In Simulink the execution of every control-step happens instantaneously (blue vertical lines). The execution time of the same control-step in the VP is then measured (green/red horizontal line). If the single duration (end-to-end latency of the SDFG) of any control-step exceeds the *fixed-step size* (see Fig. 6.15, at T2 red/fat horizontal line) then we have detected a timing violation. Another timing violation can be detected if the finish time of some control scenario execution (which is the summation result of durations of all control-steps of this scenario) violates the end time requirement for this scenario. It is important to note here that even if any control-step lasts longer than the upper-bound time requirement, the target processor still executes it till its end (according the VPIL simulation procedure). This means that timing violations would not manipulate the time instants at which the actuator values are updated and thus does not have any influence on the measured functional results of the controller.

In Fig. 6.16, the sequence diagram shows in detail what happens within a control-step (highlighted in green/fat activation block of the Environment Interface lifeline) for the example in Fig. 6.14. First, the Simulink model steps once

and then it is paused (with the help of Simulink customized block). Then, the *Environment Interface* component in the VP detects this pause (`isPaused()`), gets the current sensor values (`getSensVals()`) corresponding to this step from the Simulink environment model and issues interrupts to notify the tiles (`irq0, irq1`). On the tiles, the schedule function (for e.g. `ScheduleSO()`) is executed and tasks (or executable actors) are activated to execute according to the SDF execution semantics. On the dedicated I/O *tile*1, sensor data are read (`getSensVals()`) from the *Environment Interface* through the sensor actor (highlighted in orange/light activation block of I/O Tile1 lifeline in Fig. 6.14). Likewise, the actuator actor (highlighted in orange in Fig. 6.14) on *tile*1 updates the actuator data by writing (`setActVals()`) to the *Environment Interface*. Every task, when activated (see actor execution phases in Fig. 4.3), executes its step function (except for the sensor and actuator actors) once in the compute phase (for e.g. `ComputeT1()`). During software execution, the *Environment Interface* component records the end-to-end execution time which is the time from the moment where the sensors data were received until the moment where the last actuator was updated including the inter-processor communication and synchronization at a cycle-accurate level. After the execution of the control-step at the VP level has been completed (green/thick lifeline of sequence diagram in Fig. 6.16), the *Environment Interface* sends the updated actuator values and the timing measured values (`sendResults()`) back to Simulink. It also wakes up Simulink to resume the execution (`continueSim()`) of another control-step. This procedure can be iterated until the desired number of control-steps are executed. An elaborated description of the VPIL methodology with an application on an academical use-case can be found in [Fakih, 2011, Fakih and Grüttner, 2012].

6.5 Implementation Concepts

In the following, some implementation concepts of the SDFGs and their scheduling algorithms are are given in a simplified pseudo-code form in order to show how such systems, compatible to our constraints in Chap. 4, can be implemented on a real platform while remaining analyzable by our state-based RT analysis method. For a complete code excerpt of a basic SDF library implementation in C please refer to [Schaumont, 2013]. For the same purpose mentioned above, we also discuss some major issues of the communication driver considered in this thesis.

6.5.1 Pseudo-code of Static-order Scheduled SDFG

In Listing 1, the pseudo-code of self-timed static-order SDFGs with auxiliary functions is depicted. We notice that when actors are blocked they wait some

time (`WAIT(t)`) and then they go to a *Suspended* (`return false`), which also results in suspending the SDFG (`return ACTOR_SUSPEND`), the fact which will be registered by the SDFG scheduler (see Algorithm 2 and Algorithm 3). In case of suspension, actor's progress information are saved locally (`SAVE_CONTEXT`).

6.5.2 Pseudo-code of SDFGs Schedulers

If an static-order (SO) scheduler is chosen for SDFG scheduling (see Algorithm 2), the scheduler activates the first SDFG in the ordered list. if the SDFG execution is finished successfully (`SDFG(sdfg_vector[i])==SDFG_FINISH`) then the scheduler executes the next SDFG in the list. If some actor of some SDFG blocks, the SO scheduler simply reactivates the same SDFG which retries to run the same actor (after waiting for some time).

In the case of Round-Robin (RR) SDFG scheduler (see Algorithm 3), in both cases where the actor blocks or finishes execution successfully, the scheduler switches to the next SDFG and executes it, achieving more fairness among SDFGs in the system.

Algorithm[11] 4 depicts a pseudo-code of an interrupt-based implementation of the TDMA scheduling algorithm (suggested in Sect. 5.4.3 with an example shown in Fig. 5.14). The input is a sorted list of partial SDFGs (i.e. an ordered set of actors) per tile and TDMA slot, the duration of each TDMA slot and the duration of the scheduler "slot". The duration of each slot is analyzed statically, as described in Sect. 5.4.3. The `main` routine on tile 1 initializes the two timer channels. Timer channel 1 is configured with the duration of the scheduling slot and channel 2 is configured with the duration of the next TDMA slot to be executed. Each timer starts counting down immediately after its configuration and triggers an interrupt when the timer register hits zero. The interrupt lines of each timer channel can be logically OR'ed and connected to each tile's interrupt port (see Fig. 5.14). Each tile has an interrupt service routine (`ISR`) which is executed when the tile's interrupt occurs. The `ISR` implements a simple state machine to detect and handle the start and end of each TDMA slot. The `start` state executes the scheduling algorithm on the list of actors per tile. This can either be an SO (see Algorithm 2) or an RR (see Algorithm 3) scheduling. The `end` state switches to the next slot (including wrap around) and sets the next slot duration (only done by the ISR of tile 1).

6.5.3 Communication Driver Issues

In the following, we will elaborate on the communication driver basic structure (considered in this thesis) assuming that the SDFGs' channels are mapped (for

[11] Both algorithm 2 and Algorithm 3 (each algorithm mainly includes 3 nested loops) have a run-time complexity of $O(n^3)$. Algorithm 4 has a run-time complexity of $O(n^4)$ since it includes one of the above algorithms and execute it n-times depending on the number of slots.

Algorithm 1 Self-timed static-order SDF execution

```
1: channel[m]                                              ▷ FIFO buffers of SDFG j
2: max_size[m]                                             ▷ Sizes of FIFO buffers of SDFG j
3: suspendedWrite[l]                                       ▷ resumes flags of actor l
4: activePort[a]                                           ▷ active port of Actor a
5: procedure CanPut(c,n)
6:     if num_elements(channel[c]) + n > max_size[c] then
7:         return false
8:     else return true
9: procedure CanGet(c,n)
10:    if num_elements(channel[c]) − n < 0 then return false
11:    else return true
12: procedure Compute(id)                                  ▷ executes actor's behavior
13: procedure Wait(t)                                      ▷ polling delay of t time units
14: procedure SaveContext(actor, port, isWrite)
15:    suspendedWrite[actor] = isWrite
16:    activePort[actor] = port
17: procedure Produce(c,n)
18:    if not CanPut(c,n) then
19:        Wait(t) return false
20:    else
21:        enqueue(channel[c], n) return true
22: procedure Consume(c,n)
23:    if not CanGet(c,n) then
24:        Wait(t) return false
25:    else
26:        dequeue(channel[c], n) return true
27: procedure Actor(id)                                    ▷ executes actor with index id
28:    i ← 0
29:    if SuspendedWrite[n] then
30:        while i < MAX_INITIATOR_PORTS[id] do
31:            if Produce(i,n) then
32:                i ← i+1
33:            else SaveContext(id, i, false) return false
34:    else
35:        while i < MAX_TARGET_PORTS[id] do
36:            if consume(i,n) then
37:                i ← i+1
38:            else SaveContext(id, i, true) return false
39:        Compute(id)
40:        i ← 0
41:        while i < MAX_INITIATOR_PORTS[id] do
42:            if Produce(i,n) then
43:                i ← i+1
44:            else SaveContext(id, i, W) return false
45:    return true
46: procedure SDFG(j)                                      ▷ executes SDFG with index j
47:    i ← 0
48:    while i < MAX_SDFG_ACTORS[j] do
49:        if not Actor(i) then return ACTOR_SUSPEND
50:        else i ← i+1 return ACTOR_FINISH
51:    return SDFG_FINISH
```

Algorithm 2 Static-Order SDFG Scheduling

```
1: Input Sorted list of SDFGs to be executed on a tile t (SDFG_list[t]) and
   length of this list (SDFG_list_size[t])
2: Result Execute SDFGs in a static order
3: procedure ScheduleSO(sdfg_vector, sdfg_vector_size)
4:     i ← 0
5:     while i < sdfg_vector_size do
6:         if SDFG(sdfg_vector[i]) == SDFG_FINISH then
7:             i ← i+1
8: procedure MAIN                                  ▷ Main routine of Tile t
9:     loop                                                  ▷ Infinite loop
10:        ScheduleSO(SDFG_list[t], SDFG_list_size[t])
```

Algorithm 3 Round-Robin SDFG Scheduling

```
1: Input Sorted list of SDFGs to be executed on a tile t (SDFG_list[t]) and
   length of this list (SDFG_list_size[t])
2: Result Executes SDFGs in a static order but switches to next SDFG, if cur-
   rently executed SDFG is blocked
3: procedure ScheduleRR(sdfg_vector, sdfg_vector_size)
4:     i ← 0
5:     while i < sdfg_vector_size do
6:         if SDFG((sdfg_vector[i])) == (ACTOR_FINISH or ACTOR_SUSPEND)
   then
7:             i ← i+1
8: procedure MAIN                                  ▷ Main routine of Tile t
9:     loop                                                  ▷ Infinite loop
10:        ScheduleRR(SDFG_list[t], SDFG_list_size[t])
```

e.g. in Fig. 6.17) to the shared storage resource and must be accessed through the interconnect. A similar procedure is done if these channels are mapped to the private storage resources, but in a simpler manner without the need of the interconnect protocol. We also differentiate between basic communication drivers for interconnects with single-beat or with built-in burst transfer capabilities (for e.g. AHB-Bus which can perform up to 16 beats [Kesel, 2012]) and other more complex drivers utilizing the DMA hardware component to realize a burst transfer.

Basic Communication Driver Fig. 6.17 shows the activity diagram of the *write* phase (see `PRODUCE` in Algorithm 1) of an actor, pointing out at which entry point the communication driver is called. The activity diagram of the *read* phase can be constructed in a similar manner to that of *write* phase with some minor differences (see `CONSUME` in Algorithm 1).

Every time an actor requires to write to a channel, it must first check

Algorithm 4 TDMA Scheduling

```
1: SDFG_list[t][s]                          ▷ Sorted list of SDFGs to be executed per Tile & Slot
2: SDFG_list_size[t][s]                     ▷ Size of each SDFG_list entry
3: scheduler_duration                       ▷ TDMA scheduler WCET
4: slot_duration[s]                         ▷ Duration of each TDMA slot s
5: enum State{start, end} slot_pos[t]       ▷ Slot position per Tile t
6: slot[t]                                  ▷ Current active slot per Tile t
7: procedure INIT(t)                        ▷ Initialization of Tile t
8:     if t == 1 then
9:         setTimer(channel1, scheduler_duration)
10:        setTimer(channel2, slot_duration[0])
11:    slot_pos[t] ← start
12:    slot[t] ← 0
13: procedure ISR(t)                        ▷ Interrupt Service Routine of Tile t
14:    if slot_pos[t] == start then
15:        list ← SDFG_list[t][slot[t]]
16:        list_size ← SDFG_list_size[t][slot[t]]
17:        Schedule[SO|RR](list, list_size)     ▷ SO or RR scheduling
18:        slot_pos[t] ← end
19:        if t == 1 then
20:            clear_timer_interrupts()
           return
21:    if slot_pos[t] == end then
22:        if slot[t] < MAX_SLOTS-1 then
23:            slot[t] ← slot[t] + 1
24:        else
25:            slot[t] ← 0
26:        if t == 1 then
27:            setTimer(channel2, slot_duration[slot[t]])
28:        slot_pos[t] ← start
29:        if t == 1 then
30:            clear_timer_interrupts()
           return
31: procedure MAIN                          ▷ Main routine of Tile t
32:    init(t)
33:    loop                                 ▷ Infinite loop
```

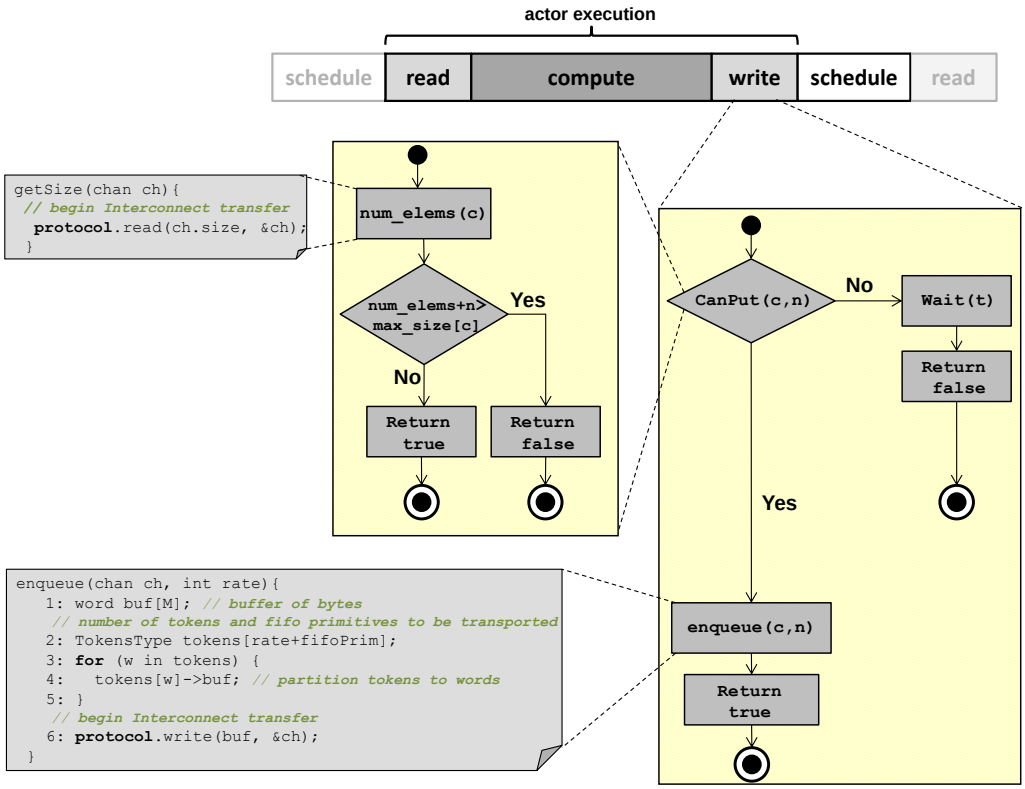


Figure 6.17: Communication driver's entry calls for a `Write` access

if there is enough buffer capacity available for this access. To do this the `size` attribute of the buffer (located in the shared storage resource) should be read (`getSize()`). In this function, an interconnect read access is issued according to the interconnect protocol (`protocol.read()`). Now, if there is enough buffer capacity for writing the tokens (`num_elems+n`$\leq$`max_size[c]`) then an `enqueue` function is called. In this function, the number of tokens to be transported are first converted into an untyped ordered byte streams having the bitwidth of smallest addressable unit (in our case equal to interconnect width c.f. Sect. 5.2.4) [Gajski et al., 2009]. In addition endianess [Gajski et al., 2009] is also handled at this level. The number of tokens (line 2 in `enqueue()`) consists of the actual tokens to be transported depending on the current port's rate of the actor plus other FIFO auxiliary variables (`fifoPrim`: FIFO implementation specific primitive variables such as the `size` variable) which should be updated. Afterwards, an interconnect write access is launched according to the interconnect protocol (`protocol.write()`). This access could be realized depending on the chosen inter-processor communication style in a single-beat or a burst-transfer fashion as seen in Fig. 2.8, Fig. 2.9 respectively (in Sect. 4.2.5).

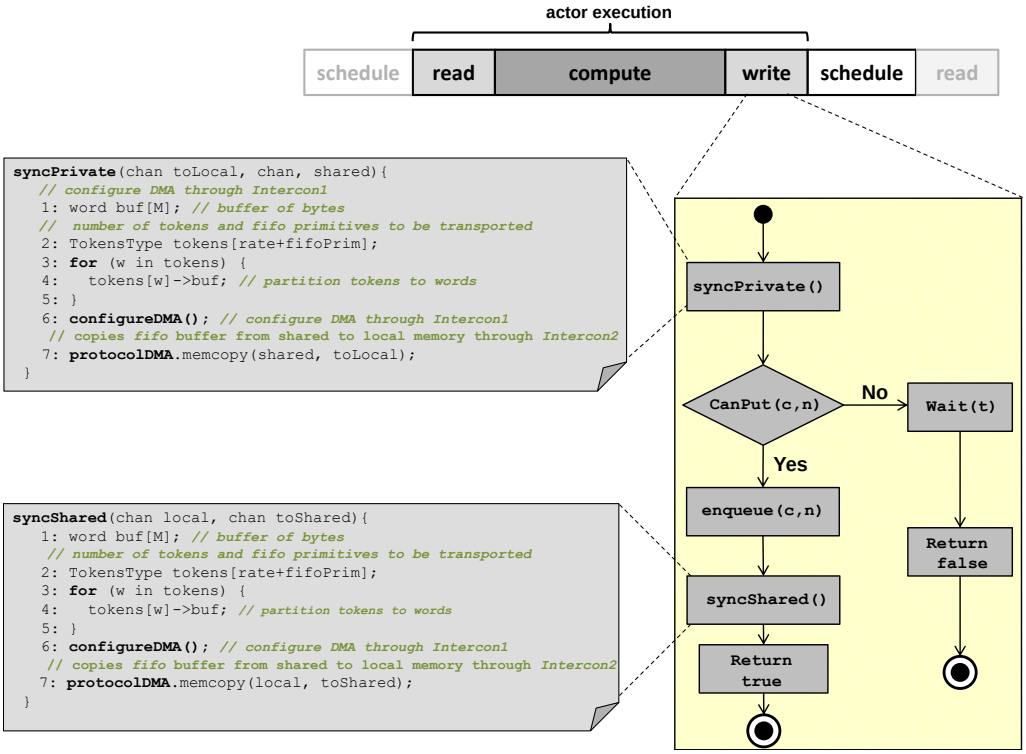


Figure 6.18: DMA communication driver's entry calls for a `Write` access

Communication Driver for DMA In case, the burst transfer is realized through the DMA hardware component, the protocol gets more complex and there are some additional issues which should be taken into consideration. Typically, a DMA *Transaction* consists of a number of *Transfers*, which in turn consists of a number of *Moves*. A *Move* is the basic action of the DMA reading from one (or group of) memory cell(s) and writing to another. In order to launch a burst transfer through the DMA, every tile first configures the DMA to send on a specific channel (with each channel having a fixed-priority as in the case of the Aurix DMA [Infineon Inc., 2013] as we will see in Sect. 7.4). In addition, the tile configures the DMA transfer parameters (number of moves per transfer, datawidth etc.). As already stated, we assume that the configuration phase of an interconnect through one tile does not interact with the transfer phase of other tiles (in Aurix experiments in Sect. 7.4 we have used the System Peripheral Bus (SPB) exclusively for the configuration phase, and the System Resource Interconnect (SRI) exclusively for the transfer phase).

Typically, within the DMA component an arbitration mechanism is supported which grants access to the channel with the highest priority (for e.g. the Aurix DMA supports a Fixed Priority (FP) arbitration scheme). After that the *Transfer* (in Aurix DMA up to 64 bytes per transfer by a channel width of 32

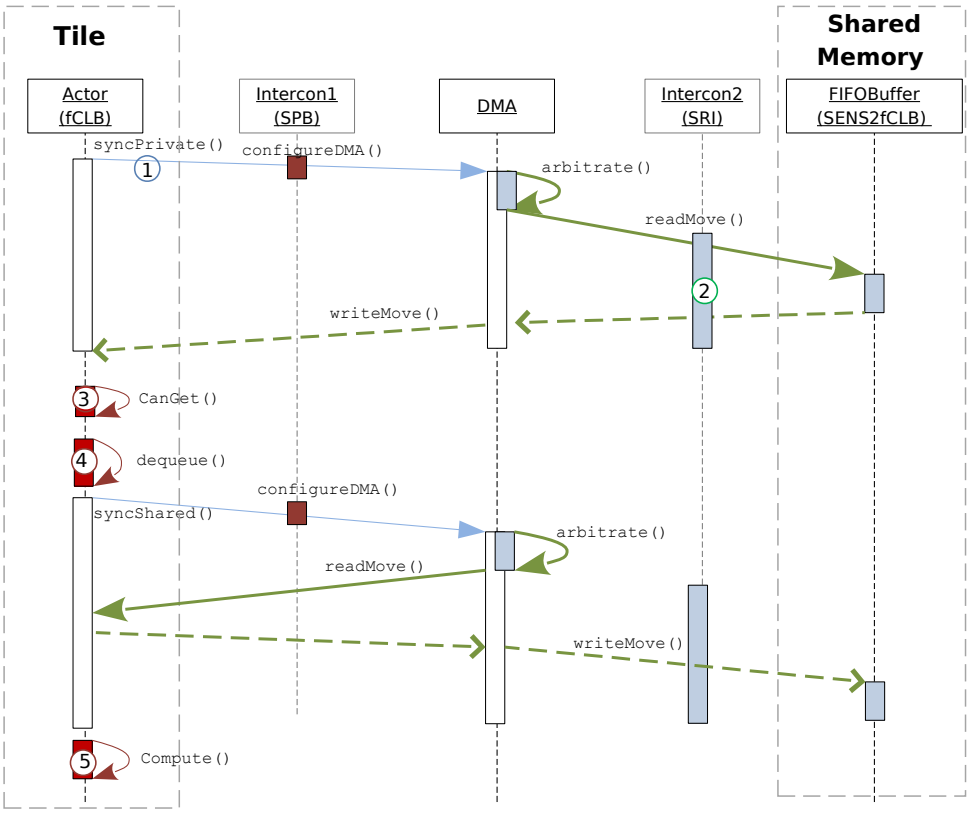


Figure 6.19: Sequence diagram of a DMA `Read` burst transfer

bits) is launched in an atomic way, so that re-arbitration is only done when the *Transfer* is finished. At this point, the *Transfer* of a low priority channel is suspended if a higher one is active. If the current request wins the arbitration the DMA starts moving memory blocks from/to the shared storage resource (from the LMU in the Aurix platform) from/to local memories through an another interconnect (SRI in Aurix experiments Sect. 7.4).

By utilizing the DMA for realizing inter-processor communication, some changes occur on the procedure done in Fig. 6.17. One major difference is that an extra configuration of the DMA (`configureDMA()`) is done every time the actor needs to access a (number of) variable(s) located in the shared storage resource. Another difference, is the `memcopy` action of different locations from/to the shared storage resource to/from the local private storage resource. Fig. 6.18 shows the extensions needed to realize the DMA `write` burst-transfer.

In order to better understand the DMA inter-processor communication semantics which are the same for both `write` and `read` accesses, let us consider the simple read access in Fig. 6.19. We assume that the producer actor (SENS actor see Fig. 7.11 in Sect. 7.4) already wrote data on the shared FIFO buffer (SENS2fCLB) allocated in the shared memory (LMU in Fig. 7.11)

and now we will show how the communication semantics for one read *Transfer* look like. Since the shared FIFO buffer is mapped to the shared memory, every tile communicating through this FIFO buffer should first synchronize its local buffer with the shared one before updating (enqueuing/dequeuing) it. The consumer actor (fCLB) when activated, launches a synchronizing action (`syncPrivate()` ①) to synchronize first the local private buffer with the shared one. Within ① (thin/blue line), the DMA configuration parameters are sent through *Intercon1* (SPB see Sect. 7.4) to the DMA. Next, arbitration is done at the DMA, which in turn by successful arbitration launches a *Transfer* on *Intercon2* (SRI see Sect. 7.4) where a number of read *Moves* and write *Moves* take place (bold/green lines ②). After finishing the read *Transfer*, a local copy of the shared FIFO buffer (SENS2fCLB) has been established in the local memory of the corresponding tile and it can now be checked whether there are enough data in the buffer or not (`CanGet()` ③). If there is not enough tokens in the buffer then procedures ①, ② and ③ are redone after some time (see `Wait(t)` in Algorithm 2). If there is enough tokens to read, the needed tokens are dequeued from the local buffer ④, and another synchronization mechanism (`syncShared()`) is executed which has the same behavior as `syncPrivate()` with the difference that now data are moved in the opposite direction in the *Transfer* i.e. from the local private FIFO to the shared one. The above steps are repeated for every inter-processor communication of the actor's input ports. After reading all input ports the actor computes internally (`Compute()` ⑤), and then writes the outputs on the output buffers. If these are mapped to the shared memory, the same procedure as the (above described) read transaction is done with the difference that now the buffer is checked whether it has enough capacity (`CanPut()`) or not, and if this is the case tokens are written to the buffer (via `enqueue()`).

According to above DMA semantics, it is now possible (if not through hardware primitives prohibited) that after a consumer actor synchronizes its local private buffer with the shared one and while it is busy checking/updating it, that the producer actor writes to the shared one. In order to avoid Write After Write (WAW) conflicts on the shared FIFO buffers, we restrict for simplicity the size of the buffer (during a DMA burst transfer) to be always equal to the maximal total tokens' size transported during a burst-transfer by either the consumer or the producer actor when activated. In the case, where the producer actor has a larger rate than that of the consumer then it will block when trying to write more data than those produced in one activation (since the buffer's size is set to be equal to the size of number of tokens produced by the producer in one activation). If on the other side the consumer actor has a larger rate than that of the producer actor, then the producer actor will always block when trying to produce more tokens not consumable by the consumer actor in

one activation (since the buffer's size is set to be equal to the size of number of tokens being consumed by the consumer in one activation). Also in the case the rates are equal, the same happens and the integrity of the transported data is achieved.

6.6 Summary

In this chapter, we have presented a model-based design flow which enables taking models implemented and simulated in the widely-spread simulation framework Simulink as an input model. With the help of the translation from Simulink (specific Simulink models subjected to restrictions) to SDFGs (see contribution C3-1), we enable the state-based RT analysis of Simulink control applications when implemented and run on a specific MPSoC and the usage of Simulink powerful features such as code-generation.

Furthermore, we gave a short description of our SDF2TA tool which automatize some of our design flow steps and facilitates the application of our state-based RT analysis (see contribution C3-2). We also pointed out to the maintainability features (which was enabled through the usage of Ecore models) of SDF2TA and other features allowing the application of extra analysis methods supported by the well-known SDF^3 tool [Stuijk et al., 2006] to the SDFGs modeled in SDF2TA.

In addition, we integrated a VPIL simulation (see contribution C3-3) to our design-flow which verifies Simulink control applications running on a concrete MPSoC, in the loop with the Simulink environmental model preserving the causality of the golden model and enabling a simulation-based verification of their timing and functional requirements. The proposed simulative approach contributes to the verification of timing requirements of critical control algorithms through a non-invasive measurement technique of their execution times on a cycle-accurate level. Using these measurements, design decisions can be made with minimized costs and shorter development times in early cycles. This technique also has the advantage of obtaining an average RT analysis in case no WCET analyzer tools are still mature for novel processors (as for the case of the Aurix see Sect. 7.4), or in the case of large parallel systems which are beyond the scope of being analyzable by formal methods. Moreover, a seamless functional verification via the semi-automated virtual-platform-in-the-loop V&V of the implemented MPSoC application is supported.

Finally, we elaborated on implementation issues which are faced by a developer and gave hints on how to implement the SDFGs with their different scheduling mechanisms and how to realize access to shared storage resources with the help of communication drivers, all being conform to our assumptions in Chap. 4.

Chapter 7

Evaluation

In this chapter, we describe the set of experiments conducted to demonstrate the viability of our state-based RT method. We start by providing evidence for the correctness of our state-based RT method, mainly showing how the TA templates' implementation described in Chap. 5, can be validated with the help of UPPAAL[1] model-checker. Next, we will take a look at how far can our RT method scale in terms of actors, tiles, arbitration protocols and BCETs/WCETs interval variations. Afterwards, the possible scalability improvement when applying the methods presented in Sect. 7.2.1 is demonstrated. Following this, we will evaluate the tightness improvement of the RT results achieved via our approach when analyzing a benchmark SDFG application compared to an analytical method from [Shabbir et al., 2010]. Finally, we demonstrate our model-based design flow on an industrial use-case and calibrate the timed automata described in Sect. 5.1 with the timing properties of the Aurix platform and the timing values of the automotive case-study to extract timing bounds of the application execution time. All experiments were done based on a burst-transfer inter-processor communication (except the single-beat transfer mapping in Sect. 7.4) and were conducted on a 64-core (AMD Opteron(tm) 6282 SE Processor) platform running at 2.6 GHz with about 500 GB of RAM.

7.1 Increasing Confidence in Correctness of Approach

Typically, for a RT analysis method, there are two main sources of doubt (according to [Kästner Daniel and Christian, 2014]) which can alleviate the confidence of the obtained results: the *logic* doubt associated with the validity of the reasoning and the *epistemic* doubt associated with uncertainty about the underlying assumptions. For e.g. in measurement based RT methods usually logic

[1] UPPAAL 4.1.19 (rev. 5648) 32-bit version, has been used in the experiments.

and epistemic doubts remain [Kästner Daniel and Christian, 2014], since a full test coverage is in most cases not achievable and because of the fact that they are either invasive or require an extra hardware setup (see our VPIL simulation in Sect. 6.4).

In our system model definition in Chap. 4, we have described how our abstractions are based on sound ground using sound over-approximations on the execution times in the SUA. This sound abstraction together with the theory of timed automata which provides a formal methodology to check the SUA model rigorously, eliminate the logic doubt. It is important to note here, that since these over-approximations depend mainly on the WCET analyzers, these analyzers are assumed to be maturely verified and their results can be trusted (as shown in [Kästner Daniel and Christian, 2014]).

After establishing the soundness of our RT analysis methodology through sound abstraction, it remains to show the correctness of the SDF2TA implementation and that of the underlying SUA model. A short description is given in Appendix A.1 showing how possible errors are prohibited in the SDF2TA tool with the help of defensive programming mechanisms. We will use model-checking constructs to gain confidence in the correctness of our TA templates' implementation representing a certain SUA (including the SDFGs and the hardware platform).

Correctness Validation of TA Templates using Model-checking In the Appendix A.1, we elaborated on how to detect, avoid/correct syntax and compatibility errors (for e.g. if the Ecore model is not compatible to the timed-automata templates implementation). The question remains, whether the implemented timed-automata templates capture correctly the semantical behavior of the SUA or not i.e. how can we gain confidence that we are analyzing the right system model?

One advantage of the state-based RT method presented in this work using the well-established academical and industrial model-checking tool UPPAAL, is that it enables to determine whether some timing and logical properties of the considered system are satisfied or not. Some of these properties which will help us to answer the question above, are described in the following:

(a) **General Properties:**

1. *Is the system deadlock free?* The network of TA representing a SUA can deadlock in UPPAAL when no progress (neither by waiting for some time nor by a transition between locations) can be achieved in the state space. The reason could be a bug in the implementation of the TA templates for e.g. a committed state which has no valid transition to leave the state, or array out of bounds access error.

The property checking the absence of deadlocks in the TA implementation is primitively supported by UPPAAL and can be done by checking the following TCTL property (see Sect. 2.5.2.1):

```
A[] not deadlock
```
(7.1)

If the network of TA is verified to be deadlock free, this doesn't mean necessarily that our SUA is deadlock free. Take as an example the case where a consumer actor is blocking on a FIFO buffer (because there is not enough tokens to consume for e.g. when the producer is only activated once instead of twice) and after some time it retries the access, but the producer actor remains blocked because of a faulty schedule. In this example, UPPAAL doesn't detect a deadlock of the timed automata since these themselves show a progress in time. In order to still assure the absence of deadlocks other properties should be checked (see below liveness property of SDFGs).

2. *May-Happen-in-Parallel (MHP) Analysis for actors:* MHP analysis checks for two given actors whether or not there is a possibility where these are executed at the same time. MHP analysis could be useful to optimize mapping decisions (for more information c.f. [C. Chang, 2015]).

   ```
   E<> Actor_a0.Compute and Actor_a1.Compute
   ```
   (7.2)

   which simply checks if eventually a state can be reached where both actors `Actor_a0` and `Actor_a1` are executing in parallel (both in the `Compute` state see Fig. 5.5).

3. *Resource Contention:* Is there any contention on the shared communication resource? This property could be very interesting, since it might be that according to a specific mapping and different execution times, that actors never compete on the shared communication resource at the same time. In this case, it doesn't matter which arbitration protocol is used, the timing bounds will be independent from the arbitration protocol. Another issue here if there is contention at all at any time i.e. whether or not other requests from other tiles are issued and block while the current tile is performing an inter-processor communication on the communication resource.

   ```
   sup:  interconnect.buffer.length
   ```
   (7.3)

   Above property searches for the maximum buffer length of the interconnect for all paths. If the result is higher than one then we

know that contention exists and arbitration plays a role in the current mapping otherwise (if equals to one) no parallel access from two tiles or more to the shared communication resource ever occurs, and the arbitration scheme does not influence timings of the SUA. Now, the question remains whether or not one access of some tile(s) was still blocked/delayed through another one, the fact which we can find out by checking the following property:

```
E<> (interconnect.FIFOBlocked or interconnect.TransOK)
                and (interconnect.buffer.length>=1)   (7.4)
```

which simply checks if eventually a state can be reached where the interconnect TA is in one of the finishing states (`FIFOBlocked` or `TransOK` see Fig. 5.7) and there is one or more access(es) pending in the buffer. If satisfied, then it indicates that communication contention took place.

4. *Maximum Number of Shared Resource Accesses (MNC):* The MNC is defined as the maximal number of actor's accesses to a shared resource among all activations of this actor in an iteration. This property was used in Sect. 7.3 to calculate the worst-case response time of every actor, depending on its MNC, communication latency and execution time. This enabled the assessment of tightness improvement of our method compared to a pessimistic analytical RT method.

   In order to obtain the maximal/minimal access numbers on the shared communication resource, we introduce a recording array `countAccess` in the communication driver timed automaton, which counts for every actor its accesses to the resource (which is incremented in every transition with an interconnect access from the state `IssueComm` to state `WaitComm` see Fig. 5.6). The `countAccess` variable of every actor is reseted after every firing of this actor and at the end of every iteration of the SDFG. The MNC includes in addition to the access itself also extra accesses caused by the blocking mechanism in case the actor blocks. With the help of the following TCTL, we can let UPPAAL find the min/max resource access number of some actor a_i mapped to a tile t_j:

$$\texttt{sup:commdriver}_{t_j}\texttt{.countAccess[}a_i\texttt{]} \tag{7.5}$$

$$\texttt{inf:commdriver}_{t_j}\texttt{.countAccess[}a_i\texttt{]} \tag{7.6}$$

(b) **Correctness of SDFG Model**

5. *SDF semantics:* Do the SDF semantics hold? In order to answer this question, we take use of the fact defined in Def. 5.2.1 which states that an SDFG after an iteration, obtains its initial state. This means that after one iteration (executing the actors according to the repetition vector see Def. 4.2.4) the number of tokens of all channels of an SDFG should be retrieved to the initial state. After implementing the SDFG on the MPSoC, the above statement is of course only verifiable if we can guarantee that any of the actors after being activated according to the repetition vector, does not get extra activated during the time where one of the actors belonging to the same SDFG (for e.g. the sink actor) still didn't finish its execution in the current iteration[2]. The above issue can't be guaranteed by SDFGs which are non-sensitive to external events, but can be guaranteed for periodically-triggered SDFGs. By a triggered SDFG, if we choose its trigger period large enough (greater than or equal to its maximal latency), then it can be guaranteed that the source actor cannot start again to execute (it does not get triggered again) unless the sink actor has completed its activation, and thus after one iteration the initial tokens' distribution is retrieved. In a first step (this step is only for validation reasons, afterwards the event trigger can be of course removed), we assume that all SDFGs under validation are sensitive to periodic events. Now, we utilize the end-to-end latency observer timed automaton (see Fig. 5.10) for a specific SDFG, if the semantics are correct then an iteration of the SDFG should be completed when the end-to-end latency observer reaches the `Finish` state and all buffers in this moment should have restored their initial states (with initial number of tokens). If this is not the case then we know that the SDF semantics hold no more.

   For an SDFG observed through the end-to-end latency observer `obs`, let $\mathcal{D} = \{d_1, d_2, d_3, d_4\}$ be the set of edges in the SDFG, with all edges having no initial tokens except d_2 having two initial tokens, then the following property must be satisfied as described above:

$$\texttt{A[] obs.Finish imply}(d_1\texttt{.len==0) and (}d_2\texttt{.len==2) and (}d_3\texttt{.len==0) and (}d_4\texttt{.len==0)} \quad (7.7)$$

[2]For e.g. in case of pipelining feature of MPSoCs, it could be that after completing one iteration, the initial token distribution for an SDFG is violated. Take as an example the SDFG in Fig. 2.4, `Actor1` won't be blocked if the buffer is large enough (larger than 2× consumption rate of `Actor2`) and it can produce a different token distribution than the initial one (assuming initial tokens' number of 0 on channel `a1-to-a2`) until the last instance of `Actor4` is finished.

6. *Maximum Buffer occupancy:* this property can obtain for every buffer (representing a channel) in the SDFG, what is the maximum buffer occupancy [Skelin et al., 2015]. This in turn can give confidence that the implemented SDFG semantics hold if we know a priori the maximal occupation of all buffers in the SDFG.

   Let b_i be the buffer to be analyzed, the maximum buffer analysis can be obtained by utilizing the supremum operator of UPPAAL:

   $$\texttt{sup:} \quad b_i \tag{7.8}$$

7. *Liveness property:* does every actor finally come to an execution? An SDFG can deadlock (as mentioned above) if the number of delay tokens (on corresponding channels) are not set right (in a cyclic SDFG) or if the buffer sizes are not chosen right or if the execution order of actors in an SDFG does not respect the reference schedule. In this case, even if the SDFG is blocked, the TA templates continue to progress (as we already described above `P1`) and that's why the primitive supported deadlock check of UPPAAL is not able to detect this kind of deadlocks.

   For a correct execution of an SDFG having the source and sink actors a_{src}, a_{snk}, the following liveness property should be valid:

   $(a_{src}$`.Finish) --> (`a_{snk}`.Finish)` which is equivalent to:

   $$\texttt{A[] (}a_{src}\texttt{.Finish imply (A<> }a_{snk}\texttt{.Finish))} \tag{7.9}$$

   which mainly checks if always after a source actor have been executed, eventually a sink actor is executed for all paths. Another way to check if every actor a_i is always eventually executed on all paths is:

   $$\texttt{A <> }a_i\texttt{.Finish} \tag{7.10}$$

(c) **MoA Correctness**

8. *Liveness property:* is there any state in which tiles' requests are served?

   In the following TCTL, we are able to check for every tile t_i whether or not the request is eventually being served:

   $$\texttt{(CommDriver}_{t_i}\texttt{.IssueComm)-->(CommDriver}_{t_i}\texttt{.CommFinishedOk)} \tag{7.11}$$

   which states that every time a tile issues (through the communication driver) a communication request to the storage resource, this request would be eventually served (see Fig. 5.6).

9. *Arbitration:* is there any situation in which the highest priority message does not win the arbitration? This property helps verify the implementation of the arbitration mechanisms of the shared communication resources.

   In order to validate that arbitration schemes in the TA implementation do not exhibit any errors, either already tested C implementations of the arbitration are imported (UPPAAL has a C like syntax [Bengtsson and Yi, 2004]) or these can be simulated and verified to check their behavior. In addition, model-checking can be utilized to validate them, either by constructing special observers or by introducing extra states. For e.g. for the fixed-priority arbitration, extra error states (with corresponding variables) are introduced to the interconnect, and if the error state is never reached, then the implementation is correct. This means that the following TCTL must never be satisfied:

   $$\texttt{A <> interconnect.Error} \tag{7.12}$$

   For the fixed-priority arbitration, we must first guarantee that there are multiple requests in the interconnect buffer (where *MultipleRequests* is true if `interconnect.buffer.length>1`) and that after arbitration, the tile with the highest priority ($tile_h$) gets the access. If *MultipleRequests* is true and the arbitration result is not $tile_h$ (in the state `IdentifyAccess` see Fig. 5.7) then the interconnect goes into `Error` state.

We have seen above how with the help of a set of TCTL statements we are able to validate whether or not the SDF semantics hold. In addition we can check if the SUA deadlocks and if the arbitration algorithms are implemented correctly. Moreover, we presented extra TCTL statements which help us verify other interesting properties (see P2, P3, P4, P6) of the SUA which are typically not supported by other RT analytical approaches (see contribution C2-3 in Chap. 1).

Apart from the model-checking based validation, we also use an empirical evaluation (as typically done by WCET tools such as aiT [Ferdinand and Heckmann, 2004], chronos [Li et al., 2007] or SWEET [Lisper, 2014]) where the analytically-determined upper bounds for an industrial use-case in Sect. 7.4 are compared with measured timings (obtained through our VPIL cycle-accurate simulation). This empirical check must insure that the measured WCET times are always below the ones computed by our state-based RT method.

7.2 Evaluation of Scalability

As the state-explosion problem is one of the main bottlenecks faced when using state-based RT methods with model-checking, we will give an insight about the state-space scalability of our approach in this section. For every experiment, we have obtained (with the help of UPPAAL) the number of states explored since states (in contrary to analysis time) are independent from the target computer where the analysis is run. First we will analyze the scalability with respect to the number of actors and tiles, then we will analyze the state-space behavior when using a specific arbitration protocol and finally we will vary the interval between best-case execution times of actors and their worst-case execution times to assess their effect on the state space. It is important to note that the experiments conducted in the following sections were made with fixed parameters of the same application (e.g. the JPEG encoder which is replicated in the experiments) and with a typical mapping for every experiment. The main goal was to give an insight how good our state-based method scales for a typical application with a shared-memory communication. It is obvious that the choice of different mappings, or integrating JPEG encoder application with another application of different timing behavior could produce totally different state-space results, since such a choice could lead to a larger number of states to be explored because of the possibly more complex contention and blocking behaviors.

Next, the possibility to improve the scalability of our approach will be examined by applying the concepts illustrated in Sect. 5.4, once by applying clustering and once by enabling composability.

7.2.1 Possible Scalability w.r.t number of Tiles and Actors

Fig. 7.1 depicts the SDFG of a JPEG encoder. The JPEG encoder SDFG consists of four actors: a macro-block sampling (get_MB) which parses an input BMP file and sends 3 macro-blocks (each 16x16 pixels) to a color conversion (CC) actor. The CC actor can fire if 128 pixels are available on its input edge and sends 64 pixels to the discrete cosine transform (DCT) actor which in turn

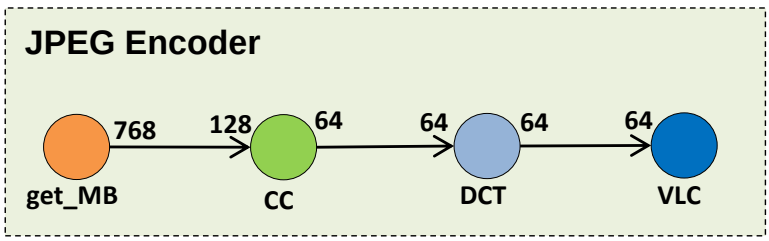


Figure 7.1: SDFG of a JPEG encoder (based on [Shabbir et al., 2010])

sends with each firing 64 pixels to the variable length coding (VLC) actor. In order to test the scalability of our method, we took the JPEG encoder SDFG which is non-sensitive to external events and used its parameters (see Tab. 7.6, only WCET are considered) to instantiate the timed-automata templates. In addition, the polling-wait time was set to 500 cycles in the case of blocking on shared FIFO buffer. Furthermore, the arbitration is set to a FCFS arbitration (since as we will see in Fig. 7.3, FCFS arbitration induces the largest state-space complexity), and all other parameters were fixed throughout the experiments to typical values. We then varied the number of JPEG SDFGs in the system and the number of tiles. For every variation, we have recorded the number of states explored needed to obtain the worst-case period of the SDFG which its sink actor is the last to be executed on a tile.

The results achieved are shown in Fig. 7.2 and indicate a better scalability than in [Gustavsson et al., 2010, Lv et al., 2010] which hardly scaled beyond two cores where altogether two tasks run on two cores and communicate through a simple spin-lock. As seen in Fig. 7.2, our approach has the potential to scale up to 96 actors mapped to 4-tiles platform, 196 actors on 3-tiles platform and even 320 actors running on a 2-tiles platform[3] without running into state-space explosion.

We notice that the number of states explored in Fig. 7.2 grows quite linearly (with some exceptions esp. in case of the red, square curve of 3-tiles, or in case of 40 actors running on 4-tiles platforms) as expected after the optimizations (see Sect. 5.4.1) with the growing number of actors (including their channels and data variables). This fact is due to the optimization of the previous implementation where instead of instantiating for every new introduced actor a new timed automaton, one timed automaton is used to represent the group of actors mapped to a tile. Major exceptions to the above observation occur in the case of actors running on a 3-tiles platform (see red, square curve). Since in most of the experimented cases depicted in this curve (except in the cases where 48 and 96 actors were analyzed) the number of actors cannot be equally distributed among the three tiles, the partitioning of actors of the same SDFG was mandatory. This clearly lead, in some cases (depending on the partitioning and mapping), to more contention (e.g. in the case of 16 actors) and thus to an increasing number of states to be explored and, in other cases, to a less contention (e.g. 32 actors running on a 3-tiles experiment has less states to be explored than the experiment with 16 actors). The same reason is also behind the fact that 40 actors running on a 4-tiles platform requires more state space than 48 actors running on a 4-tiles platform. On the other side, when increas-

[3]The suggested TA optimizations in Sect. 5.4.1 were directly implemented in SDF2TA and led to improvements in terms of analyzable actors (see Fig. 7.2) compared to our first experiments with the same setup (published in [Fakih et al., 2013a]) where analyzing above 96 actors on a 2-tiles platform lead to memory exhaustion.

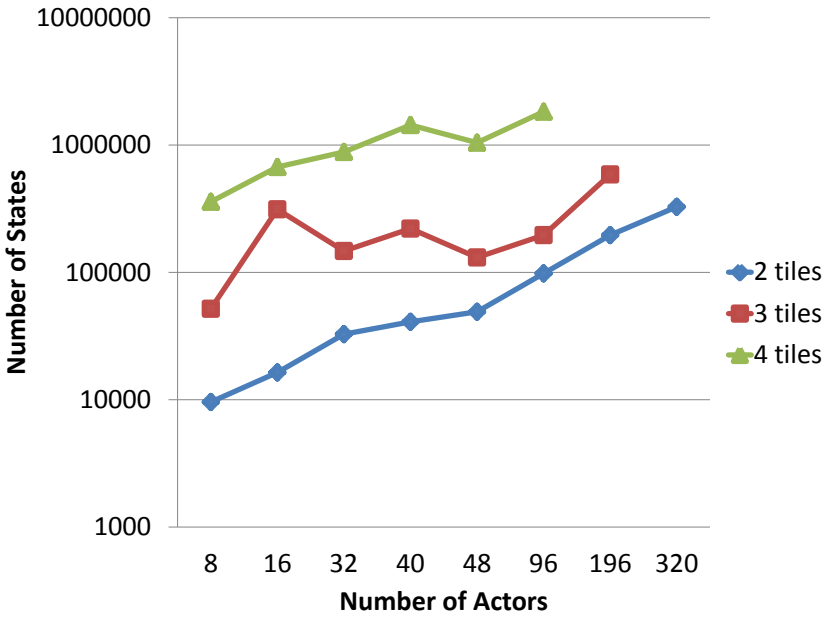


Figure 7.2: Influence of number of tiles and actors on the state space

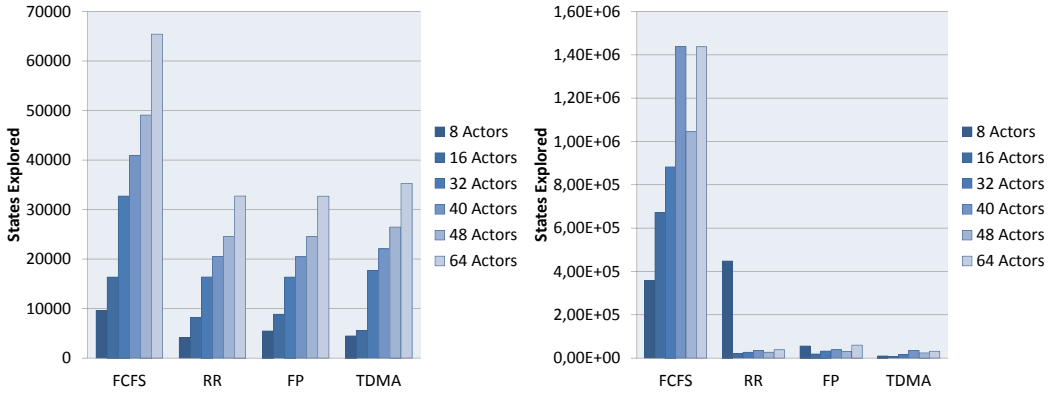
ing the number of tiles, we must introduce for every tile (independent of the number of actors running on it) a new timed automaton (see Sect. 5.4.1) which leads to the exponential growth of the state space observed in Fig. 7.2 when increasing the number of tiles.

In general, the scalability of our approach would be also influenced by the kind of IPC or by the variably large values of WCETs timings and the mapping, but again the choice of the set of experiments discussed in this section, is only to show the potential scalability under a realistic use-case configuration. The analysis time ranged between 0.1 s and 48 hours for the experiments which terminated successfully. The verification run was aborted by the tool with 8 actors mapped to 5 tiles after 2 weeks of analysis, as the memory was exhausted[4] (Out Of Memory: OOM).

7.2.2 Scalability w.r.t Arbitration Protocols

In this section, we will analyze how the scalability of our state-based RT method is affected with the arbitration protocol choices. The following experiments were done for the same application (JPEG encoder) as in Sect. 7.2.1. For every variation, we have obtained (with the help of UPPAAL) the number of states explored. In addition, we have chosen a burst transfer as the inter-processor communication style since this scales better than single-beat especially for an

[4]When using the 64 bit version of UPPAAL on a host PC with a large RAM, there is an improvement potential to analyze applications on more than 4 tiles, but the analysis time would still remain long since only one CPU is utilized by the UPPAAL model-checker.



(a) Different arbitration strategies on 2-tiles (b) Different arbitration strategies on 4-tiles

Figure 7.3: Influence of different arbitration protocols on the state space

application with high port rates (which is the case for a JPEG encoder). Furthermore, we fixed the execution time of every actor to a fixed-point interval ($WCET = BCET$) in this experiment.

Fig. 7.3 shows two sets of experiments which we have made, once varying the arbitration on 2-tiles platform (see Fig. 7.3a) and once on a 4-tiles platform (see Fig. 7.3b). Results in Fig. 7.3 show that the FCFS protocol, for all test-cases (with only one exception in case 8 actors are analyzed on 4-tiles with RR arbitration), induced the largest state space to be explored by the model-checker and the number of states explored gets largely increased when the number of tiles is increased compared to other protocols (as seen in Fig. 7.3b). This observation can be illustrated with the fact that FCFS arbitration is a heavily state-dependent protocol (see Sect. 2.3.3.1).

Another interesting issue is that other arbitration protocols (than the FCFS protocol) have the potential for enabling RT analysis of actors running on more than 4 tiles[5]. Yet we only tested scalability w.r.t number of tiles in Sect. 7.2.1 only for FCFS protocol to see how far our method scales with the most complex arbitration.

7.2.3 Scalability w.r.t BCET/WCET Interval Variation

In this section, we will analyze the effect of BCETs/WCETs intervals of variable sizes on the state space of our RT analysis method. In order to that, we took the same application (JPEG encoder) as in Sect. 7.2.1, and then fixed the arbitration to FCFS arbitration (since FCFS induces the largest state-space complexity see 7.3). We then varied the difference between WCET/BCET execution times of

[5]For e.g. the same experimental setup of that in Sect. 7.2.1 was done for 8 actors running on 6 tiles, only the arbitration protocol was changed to TDMA. Model-checking the WCP property needed about one second to finish and the number of states explored was only 33789 states.

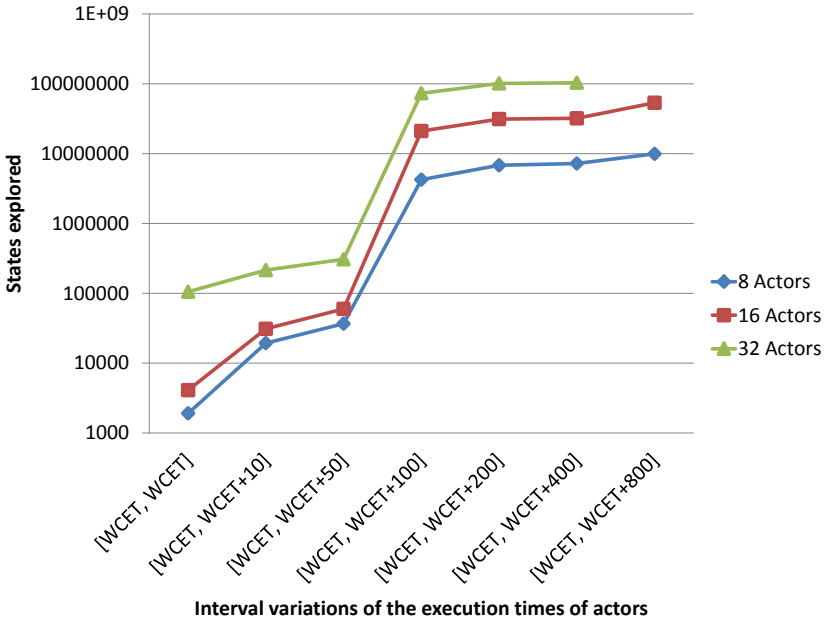


Figure 7.4: Influence of interval variation (2-tiles platform) on the state space

every actor and the number of actors (in Fig. 7.4) being run on a 2-tiles platform) and obtained the number of states being explored in each case when searching (this time) for the maximal latency of an SDFG.

We notice in Fig. 7.4 that the interval difference has a large effect on the number of states to be explored by the model-checker (which is a typical bottleneck of a state-based analysis methods see Sect. 2.5.2.1). For e.g. by only enlarging this interval from 0 to 100 for all 32 actors (see green triangular line in Fig. 7.4) running on a 2-tiles platform, the number of states explored increases exponentially (about a factor of 700 times) from `105167` to `73188499` states. For a 4-tiles platform our approach could not scale beyond 16 actors when the interval difference is increased only by 10 leading to a memory exhaustion.

Another interesting issue can be seen in Fig. 7.4 for this specific SUA, when enlarging the execution time interval from $[WCET, WCET + 50]$ to $[WCET, WCET + 100]$ for all actors. While a significant increase of the number of states can be observed for this transition, for all number of actors experimented, other intervals' variation depicted in that graph had a less significant impact (see Fig. 7.4). This is due to the contention induced by this specific execution time interval $[BCET, BCET + 100]$, where more blocking/waiting penalties lead to a more complex state space than other intervals.

Table 7.1: Execution times of actors of MP3 decoder (in cycles taken from [Stuijk et al., 2006])

	huffman	**req0**	**req1**	**reorder0**	**reorder1**	**stereo**	**aliasreduct0**
$\mathbf{t}_{wcet}$	473	279	279	139	139	148	27
$\mathbf{t}_{com}$	4	3	3	2	2	8	2
	aliasreduct1	**IMDCT0**	**IMDCT1**	**freqinv0**	**freqinv1**	**synth0**	**synth1**
$\mathbf{t}_{wcet}$	27	1424	1424	473	473	3733	3733
$\mathbf{t}_{com}$	2	3	3	2	2	1	1

7.2.4 Possible Scalability Improvement with Actors' Clustering

In the following, we will demonstrate the achievable improvements (increasing the number of actors being analyzable) by applying the clustering method (presented in Sect. 5.4.2) on a multimedia use-case. Consider the MP3 decoder example (from SDF^3 benchmark) in Fig. 2.6 (to the left). Suppose that we want to run the MP3 decoder on a 2-tiles platform. For this purpose, we map actors `freqinv0` and `freqinv1` to tile2 and all other actors to tile1. The execution times (taken from [Stuijk et al., 2006]) are shown in Tab. 7.1. In addition, in case of blocking on shared FIFO buffers a polling-wait time of 50 cycles is assumed. The clustering in Fig. 2.6 (to the right) can be made according to the clustering method explained in Sect. 5.4.2, resulting in an MP3 decoder SDFG of 7 actors only (originally it was 14 actors). Notice that since all six actors: `IMDCT0`, `IMDCT1`, `freqinv0`, `freqinv1`, `synth0`, `synth1` engage in an inter-processor communication, they are excluded from the cluster.

The timing requirement of the MP3 decoder requires that decoding one frame (in one iteration) should not exceed a maximal time of 26 *ms* (with a 500 MHz clock). Tab. 7.2 shows results obtained from the analysis of the clustered and non-clustered MP3 decoder. Both show the same timing behavior and achieve one frame decoding in a maximal period of 19.4 *ms* which clearly fulfills the requirement. In terms of analysis time the clustering gives a percentage improvement of 48.6% and in terms of states being explored an improvement of 4.3 %. Interestingly, for the above experiment even a minor improvement of the states explored causes a major improvement of the analysis time. This issue was often observed by other experiments conducted in this thesis, which seems to be a property of search algorithm implemented in the UPPAAL model-checker.

7.2.5 Possible Scalability Improvement via Temporal Segregation

In the following, we will show how enabling composability in the MPSoC (as described in Sect. 5.4.3) by extending the MPSoC with an extra TDMA hypervisor hardware component (first published by the author in [Fakih et al., 2013b]),

can improve the scalability of our approach (increasing the number of actors being analyzable). The same results can be achieved when using the alternative method with the help of customized timer for realizing composability (presented in Algorithm 4). This hypervisor achieves a resource virtualization using a static time slot per SDFG cluster. The hypervisor switches circularly between the time slots and takes care of the temporal and spatial segregation. Each SDFG cluster can access all platform resources until its time slot is over. When switching to the next slot, the hypervisor takes care of storing the local state of all platform resources of the terminated slot and restores the local state of the next time slot to be activated. We assume a worst-case context switch overhead h when switching between different slots.

Suppose that *Application Cluster 1* and *Application Cluster 2* (see Fig. 7.5) were developed independently to be executed on the same MPSoC. Now, an Original Equipment Manufacturer (OEM) designer has the task to integrate both applications on the same MPSoC extended with the hypervisor component, such that they still meet their timing requirements. Above use-case is typical in nowadays industrial domains (such automotive and avionics) where platform-based design is indeed widespread.

In the following experiment, we intend to achieve the following goals:

1. Demonstrate how our proposed method (see Sect. 5.4.3) can be applied to above use-case

2. Assess the performance degradation in terms of hypervisor's switching delay h

3. Show how this extension (together with our RT method) enables the analysis of larger applications with larger number of actors.

Tab. 7.3 shows the parameters of the eight artificial SDFGs, we constructed (with the help of SDF^3 `generate` extension of SDF2TA) to achieve the goals above. The actors' worst-case execution times were generated randomly (uniformly distributed) within a range of [5..500] cycles. We have set the ports' rates

Table 7.2: Analysis results of clustered and non-clustered MP3 decoder

	Analysis time (in s)	States explored	WCP (in ms)
without Clustering	148	281996	19,452
with Clustering	76	270326	19,452
% improvement with Clustering	48.6%	4.13%	

Table 7.3: Composable RT analysis: experiment setup (in cycles)

SDFG	A	B	C	D	E	F	G	H
Actors' Nr.	10	10	10	6	10	10	10	6
Channels' Nr.	9	9	9	5	9	9	9	5
Ports' Rates	[1200,2400]	[200,600]	[220,440]	[100,200]	[500,2000]	[300,600]	[700,1400]	[150, 300]

Table 7.4: Composable RT analysis results on 2-tiles platform (WCP in cycles)

	Cluster1		**Cluster2**	
SDFG	A	B	C	D
WCP_{req}	160 000	160 000	160 000	160 000
WCP_{isol}	54 529	59 895	85 001	44 236
$WCP_{noCompos}$	140 863	117 439	141 734	119 466
WCP_{compos}	144 896 $+(2\times h)$			
Avg. Performance degradation %	2,8% + $(0{,}0014\times h)$%	23% + $(0{,}0017\times h)$%	2,2% + $(0{,}0014\times h)$%	21,2% + $(0{,}0016\times h)$%

deliberately high, in order to impose more contention on the bus. High rates lead to longer communication time of the active actor, and this in turn leads to longer waiting time of other actors trying to access the bus. In addition, all edges of all SDFGs in all mappings were mapped to the shared memory in order to achieve a high contention on the bus, and a polling-wait time of 50 cycles is assumed in case of blocking on shared FIFO buffers. The bus has a bandwidth of 32 bits/cycle, with a FCFS arbitration protocol and all tokens' sizes were set to 32 bits. Moreover, all SDFGs were scheduled according to a static-order SDFG scheduler.

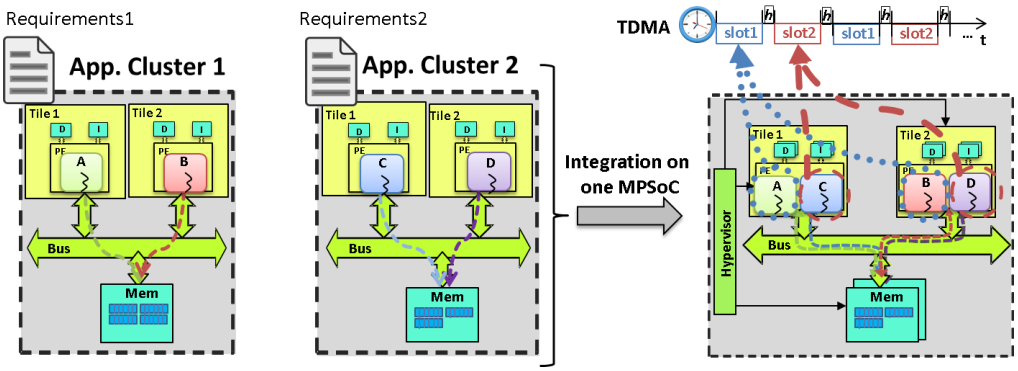


Figure 7.5: Integrating two SDFG clusters on a 2-tiles virtualized platform

Table 7.5: Composable RT analysis results on 4-tiles platform (WCP in cycles)

	Cluster1				**Cluster2**			
SDFG	A	B	C	D	E	F	G	H
$WCP_{isolation}$	135 400	171 000	135 400	69 600	107 850	64 500	66950	37 300
WCP_{compos}	278 850 +(2×h)							

In the first experiment (see Fig. 7.5), we configured our timed-automata templates to evaluate the mapping of each cluster in isolation (see left of Fig. 7.5 with *cluster*1: `SDFG A, SDFG B` and *cluster*2: `SDFG C, SDFG D`) on the 2-tile platform, each having a timing requirement (on the Worst-case Period: WCP_{req}). The Worst-case Period results for every SDFG which were calculated in isolation (WCP_{isol}) with the help of our state-based RT method (see first step of the two-tier RT method on top of Fig. 5.15) are shown in Tab. 7.4.

Next, we integrated the four SDFGs to run on the same MPSoC but without the hypervisor component extensions. Again, we utilized SDF2TA to find the new $WCP_{noCompos}$ of every SDFG (see Tab. 7.4). We can observe an average percentage increase of 121% of the WCP of every SDFG, due to the large contention and waiting times when integrating the two clusters on the MPSoC.

After that, we took use of the hypervisor extension, configuring two time slots. *cluster*1 is assigned to *slot*1, and *cluster*2 are assigned to *slot*2 (see Fig. 7.5 right). The length of every slot (WCP_{max}) is equivalent to the maximum WCP_{isol} among the SDFGs assigned to this slot (slot1: `59895`, slot2: `85001`). The new WCPs (WCP_{compos}) can be now calculated according to Eq. 5.6.

Results depicted in Tab. 7.4, show that all SDFGs will still respect their requirements as long as $h \leq 7552$ cycles.

Assuming a hypervisor delay h of `1000` cycles, a minor performance percentage degradation of average 14% can be observed in order to insure a temporal and spatial segregation through the hypervisor (where the percentage degradation is equal to $(\frac{WCP_{nocompos}}{WCP_{compos}} - 1) \times 100$).

If we assume that our RT analysis method does not scale beyond 40 actors mapped to a 4-tiles platform for this considered use-case with the specific mapping, then in order to demonstrate the scalability improvement of our proposed extension, we consider the same set of artificial SDFGs presented above which have in total 36[6] actors constituting *cluster*1, and another set of SDFGs (E, F, G and H) constituting *cluster*2 also having in total 36 actors (see Tab. 7.3).

Each cluster was mapped on the same 4-tiles platform (without hypervisor) and both were first analyzed in isolation with the help of our SDF2TA tool. Af-

[6]The same experiment would also be possible for clusters each having a maximal number of 96 actors on a 4-tiles platform, since this number of actors was analyzable through our state-based RT method (see Sect. 7.2.1) for the chosen JPEG SDFGs.

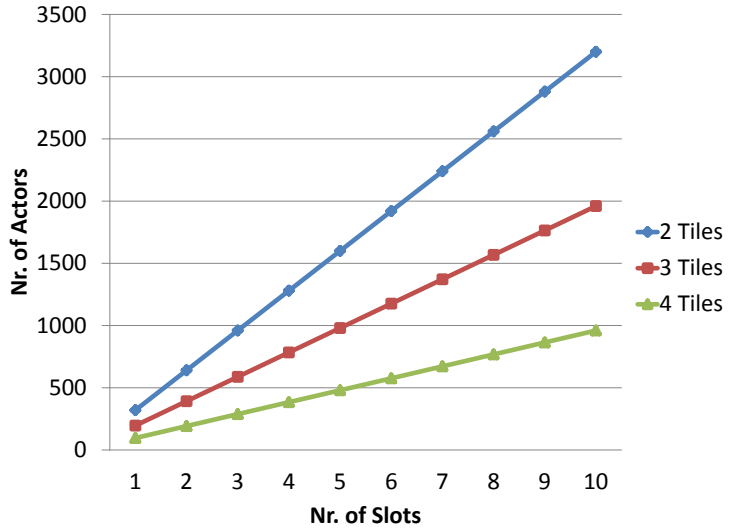


Figure 7.6: Potential scalability improvements with hypervisor extension

ter obtaining the WCP_{isol} of the single SDFGs in isolation (see Tab. 7.5), we now integrate the two clusters to run on the same 4-tiles platform with a hypervisor having two slots: *cluster*1 was assigned to *slot*1 with the length `171000` cycles and *cluster*2 to *slot*2 having a length of `107850`. Afterwards, we calculated the new WCP_{compos} of the single SDFGs according to Eq. 5.6 (see Tab. 7.5).

Results show that our composable analysis has the potential of significantly increasing the number of actors being analyzable by our method but at the cost of performance degradation. Obviously, we can now increase the number of SDFGs that can be analyzed by increasing the number of slots managed by the hypervisor.

Fig. 7.6 shows that the potential number of actors which can be analyzed by our method increases linearly with the number of slots reaching up (assuming the use-case has the same scalability behavior as the one in Fig. 7.2) to 3200 actors on a 2-tiles platform, 1960 actors on 3-tiles platform and 960 actors on a 4-tiles platform when assigning these actors to 10 slots. Nevertheless, the designer should be acquainted with the fact that by increasing the number of slots the performance overhead of the single SDFG would be increased (for Tab. 7.5 assuming $h = 1000$ an average increase of 256%) depending on the context switch overhead h of the hypervisor and the summation nature of the TDMA based analytical method (see Eq. 5.6).

7.3 Evaluation of Tightness Improvement

In this experiment, our goal is to make a comparison between the output of our analysis method with that of a pessimistic analytical method considered

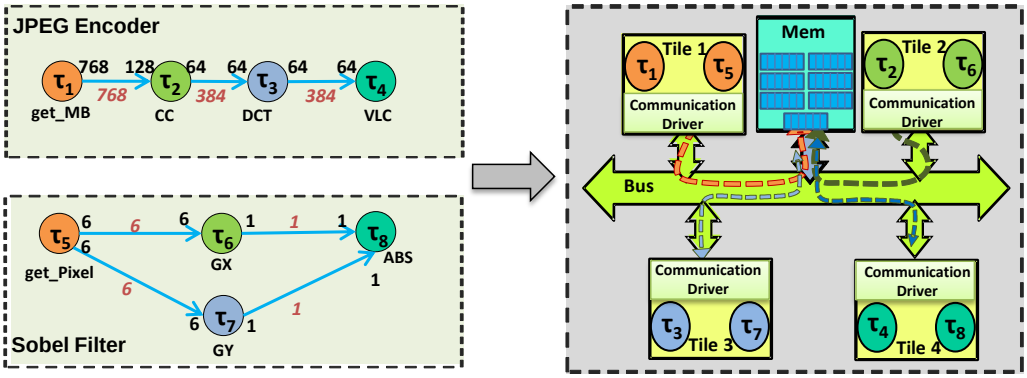


Figure 7.7: Mapping of JPEG encoder and Sobel filter on a 4-tiles platform (S3)

in [Shabbir et al., 2010]. In their work, the authors calculated the worst-case waiting times for resources with non-preemptive FCFS strategy by assuming that all other competing actors mapped to this resource come to run before the waiting actor. In our case, this means that for every tile (*tile A*) access to the interconnect, it should be assumed that the actor with the maximal communicating time on every other tile runs to completion before tile A gets access to the interconnect. The authors admit pessimistic results for large number of applications. We will show in the following how pessimistic these estimations can grow and how far our RT analysis method can tighten these estimations.

In order to do that, we use the system shown in Fig. 7.7 (depicting scenario S3 in Tab. 7.7) consisting of two SDFGs mapped to a 4-tiles shared bus platform and configured with the parameters listed in Tab. 7.6. t_{wcet} (in cycles) is the WCET given by a static analyzer for every actor (values were adopted from [Shabbir et al., 2010]). t_{com} (in cycles) is the communication time needed by every actor firing to transport a number of tokens (each of size 32 bits) over a bus with a bandwidth of 32 bits/cycle. First we configured the timed-automata templates to evaluate different mappings and schedules of the considered SDFGs (see Tab. 7.7). All edges in all mappings were mapped to the shared memory in order to achieve a high contention on the bus (as seen in Fig. 7.7). In addition, a polling-wait time of 500 cycles is assumed in the case of blocking on shared FIFO buffers.

Table 7.6: Execution times of (in cycles taken from [Shabbir et al., 2010])

	getMB	**CC**	**DCT**	**VLC**	**getPixel**	**GX**	**GY**	**ABS**
$\mathbf{t}_{wcet}$	13220	4446	20950	5420	320	77	77	123
$\mathbf{t}_{com}$	768	192	128	64	12	7	7	2

Table 7.7: Static-order schedules experimented

Scenario	Tile-1	Tile-2	Tile-3	Tile-4
S1	(getMB)(CC)^6 (getPixel)(GX)	(DCT)^6(VLC)^6 (GY)(ABS)	-	-
S2	(getMB)(CC)^6 (DCT)^6(VLC)^6	(getPixel)(GX) (GY)(ABS)	-	-
S3	(getMB)(getPixel)	(CC)^6(GX)	(DCT)^6(GY)	(VLC)^6(ABS)
S4	(getPixel)(getMB)	(GX)(CC)^6	(GY)(DCT)^6	(ABS)(VLC)^6

To obtain the worst-case period duration (WCP) for an SDFG, the schedule of the SDFG and the *Worst-Case Response Time (WCRT)* of every actor are needed. In Sect. 5.3 we pointed out how the WCP can be computed using our model-checking based approach. The WCRT for every actor can be calculated analytically according to Eq. 2.3. For the pessimistic analytical method considered in [Shabbir et al., 2010], the waiting time t_{wait} of an actor A mapped to a tile M by every activation is defined as follows:

$$t_{wait} = MNC_A \times \sum_{i \neq M}^{N} AWCT_i, \tag{7.13}$$

where *MNC* is the *Maximum Number of Communication* attempts that an actor A (by one activation) can launch on the bus in a given period (see P4 in Sect. 7.1). N is the number of tiles in the system, AWCT_i is the *Actor With the maximal Communication Time* (t_{com}) among the actors mapped to tile i (where $i \geq 0$ and $i \neq M$ excluding the tile on which the actor A is mapped).

The MNC highly depends on the number of ports of the actor and on the polling parameters (when blocking on a shared buffer). To achieve a fair comparison, we have extracted for every scenario in Tab. 7.7 the MNC of every actor with the help of the model-checker (see Eq. 7.5) and used it to calculate the WCRT of every actor according to Eq. 2.3. This guarantees that both methods work with the same MNC for every actor.

For every scenario, we have calculated the WCP once using our model-checking-based approach (MC WCP) and once with the help of the pessimistic method (Pess. WCP). The analysis time of our method for the considered scenarios ranged from 0.15 *sec* (for the case of 2-tiles platform) to a maximal of 13 *sec* (for the case of 4-tiles platform). We define the percentage improvement as $((Pess.WCP/MC.WCP) - 1) \times 100$ which describes how far our approach can reduce over-approximation compared to the pessimistic worst-case bus delay. Except for scenario S2 of the JPEG encoder, where the WCP estimated by our method gave only a minor improvement of 0.1%, all other results in Fig. 7.8 indicate significant tightness improvements of our approach over the pessimistic

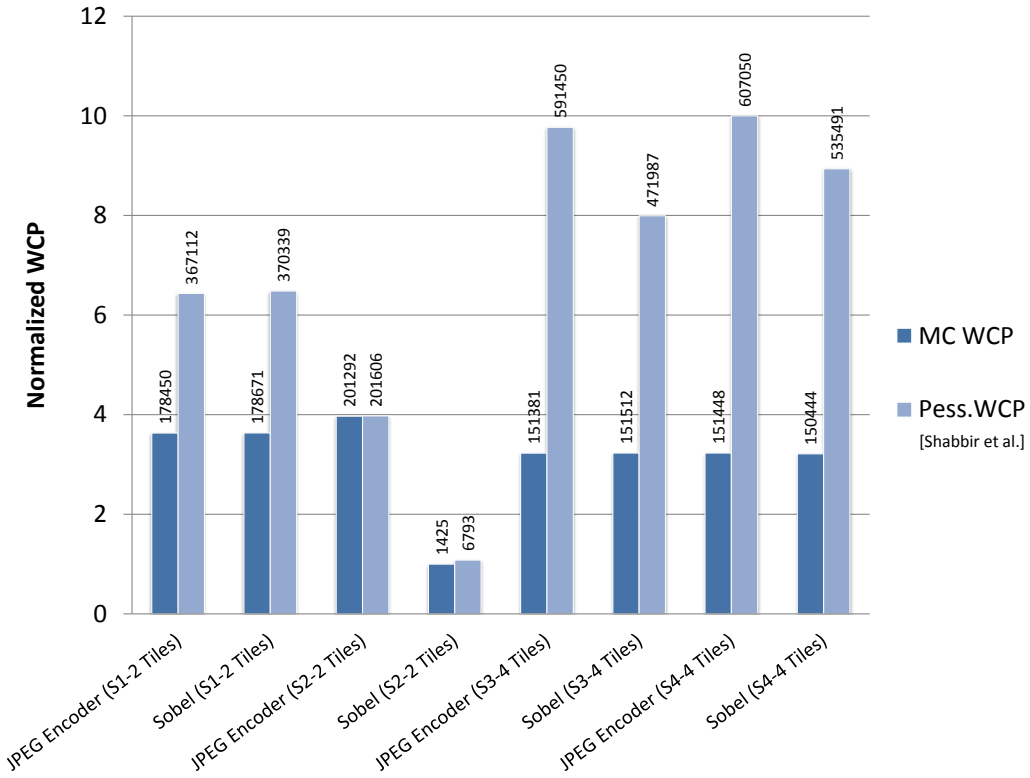


Figure 7.8: Worst-case period (WCP) analysis results

method. The minimal improvement in S2 is due to the fact that the waiting time (t_{wait}) of JPEG actors mapped to *tile-1* by every bus access was minimal (12 cycles, compared to the Sobel filter actors in S2 where t_{wait} was 768 cycles). Another factor was that the MNC of all actors in S2 was the smallest among all other configurations (ranging from 1 to 2 communication attempts on the bus, where as in S1 the MNC of the actors ranged from 2 to 56 access attempts). The maximum improvement was achieved in S4 by the JPEG encoder (up to 300%). The reason behind this is that the scheduling and mapping of the actors caused a very high MNC by the actors (ranging from 3 to 127). In this case, the waiting penalty by every actor communication according to the pessimistic method was also high (ranging from 256 to 960 cycles).

7.4 Industrial Applicability: Motor Control Case-Study

In the following, we will go through the steps of our model-based design flow (presented in Chap. 6) applied on a motor control case-study. First we describe both the input model and the MPSoC. Afterwards, two possible mappings of the control application on the hardware platform are implemented, one sup-

porting a burst-based inter-processor communication and the other one supporting a single-beat IPC. Next, we utilize our VPIL cycle-accurate simulation (see Sect. 6.4) to assess the timings of the application when run on the target platform. Finally, we use our SDF2TA tool to perform a RT analysis of the given application and compare the simulation with the analysis results.

In this experiment, we aim to achieve the following goals:

1. Apply our state-based RT approach on an industrial use-case showing that the assumptions (made in Sect. 4.1.2) needed for enabling our RT analysis are applicable on a modern multicore platform (the Aurix TriCore).

2. Demonstration of our model-based design flow steps on the given use-case.

3. Comparison of the simulative (obtained via VPIL simulation) and formal (obtained via SDF2TA) timing results, assessing over-approximation and showing that the measured WCET times are always below the ones computed by our SDF2TA tool.

7.4.1 Motor Control Simulink Model

In our case study, the motor control algorithm is modeled and implemented in Simulink abstracting away the hardware properties of the underlying platform. This enables the tuning of the control parameters depending on the simulation results of this model.

As seen Fig. 7.9 (top), the motor FOC (Field Oriented Control) algorithm [Park, 1929] is modeled and simulated with an abstract DC motor model (developed in [MotorBrain Consortium, 2013]). These models will be used as reference models for subsequent implementation steps. Besides the `FOC`, other functions are also implemented on the ECU. `CALIB_FAST` and `CALIB_SLOW` blocks realize all functions for the assessment and calibration of sensor data. `Voter` and `Monitor` observe the system functionality and sensor values and compare them with normative behavior [MotorBrain Consortium, 2013]. In the case of anomaly detection, the monitoring mechanisms switch the motor controller into a *Safe* state (e.g. limit the requested torque to a safe band or switching off the motor completely).

7.4.2 Motor Control Simulink Model to SDFG Translation

The next step is to translate the Simulink model into an equivalent SDFG (according to the translation procedure defined in Sect. 6.2). The result is shown in Fig. 7.9 (bottom). For our use-case, we consider the Simulink at the second level

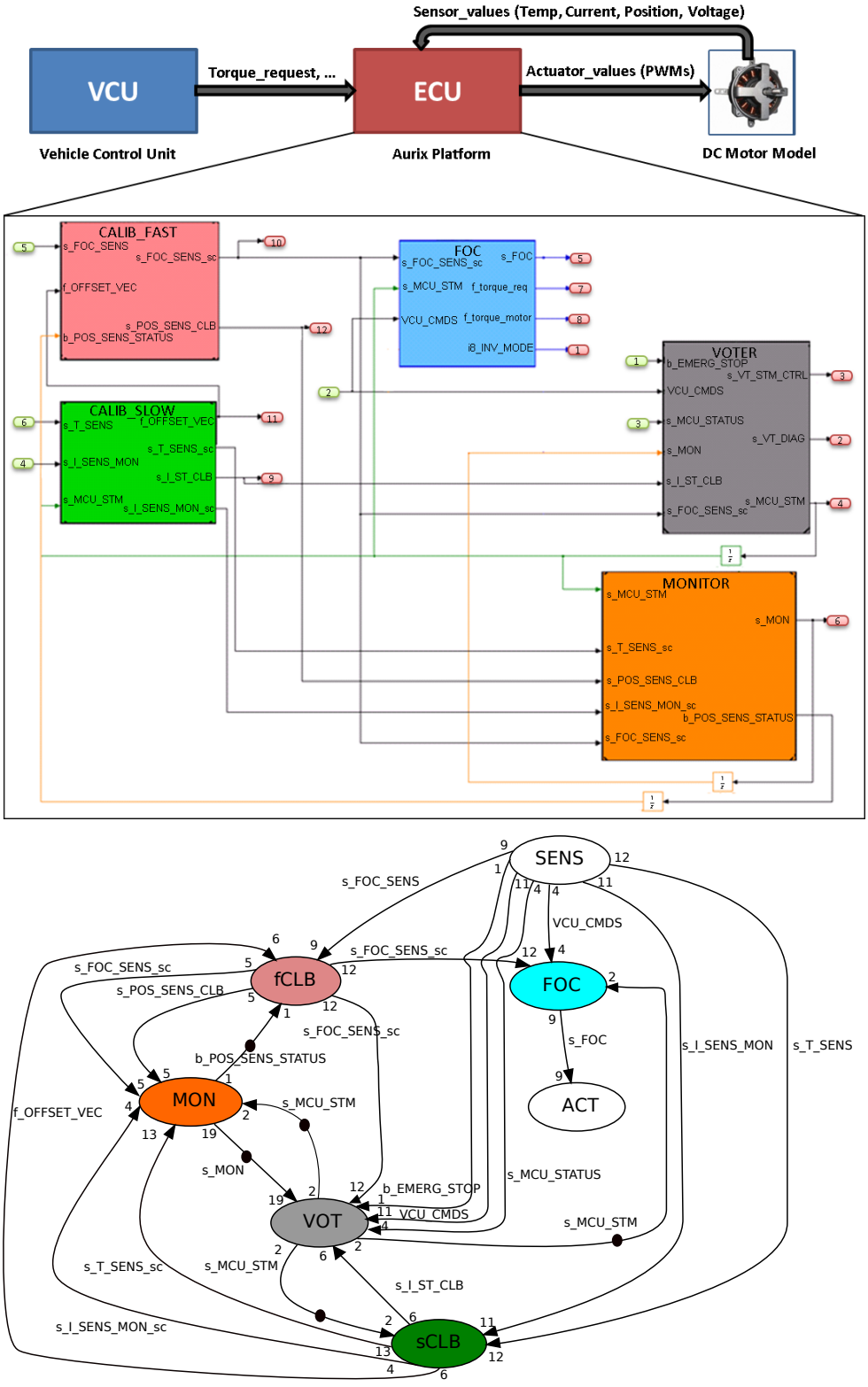


Figure 7.9: Motor control Simulink model and its corresponding SDFG

hierarchy (seen in the top of Fig. 7.9), which simplifies the translation process from Simulink to SDFG (compared to a more granular level). The actors `fCLB`, `sCLB`, `FOC`, `VOT`, and `MON` represent the blocks `CALIB_FAST`, `CALIB_SLOW`, `FOC`, `VOTER` and `MONITOR` respectively. The two actors sensor `(SENS)` and actuator `(ACT)` actors were introduced to realize the communication with the motor model and for optimization reasons.

Except for the output port with identification 5 (which forwards the Pulse Width Modulation (PWM) outputs to the motor), all other output ports were used for debugging and that is why these were omitted in the resulting SDFG. The actor representing the output port 5 after translation is renamed as `ACT` actor, whereas actors representing input ports (1 to 6) were all clustered in `SENS` actor to enable an efficient RT analysis. All edges in the Simulink model can be found in the SDFG, with the difference that ports are replicated if more than one transition goes from the same port in order to conserve the SDF semantics (see for e.g. by `VOTER` output port `s_MCU_STM` results into 3 output ports in the `VOT` actor in the SDFG each outputting the same tokens when activated). The ports' rates depicted in Fig. 7.9 (bottom), represent the number of tokens transported and consumed by the actors[7]. Since we have a cyclic SDFG, and in order to avoid deadlocks, we initialize the edges `s_MCU_STM`, `s_MON`, and `b_POS_SENS_STATUS` with initial number of delay tokens (just as in Simulink unit-delays[8] blocks are used).

7.4.3 Aurix TriCore platform

In specific, we will describe here the configuration and setup of the Aurix TC275 platform (see Fig. 7.11) to be compliant to the architectural constraints of our model-checking based performance method.

The MotorBrain team [MotorBrain Consortium, 2013] designed the Aurix platform for a nine-phase brushless DC motor as part of an electric car design. Fig. 7.11 shows the most relevant parts of the Aurix platform which we will be considering in our MoP to obtain the timing bounds of the FOC control application (see Fig. 7.9). The ECU is based on a multicore architecture containing safety features required in the automotive domain. The main processor

[7] In the Simulink model, generic arrays of data are transported, but not always all data of the array are consumed. In the translation process an optimization was made and the rate is set equal to the effectively consumed/accessed tokens in the consumer actor, which requires a knowledge of the implementation inside of the corresponding Simulink block (see for e.g. `VCU_CMDS` packet in the case of `SENS2FOC` the transported number of tokens is 4 while the transported number of tokens is equal to 11 in case of `SENS2VOT` edge).

[8] Unit-delay blocks are used in Simulink as seen in Fig. 7.9, to avoid deadlocks. These can be used in the translation to identify at which edge in the equivalent SDFG the delay tokens should be placed.

is an Infineon Aurix with three TriCore cores (TC1.6P [Infineon Inc., 2013][9]). Among others, the ECU board contains interfaces (not shown in Fig. 7.11) for controlling motor power electronics via PWM and Analog-to-digital converters (ADCs) to measure motor current and rotational position which will not be considered in this thesis.

In order to make the final software implementation predictable, a smart configuration of AURIX multicore platform is performed (compliant with our MoA constraints in Sect. 4.1.2). Looking into the specification of the Aurix platform, we observe that (A5, A6) are fulfilled. We disabled the supported caches (A6) and utilized the available linker script to link and map the code/data to the local instruction/data memory of the PE (to limit interferences among the cores) and to place only shared FIFO buffers (used for inter-processor communication) in the shared memory (LMU) (A7). A8 is also valid since the crossbar System Resource Interconnect (SRI) (supporting a fixed-priority arbitration) will be used in the experiments as a typical bus (since no parallel accesses to more than one slave are allowed in the experiments). Moreover, we simply restrict the contention on the I/O resources and bridges between different cores in our implementation (see A10).

The main challenge we encountered was that the Aurix platform does not support uninterruptable burst transfer, neither on the crossbar System Resource Interconnect (SRI), nor on the System Periphery Bus (SPB). In order to still be able to enable burst transfer of messages larger than the bus data width, we exploited the available Direct Memory Access (DMA) component, which supports burst-transfer IPC. Please refer to Sect. 6.5.3 for a description of the implementation details of DMA-based IPC semantics. The Aurix platform shown in Fig. 7.11a configured for a burst-transfer IPC (with the help of the shared DMA controller), can be described according to Def. 4.2.7 as follows:

$$
\begin{aligned}
EP_{Burst} = (&\{\text{Tile0, Tile1, Tile2}\}, \{\text{SRI, SPB}\}, \{\text{DMA}\}, \{\},\\
&\{\text{PFlash, LMU, Environment Interface}\},\\
&((\text{Tile*} \mapsto (\text{SRI, PFlash}),\\
&\quad \text{Tile0} \mapsto (\text{SPB, Environment Interface})),\\
&(\text{Tile*} \mapsto (\text{SPB, DMA})),\\
&(\text{DMA} \mapsto (\text{Tile*, SRI, LMU})), ())
\end{aligned}
$$

[9] In this work, it is assumed that all three cores are identical which is not the case in the real Aurix (two TC1.6P and one TC1.6E). This was done for compatibility reasons with the Aurix virtual-hardware platform which makes this assumption (see Sect. 7.4.6).

The Aurix platform configuration for single-beat IPC as shown in Fig. 7.11b is:

$$\begin{aligned} EP_{\text{Single}} = (&\{\text{Tile0}, \text{Tile1}, \text{Tile2}\}, \{\text{SRI}, \text{SPB}\}, \{\}, \{\}, \\ &\{\text{PFlash}, \text{LMU}, \text{Environment Interface}\}, \\ &((\text{Tile*} \mapsto (\text{SRI}, \text{PFlash}), \\ &\quad \text{Tile0} \mapsto (\text{SPB}, \text{Environment Interface}), \\ &\quad \text{Tile*} \mapsto (\text{SRI}, \text{LMU})), (), (), ()) \end{aligned}$$

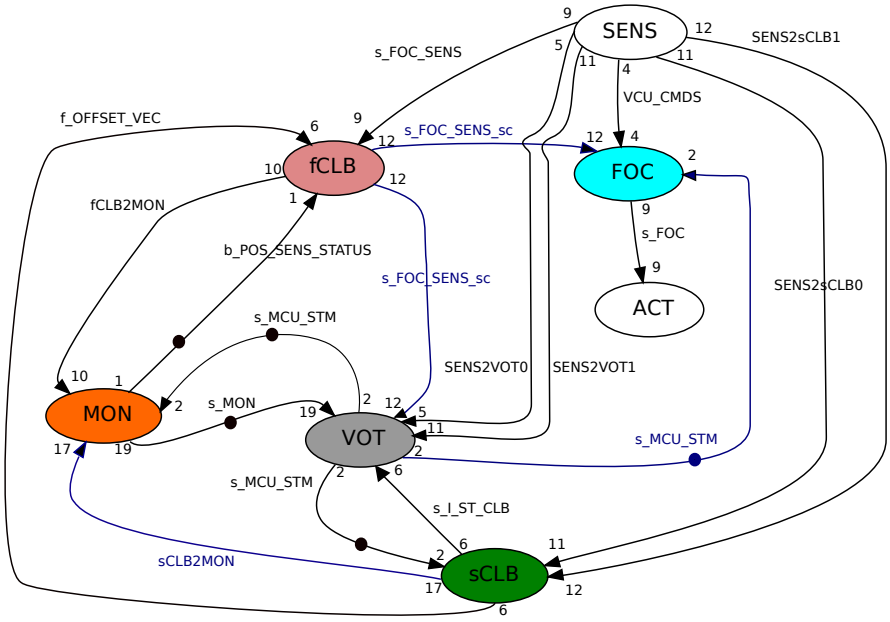
In the case of a burst transfer, the Aurix DMA component supports two kinds of arbitrations: *DMA channel arbitration* and *DMA switch arbitration*. By a *DMA channel arbitration*, if two channels are active at the same time, the channel with the highest number wins the arbitration (Fixed Priority (FP)). The *DMA switch arbitration* takes place in case concurrent SRI accesses from multi-channels are requested. In this case, the *Move Engine* of the channel with the highest number wins the arbitration. We observe that the tile which is assigned to the highest channel gets the highest priority and wins the arbitration, on both internal arbiters of the DMA. This means that it would be sufficient to abstract the DMA internal double arbitration into a single Fixed-Priority arbitration.

After winning arbitration, the *Transaction*[10] control set of the wiring channel is written to a so-called *sub-block active channel*. Then the *Transfer* is launched in an atomic way, so that re-arbitration is only done when the *Transfer* is finished. At this point, the *Transfer* of a low priority channel is suspended if a higher one is active. In our implementation, we made sure that all exchangeable data do not exceed the maximum data which can be transported by one *Transfer* of the DMA (maximum 16 moves by data width of 4 bytes = 64 bytes transportable in one *Transfer*) so that we can guarantee that no interruption of the transaction happens. By doing that, we can abstain from the explicit modeling of the DMA preemption mechanism and A9 is now also fulfilled.

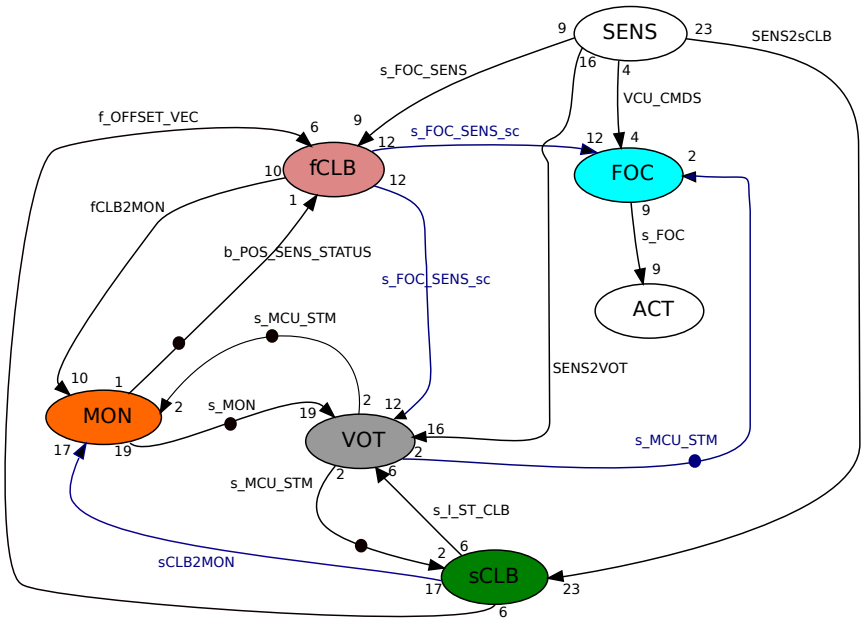
7.4.4 Mapping

As mentioned above, we will consider two different inter-processor communication styles for the motor control use-case. In the first style (single-beat see Fig. 7.11b), for realizing inter-processor communication between actors which are mapped to different tiles, we use the `SRI` and `LMU` resources, while in the second style (burst transfer see Fig. 7.11a) the `SPB`, `DMA`, `SRI` and `LMU` are used.

[10] Typically, a DMA Transaction consists of a number of Transfers, which in its turn consists of a number of Moves. A Move is the basic action of the DMA reading from one (or group of) memory cell(s) and writing to another.

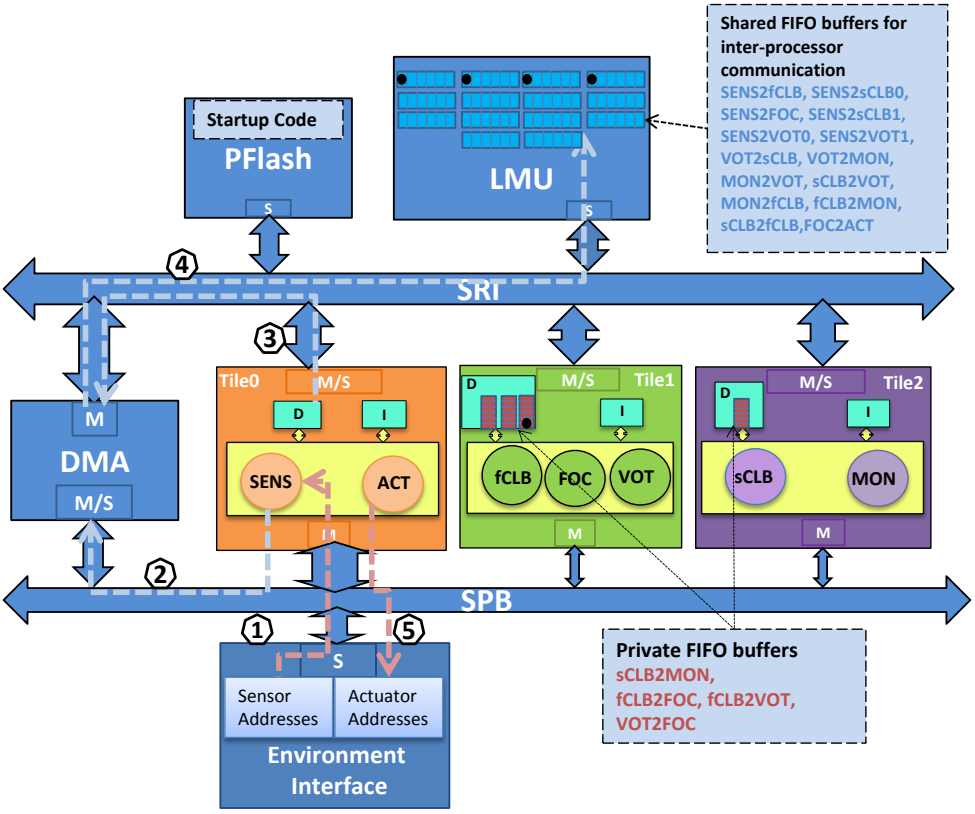


(a) Burst-transfer-aware SDFG

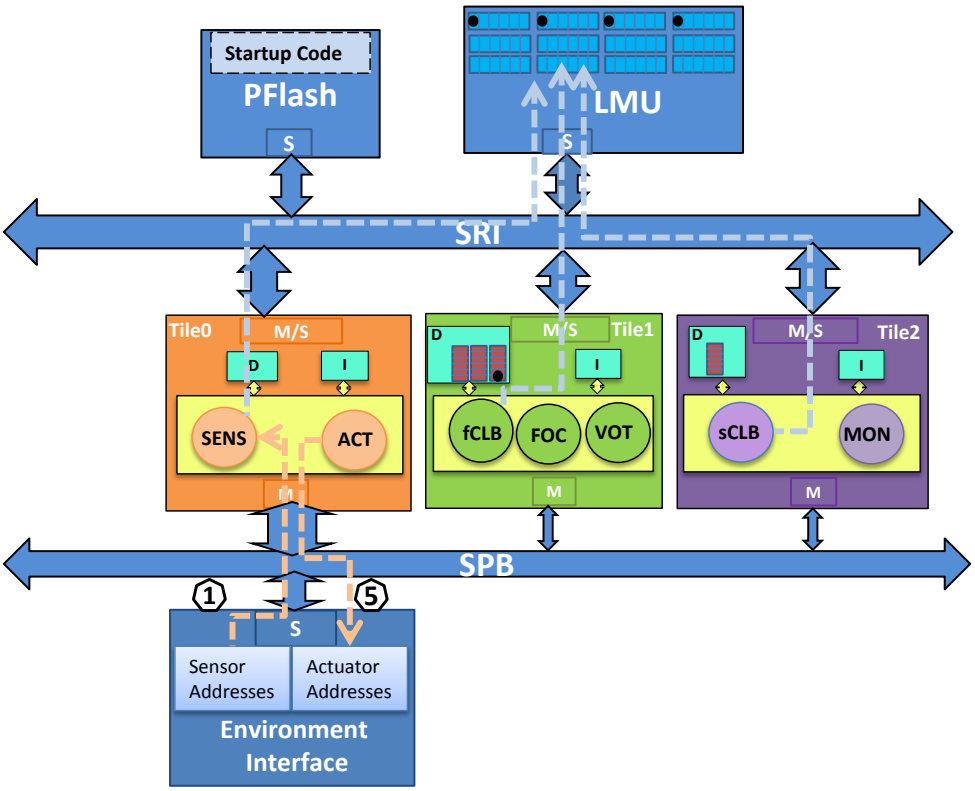


(b) Single-beat-aware SDFG

Figure 7.10: Mapping-aware SDFG



(a) With burst-transfer inter-processor communication via DMA



(b) With single-beat inter-processor communication via SRI

Figure 7.11: Mapping the motor control SDFG to Aurix platform

Fig. 7.11 shows the two styles with a possible mapping[11] of the motor control use-case:

1. Mapping actor(s) `SENS` and `ACT` to `Tile0`, `fCLB`, `FOC` and `VOT` to `Tile1`, `sCLB`, and `MON` to `Tile2`.

   Obviously, for performance reasons, edges `fCLB2VOT`, `fCLB2FOC`, `VOT2FOC`, `sCLB2MON0` and `sCLB2MON1` should be mapped to the local private memories of their corresponding tiles (`fCLB2VOT`, `fCLB2FOC`, `VOT2FOC` to private memory of `Tile1` and `sCLB2MON0`, `sCLB2MON1` to that of `Tile2`) while all other edges (since they invoke inter-processor communication) are mapped to the shared LMU, which can be accessed via interconnects either through the configuration `(SPB, DMA, SRI, LMU)` as shown in Fig. 7.11a or through configuration `(SRI, LMU)` as depicted in Fig. 7.11b.

   Fig. 7.10 shows mapping-aware SDFGs[12], where depending on the inter-processor communication style, changes are applied on the SDFG edges (with corresponding ports' rates) to fit this style. Tab. 7.8 depicts the transformations applied for each style. In the case of a DMA-based inter-processor communication style, a burst transfer of 16 tokens per transfer restricts the maximum transferable data to 12 tokens (each of 32 bytes size) per transfer[13]. The burst-transfer aware SDFG shows how at every edge a maximal amount of 12 tokens (the number of tokens transfered on every edge is depicted in brackets), whereas by the single-beat style this restriction does not exist, allowing edges with rates beyond 12.

2. Calculating a static-order schedule for the SDFG as follows:
   `(SENS)(fCLB)(FOC)(VOT)(sCLB)(MON)(ACT)`

3. No need for choosing a hierarchical scheduling strategy since we are only considering one SDFG.

7.4.5 BCET/WCET Analysis of Software Components on single PEs

With the help of *Simulink Coder* (R2011-b), we are able to generate target C code from the Simulink motor-control model which then manually customize to make compliant to the SDF semantics of the translated SDFG (see pseudo-code

[11] This mapping was suggested by the MotorBrain team [MotorBrain Consortium, 2013] for its parallelization advantages.

[12] In both figures, `sCLB2MON0`, `sCLB2MON1` are merged (for optimizations purpose) to one edge (`sCLB2MON`) with the number of tokens being summed up. This optimization can be made preserving semantics, since these two edges are mapped to the same private memory.

[13] Four tokens are reserved for the FIFO queue implementation-specific control tokens.

Table 7.8: Burst-aware and single-beat-aware SDFG transformations

Original SDFG	Burst-aware SDFG	Single-beat-aware SDFG
b_EMERG_STOP (1) s_MCU_STATUS (5) VCU_CMDS (11)	SENS2VOT0 (6) SENS2VOT1 (11)	SENS2VOT (17)
s_I_SENS_MON (11) s_T_SENS (12)	SENS2sCLB0 (11) SENS2sCLB1 (12)	SENS2sCLB (23)
s_FOC_SENS_sc (5) b_POS_SENS_CLB (5)	fCLB2MON (10)	fCLB2MON (10)

implementation of SDFGs in Sect. 6.5). The generated C code is cross-compiled using the *Hitex compiler* (v4.6.2.0 [Hitex Inc., 2013]) and can be executed on the *Tricore 1.6P* processors.

In this step, each SDF actor's C code generated from Simulink and cross-compiled for the platform should be statically analyzed to obtain its BCET/WCET when executed on the single processors of type TC1.6P. The same analysis should be performed for the communication drivers of the platform that are used to establish the FIFO-style message passing communication between actors. Yet, available commercial WCET analyzers (such as aiT [Ferdinand and Heckmann, 2004]) still do not support the RT analysis of the SW components on novel processors in novel platforms such as the Aurix (up to the date the experiments were made). Due to this fact, we used (minimal/-maximal) execution times measured when running our software components on an accurate virtual-hardware platform model of the Aurix (see Sect. 7.4.6). Thereby the virtual platform enables early flexible and accurate measurements of the system timings without having the hardware available. These execution times are then passed over and annotated in the formal MoP representation and used to configure the TA templates of our analysis framework from (see Sect. 7.4.7) in order to validate the SUA against its real-time requirements.

With the above decision made, care should be taken when considering the results achieved via our formal analysis, since these are no longer valid upper bounds on the application execution time, as they depend on measured timings instead of WCET timings. But since the annotated timings in the formal MoP of SDF2TA (see Sect. 6.4) are the worst-case measured timings bounds, our claim stating that "*the measured WCET times should always be below the ones computed by our SDF2TA tool*" still holds, and will be shown in the following section.

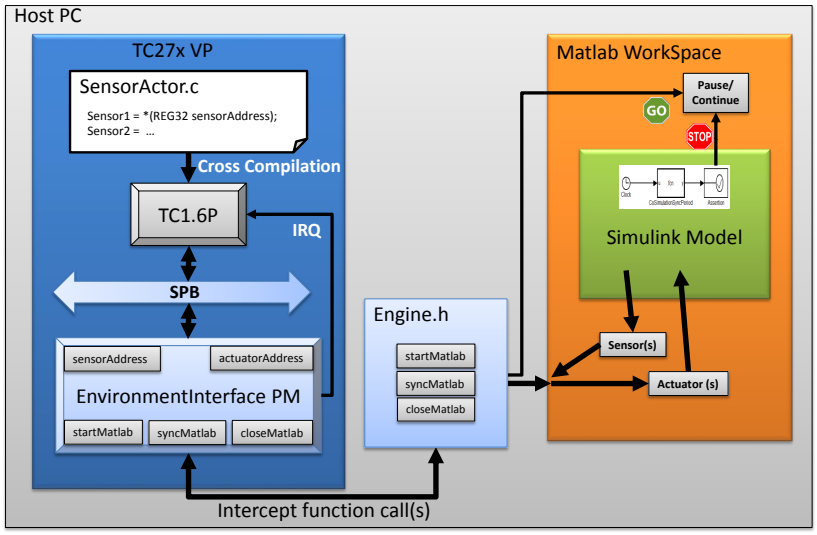


Figure 7.12: VPIL simulation setup for Aurix platform (based on [Poppen, F. and Grüttner, K., 2012])

7.4.6 VPIL Simulation for Aurix TriCore

In order to realize the VPIL simulation (see Sect. 6.4), we use a cycle-accurate virtual platform (VP) of the *Aurix TC275* (Aurix_TC27x_VSP_v1.6_2_Release), which has been developed by Infineon using the Synopsys Virtualizer-CoMET (G2012.06-SP1 [Synopsys Inc., 2015]) tool. Especially for new hardware platforms, like the Aurix, the virtual platform enables early V&V of the SW implementation and the approached hardware design, instead of waiting for the first engineering samples to be developed, manufactured, and delivered.

For code cross-compilation, we use the *Hitex compiler* (v4.6.2.0 [Hitex Inc., 2013]) which enables the execution of the generated code of the motor control SDFG, according to the mapping seen in Fig. 7.11 on the TriCore 1.6P processors. The VPIL framework infrastructure is depicted in Fig. 7.12. On the left side, only relevant components to the VPIL of the *Infineon TC275 VP* can be seen (only `Tile0` can be seen, the other two tiles are not depicted for visualization clarity purposes). We extended the Aurix VP by developing a *EnvironmentInterface Peripheral Model (PM)* (written in a PM C++ API of Virtualizer-CoMET tool) and connected it to the System Peripheral Bus (SPB). Attached to the simulated bus of an instruction-set simulator of the *TC1.6P* processor model, the PM enables the exchange of sensor(s) and actuator(s) data with the Matlab/Simulink model via memory-mapped IO. Since the *EnvironmentInterface PM* occupies an address space in the *TC275* simulated system on chip, any read/write transaction to this address space on the SPB bus is forwarded to Matlab/Simulink using Mathworks engine API for data exchange.

All values exchanged had a single-precision floating-point types.

The single steps of the bi-simulation are described in the following: first, all the VP components are initialized. When initialized, the *EnvironmentInterface PM* calls a defined function from `engine.h` library (`startMatlab()`) to launch an instance of the Simulink model. The Simulink model executes till it reaches a synchronization point (defined through a sample time of the algorithm: 100 μs) where it automatically pauses (time pausing module: top right of Fig. 7.12). The *EnvironmentInterface PM* detects this pause (`syncMatlab()`) and it gets the current sensor(s) values from the Simulink motor model. Since the used virtual-platform framework (*Virtualizer-CoMET*) supports built-in *Function Tracing* with cycle-accurate time measurement capabilities, it is not necessary in this case to use our generic interrupt-based time measurement method presented in Sect. 6.4. Instead two synchronization techniques between the top-level Simulink model and the virtual-platform framework are now possible: either `Tile0` polls some register(s) at the PM, till sensor(s) data are available or the PM component issues an interrupts to notify `Tile0`. The activated actors on other tiles can then get the sensor values from the sensor actor running on `Tile0` which communicates exclusively with the *EnvironmentInterface PM*. The updated actuator values are sent to back to the actuator actor which forwards them to the *EnvironmentInterface PM*. During software execution, the *end-to-end* execution time which is the time from the moment where the sensor(s) data is received until the moment where the last actuator was updated (including the inter-processor communication and synchronization) can be recorded. After the execution completion of the *control-step* at the VP implementation level, the *EnvironmentInterface PM* sends the updated actuator values back to Simulink (calling `syncMatlab()`). At the same time, it wakes up Simulink to resume the execution of another control-step. This procedure can be iterated until the desired number of control-steps has been executed and the bi-simulation is then terminated (calling `closeMatlab()`).

7.4.6.1 Simulation Results

Fig. 7.13 shows the functional results of the VPIL simulation done for one given scenario which is triggered by setting the requested torque to 50 $N.m$. Simulating 10000 control-steps (1 sec simulation time) with basic sampling rate of 100 μs took 944 sec (15.7 min) of real time. This means that we need 15 minutes for every test-case, which is a reasonable time, taking into consideration that the virtual-platform model is a very accurate one. The measured values plotted together with the reference top-level Simulink model results are shown in Fig. 7.13, where only the outputs of the FOC component were considered. The reference `FOC` model actuator values (dotted line) and the measured actuator

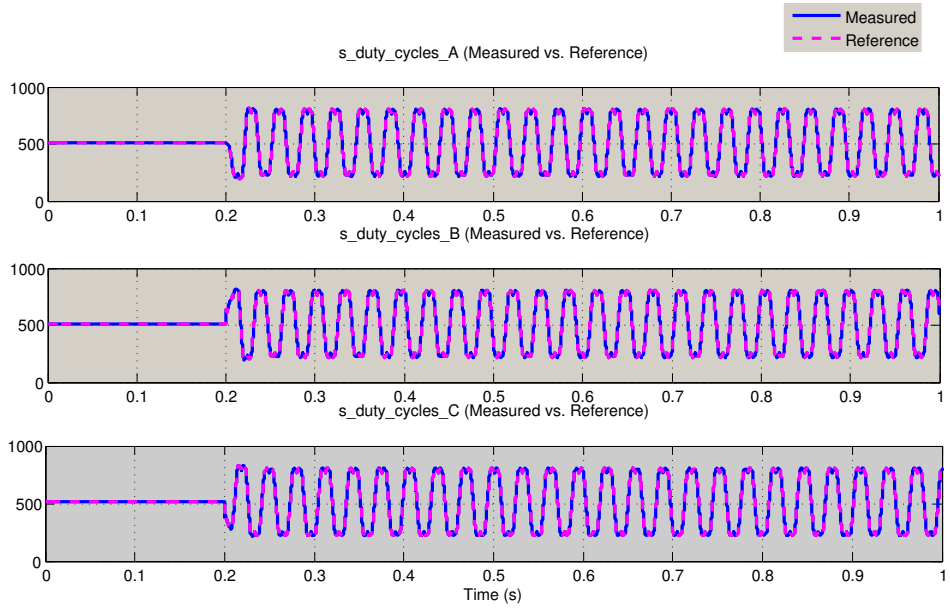


Figure 7.13: VPIL simulation functional results (for one test-case) of the reference duty cycles (A, B, C) output values of the 3-phase FOC at Simulink level (violet/dashed) and the measured ones at the virtual-platform level (blue/non-dashed)

values (thick solid line) show exactly the same results[14]. Thus, the functionality of the generated code mapped to the multicore platform has been successfully verified against the golden reference model in Simulink for the given scenario.

Another important issue is to examine whether or not our timing requirement was satisfied. In order to answer above question, we need to answer the following question: Is the end-to-end latency of the application starting from reading sensor values, executing the SDFG application till updating the actuator values always less than 100 μs[15] or not?

In the following, we will analyze the implementation of two inter-processor communication styles and give the measurements' results. For this we utilized the function tracing tool (*VPviewer*) in CoMET-Virtualizer to get a trace the duration of single functions' execution on the *TC275* virtual platform. We first implemented the inter-processor communication using the DMA controller (with a fixed-priority arbitration) as shown in Fig. 7.11a. In step ① the `SENS` actor polls for new sensor data which are interfaced with an environment interface (more details about this are found in Sect. 6.4). As soon as the sensor data are available, the `SENS` actor propagate these values through the DMA by writing to corresponding buffers (e.g. `SENS2fCLB`, `SENS2sCLB`). To perform

[14]In Fig. 7.13, the outputs were of a 3-phase FOC controller, but since the 9-phase FOC control algorithm consists of 3× 3-phase FOC, the validation is done similarly.

[15]This requirement was the basic timing requirement imposed by the Motorbrain Team.

Table 7.9: Timing measurements of the motor control application (in cycles with a 300 MHz clock)

Operation	CALIB_SLOW	FOC	VOTER	MONITOR	CALIB_FAST	SENS	ACT
t_{comp}	[811, 844]	[41, 1397]	[231, 1389]	[78, 255]	[223, 247]	[344, 344]	[34, 34]
$t_{com_{single}}$ (with 32 bits/transfer)	839	290	1031	716	588	1114	207
$t_{com_{dma}}$ (with 512 bits/transfer)	2680	1072	3216	2144	2144	3216	536
$end\text{-}to\text{-}end_{single}$				[20, 25] μs			
$end\text{-}to\text{-}end_{burst}$				[60, 90] μs			

these transactions, `Tile0` first configures the DMA through SPB in step ②. Afterwards, in ③ and ④ the DMA copies the values of the sensors stored in the local memory of `Tile0`, to the LMU corresponding locations.

Fig. 7.11b shows the single-beat inter-processor communication procedure, with step ① and step ⑤ remaining the same as in burst-transfer IPC procedure. Instead of using the DMA, the actors now write/read directly to/from the shared FIFO buffers through the SRI crossbar switch. The SRI also uses a fixed-priority arbitration scheme (where the tile with the least id has the highest priority) that arbitrates after every data word (of size equal to 32 bits) transfer. Before writing or reading to/from the FIFO buffer, every actor checks a variable named `size`, to see if enough space/data is available. If this is not the case, polling-wait time (of 3454 cycles for the burst IPC and 348 cycles for the single-beat one[16]) is used to wait until space/data is available.

Tab. 7.9 depicts the results of the timing measurements which were the outcome of a series of test-cases (mainly defined by Motorbrain team [MotorBrain Consortium, 2013]) conducted on the corresponding implementations. Some of these test-cases were stress tests which had the role to stress the single actors in order to obtain maximal-execution times in the simulation on the one side, and to invoke others for obtaining minimal ones on the other side. We found out through the measurements obtained (see in Tab. 7.9) that the end-to-end latency in the case of a burst-transfer varied between 60 and 90 μs while in the case of a single-beat implementation it varied between 20 and 25 μs, showing a high improvement compared to the burst-transfer one. This means that the timing requirement is fulfilled, at least for the simulated test-cases, according to the simulative RT results.

Please note that $t_{com_{dma}}$ is large due to the overhead of the DMA controller (with its synchronization and software driver overheads). For more information about the simulation parameters please refer to Appendix B.1

[16]The polling-waiting time value in the case of burst transfer is about ten times larger than that of the single-beat transfer. This is due to the implementation-specific timing requirement imposed by the considered DMA to wait for extra time before being able to receive any configurations.

7.4.7 SDF2TA RT Results with different Communication Styles

In this section, we will describe how we utilized our SDF2TA tool to obtain upper and lower bounds on the execution times of the two implementations described above (for details see Appendix B.2). For the RT analysis of the burst-transfer based implementation the following further assumptions were made:

B1 Assume worst-case communicating time on SPB while setting up the DMA. This means that if any tile initiates the configuration of the DMA through the SPB, it must be assumed that the other two tiles are doing the same, and thus must wait for the maximal time delay before getting the configuration completed (which is equal to the sum of the response times of the active actors on the other two tiles). This leads indeed to more pessimistic results, but allows us to abstract from modeling the SPB and simplifies our MoP state space which should be explored in our model-checking based performance analysis method.

B2 During a DMA access on the SRI, no other master is allowed to access the SRI. With this, we are able to simplify the SRI model. The simplified SRI model adds the burst's transport delay of the SRI to the overall DMA delay (for details see Appendix B.2).

By doing above assumptions for the burst-transfer implementation (see Fig. 7.11a), we only need one interconnect timed automaton for modeling the DMA-based IPC, annotated with both the latency delays of the DMA and the SRI and configured with the DMA specific arbitration (see Appendix B.2). In this case, the delay of the DMA configuration on the SPB can be merged with the delay of the communication driver and annotated to the communication driver timed automaton. Similarly, for the case of a single-beat implementation (see Fig. 7.11b), also one interconnect timed automaton is needed but this time this automaton is configured by the timing delays of the SRI and its arbitration.

We configured our SDF2TA tool with the simulative measurements (of the actors) and the annotations of the hardware latencies (bus delay and memory accesses, for detailed annotations please refer to Appendix B.2) and evaluated the two different styles (see Sect. 7.4.4) of the considered motor controller use-case. The obtained RT results are depicted in Tab. 7.10. We notice that in case of the DMA implementation, the model-checker was able to find a violation of the real-time requirement of the motor control application (of 100 μs) exhibiting an end-to-end latency of $115,7\mu$s. Unfortunately, the analysis of the single-beat implementation through our RT method for the full ranged execution times' intervals of every actor (see Tab. 7.9) lead to memory exhaustion. The end-to-end latency values for the single-beat implementation in Tab. 7.10 were obtained

Table 7.10: State-based versus simulative RT analysis results for single-beat and burst-transfer implementations

	Single-beat	**Burst-transfer**
Simulative end-to-end latency	[20, 25] μs	[60, 90] μs
State-based end-to-end latency	[19.4, 39.25] μs	[70.8, 115,7] μs
% Overapproximation State-based vs. simulative	Up to 57%	Up to 28.5%

for fixed-point intervals of every actor once by setting its execution time to its worst-case measured execution time and once to its best-case measured execution time. Nevertheless, the obtained results for the single-beat implementation with fixed-point intervals are far away from violating the imposed RT requirement. In addition, we observe a higher over-approximation (w.r.t simulation) in case of the single-beat inter-processor communication style (up to about 57%) compared to the burst-transfer (up to about 28.5%).

7.4.8 Discussion

Concerning the simulative results and keeping in mind that the whole application execution should execute within a maximal time of 100 μs, we observe that, in the case of a DMA based inter-processor communication, the measured end-to-end latency is indeed high. This is due to the overhead of the DMA driver software, the synchronization mechanisms and the hardware latencies which are imposed at every transfer. Obviously the usage of a DMA is only reasonable when large amount of the data is being transported. If this is not the case, large execution time penalties would be imposed on the overall application (compare $t_{com_{single}}$ and $t_{com_{dma}}$ in Tab. 7.9). Nevertheless, it is important to note that the measurements obtained in the above experiment were the outcome of a first-shot implementation, which still has much room for optimizations and thus being capable of improving its efficiency drastically (for e.g. optimizing the driver code of the DMA or the DMA synchronization mechanism which was out of the scope of this thesis).

Concerning the RT results of our SDF2TA tool, the violation detected in the burst-transfer implementation, was expected since the simulative timing measurements of SUA, were already high (up to 90 μs) and very close to violate the timing requirement (of 100 μs). Another reason for this, is the over-approximation of the DMA configuration timings in the analytical model (see assumption B1). This assumption is also responsible for the result in Tab. 7.10 showing that the best-case (BC) the *BCend*2*end* obtained by SDF2TA was greater

than the one obtained via simulation, which definitely should not be the case. Yet, relaxing assumption B1 would require the explicit modeling of the SPB together with an accurate representation of communication driver accesses from different tiles trying to configure the DMA. Modeling the SPB is not an issue as our method supports the modeling of multiple interconnects (see Sect. 5.2.5), yet we still have chosen not to do that, since this special case would require the modeling of the DMA complex communication driver at the instruction-level granularity for the accurate representation of the configuration phase and contention on the SPB. This in turn would lead to an unmanageable state space of the SUA. Ideally, to overcome such an issue, the target platform can be built enabling a unique access via a private interconnect of the PE to the DMA component (as made in [Shabbir et al., 2010]).

As for the single-beat implementation, with the full-interval variation of the actors' execution times, SDF2TA fails to compute lower/upper bounds on the end-to-end latency. This can be justified by the fact that in general, single-beat style does not perform as well as the burst-transfer in terms of scalability, due to the fact that the number of possible interleaving of different cores' accesses to the shared interconnect is much higher than that of the burst transfer. Another reason is the interval distance between best-case measured values and the worst-case measured values which could lead very quickly to a state explosion (as seen even in the case of 2-tiles Sect. 7.2.3).

Similarly, the higher over-approximation of the analytical results w.r.t simulative results (see Tab. 7.10) in case of the single-beat inter-processor communication style, is due to the fact that the number of possible interleaving of different cores' accesses to the shared interconnect is much higher than that of the burst-transfer IPC.

7.5 Summary

In this chapter, we elaborated on the model-checking capabilities which enable us to check more complex properties (see claim C2-3) than those analyzed by analytical RT methods (see analytical Sect. 3.1.1.2). The model-checking TCTL statements also allowed us to gain confidence whether or not the instantiated timed-automata templates correctly capture the semantical behavior of a specific SUA through checking the validity of SDF semantics and other properties such as liveness and arbitration protocols' correctness.

The viability of our state-based RT method (see claim C4 in Chap. 1) was approved by conducting a set of scalability tests which showed that our method has the potential of scaling up to 320 actors on a 2-tiles platform and up to 96 actors on a 4-tiles platform (for the chosen use-case), significantly improving the number of analyzable actors compared to related work (see claim C4-3).

We have also shown how the clustering mechanism when applied can help reducing the state space and the needed analysis time (in the MP3 example 4.3% less state space and 48.6% less analysis time). In addition, by enabling composability and combining a TDMA-based SDFGs' cluster scheduling with our state-based method, the RT analysis of larger SDF applications (by TDMA slot number of 10, potentially thousands of actors on a 2-tiles platform and hundreds of actors on a 4-tiles platform if the use-case has the same scalability behavior as the one in Fig. 7.2) with large number of actors is enabled.

In addition, our approach showed a significant precision improvement (up to a percentage improvement of 300%) compared with the worst-case bound calculation based on a pessimistic analytical upper-bound delays for every shared resource access (see claim C4-2).

Finally, we have demonstrated the applicability of our model-based design flow on an industrial use-case (see claim C4-1) using a multi-phase electric motor control algorithm (modeled as SDFA) mapped to the Infineon's TriCore hardware platform with both the burst and single-beat inter-processor communication styles. We have shown that the upper bounds timing results estimated through our approach were always a safe over-approximation of the measured (through cycle-accurate simulation) ones and that our state-based RT analysis was able to detect timing violation in the case where a burst inter-processor style was chosen.

Chapter 8

Conclusion and Outlook

In thesis, we started from the observation that MPSoCs are emerging due to their performance and power efficiency, and that the real-time analysis of applications with hard real-time requirements on such architectures is not an easy task requiring novel RT analysis methods. The underlying research of this thesis tried to address the problem whether or not a state-based RT method is applicable for RT analysis of multiple applications restricted to synchronous data-flow model of computation when run on MPSoCs with shared communication resources. More centrally, we tried to handle the challenge of choosing an appropriate abstract representation (in timed automata) of the SDFG applications, the MPSoC and their temporal behaviors and interactions, while still enabling tight timing results' prediction.

By combining the flexibility of timed automata with the efficiency of SDF graphs, we enabled a state-based RT analysis of multiple hard real-time SDF applications mapped to an MPSoC platform with shared communication resources, considering variable access delays due to the contention on communication resources and utilizing different inter-processor communication styles (such as burst/single-beat). This was realized through the implementation and state-space exploration of a set of flexible timed-automata templates capturing execution times boundaries of SDF actors and their scheduling decisions, mapping and utilization of MPSoC resources, shared communication resources access protocols (including arbitration of various complexities) and local/shared memories. These TA templates are also capable of representing a class of MPSoCs respecting (or which can be configured to respect) the constraints imposed in this thesis (see Sect. 4.1).

Since the state-space explosion problem is the main bottleneck which faces a system designer when utilizing state-based RT analysis methods, we examined methods which helped improving the state space of our implemented TA

templates. In a first approach, we proposed some optimizations on our TA templates to minimize the state space. In addition, techniques from literature such as clustering of actors and extending the MPSoC with extra hardware components which guarantee temporal and spatial isolation of clusters of actors (combined with a TDMA clusters' scheduler), were examined to be useful in terms of improving the scalability of our approach and their application to our system model was described.

The viability of our RT method was approved by conducting a set of scalability tests which showed that our method scales up to 320 actors on a 2-tiles platform and up to 96 actors on a 4-tiles platform, significantly improving the number of analyzable actors compared to related work. We have also shown how the clustering mechanism when applied can help reducing the state space and the needed analysis time (in the MP3 example, 4.3% less state space and 48.6% less analysis time). In addition, by enabling composability and combining a TDMA-based clusters' scheduling with our state-based method, the RT analysis of even larger SDF applications (e.g. by ten TDMA slots, potentially thousands of actors on a 2-tiles platform and hundreds of actors on a 4-tiles platform) with large number of actors was demonstrated. Moreover, our method showed a significant reduction in the worst-case response time prediction (up to a percentage improvement of 300%), compared with the worst-case bound calculation based on a pessimistic analytical upper-bound delays for every shared resource access known from literature. In addition, our approach enabled the analysis of more complex properties than those supported by traditional analytical RT methods such as the safety, liveness and reachability properties.

Finally, we have demonstrated the applicability of our suggested model-based design flow being able to validate the timing requirements of a small industrial use-case of a control algorithm (modeled as SDFA) of a multi-phase electrical motor mapped to a TriCore-based Aurix hardware platform with different inter-processor communication styles (burst and single-beat IPC). We have shown that the upper bounds timing results estimated through our approach were always (for the scenarios experimented) a safe over-approximation of the measured (through cycle-accurate simulation) ones and that our state-based RT analysis was able to detect a timing violation in the case where a burst inter-processor style was chosen.

Overall, our approach opened up the way for using timed automata with its model-checking features for the RT analysis for SDFGs running on MPSoCs (see Sect. 3.1.2.3). In addition, our proposed RT analysis method feasibility was demonstrated for small parallel systems, enabling their usage in safety-critical real-time domain (such as avionics) providing formal guarantees on the absence of timing hazards.

8.1 Discussion

The challenge which we faced when developing our state-based RT analysis method, is how to choose the right abstraction level of the input model such that the method scales to be able to analyze systems with adequate sizes and at the same time can still obtain tight timing results. For this purpose, we deliberately made the assumptions and restrictions described in Sect. 4.1 to enable such a state-based RT analysis of SDF applications mapped to an MPSoC. While the applicability of our method was demonstrated in the conducted experiments, there are still some issues that should be discussed. Restrictions made in this thesis are considered to be very realistic for safety-critical domains, for e.g. in the avionics domain. In these domains, costs resulting from adapting such restrictions are typically compromised as long as they help passing the certification procedures imposed by authorities to approve the deployment of the target MPSoC system. Nevertheless, the price to be paid, when imposing such restrictions, could be critical in other domains.

While SDFGs (see A1) are commonly used for capturing the behavior and implementation of signal-processing applications where infinite streams of signal samples (which can be represented as tokens) are processed [Schaumont, 2013], their expressiveness suffers from control-related limitations. Some of these limitations were stated in [Schaumont, 2013], for e.g. stopping and restarting an SDFG is not possible since an SDFG can have only two states either running or waiting for input. In addition, reconfiguration of an SDFG to be able to (de)activate different parts depending on specific modes is not possible. Moreover, different rates depending on run-time conditions are not supported. Also modeling exceptions which might require deactivating some parts of the graph is not possible. However, emulating control flow within the SDFG is possible even though not always efficient (c.f. [Schaumont, 2013]). In addition, control-flow within an actor functionality is allowed, the fact which enabled us translating event-triggered systems in Simulink into SDFGs (refer to Sect. 6.2). We also relaxed some of the SDFG MoC limitations in this thesis by enabling SDFG graphs to be sensitive to external periodic events allowing us to support the RT analysis of periodic control systems.

Static allocations of actors (see A1), static-order and non-preemptive scheduling (see A3) (incl. non-preemptive arbitration in A9, non-support of hardware interrupts in A4) can be very costly in terms of resource utilization. The fact which can lead to expensive and thus non-competitive designs. On the other side, a variety of dynamic implementations which are reconfigurable (e.g. adapting different allocations, scheduling for different situations) depending on dynamic changes in the environment cannot be supported when making above restrictions. Also the fact that we restrict external events to be periodic (see A2) decreases the flexibility of our approach to handle a set of applications

in the safety-critical domain such as those sensitive to sporadic events.

Moreover, constraining the application code to be mapped to the private memory of the corresponding processor (see A6), leads to a limitation concerning the size of private memories particularly for large applications. Nevertheless, recent research recommendations and current design trends are moving in this direction where private tasks' code is stored in private (growing-larger) memories and only message-passing (see A7) is realized via communication resources (esp. in the emerging NoCs designs). In addition, the non-usage of shared caches could also lead to a performance degradation of the application overall execution time. In the industrial example demonstrated in Sect. 7.4, however, the Aurix local memories were very fast so that the execution time measured without caches was even better than that measured when using caches[1]. In addition, prohibiting contention on interconnect bridges and IO devices (see A8, A10) could be too strict for some applications even though using a dedicated processor element (I/O PE) which is exclusively allowed to communicate with I/O devices seems to be a typical decision in real-life implementations.

Overall, the restrictions imposed in this thesis were deliberately made to obtain a manageable state space. However, most of these restrictions can be easily relaxed in future work if the ongoing research achieves more powerful model-checkers with more capabilities. Future model-checkers could highly benefit from the growing computing power and can utilize for e.g. many-cores to enable the concurrent exploration of the SUA given state space instead of the currently supported single-threaded approach.

8.2 Future Work and Open Questions

Based on the method developed in this thesis, the most relevant extensions which can be addressed in future work are presented in the following:

Improving scalability Concerning scalability, it was shown in the experiments that our method is applicable to small size industrial use-cases. This method could be easily driven to its limits (as this is a general problem of state-based RT analysis methods) when the non-determinism in the system is increased for e.g. in the case where the difference between BCET and WCET times is high. In such a case, one could think of trying to constrain the implementation in order to decrease/eliminate the difference between BCET and WCET, for e.g. by enforcing all states of the executed code

[1] The comparison was made between the implementations in Sect. 7.4 and another implementation where the application code is mapped to the shared memory (not to the private one) and the cache is activated.

to follow the WCET with the help of a run-time monitor[2] as suggested in [Nowotsch et al., 2014, Wolf et al., 2012].

Another approach would be to optimize the model-checker for the given problem set, for e.g. through utilizing a multi-core capable model-checker (instead of the current UPPAAL tool) such as `opaal+LTSmin` [Dalsgaard et al., 2012] to tackle the state-space explosion problem. Statistical model-checking [David et al., 2011] (found to be a good alternative to exhaustive computation of WCET on single-processor platforms [Béchennec and Cassez, 2011]) can also be used to obtain the probability distribution of the execution times and improve the scalability of our approach. In general, if the probability that a critical violation of the real-time requirement is adequately low then this would be acceptable even for hard real-time applications.

In the case of hierarchical scheduling, a composable TDMA scheduling improves the number of analyzable SDFGs on the same platform compared to other strategies, also using a TDMA arbitrated interconnect can help.

Possible Architectural Extensions Even though high-speed private scratch-pads with increasingly larger sizes are emerging to current hardware platforms, the sizes of these are still considered as a bottleneck especially when dealing with applications of large memory footprint. In this case, the architecture could be extended with an off-chip large memory with slower access latencies, which can be shared between the processors. In order to retain the predictability through RT analysis method, this large memory should be binded with the help of a TDMA memory controller to the current architecture, allowing a clear temporal separation of accesses to shared memory. In this case a cache (which is supported by the WCET analyzer) could buffer the instructions between the local and the off-chip memory to achieve a better performance, keeping the SUA analyzable by our approach.

Additionally, the MPSoC architectural constraints could be relaxed towards other kinds of communication resources such as cross-bar switches and NoCs with flexible arbitrations. Extending the interconnect model towards Network on Chips (NoCs) should be straight forward as burst-transfer modes are already supported. Nevertheless, NoCs are only meaningful for large number of tiles which are connected through it.

Furthermore, in order to be able to introduce caches in the current system model, the model should be able to consider single tiles loading/writing instructions from/to the shared memory which requires an instruction abstrac-

[2] A hardware monitor component can be configured (with the WCET) as a watchdog which monitors the execution time of actors and in case it finishes before the assigned time, it blocks and in case of lasting longer than the expected WCET then additional time could be assigned or an error is detected since WCET must be always larger than measured time.

tion level in order to be able to model the behavior of caches and contention on the interconnects. This in turn will hit the state-space wall as already noted in Sect. 3.1.2.1.

Extending the Scheduling Mechanisms Extending our model towards preemptive schedulers is a difficult issue, since by preemption the current state of the actor should be saved and a context switch should be done. This is difficult in our current model, since we abstract the execution of actors in terms of upper/lower bounds and we don not consider the actor at the instruction level of granularity (as done for e.g. in [Lv et al., 2010]).

Typically, scheduling involves three steps with each one can be performed either at run-time or at compile-time [Sriram and Bhattacharyya, 2000]:

1. Assigning the tasks to processors
2. Determining the order in which tasks may run
3. Setting the start times at which the tasks will be executed

There are many scheduling strategies for SDFG (see [Sriram and Bhattacharyya, 2000]), ranging from static (fully static, order transactions, self-timed) to quasi-static to dynamic (static assignment, fully dynamic) strategies. Future work should explore these options and their support should be analyzed.

In this work, we made a simplification of the general case of Def. 5.4.1 concerning at which granularity we allow to construct the clusters and assumed that a cluster can consist of a number of SDFGs and these are independent from other SDFGs mapped to other clusters. In future work, the general case, permitting clustering at the granularity level of actors (see Fig. 5.13) should be examined and it procedures should be analyzed to assure that such clustering never leads to a deadlocks.

The blocking behavior on the shared FIFO should support besides busy-waiting (considered in this thesis) suspension-based approaches[3] to enable comparison between them when making decisions for a SDFGs binding and scheduling. In general, in order to model preemption *stopwatch automata* (TA with stoppable clocks) are required (in order to stop the execution time of preempted transaction) which are supported by current versions of UPPAAL and

[3]Suspension-based blocking mechanism (realized through interrupts) are useful in the case of shared hardware FIFOs which could notify the blocked actor on the target processor when data are available.

for which an over-approximated but efficient reachability analysis[4] can still be applied [Cassez and Larsen, 2000].

Enabling RT-analysis of more Dynamic Data-flow MoC Although the SDFGs offer good features for analyzability (e.g. deadlocks and bounded buffer properties are decidable for such models [Lee and Messerschmitt, 1987b]), they lack expressiveness. Future work should take into consideration more expressive extensions of SDFGs and analyze their predictability and evaluating how far our state-based RT method can handle such systems. One example is the Scenario-Aware Data-Flow (SADF) MoC [Skelin et al., 2015]. This MoC uses a data-flow model to represent a specific scenario and it uses either a stochastic (Markov chain) approach or a finite state machine to model the order in which scenarios occur. A first sketch of an approach, based on our work, targeting the state-based RT analysis of FSM-SADFGs on MPSoCs with shared memory communications was accepted to be published in [Stemmer et al., 2016].

Another interesting more dynamic MoC is SysteMoC [Falk et al., 2005] where applications are modeled similar to SDFGs as a graph of atomic actors which communicate through FIFO queues but in difference to SDFGs, actors' production and consumption rates are variable (which is the property of dynamic data-flow graphs) [Gajski et al., 2009].

Towards Design-space Exploration Benefiting from clean semantics of the SDF MoC being able to easily distinguish communication from computation parts in the application, the complexity of the mapping and platform alternatives can be compositionally managed. Flexible mapping to different target platforms is now enabled, and with the help of our composable timing analysis method analyzing different mappings is possible. Future work should address supporting design-space exploration in our model-based design flow (similar to [Büker, 2013, Rosvall and Sander, 2014]) exploring mappings with predictability, performance efficiency and costs as optimization goals. Genetic algorithms could also be used for encoding mapping problem as in [Stulova et al., 2012].

Additionally, our timed-automata representation could be extended (with the help of Priced Timed Automata: PTA [Behrmann et al., 2005]) to support energy optimal mapping exploration of power-aware SDFGs on MPSoCs (similar to [Zhu et al., 2014, Zhu et al., 2015]).

[4]The reachability problem is in general undecidable for stopwatch automata [Suman and Pandya, 2006], an over-approximating but efficient reachability analysis is shown to be decidable in [Cassez and Larsen, 2000].

Bibliography

[IEC, 2010] (2010). IEC 61508. Functional safety of electrical/electronic/programmable electronic safety-related systems.

[Abel et al., 2013] Abel, A., Benz, F., Doerfert, J., Dörr, B., Hahn, S., Haupenthal, F., Jacobs, M., Moin, A. H., Reineke, J., and Schommer, B. (2013). Impact of resource sharing on performance and performance prediction: A survey. In *CONCUR 2013–Concurrency Theory*, pages 25–43. Springer.

[Aeronautical Radio, 1992] Aeronautical Radio, I. (1992). RTCA DO-178B. Software Consid- erations in Airborne Systems and Equipment Certification.

[Aeronautical Radio, 2003] Aeronautical Radio, I. (2003). Arinc 653: Avionics application software standard interface. Technical report, ARINC, 2551 Riva Road Annapolis, MD 21401, U.S.A.

[Ahmad et al., 2014] Ahmad, W., de Groote, E., Hölzenspies, P. K., Stoelinga, M. I. A., and van de Pol, J. C. (2014). Resource-constrained optimal scheduling of synchronous dataflow graphs via timed automata. In *Proceedings of 14th IEEE International Conference on Application of Concurrency to System Design (ACSD)*. IEEE.

[Akesson et al., 2010] Akesson, B., Molnos, A., Hansson, A., Angelo, J. A., and Goossens, K. (2010). Composability and Predictability for Independent Application Development, Verification, and Execution. *Multiprocessor System-on-Chip: Hardware Design and Tool Integration*, page 25.

[Alur et al., 1990] Alur, R., Courcoubetis, C., and Dill, D. (1990). Model-checking for real-time systems. In *Logic in Computer Science, 1990. LICS '90, Proceedings., Fifth Annual IEEE Symposium*, pages 414–425.

[Alur and Dill, 1990] Alur, R. and Dill, D. L. (1990). Automata for modeling real-time systems. In *Proceedings of the Seventeenth International Colloquium on Automata, Languages and Programming*, pages 322–335, New York, NY, USA. Springer-Verlag New York, Inc.

[Alur and Dill, 1994] Alur, R. and Dill, D. L. (1994). A Theory of Timed Automata. *Theoretical Computer Science*, 126:183–235.

[Andersson et al., 2010] Andersson, B., Easwaran, A., and Lee, J. (2010). Finding an upper bound on the increase in execution time due to contention on the memory bus in COTS-based multicore systems. *ACM Sigbed Review*, 7(1):4.

[ARM, 2006] ARM (2006). AMBA 3 AHB-lite protocol v1.0 specification. Technical report, ARM.

[Baleani et al., 2005] Baleani, M., Ferrari, A., Mangeruca, L., Sangiovanni-Vincentelli, A. L., Freund, U., Schlenker, E., and Wolff, H.-J. (2005). Correct-by-construction transformations across design environments for model-based embedded software development. In *Proceedings of the Conference on Design, Automation and Test in Europe - Volume 2*, DATE '05, pages 1044–1049, Washington, DC, USA. IEEE Computer Society.

[Banks Jerry, 2005] Banks Jerry, John S. Carson, B. L. N. u. D. M. N. (2005). *Discrete-Event System Simulation.* Pearson Prentice Hall,.

[Bartolini et al., 2010] Bartolini, A., Cacciari, M., Tilli, A., Benini, L., and Gries, M. (2010). A virtual platform environment for exploring power, thermal and reliability management control strategies in high-performance multicores. In *Proceedings of the 20th symposium on Great lakes symposium on VLSI*, pages 311–316.

[Basten, 2008] Basten, T. (2008). Reliable embedded multimedia systems? `http://www.es.ele.tue.nl/~tbasten/presentations/infcoll20080221.pdf` (07.05.2015). Openning remarks, 2nd Artist workshop on models of computation and communication last accessed on 01.11.2015).

[Béchennec and Cassez, 2011] Béchennec, J. and Cassez, F. (2011). Computation of WCET using program slicing and real-time model-checking. *CoRR*, abs/1105.1633.

[Behrmann et al., 2005] Behrmann, G., Larsen, K., and Rasmussen, J. (2005). Priced timed automata: Algorithms and applications. In de Boer, F., Bonsangue, M., Graf, S., and de Roever, W.-P., editors, *Formal Methods for Components and Objects*, volume 3657 of *Lecture Notes in Computer Science*, pages 162–182. Springer Berlin Heidelberg.

[Bekooij et al., 2004] Bekooij, M., Moreira, O., Poplavko, P., Mesman, B., Pastrnak, M., and Meerbergen, J. v. (2004). Predictable embedded multiprocessor system design. In Schepers, H., editor, *Software and Compilers for Embedded Systems*, number 3199 in Lecture Notes in Computer Science. Springer Berlin Heidelberg.

[Bengtsson and Yi, 2004] Bengtsson, J. and Yi, W. (2004). Timed Automata: Semantics, Algorithms and Tools. In *In Lecture Notes on Concurrency and Petri Nets. LNCS 3098*, pages 87–124. Springer-Verlag.

[Bhattacharyya et al., 1997] Bhattacharyya, S., Murthy, P., and Lee, E. (1997). APGAN and RPMC: Complementary heuristics for translating DSP block diagrams into efficient software implementations. *Design Automation for Embedded Systems*, 2(1):33–60.

[Bjerregaard and Mahadevan, 2006] Bjerregaard, T. and Mahadevan, S. (2006). A survey of research and practices of Network-on-chip. *ACM Computing Surveys*, 38(1):1–es.

[Boland et al., 2005] Boland, J.-F., Thibeault, C., Zilic, Z., and others (2005). Using MATLAB and Simulink in a SystemC verification environment. In *Proceedings of Design and Verification Conference, DVCon*.

[Boström et al., 2010] Boström, P., Grönblom, R., Huotari, T., and Wiik, J. (2010). *An Approach to Contract-Based Verification of Simulink Models*. Number 985 in TUCS Technical Reports. Turku Centre for Computer Science.

[Boström and Wiik, 2015] Boström, P. and Wiik, J. (2015). Contract-based verification of discrete-time multi-rate Simulink models. *Software & Systems Modeling*, pages 1–21.

[Bouchhima et al., 2006] Bouchhima, F., Briere, M., Nicolescu, G., Abid, M., and Aboulhamid, E. (2006). A SystemC/Simulink Co-Simulation Framework for Continuous/Discrete-Events Simulation. In *Behavioral Modeling and Simulation Workshop, Proceedings of the 2006 IEEE International*, pages 1–6.

[Bovet and Crescenzi, 1994] Bovet, D. P. and Crescenzi, P. (1994). *Introduction to the Theory of Complexity*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK.

[Brekling et al., 2008] Brekling, A., Hansen, M. R., and Madsen, J. (2008). Models and formal verification of multiprocessor system-on-chips. *The Journal of Logic and Algebraic Programming*, 77(1–2):1–19.

[Buck, 1993] Buck, J. (1993). *Scheduling dynamic dataflow graphs with bounded memory using the token flow model*. PhD thesis, UNIVERSITY of CALIFORNIA at BERKELEY.

[Buttle, 2012] Buttle, D. (2012). Real-Time in the Prime-Time. `http://ecrts.eit.uni-kl.de/fileadmin/user_media/ecrts12/ECRTS12-Keynote-Buttle.pdf`. ECRTS (KEYNOTE TALK).

[Büker, 2013] Büker, M. (2013). *An Automated Semantic-Based Approach for Creating Task Structures*. Dissertation, University of Oldenburg.

[C. Chang, 2015] C. Chang, R. D. (2015). May-Happen-in-Parallel Analysis of ESL Models using UPPAAL Model Checking. Grenoble, France. Design, Automation and Test in Europe Conference 2015.

[Cai and Gajski, 2003] Cai, L. and Gajski, D. (2003). Transaction Level Modeling: an Overview. In *First IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis, 2003*, pages 19–24.

[Caspi et al., 2003] Caspi, P., Curic, A., Maignan, A., Sofronis, C., Tripakis, S., and Niebert, P. (2003). From Simulink to SCADE/Lustre to TTA: a layered approach for distributed embedded applications. In *ACM Sigplan Notices*, volume 38, pages 153–162.

[Cassez and Larsen, 2000] Cassez, F. and Larsen, K. G. (2000). The impressive power of stopwatches. In *Proceedings of the 11th International Conference on Concurrency Theory*, CONCUR '00, pages 138–152, London, UK, UK. Springer-Verlag.

[Cha and Kim, 2011] Cha, M. and Kim, K. (2011). Automatic Building of Real-Time Multicore Systems Based on Simulink Applications. *Ubiquitous Computing and Multimedia Applications*, pages 209–220.

[Chattopadhyay and Roychoudhury, 2011] Chattopadhyay, S. and Roychoudhury, A. (2011). Static bus schedule aware scratchpad allocation in multiprocessors. In *Proceedings of the 2011 SIGPLAN/SIGBED conference on Languages, compilers and tools for embedded systems*, pages 11–20.

[Clarke and Emerson, 1982] Clarke, E. M. and Emerson, E. A. (1982). Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71, London, UK, UK. Springer-Verlag.

[Clarke et al., 2012] Clarke, E. M., Klieber, W., Nová\vcek, M., and Zuliani, P. (2012). Model Checking and the State Explosion Problem. In *Tools for Practical Software Verification*, pages 1–30. Springer.

[Colin and Puaut, 2000] Colin, A. and Puaut, I. (2000). Worst case execution time analysis for a processor with branch prediction. *Real-Time Syst.*, 18(2/3):249–274.

[Commoner et al., 1971] Commoner, F., Holt, A. W., Even, S., and Pnueli, A. (1971). Marked directed graphs. *J. Comput. Syst. Sci.*, 5(5):511–523.

[Cullmann et al., 2010] Cullmann, C., Ferdinand, C., Gebhard, G., Grund, D., Maiza, C., Reineke, J., Triquet, B., and Wilhelm, R. (2010). Predictability considerations in the design of multi-core embedded systems. In *Proceedings of the Embedded Real Time Software and Systems Congress (ERTS^2) 2010*.

[Dalsgaard et al., 2012] Dalsgaard, A., Laarman, A., Larsen, K., Olesen, M., and van de Pol, J. (2012). Multi-core reachability for timed automata. In Jurdziński, M. and Ničković, D., editors, *Formal Modeling and Analysis of Timed Systems*, volume 7595 of *Lecture Notes in Computer Science*, pages 91–106. Springer Berlin Heidelberg.

[Dasari et al., 2011] Dasari, D., Andersson, B., Nelis, V., Petters, S. M., Easwaran, A., and Lee, J. (2011). Response time analysis of cots-based multicores considering the contention on the shared memory bus. In *Proceedings of the 2011IEEE 10th International Conference on Trust, Security and Privacy in Computing and Communications*, TRUSTCOM '11, pages 1068–1075, Washington, DC, USA. IEEE Computer Society.

[David et al., 2011] David, A., Larsen, K. G., Legay, A., Mikučionis, M., and Wang, Z. (2011). Time for statistical model checking of real-time systems. In *Proceedings of the 23rd International Conference on Computer Aided Verification*, CAV'11, pages 349–355, Berlin, Heidelberg. Springer-Verlag.

[Davis and Burns, 2011] Davis, R. I. and Burns, A. (2011). A survey of hard real-time scheduling for multiprocessor systems. *ACM Computing Surveys*, 43(4):1–44.

[Dominik, 2011] Dominik, C. (2011). Conception and Implementation of Parallelism Analyses in MATLAB/SIMULINK Models for programming Embedded Multicore-Systems. Bsc. thesis, TU München.

[EN50128, 2009] EN50128 (2009). CENELEC DRAFT prEN 50128. Railway applications – Communication, signaling and processing systems – Software for railway control and protection systems.

[Ermedahl and Engblom, 2007] Ermedahl, A. and Engblom, J. (2007). Execution time analysis for embedded real-time systems. *Handbook of Real-Time Embedded Systems, SHS Insup Lee, Jospeh YT. Leung, Ed. Chapman & Hall/CRC-Taylor and Francis Group*, pages 35–1.

[F. Bouchhima, 2005] F. Bouchhima, G. N. (2005). Discrete-continuous simulation model for accurate validation in component-based heterogeneous SoC design. pages 181– 187.

[Fakih, 2011] Fakih, M. (2011). Timing Validation of Functional Models on Virtual Platforms. Master's thesis, University of Oldenburg.

[Fakih et al., 2011] Fakih, M., Grüttner, K., , Fränzle, M., and Rettberg, A. (2011). Simulink and virtual hardware platform co-simulation for accurate timing analysis of embedded control softwares. In *ASIM STS/GMMS Workshop 2011*.

[Fakih and Grüttner, 2012] Fakih, M. and Grüttner, K. (2012). Virtual Platform in the Loop Simulation for Accurate Timing Analysis of Embedded Software on Multicore Platforms. In *ASIM Konferenz STS/GMMS, Wolfenbüttel*.

[Fakih et al., 2013a] Fakih, M., Grüttner, K., Fränzle, M., and Rettberg, A. (2013a). Towards performance analysis of SDFGs mapped to shared–bus architectures using model–checking. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '13, Leuven, Belgium. European Design and Automation Association.

[Fakih et al., 2014] Fakih, M., Grüttner, K., Fränzle, M., and Rettberg, A. (2014). Multicore performance analysis of a multi–phase electrical motor controller. In *Proceedings of the Embedded Real Time Software and Systems Congress (ERTS^2) 2014*.

[Fakih et al., 2015] Fakih, M., Grüttner, K., Fränzle, M., and Rettberg, A. (2015). State-based real-time analysis of SDF applications on MPSoCs with shared communication resources. *Journal of Systems Architecture - Embedded Systems Design*, 61(9):486–509.

[Fakih et al., 2013b] Fakih, M., Grüttner, K., Fränzle, M., and Rettberg, A. (2013b). Exploiting Segregation in Bus-Based MPSoCs to Improve Scalability of Model-Checking-Based Performance Analysis for SDFAs. In *Embedded Systems: Design, Analysis and Verification*, volume 403 of *IFIP Advances in Information and Communication Technology*, pages 205–217. Springer Berlin Heidelberg.

[Falk et al., 2005] Falk, J., Haubelt, C., and Teich, J. (2005). Syntax and execution behavior of SysteMoC. Technical Report Co-Design-Report 04 -2005, Department of Computer Science, Hardware-Software-Co-Design University of Erlangen-Nuremberg.

[Ferdinand and Heckmann, 2004] Ferdinand, C. and Heckmann, R. (2004). ait: Worst-case execution time prediction by static program analysis. In *Building the Information Society*, page 377–383. Springer.

[Fränzle, 2012] Fränzle, M. (2012). Introduction to model checking. Lecture notes (TC-DSD), Carl von Ossietzky Universität, FK II, Dpt. Informatik , Abt. Hybride Systeme.

[Fuller and Lynette I. Millett, 2011] Fuller, S. H. and Lynette I. Millett, E. C. o. S. G. i. C. P. N. R. C. (2011). *The Future of Computing Performance: Game Over or Next Level?*

[Gajski et al., 2009] Gajski, D. D., Abdi, S., Gerstlauer, A., and Schirner, G. (2009). *Embedded System Design: Modeling, Synthesis and Verification*. Springer Science & Business Media.

[Gansner et al., 2015] Gansner, E. R., Koutsofios, E., and North, S. (2015). *Drawing graphs with dot*.

[Geilen et al., 2005] Geilen, M., Basten, T., and Stuijk, S. (2005). Minimising buffer requirements of synchronous dataflow graphs with model checking. In *Proceedings of the 42nd annual Design Automation Conference*, pages 819–824.

[Gerstlauer, 2009] Gerstlauer, A. (2009). System-level design. http://users.ece.utexas.edu/~gerstl/ee382v_f09/schedule.html (30.04.2015). Lecture notes.

[Gerstlauer et al., 2009] Gerstlauer, A., Haubelt, C., Pimentel, A., Stefanov, T., Gajski, D., and Teich, J. (2009). Electronic System-Level Synthesis Methodologies. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 28(10):1517 –1530.

[Ghamarian, 2008] Ghamarian, A. (2008). *Timing Analysis of Synchronous Data Flow Graphs*. PhD thesis, Eindhoven University of Technology.

[Giannopoulou et al., 2012] Giannopoulou, G., Lampka, K., Stoimenov, N., and Thiele, L. (2012). Timed model checking with abstractions: Towards worst-case response time analysis in resource-sharing manycore systems. In *Proc. International Conference on Embedded Software (EMSOFT)*, pages 63–72, Tampere, Finland. ACM.

[Glass et al., 2012] Glass, M., Teich, J., and Zhang, L. (2012). A co-simulation approach for system-level analysis of embedded control systems. In *2012 International Conference on Embedded Computer Systems (SAMOS)*, pages 355–362.

[Greenyer, 2010] Greenyer, J. (2010). Synthesizing modal sequence diagram specifications with uppaal-tiga. Technical Report tr-ri-10-310, University of Paderborn.

[Grimm et al., 2009] Grimm, C., Barnasconi, M., Vachoux, A., and Einwich, K. (2009). Introduction to the SystemC AMS Draft Standard. http://www.systemc-ams.org/documents/einwich_ieeesocc_belfast_070909.pdf (01.11.2015). presentation slides.

[Grüttner et al., 2011] Grüttner, K., Hartmann, P. A., Reinkemeier, P., Oppenheimer, F., and Nebel, W. (2011). Challenges of multi- and many-core architectures for electronic system-level design. In *2011 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation, SAMOS XI, Samos, Greece, July 18-21, 2011*, pages 331–338.

[Gu et al., 2007] Gu, Z., Yuan, M., Guan, N., Lv, M., He, X., Deng, Q., and Yu, G. (2007). Static scheduling and software synthesis for dataflow graphs with symbolic model-checking. pages 353–364. IEEE.

[Gustavsson, 2010] Gustavsson, A. (2010). WCET Analysis of Multicore Architectures.

[Gustavsson et al., 2010] Gustavsson, A., Ermedahl, A., Lisper, B., and Pettersson, P. (2010). Towards WCET analysis of multicore architectures using UPPAAL. In Lisper, B., editor, *WCET*, volume 15 of *OASICS*, pages 101–112. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany.

[Hansson et al., 2009] Hansson, A., Goossens, K., Bekooij, M., and Huisken, J. (2009). CoMPSoC: A template for composable and predictable multi-processor system on chips. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 14(1):2.

[Hausmans et al., 2013] Hausmans, J. P. H. M., Wiggers, M. H., Geuns, S. J., and Bekooij, M. J. G. (2013). Dataflow analysis for multiprocessor systems with non-starvation-free schedulers. In *Proceedings of the 16th International Workshop on Software and Compilers for Embedded Systems*, M-SCOPES '13, pages 13–22, New York, NY, USA. ACM.

[Hendriks and Verhoef, 2006] Hendriks, M. and Verhoef, M. (2006). Timed automata based analysis of embedded system architectures. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, pages 8–pp. IEEE.

[Henia et al., 2005] Henia, R., Hamann, A., Jersak, M., Racu, R., Richter, K., and Ernst, R. (2005). System Level Performance Analysis - the SymTA/S Approach. In *IEE Proceedings Computers and Digital Techniques*.

[Herber, 2010] Herber, P. (2010). *A Framework for Automated HW/SW Co-Verification of SystemC Designs using Timed Automata*. Logos Verlag Berlin GmbH.

[Hitex Inc., 2013] Hitex Inc. (2013). Hitex Compiler. http://www.hitex.co.uk/. (last accessed on 01.11.2015).

[Huang et al., 2009] Huang, K., Yan, X., Han, S., Chae, S., Jerraya, A., Popovici, K., Guerin, X., Brisolara, L., and Carro, L. (2009). Gradual refinement for application-specific MPSoC design from Simulink model to RTL implementation. *Journal of Zhejiang University-Science A*, 10(2):151–164.

[Huber and Schoeberl, 2009] Huber, B. and Schoeberl, M. (2009). Comparison of implicit path enumeration and model checking based WCET analysis. In *Proceedings of the 9th International Workshop on Worst-Case Execution Time (WCET) Analysis*, pages 23–34.

[ICVerification , 2015] ICVerification (2015). AMBA AHB Protocol. http://www.icverification.com/BusProtocols/AmbaAHB2.php.

[IEEE-1666, 2012] IEEE-1666 (2012). IEEE Standard SystemC Language Reference Manual. IEEE Std. 1666–2011, IEEE Computer Society. ISBN 978-0-7381-6801-2.

[Infineon Inc., 2013] Infineon Inc. (2013). AURIX – Safety joins Performance. http://www.infineon.com/cms/en/product/microcontrollers/32-bit-tricore-tm-microcontrollers/aurix-tm-family/channel.html?channel=db3a30433727a44301372b2eefbb48d9. (last accessed on 01.11.2015).

[ISO11898-1, 2003] ISO11898-1 (2003). Iso11898-1: 2003-road vehicles–controller area network. *International Organization for Standardization, Geneva, Switzerland*.

[ISO26262, 2011] ISO26262 (2011). ISO/FDIS 26262. Road vehicles – Functional safety.

[Jeffay et al., 1991] Jeffay, K., Stanat, D. F., and Martel, C. U. (1991). On non-preemptive scheduling of period and sporadic tasks. In *Real-Time Systems Symposium, 1991. Proceedings., Twelfth*, pages 129–139. IEEE.

[Kai Hylla, 2008] Kai Hylla, J.-H. O. (2008). Using SystemC for an extended MATLAB/Simulink verification flow. pages 221 – 226.

[Karray et al., 2013] Karray, H., Paulitsch, M., Koppenhoefer, B., and Geiger, D. (2013). Design and implementation of a degraded vision landing aid application on a multicore processor architecture for safety-critical application. In *Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), 2013 IEEE 16th International Symposium on*, pages 1–8.

[Kästner Daniel and Christian, 2014] Kästner Daniel, Pister Markus, G. G. and Christian, F. (2014). Reliability of wcet analysis. In *Proceedings of the Embedded Real Time Software and Systems Congress (ERTS^2) 2014*.

[Kesel, 2012] Kesel, F. (2012). *Modellierung von digitalen Systemen mit SystemC: Von der RTL- zur Transaction-Level-Modellierung*. Oldenbourg Verlag.

[Kirner and Puschner, 2010] Kirner, R. and Puschner, P. (2010). Time-Predictable Computing. In Ungerer, S. L. M. R. P. P. P. T., editor, *Software Technologies for Embedded and Ubiquitous Systems*, volume 6399 of *Lecture Notes in Computer Science*, pages 23–34. Springer.

[Kotaba et al., 2013] Kotaba, O., Nowotsch, J., Paulitsch, M., Petters, S. M., and Theiling, H. (2013). Multicore in real-time systems - temporal isolation challenges due to shared resources. In *Proceedings of the Workshop on Industry-Driven Approaches for Cost-effective Certification of Safety-Critical, Mixed-Criticality Systems (WICERT)*, DATE '13, Leuven, Belgium. European Design and Automation Association.

[Kumar, 2009] Kumar, A. (2009). *Analysis, Design and Management of Multimedia Multiprocessor Systems*. PhD thesis, Ph. D. thesis, Eindhoven University of Technology.

[Lee and Messerschmitt, 1987a] Lee, E. and Messerschmitt, D. (1987a). Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245.

[Lee and Messerschmitt, 1987b] Lee, E. A. and Messerschmitt, D. G. (1987b). Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Trans. Computers*, 36(1):24–35.

[Lee and Neuendorffer, 2005] Lee, E. A. and Neuendorffer, S. (2005). Concurrent models of computation for embedded software.

[Lee and Seshia, 2012] Lee, E. A. and Seshia, S. A. (2012). *Introduction to embedded systems: a cyber physical systems approach*. LeeSeshia.org, Lulu, 1. ed., print. 1.08 edition.

[Lele et al., 2014] Lele, A., Moreira, O., Bastos, J., Almeida, R., Pedreiras, P., and van Berkel, K. (2014). Analyzing preemptive fixed priority scheduling of data flow graphs. In *Embedded Systems for Real-time Multimedia (ESTIMedia), 2014 IEEE 12th Symposium on*, pages 50–59. IEEE.

[Li, 2013] Li, S. (2013). Simulink2sdf - converter. GitHub, open-source Code, (last accessed on 01.11.2015).

[Li et al., 2007] Li, X., Liang, Y., Mitra, T., and Roychoudhury, A. (2007). Chronos: A timing analyzer for embedded software. *Science of Computer Programming*, 69:56–67.

[Li and Malik, 1995] Li, Y.-T. S. and Malik, S. (1995). Performance analysis of embedded software using implicit path enumeration. In *Proceedings of the 32Nd Annual ACM/IEEE Design Automation Conference*, DAC '95, pages 456–461, New York, NY, USA. ACM.

[Li et al., 1997] Li, Y.-T. S., Malik, S., and Wolfe, A. (1997). Cinderella: A retargetable environment for performance analysis of real-time software. In Goos, G., Hartmanis, J., van Leeuwen, J., Lengauer, C., Griebl, M., and Gorlatch, S., editors, *Euro-Par'97 Parallel Processing*, volume 1300, pages 1308–1315. Springer Berlin Heidelberg, Berlin, Heidelberg.

[Lin et al., 2011] Lin, J., Srivatsa, A., Gerstlauer, A., and Evans, B. L. (2011). Heterogeneous multiprocessor mapping for real-time streaming systems. In *Acoustics, Speech and Signal Processing (ICASSP), 2011 IEEE International Conference on*, pages 1605–1608.

[Lisper, 2014] Lisper, B. (2014). SWEET – a tool for WCET flow analysis (extended abstract). In Margaria, T. and Steffen, B., editors, *Leveraging Applications of Formal Methods, Verification and Validation. Specialized Techniques and Applications*, number 8803 in Lecture Notes in Computer Science, pages 482–485. Springer Berlin Heidelberg.

[Liu et al., 2008] Liu, W., Yuan, M., He, X., Gu, Z., and Liu, X. (2008). Efficient SAT-Based Mapping and Scheduling of Homogeneous Synchronous Dataflow Graphs for Throughput Optimization. In *Proceedings of the 29th IEEE Real-Time Systems Symposium, RTSS 2008, Barcelona, Spain, 30 November - 3 December 2008*, pages 492–504. IEEE Computer Society.

[Lublinerman and Tripakis, 2008] Lublinerman, R. and Tripakis, S. (2008). Translating data flow to synchronous block diagrams. In Eles, P. and Pimentel, A. D., editors, *ESTImedia*, pages 101–106. IEEE.

[Lv et al., 2010] Lv, M., Yi, W., Guan, N., and Yu, G. (2010). Combining Abstract Interpretation with Model Checking for Timing Analysis of Multicore Software. In *2010 31st IEEE Real-Time Systems Symposium*, pages 339–349.

[Malik and Gregg, 2013] Malik, A. and Gregg, D. (2013). Orchestrating Stream Graphs Using Model Checking. *ACM Trans. Archit. Code Optim.*, 10(3):19:1–19:25.

[Marwedel, 2010] Marwedel, P. (2010). Embedded and cyber-physical systems in a nutshell. *DAC. COM Knowledge Center Article*.

[MathWorks, Inc., 2010] MathWorks, Inc. (2010). MATLAB SIMULINK 7 User Manual. http://www.manualslib.com/download/392936/Matlab-Simulink-7.html. (last accessed on 01.11.2015).

[MathWorks, Inc., 2015a] MathWorks, Inc. (2015a). Automatic Code Generation - Simulink Coder. http://www.mathworks.de/products/simulink-coder/. (last accessed on 01.11.2015).

[MathWorks, Inc., 2015b] MathWorks, Inc. (2015b). Matlab engine API. http://de.mathworks.com/help/matlab/apiref/engine.html. (last accessed on 01.11.2015).

[MathWorks, Inc., 2015c] MathWorks, Inc. (2015c). Matlab/Simulink. http://www.mathworks.de/products/simulink/. (last accessed on 01.11.2015).

[MathWorks, Inc., 2015d] MathWorks, Inc. (2015d). Modeling Guidelines for Code Generation. Technical Report Version 1.10 (Release 2015b).

[MathWorks, Inc., 2015e] MathWorks, Inc. (2015e). Simulink- simulation and model-based design- blocklist. http://de.mathworks.com/products/simulink/blocklist.html. (last accessed on 01.11.2015).

[MathWorks, Inc., 2015f] MathWorks, Inc. (2015f). Stateflow official website. http://www.mathworks.com/products/stateflow/. (last accessed on 01.11.2015).

[Mendoza et al., 2011] Mendoza, F., Kollner, C., Becker, J., and Muller-Glaser, K. (2011). An automated approach to SystemC/Simulink co-simulation. In *Rapid System Prototyping (RSP), 2011 22nd IEEE International Symposium on*, pages 135–141.

[Metzlaff et al., 2011] Metzlaff, S., Mische, J., and Ungerer, T. (2011). A Real-Time Capable Many-Core Model. *RTSS 2011 Organization Committee*, page 21.

[Metzner, 2004] Metzner, A. (2004). Why model checking can improve WCET analysis. In *Computer Aided Verification*, pages 298–301.

[Miller et al., 2005] Miller, S., Anderson, E., Wagner, L., Whalen, M., and Heimdahl, M. P. E. (2005). Formal verification of flight critical software. In *Proceedings of the AIAA Guidance, Navigation and Control Conference and Exhibit*.

[Moonen, 2009] Moonen, A. (2009). *Predictable Embedded Multiprocessor Architecture for Streaming Applications*. PhD thesis, Eindhoven University of Technology.

[Moreira, 2012] Moreira, O. (2012). *Temporal analysis and scheduling of hard real-time radios running on a multi-processor*. PhD thesis, Ph. D. dissertation, TU Eindhoven.

[Moreira et al., 2007] Moreira, O., Valente, F., and Bekooij, M. (2007). Scheduling multiple independent hard-real-time jobs on a heterogeneous multiprocessor. In *Proceedings of the 7th ACM & IEEE international conference on Embedded software*, pages 57–66. ACM.

[MotorBrain Consortium, 2013] MotorBrain Consortium (2013). MotorBrain. `http://www.motorbrain.eu/`. (last accessed on 01.11.2015).

[Mühleis et al., 2011] Mühleis, N., Glass, M., Zhang, L., and Teich, J. (2011). A co-simulation approach for control performance analysis during design space exploration of cyber-physical systems. *ACM SIGBED Review*, 8(2):23–26.

[Nelis et al., 2011] Nelis, V., Dasari, D., Nikolic, B., and Petters, S. (2011). A Tighter Analysis of the Worst-Case End-to-End Communication Delay in Massive Multicores. *RTSS 2011 Organization Committee*, page 25.

[Nelson et al., 2010] Nelson, A., Hansson, A., Corporaal, H., and Goossens, K. (2010). Conservative application-level performance analysis through simulation of MPSoCs. In *Embedded Systems for Real-Time Multimedia (ESTIMedia), 2010 8th IEEE Workshop on*, pages 51–60. IEEE.

[Norstrom et al., 1999] Norstrom, C., Wall, A., and Yi, W. (1999). Timed automata as task models for event-driven systems. In *Real-Time Computing Systems and Applications, 1999. RTCSA'99. Sixth International Conference*, pages 182–189. IEEE.

[Nowotsch et al., 2014] Nowotsch, J., Paulitsch, M., Henrichsen, A., Pongratz, W., and Schacht, A. (2014). Monitoring and WCET Analysis in COTS multi-core-SoC-based Mixed-criticality Systems. In *Proceedings of the Conference on Design, Automation & Test in Europe*, DATE '14, pages 67:1–67:5, 3001 Leuven, Belgium, Belgium. European Design and Automation Association.

[Olderog and Dierks, 2008] Olderog, E.-R. and Dierks, H. (2008). *Real-Time Systems — Formal Specification and Automatic Verification*. Cambridge University Press. ISBN 978-0-521-88333-7.

[Page et al., 2001] Page, B., Liebert, H., and heymann, A. (2001). *Diskrete Simulation. Eine Einführung in Modula-2*. Springer, Berlin.

[Park, 1929] Park, R. (1929). Two-reaction theory of synchronous machines generalized method of analysis-part I. *American Institute of Electrical Engineers, Transactions of the*, 48(3):716–727.

[Pellizzoni et al., 2010] Pellizzoni, R., Schranzhofer, A., Chen, J.-J., Caccamo, M., and Thiele, L. (2010). Worst case delay analysis for memory interference in multicore systems. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2010*, pages 741—-746, Dresden, Germany. ACM.

[Perathoner et al., 2009] Perathoner, S., Wandeler, E., Thiele, L., Hamann, A., Schliecker, S., Henia, R., Racu, R., Ernst, R., and González Harbour, M. (2009). Influence of different abstractions on the performance analysis of distributed hard real-time systems. *Design Automation for Embedded Systems*, 13(1):27–49.

[Pitter and Schoeberl, 2010] Pitter, C. and Schoeberl, M. (2010). A real-time java chip-multiprocessor. *ACM Trans. Embed. Comput. Syst.*, 10(1):9:1–9:34.

[Pnueli, 1977] Pnueli, A. (1977). The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, SFCS '77, pages 46–57, Washington, DC, USA. IEEE Computer Society.

[Pop et al., 2002] Pop, T., Eles, P., and Peng, Z. (2002). Holistic scheduling and analysis of mixed time/event-triggered distributed embedded systems. In *Proceedings of the tenth international symposium on Hardware/software codesign*, pages 187–192. ACM.

[Poplavko et al., 2003] Poplavko, P., Basten, T., Bekooij, M., Meerbergen, J. V., and Mesman, B. (2003). Task-level timing models for guaranteed performance in multiprocessor networks-on-chip. In *CASES, Proc*, pages 63–72. ACM.

[Popovici and Jerraya, 2010] Popovici, K. and Jerraya, A. A. (2010). Virtual platforms in system-on-chip design. In *Design Automation Conference*.

[Poppen, F. and Grüttner, K., 2012] Poppen, F. and Grüttner, K. (2012). Co-Simulation of C-based SoC Simulators and MATLAB Simulink.

[Pouzet and Raymond, 2009] Pouzet, M. and Raymond, P. (2009). Modular static scheduling of synchronous data-flow networks: an efficient symbolic representation. In Chakraborty, S. and Halbwachs, N., editors, *EMSOFT*, pages 215–224. ACM.

[Puschner and Schedl, 1997] Puschner, P. and Schedl, A. (1997). Computing maximum task execution times - a graph-based approach. *Journal of Real-Time Systems*, 13:67–91.

[Queille and Sifakis, 1982] Queille, J. and Sifakis, J. (1982). Specification and verification of concurrent systems in cesar. In Dezani-Ciancaglini, M. and Montanari, U., editors, *International Symposium on Programming*, volume 137 of *Lecture Notes in Computer Science*, pages 337–351. Springer Berlin Heidelberg.

[Richter et al., 2003] Richter, K., Jersak, M., and Ernst, R. (2003). A formal approach to MpSoC performance verification. *Computer*, 36(4):60–67.

[Rochange, 2011] Rochange, C. (2011). An Overview of Approaches Towards the Timing Analysability of Parallel Architecture. In Lucas, P., Thiele, L., Triquet, B., Ungerer, T., and Wilhelm, R., editors, *Bringing Theory to Practice: Predictability and Performance in Embedded Systems*, volume 18 of *OpenAccess Series in Informatics (OASIcs)*, pages 32–41, Dagstuhl, Germany. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.

[Rosvall and Sander, 2014] Rosvall, K. and Sander, I. (2014). A Constraint-based Design Space Exploration Framework for Real-time Applications on MPSoCs. In *Proceedings of the Conference on Design, Automation & Test in Europe*, DATE '14, pages 326:1–326:6, 3001 Leuven, Belgium, Belgium. European Design and Automation Association.

[Roychoudhury, 2009] Roychoudhury, A. (2009). *Embedded Systems and Software Validation*. Morgan Kaufmann.

[Schaumont, 2013] Schaumont, P. R. (2013). *A Practical Introduction to Hardware/Software Codesign*. Springer US.

[Schlaak, 2014] Schlaak, C. (2014). Codegenerator zur automatischen Konfiguration eines Ausführungszeit-Analyseframeworks für Anwendungen aus der digitalen Signalverarbeitung. Bsc. thesis, Carl von Ossietzky Universität Oldenburg.

[Schliecker et al., 2010] Schliecker, S., Negrean, M., and Ernst, R. (2010). Bounding the shared resource load for the performance analysis of multiprocessor systems. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '10, pages 759–764, 3001 Leuven, Belgium, Belgium. European Design and Automation Association.

[Schranzhofer, 2011] Schranzhofer, A. (2011). *Efficiency and predictability in resource sharing multicore systems*. PhD thesis, ETH Zurich.

[Schranzhofer et al., 2011] Schranzhofer, A., Pellizzoni, R., jia Chen, J., Thiele, L., and Caccamo, M. (2011). Timing analysis for resource access interference on adaptive resource arbiters. In *in Real-Time and Embedded Technology and Applications Symposium (RTAS), 2011 17th IEEE. IEEE*, pages 213–222.

[Shabbir et al., 2010] Shabbir, A., Kumar, A., Stuijk, S., Mesman, B., and Corporaal, H. (2010). CA-MPSoC: An Automated Design Flow for Predictable Multi-processor Architectures for Multiple Applications. *Journal of Systems Architecture*, 56(7):265–277.

[Shaw, 1989] Shaw, A. (1989). Reasoning about time in higher-level language software. *Software Engineering, IEEE Transactions on*, 15(7):875–889.

[Skelin et al., 2015] Skelin, M., Wognsen, E. R., Olesen, M. C., Hansen, R. R., and Larsen, K. G. (2015). Model checking of finite-state machine-based scenario-aware dataflow using timed automata. In *Industrial Embedded Systems (SIES), 2015 10th IEEE International Symposium on*, pages 1–10. IEEE.

[Srba, 2008] Srba, J. (2008). Comparing the Expressiveness of Timed Automata and Timed Extensions of Petri Nets. In Cassez, F. and Jard, C., editors, *Formal Modeling and Analysis of Timed Systems*, number 5215 in Lecture Notes in Computer Science, pages 15–32. Springer Berlin Heidelberg.

[Sriram and Bhattacharyya, 2000] Sriram, S. and Bhattacharyya, S. S. (2000). *Embedded Multiprocessors: Scheduling and Synchronization*. CRC Press, 1 edition.

[Stemmer et al., 2016] Stemmer, R., Fakih, M., Grüttner, K., and Nebel, W. (2016). Towards State-Based RT Analysis of FSM-SADFGs on MPSoCs with Shared Memory Communication. In *2nd International Workshop on Investigating Dataflow in Embedded computing Architecture (IDEA)*. (accepted publication).

[Stuijk, 2007] Stuijk, S. (2007). *Predictable Mapping of Streaming Applications on Multiprocessors*. PhD thesis, Faculty of Electrical Engineering, Eindhoven University of Technology, The Netherlands.

[Stuijk et al., 2006] Stuijk, S., Geilen, M., and Basten, T. (2006). SDFˆ 3: SDF for free. In *Application of Concurrency to System Design, 2006. ACSD 2006. Sixth International Conference on*, page 276–278.

[Stulova et al., 2012] Stulova, A., Leupers, R., and Ascheid, G. (2012). Throughput driven transformations of synchronous data flows for mapping to heterogeneous MPSoCs. In *Embedded Computer Systems (SAMOS), 2012 International Conference on*, pages 144–151. IEEE.

[Suman and Pandya, 2006] Suman, V. and Pandya, P. K. (2006). Foundations Of Timed And Hybrid Automata: A Survey. Technical Report TIFR-PPVS-GM-2006/1, Technical Report TIFR-PPVS-GM-2006/1, TIFR.

[Synopsys Inc., 2015] Synopsys Inc. (2015). Virtualizer. http://www.synopsys.com/Systems/VirtualPrototyping/Pages/Virtualizer.aspx. (last accessed on 01.11.2015).

[Tang and Wu, 2014] Tang, L. and Wu, J. Z. (2014). The Status and Challenges of Multi-Processor System-on-Chip's Formal Verification. *Applied Mechanics and Materials*, 602-605:2926–2929.

[Tatenguem et al., 2011] Tatenguem, H. F., Ludovici, D., Strano, A., Bertozzi, D., and Reinig, H. (2011). Contrasting multi-synchronous MPSoC design styles for fine-grained clock domain partitioning: The full-HD video playback case study. In *Proceedings of the 4th International Workshop on Network on Chip Architectures*, NoCArc '11, pages 37–42, New York, NY, USA. ACM.

[Thakur and Srikant, 2015] Thakur, R. K. and Srikant, Y. N. (2015). Efficient Compilation of Stream Programs for Heterogeneous Architectures: A Model-Checking based approach. Technical Report IISc-CSA-TR-2015-2, Indian Institute of Science, India.

[Theelen et al., 2006] Theelen, B. D., Geilen, M. C. W., Basten, T., Voeten, J. P. M., Gheorghita, S. V., and Stuijk, S. (2006). A Scenario-aware Data Flow Model for Combined Long-run Average and Worst-case Performance Analysis. In *Proceedings of the Fourth ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2006. MEMOCODE '06. Proceedings.*, MEMOCODE '06, pages 185–194, Washington, DC, USA. IEEE Computer Society.

[Thiele et al., 2000] Thiele, L., Chakraborty, S., and Naedele, M. (2000). Real-time calculus for scheduling hard real-time systems. In *Circuits and Systems, 2000. Proceedings. ISCAS 2000 Geneva. The 2000 IEEE International Symposium on*, volume 4, pages 101–104. IEEE.

[Tindell and Clark, 1994] Tindell, K. and Clark, J. (1994). Holistic schedulability analysis for distributed hard real-time systems. *Microprocess. Microprogram.*, 40(2-3):117–134.

[Tomasena et al., 2009] Tomasena, K., Sevillano, J., Arrue, N., Cortés, A., Vélez, I., and others (2009). Embedding Matlab in SystemC transaction level modeling for verification. In *Design of Circuits and Integrated Systems Conference, DCIS*.

[Ungerer et al., 2010] Ungerer, T., Cazorla, F., Sainrat, P., Bernat, G., Petrov, Z., Rochange, C., Quiñones, E., Gerdes, M., Paolieri, M., Wolf, J., Casse, H., Uhrig, S., Guliashvili, I., Houston, M., Kluge, F., Metzlaff, S., and Mische, J. (2010). Merasa: Multicore Execution of Hard Real-Time Applications Supporting Analyzability. *IEEE Micro*, 30(5):66–75.

[Walter et al., 2014] Walter, J., Fakih, M., and Grüttner, K. (2014). Hardware–based real–time simulation on the raspberry pi. In *2nd Workshop on High-performance and Real-time Embedded Systems (HiRES 2014)*.

[Warsitz, 2015] Warsitz, S. (2015). Simulink-Modellübersetzung in Synchrone Datenfluss Graphen(SDFG) zur Ausführungszeit-Analyse auf Multi-core Architekturen. Bsc. thesis, Carl von Ossietzky Universität Oldenburg.

[Warsitz and Fakih, 2016] Warsitz, S. and Fakih, M. (2016). Simulink-Modell-Übersetzung in synchrone Datenflussgraphen. In *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV'2016)*. Universität Rostock.

[Westphal, 2012] Westphal, B. (2012). Real-time systems. `http://electures.informatik.uni-freiburg.de/portal/web/guest/detail/-/modulnavigation/view/4402/13302/`. Lecture notes, Albert-Ludwigs-Universität Freiburg.

[Wilhelm et al., 2008] Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D., Bernat, G., Ferdinand, C., Heckmann, R., Mitra, T., Mueller, F., Puaut, I., Puschner, P., Staschulat, J., and Stenström, P. (2008). The worst-case execution-time problem—overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems*, 7(3):36:1–36.

[Wilhelm and Reineke, 2012] Wilhelm, R. and Reineke, J. (2012). Embedded systems: Many cores; Many problems. In *2012 7th IEEE International Symposium on Industrial Embedded Systems (SIES)*, pages 176–180.

[Wolf et al., 2012] Wolf, J., Fechner, B., Uhrig, S., and Ungerer, T. (2012). Fine-grained timing and control flow error checking for hard real-time task execution. In *Industrial Embedded Systems (SIES), 2012 7th IEEE International Symposium on*, pages 257–266.

[Yang et al., 2010] Yang, Y., Geilen, M., Basten, T., Stuijk, S., and Corporaal, H. (2010). Automated bottleneck-driven design-space exploration of media processing systems. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '10, pages 1041–1046, 3001 Leuven, Belgium, Belgium. European Design and Automation Association.

[Yen and Wolf, 1998] Yen, T.-Y. and Wolf, W. (1998). Performance estimation for real-time distributed embedded systems. *Parallel and Distributed Systems, IEEE Transactions on*, 9(11):1125–1136.

[Zamorano and Juan, 2014] Zamorano, J. and Juan, A. (2014). Memory Isolation in Many-Core Embedded Systems. *High performance and Real-time Embedded System (HiRES)*.

[Zhang et al., 2013] Zhang, L., Glab, M., Ballmann, N., and Teich, J. (2013). Bridging algorithm and ESL design: Matlab/Simulink model transformation and validation. In *Specification & Design Languages (FDL), 2013 Forum on*, pages 1–8.

[Zhang, 2011] Zhang, W. (2011). Bounding Worst-Case Performance for Multi-Core Processors with Shared L2 Instruction Caches. *Journal of Computing Science and Engineering*, 5(1):1–18.

[Zhu et al., 2014] Zhu, X.-Y., Yan, R., Gu, Y.-L., and Zhang, G. (2014). Static Optimal Scheduling and Mapping of Synchronous Dataflow Graphs on a Heterogeneous Multiprocessor Platform with Model Checking.

[Zhu et al., 2015] Zhu, X.-Y., Yan, R., Gu, Y.-L., Zhang, J., Zhang, W., and Zhang, G. (2015). Static Optimal Scheduling for Synchronous Data Flow Graphs with Model Checking. In *FM 2015: Formal Methods*, pages 551–569. Springer.

Appendices

Appendix A

SDF2TA Tool

A.1 Correctness of SDF2TA Implementation

In the workflow of SDF2TA in Fig. 6.12 different syntax errors may occur at the highlighted entries (a) to (d). The following listing elaborates on these errors which can happen at each step (from (a) to (d)) and specifies how these errors are detected/avoided throughout the workflow (mainly taken from [Schlaak, 2014]).

(a) **Designer Input errors:** As seen in Fig. 6.12, the workflow of SDF2TA starts with the designer who provides SDF2TA with the SUA properties needed. The designer could provide syntactically incorrect data, which may be structural errors (such as connecting a channel to an already connected port) or data type errors (e.g. inputting by mistake the execution time of an actor as a decimal number which is not supported by UPPAAL). The designer is the main source of errors in this step and there is no guarantee that he will construct an intact system model.

Input errors prevention: Many of above errors can be intercepted through the SDF2TA input mechanisms which forces the designer to respect the structural design of the Ecore model (see Appendix A.2 Ecore model). For e.g. it is not possible to instantiate a tile containing more than one processing element (see Fig. A.3) or to join a channel with more than two ports (see Fig. A.2). Erroneous data violating data types specified in the Ecore model are also detected, and a visual feedback is given. In addition, SDF2TA is blocked and it only continues with next steps (XML export and XSLT processing (c)) if these errors are corrected. Additional validation methods (see validation methods in Appendix A.2) that are beyond the capabilities of Ecore models supported methods, or are application-

specific, can be implemented manually. For e.g. ensuring that the WCET is never less than the BCET (`isBCETsmallerWCET` see Fig. A.4) or checking whether or not all actors and channels are mapped to tiles and memories (`mapsAllActors()`, `mapsAllChannels()` see Fig. A.4).

(b) **Ecore Model errors:** In this step as already described in Sect. 6.3, the SDF2TA editor is generated from the Ecore model (as depicted in Fig. 6.12). It could be that the Ecore model contains some errors (e.g. missing some elements/attributes) which makes it incompatible/inconsistent to the implementation of timed-automata templates in UPPAAL (c.f. Sect. 5.2). This kind of errors will manifest at the last step (d) during the configuration of the TA templates and are detected by UPPAAL.

(c) **XSLT errors:** In this step, some errors might occur during XSLT processing of the XML (generated from the MoP). These errors stem either from errors in the XSLT stylesheet itself or come from a defect in the generated source XML file.

XSLT errors correction: If a critical error occurs during XSLT processing then the operation is aborted. All error messages that occur during XSLT processing are intercepted and output in a dialogue box in SDF2TA. This way, the designer can directly identifies the error sources and correct them.

(d) **UPPAAL errors:** UPPAAL is the last station for any syntax errors. In case these are existent, UPPAAL detects them and outputs corresponding error messages, where the designer can debug the source and trace back the error to its origin and correct it. Syntax errors detected here could be as a result of an error in the original Ecore Model (for e.g. some attribute or some element are missing in the Ecore model). In addition, these could be the result of some syntax error in the implementation of the timed-automata templates. Moreover, if any error happens in the process of generating configuration parameters (performed by SDF2TA), not recovered by the procedures in (a) and (c), this will be detected by UPPAAL. Unit tests assuring that above errors were excluded from SDF2TA implementation were conducted in [Schlaak, 2014].

A.2 SDF2TA Ecore model

In Sect. 6.3 we have described the work flow of our SDF2TA tool and we mentioned that it was developed based on the EMF Ecore model. In the following,

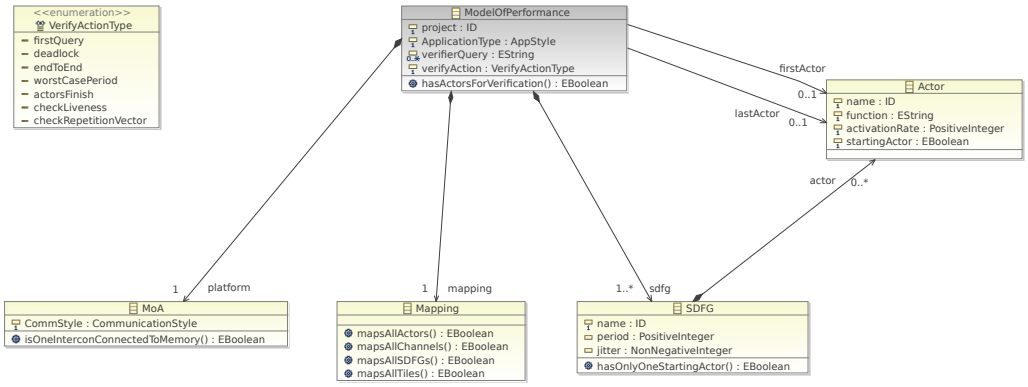


Figure A.1: UML Snapshot of the *Ecore* element "MoP"

we will elaborate on the main parts of this Ecore model by considering the equivalent UML models automatically generated from it[1].

Fig. A.1 shows the main parts of the MoP (see Sect. 5.1) of SUA which the designer supplies, depicted as a UML diagram. It constitutes mainly of a number of SDFGs, the MoA, and the synthesis steps (denoted in the figure as mapping for a better readability/clarity) which will be described in details in the following sections. Every MoP is identified by a `project` name, an application type `AppStyle` (which should be set at the mapping level see Sect. A.2.3) which states whether or not the considered applications are sensitive to the environment, an optional verifier query which gives the user the option to input customized queries to be checked and the verify action of type `verifyActionType`. The `verifyActionType` enumeration (depicted at the top left of Fig. A.1) represents the type of verification actions which can be performed via the model-checker (as described in Sect. 5.3). Two distinguished actors should be explicitly identified for certain verification queries: the *source* (denoted as `firstActor` in the figure) and *sink* (denoted as `lastActor` in the figure) actor. These actors are needed for some verification actions as in the case of end-to-end latency and worst-case period observers (see Sect. 5.2.8). After supplying the needed information of MoP, an XML comprising all the MoP attributes can the be exported which can be then converted to an XML configuration file of our UPPAAL-based TA templates' network (see Fig. 6.12).

A.2.1 SDFG Ecore element

Fig. A.2 shows the main elements of an SDFG (corresponding to Def. 4.2.3). An SDFG is identified by its name. In addition, if the SDFG is sensitive to an

[1]Main description and detailed illustrations of the SDF2TA first version were published in [Schlaak, 2014].

event trigger then a period and jitter should be set. The validation function `hasOnlyOneStartingActor` checks whether or not every SDFG has only a unique starting actor (source) and if not it notifies the user. Since we are considering MPSoCs applications, every SDFG should constitute of at least one channel and two actors. An actor could have a number of initiator ports (referred to `PortOut` in Fig. A.2) or a number of target ports (referred to `PortIn` in Fig. A.2) or both, each having an identifier and a rate. Also the channel is identified by its name and a `delay` attribute which indicates the number of initial tokens on the channel. The validation function `isActorNotLinked-ToItself` makes sure that a channel connects unique initiator port of one actor to the target port of the second actor and that an actor is not connected to itself through a channel.

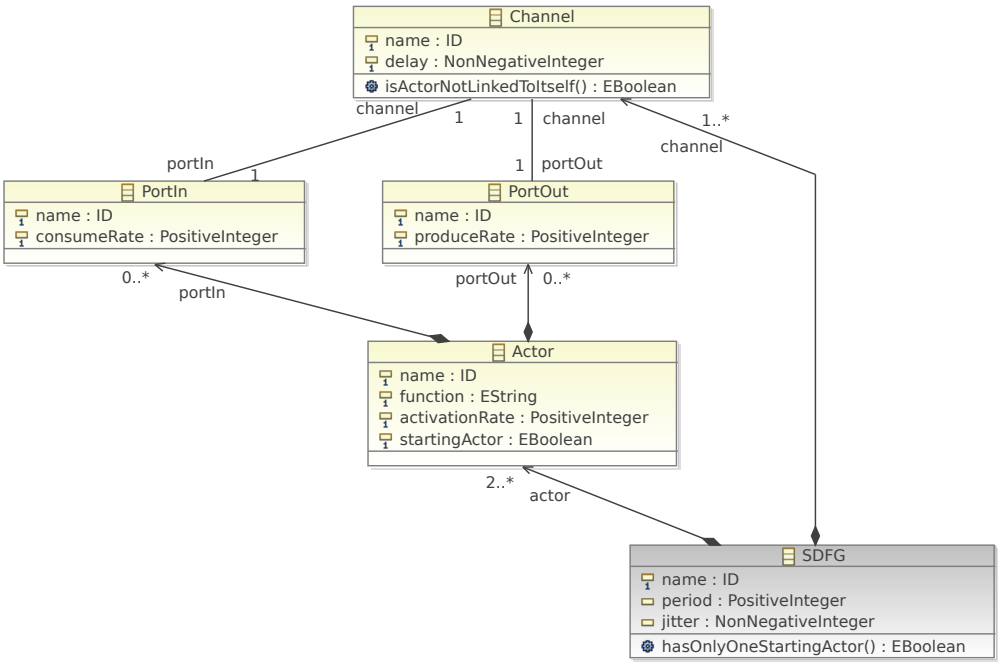


Figure A.2: UML Snapshot of the *Ecore* element "SDFG"

An actor is identified by its name and its functionality (for e.g. DCT). The `activationRate` gives the number of actor's activation within an *iteration*. In addition, within an SDFG there should be exactly one source actor for which the attribute `startingActor`) is set to true.

A.2.2 Model of Architecture Ecore Element

Fig. A.3 shows the Ecore model of the MPSoC model of architecture. It consists of a number of tiles each having an identifier, a number of shared stor-

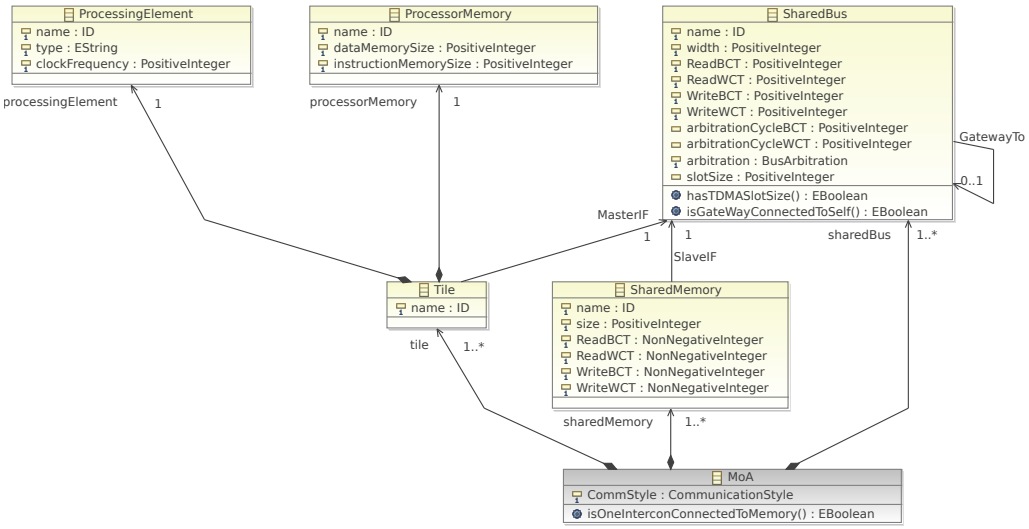


Figure A.3: UML Snapshot of the *ecore* element "MoA"

age resources (referred to by shared memory in Fig. A.3 for readability/clarity purposes) and a number of shared interconnects (referred to by shared bus in Fig. A.3 for readability/clarity purposes). The inter-processor communication style can be chosen (which should be set at the mapping level see Sect. A.2.3) by setting the attribute `commStyle` either to burst or single-beat based communication. Every tile consists of a processing element having an identifier, a type and clock frequency attributes, and a processor private memory having an *identifier* and *size* attributes. In case of multiple interconnects, the attribute `MasterIF` should be set for every tile to indicate its connection to the specific interconnects, where in the current implementation a tile is only allowed to be directly connected to one interconnect.

The `sharedBus` Ecore element is used to connect tiles to shared memories. Each `sharedBus` consists of an *identifier* attribute, a `width` attribute which describes the bus width, the (best-case and worst-case) *latencies* to read/write one word of a length equal to the bus length, the (best-case and worst-case) *latency* needed to perform an *arbitration cycle*, the arbitration strategy (which should be set at the mapping level see Sect. A.2.3) and in case of a TDMA arbitration scheme a `slotSize` attribute representing the size of time slot in a TDMA wheel need to be set. In case of multiple interconnects, the attribute `GatewayTo` should be set for every `SharedBus` to indicate its connection to other shared buses, where in the current implementation a `SharedBus` is only allowed to be directly connected to one interconnect. The validation function `hasTDMASlotSize` prohibits the designer from setting the slot-size attribute except when a TDMA arbitration is chosen. In addition, the validation function

isGateWayConnectedToSelf assures that the case where a shared bus is connected to itself via a gateway never takes place.

The SharedMemory Ecore element represents the shared storage resources having an *identifier*, a size and (best-case and worst-case) latencies attributes needed to read/write one word of a length equal to the bus length. In case of multiple interconnects, the attribute SlaveIF should be set for every SharedMemory to indicate its connection to the specific interconnects, where in the current implementation a SharedMemory is only allowed to be directly connected to one interconnect. The validation function isOneInterconnectedToMemory notifies the designer, in the case where the last interconnect in the hierarchy/topology/route, is not connected to any shared memory since is not allowed.

A.2.3 Mapping Ecore Element

Fig. A.4 shows the mapping element of the Ecore model and the decisions which can be done in the synthesis step such as mapping the channels to memories and the actors to tiles, choosing whether or not the SDFA is sensitive to an event trigger through AppStyle attribute, the bus arbitration strategy, the scheduling strategy and the IPC communication style. The validation function mapsAllActors, mapsAllChannels, mapsAllSDFGs and mapsAllTiles assure that all actors are mapped, all channels are mapped, all SDFGs schedules are chosen and that every tile has a number of actors to execute, respectively. For every tile, there exists exactly one tileMap. Also for every physical storage resource there exists exactly one memoryMap either processorMemoryMap or SharedMemoryMap.

With the help of the tileMap element, actors are mapped to tiles whereas with the help of either sharedMemoryMap or processorMemoryMap, channels are mapped either to shared memories or to private memories respectively. In the tileMap element, the BCET and WCET of both the software driver and the scheduler code can be obtained with the help of a WCET analyzer and set to the corresponding attributes. At this level, the validation function isBCETsmallerWCET assures that the BCET is always less than or equal the WCET, whereas isTileUnique assures that every actor is exactly mapped uniquely on one tile. In addition, the validation function isMemoryUnique assures that every channel is uniquely mapped on one memory. A tileMap element consists of a number of ActorOnTile elements which contain the properties that should be considered when an actor is mapped to a tile. For e.g. when executed on a specific tile, an actor has now BCET and WCET bounds to be set for the execution time. In addition, in case of blocking, a polling time should be waited. A MemoryMap element consists of a number of ChannelOnMemory elements which contain the properties that should be considered when a channel

is mapped to a specific memory. For e.g. the buffers' sizes and the token sizes are set in this case. In the `SDFGMap`, the scheduling mechanism can be chosen to schedule among different SDFGs mapped to the same tile.

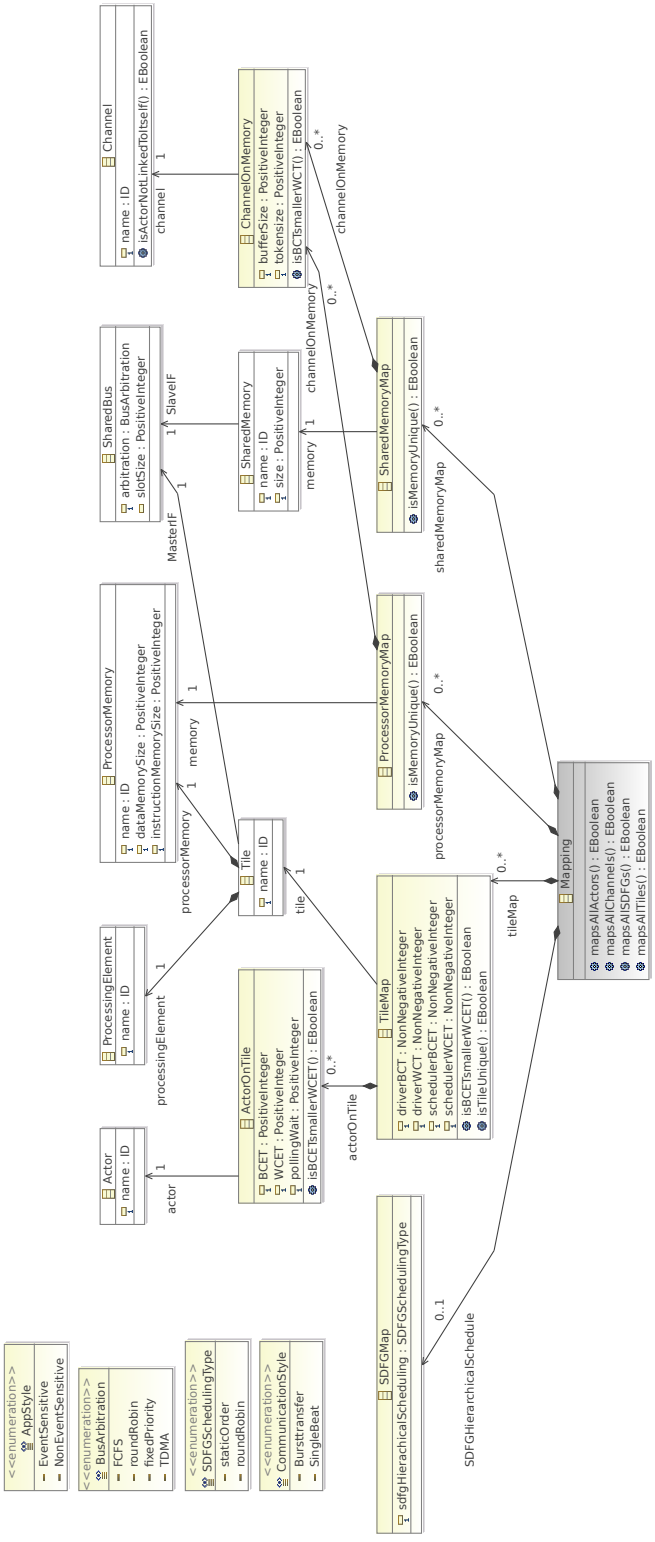


Figure A.4: UML Snapshot of the *Ecore* element "Mapping"

Appendix B

Aurix TriCore Experiment

In Sect. 7.4 the main steps of the Aurix TriCore experiment were briefly described. In the following, a detailed description of the measurement and tracing issues of the execution times is given. In addition, the detailed abstraction and timing delays annotations of the different MPSoC components to the MoP in the SDF2TA tool is given.

B.1 Simulation Measurements

Table B.1: Interconnect accesses for single-beat and burst transfer IPC

	fCLB	sCLB	FOC	MON	VOT	SENS	ACT
Single-beat Read accesses	16	25	4	12	41	0	9
Single-beat Write accesses	10	12	9	20	4	52	0
DMA Transfers	4	5	2	4	6	6	1

In Sect. 7.4.6.1 the simulative timing results of the proposed test-cases were presented. In the following, we will elaborate on the calculation of $t_{com_{single}}$ and $t_{com_{dma}}$ found in Tab. 7.9. Tab. B.1 depicts the number of accesses for every actor (in one firing) on the interconnect which was obtained from the simulation measurements' traces of the virtual-hardware platform and analyzed with the help of VCD (value change dump file) viewer.

For every actor A the communication time for a transfer on the interconnect

Table B.2: Measured parameters of Single-beat transfer (in cycles)

$\Delta_{CanGet/CanPut}$	$\Delta_{readSRI}$	$\Delta_{writeSRI}$	Δ_{arb}	Δ_P
5	5	4	1	348

can be calculated as follows:

$$t_{com_{singleA}} = \delta_R + \delta_W \tag{B.1}$$

where δ_R is the delay needed to read all tokens on all ports from the FIFO buffers and δ_W is the delay of needed to write all tokens to the target FIFO buffers.

B.1.1 Single-beat Transfer Measurements

With the help of the VCD viewer, we found out that reading ($\Delta_{readSRI}$) a token of 32 bits length in the single-beat IPC transfer takes about 5 cycles, while writing ($\Delta_{writeSRI}$) a token consumes 4 cycles of time (by 300 MHz clock frequency).

Because of the nature of enqueue/dequeue implementation (see Algorithm 1 and Fig. 6.17), for every (read/write) access additional interconnect (read/write) accesses are issued (for e.g. reading the size of the buffer before enqueue/dequeue action). Every dequeue function call (by a read transfer), imposes three read access and two write accesses through the interconnect to the target FIFO buffer according to the current implementation. On the other side for every enqueue function call (by write transfer), two read accesses and three write accesses are observed on the interconnect.

Now the read delay δ_R (in cycles) of the interconnect single-beat accesses of an actor can be calculated as follows:

$$\delta_R = R_{accessNr} \times (3 \times \Delta_{readSRI} + 2 \times \Delta_{writeSRI}) = R_{accessNr} \times 23 \tag{B.2}$$

and the write delay δ_W (in cycles) of the interconnect accesses of an actor can be calculated as follows:

$$\delta_W = W_{accessNr} \times (2 \times \Delta_{readSRI} + 3 \times \Delta_{writeSRI}) = W_{accessNr} \times 22 \tag{B.3}$$

Now with the help of above equations and Tab. B.1 the communication time for every actor $t_{com_{single}}$ in a single-beat transfer can be calculated (as found in Tab. 7.9).

In addition to the measuring the delays of the read/write access, we also measured the polling-waiting time to insure a correct annotation. The polling-waiting time in this case was found to be 348 cycles. Moreover, the arbitration cycle of the SRI was found to last only for 1 cycle.

Table B.3: Measured parameters of one DMA transfer (in cycles)

$\Delta_{CanGet/CanPut}$	$\Delta_{dequeue/enqueue}$	$\Delta_{driverDma}$	Δ_{arbDma}	$\Delta_{TransSRI}$	Δ_P
5	167	344	2	190	3453

B.1.2 DMA-based Burst Transfer Measurements

By a burst transfer (of the Aurix DMA with a SIXTEEN_MOVES_PER_TRANSFER configuration), a fixed packet length of 512 bits (4 control variables each of size 32 bits and 12 floats effectively-used tokens each of size 32 bits) is communicated per transfer. In Sect. 5.2.7 we have elaborated on the timing parameters of timed-automata representing a DMA-based burst transfer. The parameters in Tab. B.3 were measured with the help of the obtained VCD traces from the cycle-accurate virtual-hardware platform.

The $\Delta_{driverDma}$ delay consists of a software part of the driver code initialization ($\Delta_{InitDma}$= 79 cycles) and the configuration phase[1] $\Delta_{configDma}$ =255 cycles) of the DMA to start transfer, $\Delta_{TransSRI}$ is the time needed to complete the transfer of the number of packet through the SRI from source memory to target memory including the memory latency (Δ_M) and Δ_{arbDma} is the time delay of the DMA needed to perform internal arbitration.

For simplification, we have implemented the **local**[2] dequeue/enqueue such that always 12 floats of the transfered packet (even if the effectively used number of tokens is less) are consumed/produced. This is the reason why we have for all actors the same local delay when enqueuing/dequeuing. In addition, the polling-waiting time Δ_P was set to be equal 3453 cycles (according to the measurements). Indeed this value is about ten times larger than that of the single-beat transfer. This is due to the implementation-specific timing requirement imposed by the considered DMA to wait for extra time before being able to receive any configurations..

Thus, a fixed delay can be measured (with the help of VCD traces) of every burst access of 536 cycles of time. This delay includes the configuration time of the DMA ($t_{configDma}$ = 344) and the communication time ($\Delta_{DelayOfTrans}$ = 190 + Δ_{arbDma} = 2) of the tokens through the SRI interconnect. By multiplying the number of accesses of every actor to the interconnect (depicted in Tab. B.1) to the delay time, we are able to calculate the communication time $t_{com_{dma}}$ for every actor in a DMA-based burst transfer (as found in Tab. 7.9).

[1]Configuration phase is defined from the moment the first write from the SPB master (tile) on the SPB the till last write (on SPB slave) before starting the SRI transaction.

[2]Local means that no interconnect access is issued within the function call and that the dequeue/enqueue functions are applied on the local copy of the shared buffer.

B.2 Abstractions and Annotations for the MoP

After describing the relevant measurements and parameters for the MoP of the Aurix application in the previous section, we will now take a look at the abstractions done for both IPC implementations and describe the way of capturing their timing behavior in our timed-automata templates (see experiment in Sect. 7.4.7). It is important to note that we applied the optimizations (for static-order schedule) described in Sect. 5.4.1 and that is why no explicit TA were instantiated for the FIFO buffers. Instead for both implementation styles, array variables (14 shared FIFO buffers and 4 private FIFO buffers) each representing a FIFO buffer were initialized.

B.2.1 DMA-based Burst Transfer

Fig. B.1 shows the abstractions made for the DMA-based burst IPC implementation. When applying assumptions B1 and B2 (see Sect. 7.4.7 and Fig. 7.11a), we are now able to abstract away from modeling explicitly the SRI and SPB interconnects. In Fig. B.1 the following abstractions steps are done in step ①:

1. PFlash is not modeled since it was only used for startup code deployment which doesn't influence the timing of the application after initialization.

2. Since we assume no contention on the SRI (see B2), no need for explicit modeling of the SRI. Instead, only the delay of DMA transfer on SRI ($\Delta_{TransSRI}$) should be taken into consideration in the MoP.

3. Since we assume WCCT on SPB (see B1), no need for explicit modeling of the SPB. Instead, only the delay of DMA configuration on the SPB ($3 \times \Delta_{configDmaSPB}$) should be taken into consideration in the MoP.

4. If `SENS` or `ACT` actors are active on tile0, their duration activity (accessing the environment interface HW component) is always less than the time needed by tile0 to configure the DMA. This is the reason why we don't consider this case explicitly in the modeling. In that case, whether tile0 is performing access to the environment interface or is configuring the DMA, always the same pessimistic delay is assumed for accesses to the SPB (which is $3 \times \Delta_{configDmaSPB}$).

After doing above steps, the following TA templates network can be constructed (see step ② in Fig. B.1):

1. One `Eventtrigger` timed automaton is needed to model the periodic event from the environment interface component with a period of 100 μs.

2. For every tile an `SOonTile` timed automaton (see Sect. 5.4.1) is initialized with the Δ_A delay of the actors mapped to this tile and their polling-waiting delay Δ_P.

3. Similarly, for every tile a communication driver timed automaton is initialized. In this case, the delay of the DMA configuration on the SPB can be merged with the delay of the communication driver software and can be annotated to the communication driver timed automaton as follows (as described in Sect. 5.2.7):

$$\Delta_C = \Delta_{InitDma} + 3 \times \Delta_{configDma} + \Delta_{dequeue/enqueue} + \Delta_{CanGet/CanPut} \tag{B.4}$$

   where $\Delta_{InitDma}$ is the delay needed for software part of the communication driver and $\Delta_{CanGet/CanPut}$ are delays for local access on local (to PE) buffers.

4. With our DMA configurations (see Sect. 7.4.3), both arbitrations in the DMA will grant the tile with the highest identification, and this tile will in turn perform the transfer without interruption. For this and because of the abstractions above, only one interconnect timed automaton is needed for modeling the DMA-based IPC, annotated with both the latency delays of the DMA and the SRI ($\Delta_I = \Delta_{arbDma} + \Delta_{Trans}$) and configured with the DMA specific non-preemptive fixed-priority arbitration. For the calculation of Δ_{trans} in case of DMA-based burst transfer, please refer to Sect. 5.2.7.

While exploring the resulting TA network, the state space varied between `19240586` states and `19229533` states for the `sup` and `inf` TCTL verification queries respectively.

B.2.2 Single-beat transfer through SRI

Similar abstractions and annotations for the single-beat IPC implementation were also made with the following minor differences.

1. PFlash is not modeled only for startup code

2. Since no contention on the SPB and only tile0 is allowed to access the environment interface we can abstract away from the SPB and the environment interface component. The access delays of tile0 communicating with the environment interface component via the SPB is already captured in the execution times of `SENS` and `ACT` actors.

After doing that, the following TA templates network can be constructed (see step ② in Fig. B.2):

1. One `Eventtrigger` timed automaton is needed to model the periodic event from the environment interface component with a period of 100 μs.
2. For every tile an `SOonTile` timed automaton is initialized with the Δ_A delay of the actors mapped to this tile and their polling-waiting delay Δ_P.
3. Similarly, for every tile a communication driver timed automaton with the delay Δ_C is initialized. For more details about the calculation of Δ_C in case of a single-beat transfer, please refer to Sect. 5.2.4.
4. Only one interconnect timed automaton is needed for modeling the single-beat IPC on the SRI. The SRI can be abstracted through an interconnect with a non-preemptive fixed-priority arbitration with the delay $\Delta_I = \Delta_{arbSRI} + \Delta_{Trans}$. For more details about the calculation of Δ_{Trans} in case of single-beat transfer, please refer to Sect. 5.2.5.

While exploring the resulting TA network, the state space varied between `41441659` states and `58910060` states for the `sup` and `inf` TCTL verification queries respectively.

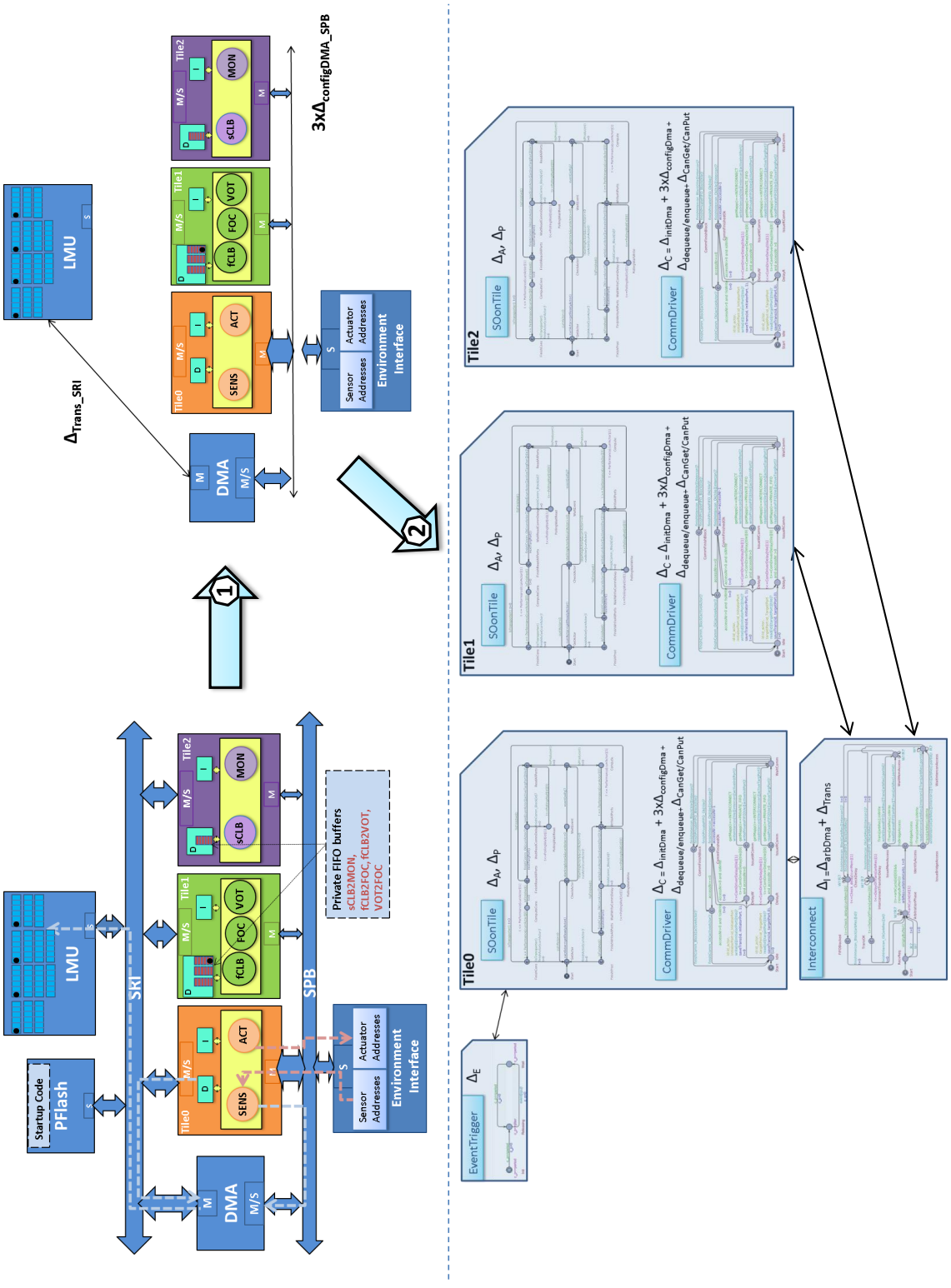


Figure B.1: Abstractions made for the DMA-based burst IPC implementation

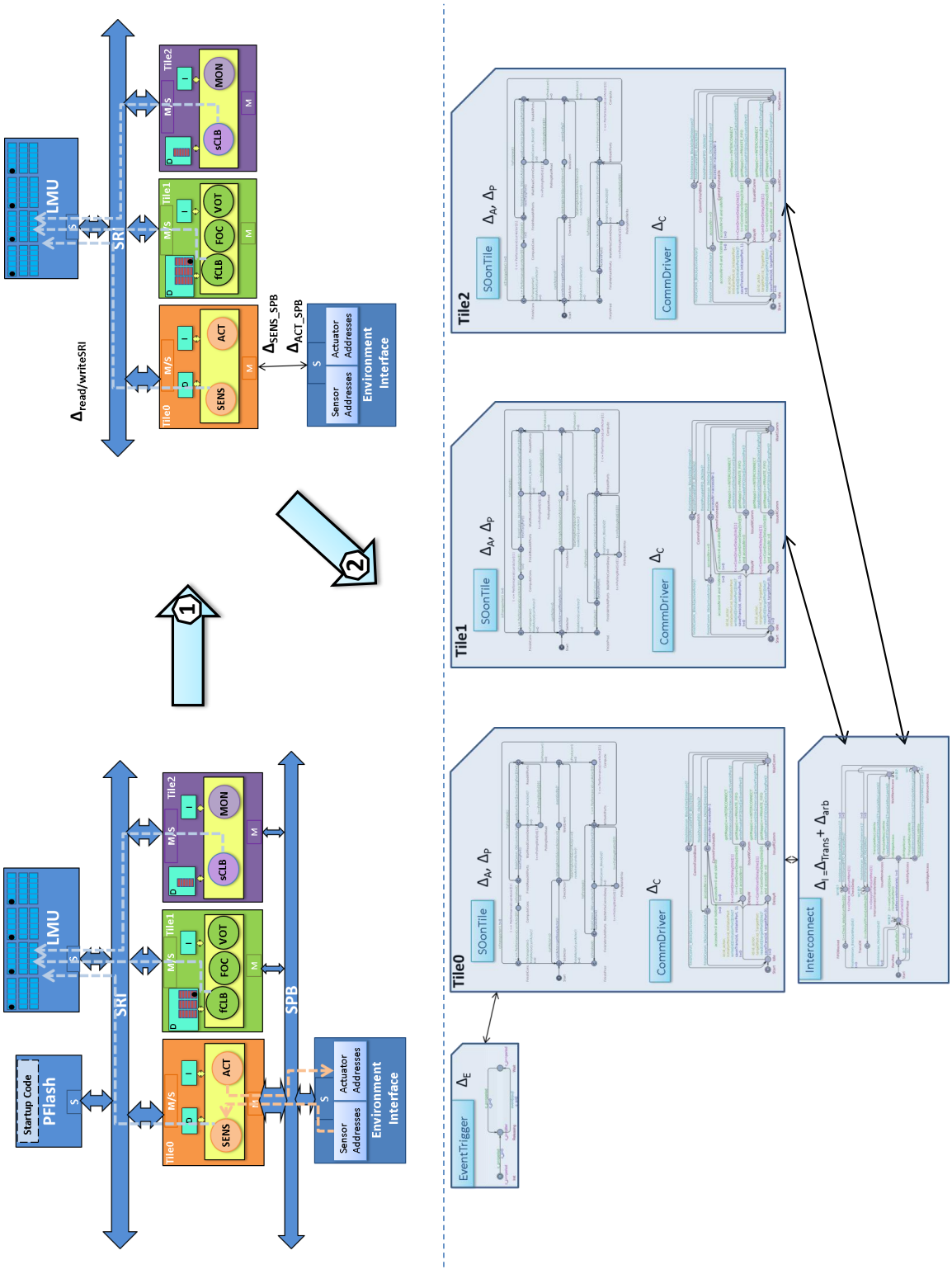


Figure B.2: Abstractions made for the single-beat IPC implementation

List of Abbreviations

Abbreviation	Description
ADC	Analog-to-digital converters
AHB	Advanced High-performance Bus
API	Application Programming Interface
BCET	Best Case Execution Time
CABA	Cycle Accurate Bit Accurate
CAN	Controller Area Network
CFG	Control-Flow Graph
CSFSM	Communicating Synchronous
CTL	Computation Tree Logic
DBMs	Difference Bound Matrices
DC	Direct Current
DDR	Double Data Rate
DMA	Direct Memory Access
DSP	Digital Signal Processor
ECP	EMF Client Platform
ECU	Electronic Control Unit
EMF	Eclipse Modeling Framework
FCFS	First Come First Serve
FIFO	First In First Out queue
FP	Fixed-Priority
FPGA	Field-Programmable Gate Array
FSM	Finite State Machine
GCC	GNU C Compiler
HSDF	Homogeneous Synchronous Data Flow
ILP	Integer Linear Programming
ISS	Instruction Set Simulator
KPN	Kahn Process Network
MBD	Model Based Design
MIL	Model In the Loop
MNC	Maximum Number of Shared Resource Accesses
MoA	Model of Architecture
MoC	Model of Computation

Abbreviation	Description
MoP	Model of Performance
MoS	Model of Structure
MPSoC	Multiprocessor System-on-Chip
NoC	Network On Chip
NP	Non-deterministic Polynomial time
PAPS	Periodic Admissible Parallel Schedule
PASS	Periodic Admissible Sequential Schedule
PBD	Platform Based Design
PE	Processing Element
PIL	Processor In the Loop
PSM	Process State Machines
PTA	Priced Timed Automata
PWM	Pulse-width modulation
RMS	Rate Monotonic Scheduling
RR	Round-Robin
RTC	Real-Time Calculus
RTL	Register Transfer Level
RTOS	Real Time Operating Systems
RTW	Embedded Real Time Workshop
SADF	Scenario-Aware Data-Flow graph
SBD	Synchronous Block Diagram
SDF	Synchronous Data Flow
SDFG	Synchronous Data-Flow graph
SDRAM	Synchronous Dynamic Random-Access Memory
SIL	Simulation In the Loop
SLD	System Level Design
SPB	System Periphery Bus
SRI	System Resource Interconnect
SUA	System Under Analysis
SW/HW	Software/Hardware
TCTL	Timed Computation Tree Logic
TDF	Timed Data Flow
TDMA	Timed Devision Multiple Access
TLB	Translation Look-aside Buffer
TLM	Transaction Level Model
UMA	Uniform Memory Access
UML	Unified Modeling Language
V&V	Verification and Validation
VCD	Value Change Dump
VP	Virtual Platform
WCET	Worst Case Execution Time
XML	EXtensible Mark-up Language
XSLT	EXtensible Stylesheet Language Transformation

Glossary

For consistency purposes and in order to unmask ambiguousness, essential key words used in this thesis are defined in the following (some taken from our own work in [Fakih, 2011]).

Basic block Is a representation of code fragments (instructions) which are executed by target processor atomically i.e. no possibility of branching or preemption is allowed within the basic block.

BCET Is a lower bound of execution time for all possible inputs of a certain executable code (task) which can be obtained through a static code analyzer.

Bi-Simulation In difference to a co-simulation, in a bi-simulation a lock-step simulation takes place, where the first simulation environment (master of simulation) executes a simulation step and the other one executes the same step.

Clock cycle A clock cycle represents the time delay between two equal edges of a clock signal.

Co-Simulation In a co-simulation, two simulation environments interact with each other in order to simulate a common complex system.

Communication Resource such as buses, interconnects where information is transfered from sender to receiver at each (number of) cycle(s).

Composability According to [Akesson et al., 2010], applications in a composable system are completely isolated and cannot affect each other's functional or temporal behaviors.

Control step A control-step is defined as a one update of the controller model in a given period which also corresponds to one update of its generated code.

Cross-compiler is a compiler that runs on a given hardware platform, but creates compiled files (object files or executable programs) for another platform.

Cycle-accurate A model is said to be cycle accurate, if it describes its state evolvement on each clock cycle.

End-to-end Latency The end-to-end latency is defined as the time starting from activating the first instance of the source actor (upon receiving the first event e.g. reading sensor values), executing the SDFG application till the last instance of the sink actor is finished (e.g. updating actuators).

Endianess defines the order in which bytes are aligned in a memory where *big* endianess means that the most significant byte is saved first (in the smallest address) and then the next highest whereas by *little* endianess the least significant byte is first saved.

Execution time analysis As defined in [Ermedahl and Engblom, 2007]: "*Execution time analysis is any structured method or tool applied to the problem of obtaining information about the execution time of a program or parts of a program.*"

Flexibility The flexibility in this work was used in context of a scheduling strategy. It defines the ability to handle potentially changing dependencies between tasks of an application. A strategy is said to be flexible when it can deal with dynamically changing dependencies between tasks [Stuijk, 2007].

Hard real-time As defined in ([Moonen, 2009],13): "*A hard real-time system must satisfy the temporal constraints for any input stream and any initial state of the system*".

Homogeneous SDF An SDFG is called homogeneous if all actors produce or consume a single sample (token) on each input or output channel in each invocation [Lee and Messerschmitt, 1987a].

Host-based simulation A host-based simulation enables execution and verification of the embedded application (running on the virtual-hardware platform) natively on the designer's host machine [IEEE-1666, 2012].

Hyperthreading is the case where different PEs actually use the same execution units or shared caches [Kotaba et al., 2013].

Implementation Model An implementation consists of a structural model (see MoS) and quality numbers (in this thesis timing delays) [Gerstlauer et al., 2009].

Instruction-Set-Simulator (ISS) is a processor simulation model. The ISS is able to execute the machine code of different processors. Using cross-compilers high-level language code is translated to binary code which can be executed by the target ISS.

Integer Linear Programming (ILP) In [Wilhelm et al., 2008] ILP is described as follows: *"Linear programming [Chvatal 1983] is a generic methodology to code the requirements of a system in the form of a system of linear constraints. Additionally given is a goal function that has to be maximized or minimized to obtain an optimal assignment of values to the system's variables. One speaks of Integer Linear Programming if these values are required to be integers. While linear programs can be solved in polynomial time, requiring the solution to be integer makes the problem NP-hard. This indicates that the use of ILP should be restricted to small problem instances or to subproblems of timing analysis generating only small problem instances."*

Iteration An iteration is a set of actor firings such that for each $a \in SDFG$, the set contains the $\gamma(a)$ firings of a, where $\gamma(a)$ represents the repetition vector which can be calculated mathematically (Sect. 2.2.1.1).

Model is a simplified form of a real system at which abstractions and idealizations take place. It offers a fast analysis platform for the performance and the affecting relations within a real system. In this thesis, the term *functional model* is used to refer the top level model which will be modeled in Simulink where only functionality is modeled here and performance energy or timing aspects are neglected.

Model of Architecture (MoA) represents a platform model where the architectural template, decisions and constraints are taken into consideration such as the number of available resources with their interconnections [Gerstlauer et al., 2009].

Model of Computation (MoC)

> *"A Model of Computation (MoC) is a generalized way of describing system behavior in an abstract, conceptual form."* [...] *MoCs are generally based on a decomposition of behavior into pieces and their relationships in the form of well-defined objects and composition rules."* ([Gajski et al., 2009]:50)

"The MoC describes how each component performs internal computation, how components transfer information between them and how they relate in terms of concurrency." ([Baleani et al., 2005]:1)

Model of Performance (MoP) *"Performance models represent the contributions of individual elements to overall design quality in a given implementation."*([Gerstlauer et al., 2009]:3) These design qualities could be for e.g. throughput, response time, latency, area and power. In this thesis we only considered timing design qualities in the MoP.

Model of Structure (MoS) *"As such, a structural model is a representation of the resulting architecture as a composition of components that are internally described in the form of behavioral models for input to the next synthesis step."*([Gerstlauer et al., 2009]:3)

MPSoC *"Multiple-Processor System-on-a-Chip (MPSoC) architectures which are heterogeneous, custom design, and often made out of standard cores whereof some often are special-purpose"* ([Gustavsson, 2010]:2). In a heterogeneous MPSoC, *"the processors are different; hence the rate of execution of a task depends on both the processor and the task. Indeed, not all tasks may be able to execute on all processors"* ([Davis and Burns, 2011]:3). On the other side, in a homogeneous MPSoC *"the processors are identical; hence the rate of execution of all tasks is the same on all processors"*. ([Davis and Burns, 2011]:3)

Multi-core are *"commodity processors, with mostly homogeneous sets of cores"* ([Gustavsson, 2010]:2).

NP-hard Problem Problems which belong to NP-class (non-deterministic polynomial-time) are problems which can be solved through a non-deterministic Turing-machine within a polynomial time. An NP-hard problem is at least so hard as the hardest problem in NP-class, i.e. an algorithm which solves an NP-hard problem could solve all problems in NP-class [Bovet and Crescenzi, 1994].

Period The period of an SDFG is defined as the time an SDFG takes to complete one iteration.

Predictable when used means exhibiting deterministic temporal behavior easing the ability to reason on the timing behavior of a specific component. In the context of real-time, an application execution on an MPSoC is said to be predictable we are able to determine beforehand whether or not the right outputs happen at the right (predicted) moment.

In order to explain that in more detail the following example is quoted from [Moreira, 2012]:
"Consider two single-processor architectures that make use of the same processor core, but with different memory hierarchies. In the first of them, the processor accesses the main memory through a cache. Say that, in this case, a memory read can take anywhere between 2 and 50 processor cycles, depending on whether the access is a cache hit or a cache miss. The read operation in such an architecture is clearly predictable, as we can bound the time it takes to completion of the operation. Now consider the second architecture, where the access to the main memory is direct, and due to the arbitration technique employed, it takes exactly 100 cycles for every access. According to our definition (and according to any intuitive notion of predictability), the second system is more predictable than the first, as the bound on timing behavior is tighter (0 cycles of variance against the 48 cycles of variance in the first case)."

Register Transfer Level (RTL) RTL is a low abstraction level of a computer system where the functionality is described as logic operations which are provided by registers. Typical components at this level are adders, multipliers, registers, etc.

Scheduling A scheduling mechanism determines the order of the tasks to execute on a given resource, where the tasks with highest priority are granted access first. Scheduling strategies are typically either compile-time scheduling (e.g. static-order scheduling) or run-time scheduling where at run-time scheduling decisions are made.

Simulation generally means experimenting on models if during a system-analysis a model is accomplished that replaces the original system and experiments are carried out on this model [Page et al., 2001]. Simulation Models can be classified according to the way their state transitions react with respect to time in *discrete* or *continuous*. In a discrete simulation model the state variables change abruptly only at discrete times, while in a continuous system the state variables change over the time continuously. In thesis, we will mainly deal with discrete time (discrete-event simulation) and continuous time but our functional model for which the timing validation is applied, is a pure discrete model. A continuous time environment is periodically sampled by a discrete controller model.

Soft real-time *"A soft real-time system has a target for its average behavior but does not have temporal constraints. Furthermore, a soft real-time system must have a fall-back mechanism to recover from deadline misses."* ([Moonen, 2009]:14)

Storage Resource is a resource (e.g. memories, buffers and caches) which keeps information for a while (for several cycles or permanently).

Synchronous data-flow Graph is a special case of the general data-flow MoC in which the number of data samples (tokens) produced or consumed by each actor by each activation is known a priori [Lee and Messerschmitt, 1987a].

System the general definition of a system could be stated as a purposeful collection of inter-related components working together toward some common objective [Banks Jerry, 2005]. In this thesis, it is specified to be a control system which has some critical timing requirements. Without diving in the control theory deepness, a control system can be simply defined as system having some sensors which gives feedback from the environment to some control unit(s) which affect the environment through actuators, to correct or regulate some behaviors within defined time intervals.

System synthesis The synthesis step includes the processes of allocating resources, binding (assigning the tasks to allocated hardware resources) and scheduling the behavioral model (execution order of tasks) on the defined architecture, and thus transforming a specification into an implementation [Gerstlauer et al., 2009].

System-on-a-Chip A System-on-a-Chip (SoC) is a system whose (digital/analog) components are integrated on a single chip (integrated circuit: IC).

Throughput The throughput of an SDFG is the inverse of the period i.e. the number of SDFG iterations within a time unit.

Tokens *"Tasks also need some input data (or control information) before they can start and usually also produce some output data; such terms of information are referred to as tokens."* ([Shabbir et al., 2010]:5)

Turing-complete A MoC or a programming language is said to be Turing-complete if it is able to calculate all the functions that can be calculated by a universal Turing machine.

Use-case A use-case is a defined as a scenario used (with specific input data) to trigger the system (design) under test.

Validation In contrast to verification, a validation process concentrates more on the satisfaction of stakeholders, i.e. checking if we are building the right product, and if it meets the requirements of the domain where it will be used and will perform as expected [Marwedel, 2010].

Verification A verification process checks the compliance of the (SUA) implementation against functional and non-functional requirements i.e. inspecting if the model has been built right. In this thesis when used, both formal methods and simulative verification are used (see Sect. 2.5).

WCET Is an upper bound of execution time for all possible inputs of a certain executable code (task) which can be obtained through a static code analyzer.

WCRT Is an upper bound of response time for all possible inputs of a certain executable code (task) which is defined as the sum of the WCET of task and the waiting time which can caused be caused for e.g. due to contention on shared resources.

List of Figures

List of Tables