CARL VON OSSIETZKY universität OLDENBURG

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

Efficient State Space Exploration of Reactive Object-Oriented Programs

Dissertation zur Erlangung des Grades eines
Doktors der Naturwissenschaften

von

Dipl.-Inform. Marc Lettrari

Gutachter:

Prof. Dr. Werner Damm
Prof. Dr. Ernst-Rüdiger Olderog

Tag der Disputation: 25. November 2005

Zusammenfassung

Für im Bereich sicherheitskritischer Systeme eingesetzte Programme, die häufig reaktiv und zunehmend in objekt-orientierten Sprachen wie z.B. C++ realisiert sind, ist ein möglichst fehlerfreier Ablauf wichtig. Ein Ansatz zum Aufspüren möglicher Fehler in solchen Programmen ist das sogenannte Software Model Checking, das im Gegensatz zum herkömmlichen Testen einen im Idealfall vollständigen Nachweis der Fehlerfreiheit eines Programmes durchführen kann. In der Praxis gelingt solch ein Nachweis, der auf einer vollständigen Zustandsexploration der zugrundeliegenden Programme beruht, aufgrund der Größe der Zustandsräume der betrachteten Programme eher selten. Doch selbst wenn eine vollständige Zustandsexploration nicht durchgeführt werden kann, können partielle Zustandsexplorationen durch Software Model Checker sehr hilfreich sein, da mit ihnen Fehler entdeckt werden können, die durch herkömmliches Testen nur schwer zu finden sind.

In dieser Arbeit wird ein neuer Ansatz zur Zustandsexploration von eingebetteten C++ Programmen vorgestellt, der eine effiziente Suche nach Fehlern oder Zuständen mit bestimmten Eigenschaften erlaubt. Da C++ standardmäßig keine Sprachmittel zur Beschreibung von Parallelität und Synchronisation bereitstellt, benutzen eingebettete C++ Programme in der Regel betriebssystemspezifische Funktionen zur Realisierung von Parallelität bzw. Synchronisation. Um eine einheitliche Behandlung solcher Programme zu ermöglichen, wird zunächst eine Erweiterung von C++ um die Konzepte Parallelität und Synchronisation vorgestellt, die wir SymC++ nennen.

Basierend auf einer Analyse bestehender Verfahren zur Zustandsexploration von Programmen sowie der Identifikation von Ansatzpunkten zur Verbesserung dieser Verfahren, werden im Anschluss daran zwei neue Techniken für eine effiziente Zustandsexploration von SymC++ Programmen vorgestellt. Der erste Ansatz beruht auf einer kombinierten explizit-symbolischen Darstellung von Zustandsmengen von SymC++ Programmen. Diese Darstellungsform erlaubt auf der einen Seite eine kompakte Repräsentation von großen Zustandsmengen und auf der anderen Seite eine effiziente Realisierung von Operationen auf diesen Zustandsmengen, die für eine effiziente Zustandsexploration benötigt werden. Insbesondere können bekannte Approximationsalgorithmen, die zu Leistungsverbesserungen des Explorationsalgorithmus führen, mittels geeigneter Anpassungen an diese Darstellungsform angewendet werden. Die Effizienz der entwickelten Techniken wird anhand mehrerer Testprogramme experimentell evaluiert.

Der zweite Ansatz beruht auf der Anwendung heuristischer Suchmethoden in der Zustandsexploration. Im Gegensatz zu den vorher betrachteten uninformierten Suchalgorithmen wie Breitensuche oder Tiefensuche benutzen heuristische Suchverfahren dy-

namisch Informationen, die die Suche gezielt in Richtung eines Zustands mit bestimmten Eigenschaften lenken sollen. Wir entwickeln einen neuen Ansatz zur automatischen Berechnung solcher Informationen aus SymC++ Programmen. Dazu wird zunächst eine Abstraktion des zugrundeliegenden Programms generiert. Auf der Basis der generierten Abstraktion und einer gegebenen Eigenschaft wird dann eine heuristische Funktion berechnet, die zur heuristischen Zustandsexploration des ursprünglichen Programms eingesetzt werden kann. Wiederum wird die Effizienz der entwickelten Methode anhand mehrerer Testprogramme experimentell evaluiert.

Abstract

For programs employed in safety critical systems, which are often reactive and implemented in object-oriented programming languages like e.g. C++, an error-free functioning is important. So-called *Software Model Checking* is an approach that can be used to find potential errors in such programs. In contrast to traditional testing methods commonly used for software validation, software model checkers can potentially prove the absence of errors in programs. Unfortunately, in practice such a proof, which is based on a complete state space exploration of the considered programs, is seldom possible due to the size of the state spaces of the programs. However, even if a complete state space exploration is impossible, partial state space explorations performed by software model checkers can be very useful since they can track down errors that are hard to detect by traditional testing methods.

In this thesis we present a new approach for state space exploration of embedded C++ programs. The developed state space exploration methods allow an efficient search towards program states characterized by a user-defined property. Since C++ by default does not provide language constructs dealing with concurrency and synchronization, embedded C++ programs often use operating system specific library functions for realizing concurrency and synchronization. To facilitate a uniform treatment of such programs, we define an extension SymC++ that extends C++ with the concepts of concurrency and synchronization.

Based on an analysis of existing methods for state space exploration of programs and the identification of starting points for improvements, in the sequel we present two new approaches for an efficient state space exploration of SymC++ programs. The first approach is based upon a composite explicit-symbolic representation of sets of states of SymC++ programs. The proposed representation allows on the one hand a compact characterization of large sets of states, and on the other hand an efficient realization of operations on these representations that are necessary for an efficient state space exploration. In particular, it is possible to adapt known approximation algorithms to this representation, increasing the efficiency of state space exploration. The efficiency of the developed techniques are evaluated experimentally by means of several test programs.

The second approach is based on the application of heuristic search techniques for state space exploration. In contrast to uninformed search algorithms like breadth-first search or depth-first search, heuristic search algorithms dynamically utilize information that should guide the search towards states with a specific property. We develop a new approach for computing such information automatically for SymC++ programs. For this purpose we firstly generate an abstraction w.r.t. a given property of the considered

SymC++ program. Based on this abstraction we compute a heuristic function that can be used for a heuristic state space exploration of the original program. Again, the efficiency of the approach is evaluated experimentally by means of several test programs.

Danksagung

Ich möchte folgenden Personen danken, die in unterschiedlicher Weise zur Erstellung dieser Arbeit beigetragen haben:

- Prof. Dr. Werner Damm, der mir die Möglichkeit gegeben hat, diese Arbeit zu erstellen und mir die nötigen Freiräume ließ, eigene Ideen zu entwickeln und umzusetzen.
- Allen aktuellen und ehemaligen Kolleginnen und Kollegen der Abteilung Sicherheitskritische Systeme für eine entspannte und angenehme Arbeitsatmosphäre. Ein besonderer Dank geht an Christian Mrugalla für die oft stundenlange Klärung semantischer Aspekte von C++.
- Allen Mitarbeitern der OSC Embedded Systems AG für die kostenlose Bereitstellung von ca. 100 Litern Kaffee, die wesentlich zur Aufmerksamkeitssteigerung des Verfassers beigetragen haben.
- Allen aktuellen und ehemaligen Mannschaftskollegen des TSV Ueffeln für die fußballerischen Erfolge der letzten Jahre und die nötige sportliche Abwechslung.
- Meiner Familie, insbesondere meinen Eltern, die mir den Weg durch das Studium ebneten und durch ihr beständiges Interesse am Fortschritt dieser Arbeit nicht unwesentlich zur Fertigstellung beigetragen haben.
- Barbara, dass Du Dein Leben mit mir teilst, mich in Allem unterstützt und immer an mich geglaubt hast. Danke für alles.

Contents

List of Figures

1 Introduction

Nowadays, computers and computer programs are ubiquitous. We are confronted with them when the programmable alarm-clock wakes us up in the morning. Several hidden embedded computers surround us when we are driving in our cars to our workplaces, where we use a key with a small computer inside to open the programmable lock of the office door. We need several computer programs during work, and when we are back at home in the evening we look at moving pictures generated by a computer program inside our digital television. Hundreds of computers and computer programs control and influence our daily life, most of them without being noticed by us.

The development of these systems is a complicated task. In fact, modern computer systems are the most complicated structures mankind has ever built in its history: Modern microprocessors are implemented by millions of transistors, and computer programs can consist of millions of lines of code. Therefore, it is no wonder that these systems often have errors that can lead to malfunctions. The steadily growing size of these systems makes it important to avoid errors. If we assume that the number of errors in these systems grows with their size, it becomes clear that the larger the systems are, the more errors they will contain, and the probability that they will actually work is reduced. This can have seriously economical risks for the manufacturers.

Besides the possibly economical risks, errors in so-called *safety critical* systems are even more critical. A system is safety critical if a faulty behavior of the system can lead to considerable damage of objects or humans. A good illustration of a serious failure of a safety critical system is what happened September 14, 1993 on the runway at Warsaw airport in Poland. A Lufthansa Airbus A320-200 with 72 people on board was landing in heavy rain. The plane did not get much traction from the wheels in the landing gear on the wet runway, but the pilots knew that they could count on the thrust reversers on the main engines to bring the plane to a stop. As it happened, the thrust reversers failed to deploy in time, and the plane overshot the end of the runway. A thrust reverser should never be activated when a plane is in flight. Most planes have elaborate protection built-in to prevent this from happening. Among other things the protection includes the check of three conditions: the landing gear must be down, the wheels must be turning, and the weight of the plane must be carried on the wheels. In this case the landing gear was down, but the wheels were aquaplaning, and an unexpected tailwind provided enough lift on the wings that the control software did not decide until nine seconds after touchdown that the plane had landed. Two people lost their lives when the plane went off the end of the runway.

Whether due to economical or safety reasons, it is clear that every computer controlled system should have as few errors as possible. This thesis deals with the problem of the

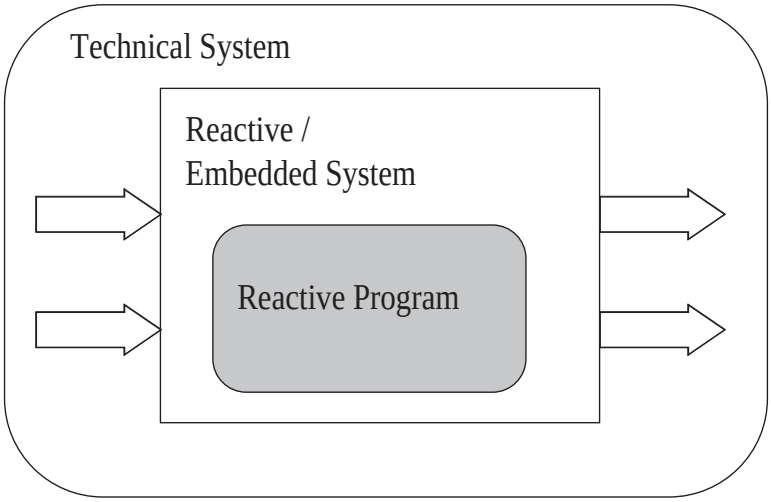


Figure 1.1: **Reactive Programs.** Schematic view of an reactive control program as a component of an embedded system.

correctness of programs that realize such safety critical systems. In the following we will shortly describe some characteristics of these systems.

1.1 Reactive Systems

Reactive systems are systems that somehow have to react to events coming from its environment. Figure 1.1 shows the typical structure of a reactive system. A reactive system is often a so-called *embedded system*, i.e., it is often a part of a bigger technical system. In contrast to e. g. transformational systems like compilers, which perform some computations on the given input and then terminate, reactive systems continuously interact with their environment. Furthermore, unlike an interactive system like e. g. a graphical user interface of an operating system, a reactive system normally must be fast enough to react on a given environment event before the next event from the environment occurs. For this reason, reactive systems often have to satisfy some real-time constraints and therefore fall into the category of *real-time systems*.

As described in the example of the Airbus crash in the preceding section, reactive systems often have to react and control several systems simultaneously. To achieve this goal, reactive systems are often realized as *concurrent systems*. This means that these systems consist of a couple of concurrent threads that may be dynamically generated and aborted. The proper scheduling and synchronization of the threads must be designed carefully because wrong scheduling or synchronization can lead to faults like deadlocks or unintentional changes of variable values. For instance, when two threads concurrently access a shared resource, then the access to the shared resource has to be protected so

that only one of the two threads can use the resource at a time.

There are many different ways to actually implement a reactive system. If execution speed should be as fast as possible, one might suggest a complete hardware solution. However, this would also be the most expensive solution, because the development of a hardware circuit is much more expensive than that for an equivalent software component. Furthermore, the performance of modern microprocessors is so good that often elaborate hardware designs are not much faster than optimized programs running on a microprocessor. Additionally, the systems to be realized are getting increasingly complex, often using complex data types and heavily interacting threads. Because of the increasing complexity and the lower costs, more and more reactive systems are at least partly realized by *reactive programs* as it is depicted in fig. 1.1, where a reactive program is running as a part of the reactive system. Aside from the better handling of the complexity and the lower costs, the use of software for reactive systems also increases the flexibility and maintainability by allowing software updates, as it is already common for e.g. engine control software in modern cars.

In this thesis, we are dealing with reactive programs realizing reactive systems. As we are concerned with the correctness of such programs, we will give a short overview about verification and validation methods of computer programs in the next section.

1.2 Verification and Validation

The terms verification and validation describe procedures for establishing the correctness of programs. To check the correctness of a program firstly one has to define the properties the program should have. For instance, a program to calculate the sum of n integers should have the property that, given $i_1, \cdots, i_n$ integers as inputs, the program should actually compute $\sum_{i=0}^{n} i_n$ as the result. A program is said to be correct if it fulfills all properties stated for that program. Verification and validation differ in their methods and achievable results when it comes to demonstrate the correctness of programs.

The term validation includes techniques like simulation, testing and debugging, which are currently the standard methods used in industry to validate the correctness of programs. Simulation and testing usually involve providing certain inputs and observing the corresponding outputs. All these techniques are normally carried out manually, which has both assets and drawbacks. An experienced programmer often has a good intuition where possible errors reside, and therefore he can often find these errors quickly. However, the programmer's intuition can also be a serious problem when there are errors in the program that only show up under unusual circumstances. For instance, the Airbus crash described in the introduction only occurred because of the unforeseen combination of multiple events. In fact, when using testing and simulation for validation of programs one can never be sure if errors still reside in the program because testing and simulation are inherently incomplete, i.e. with these techniques one can normally only check a subset of all possible behaviors. To overcome the limitations of these incomplete methods the approach of *formal verification* has been suggested. While simulation and testing

explore some of the possible behaviors of the system, formal verification tries to conduct an exhaustive exploration of all possible behaviors. Thus, when a program is declared correct with respect to given properties by a complete formal verification method, it implies that all behaviors have been explored which could potentially influence the stated properties.

Several approaches to formal verification have been proposed over the years. They can be roughly classified into deductive and enumerative methods. In a deductive approach, both the program and the properties to be checked are described by means of formulas of a particular logic. Based on the logic's underlying axioms and proof rules one tries to deductively verify the given properties w.r.t the program. Early approaches in this area stem from Floyd [Flo67] and Hoare [Hoa69], who proposed guarantee commitment style proof rules for computer programs: Given that Φ and q were formulas of some predicate logic and P is a program statement, then the *Hoare triple* $\{\Phi\}P\{q\}$ means that if Φ holds in the starting state of P and P terminates then the property q holds. In a Hoare triple Φ is called the precondition and q is called the postcondition. Based on Hoare's work several extensions for other classes of programs were proposed, e.g. for concurrent [OG76a, OG76b] or distributed programs [AFdR80, LG81]. One problem of these approaches is that they consider solely transformational programs, i.e. programs that compute an output from some inputs and then terminate. To be able to handle reactive programs adequately, Pnueli [Pnu77] used *temporal logic* to describe properties of reactive programs. Within the framework of temporal logic many interesting properties of reactive programs such as mutual exclusion or deadlock freedom can be described. A detailed description of this approach can be found in [MP91].

A major problem of all deductive methods is the fact that it is quite hard to manually prove a program to be correct. Even for small programs and persons with experience using the proof-method it may require a lot of time to realize the proof completely, and for larger programs it is practically impossible. Therefore, one often makes use of so-called *theorem provers*, which are programs that can realize proofs automatically or semi-automatically. There exist theorem provers for many different logics, e.g. provers for first order predicate logic (ACL2 [KM94], LP [GG88]) or several higher order logics (HOL [GM93], PVS [ORSS95], Isabelle [Pau94]). With the help of theorem provers many necessary proof steps can be done automatically, and there are some impressive examples using theorem proving approach, e.g. the verification of microprocessors AAMP5 [MS95]. However, even with automated theorem provers it is a hard task to verify a program. It is important to realize that some mathematical tasks cannot be performed automatically, which is an important result of the theory of *computability* [HU79]. In particular, it shows that there cannot be an algorithm that decides whether an arbitrary computer program terminates or has any other nontrivial property. Thus, most proof systems cannot be completely automated, and therefore the decisive proof steps must often be done manually.

In contrast to the proof-based methods, the enumerative methods try to verify the correctness of a program w.r.t. a given property by exhaustively exploring the possible computations of a program. A configuration or state of a program can be seen as a

vertex in a directed graph, and a computation step corresponds to an edge from the vertex representing the state before the computation step to the edge corresponding to the state after the computation step. In this directed graph, the set of states reachable from the starting states of the program is called the state space of the program. Given a property usually formulated in some temporal logic, a so-called *model checking* algorithm traverses the state graph of the program in order to find out if the property holds for this program or not. When the state space of the program is finite and sufficient computational resources are available, a model checking procedure will always terminate with a definite yes/no answer. Model checking was developed independently by Clarke, Emerson & Sistla [CE81, CES86] and Queille & Sifakis [QS82, QS83]. The main limitation of this approach is that for larger programs or programs with parallel components the reachable state space can be tremendously large, a problem usually called the *state explosion problem*. For instance, a program using only two 32-bit integer variables can have potentially $2^{64} \approx 1.8 \cdot 10^{19}$ reachable states. To cope with such large state spaces, a couple of different techniques have been developed. For instance, for concurrent programs which run asynchronously so-called *partial order reductions* can be applied. This technique exploits the independence of concurrently executed events. Two events are independent of each other when executing them in either order results in the same global state. When a property to be checked cannot distinguish between two interleaving sequences that differ only by the order in which concurrently executed events are taken, it is sufficient to analyze only one of them, thus reducing the number of states that must be explored for model checking. Another approach to tackle large state spaces is to represent sets of states rather than individual states. The key idea is that sets of states can be efficiently represented symbolically, i.e. the symbolic representation of all states in a set can be much more compact than an individual representation of all states in the set, an approach which is called *symbolic model checking* [BCMD90, BCM+92, BCM90]. Often the well-known *binary decision diagrams* (BDDs, [Bry86]) are employed for the symbolic representation, e.g. in the model checker SMV [McM93], but also other symbolic representations like *linear arithmetic constraints* [DP01] have been used for model checking. With an appropriate symbolic representation model checking is even applicable to systems with infinite state spaces provided that the number of symbolic states is finite. For instance, model checkers for so-called *timed automata* which have to deal with the infinite domain time use finite state symbolic representations, e.g. the timed automata model checkers *Uppaal* [BLL+95] and *Kronos* [Yof97]. Though there have been many improvements regarding the size of systems which can be analyzed by model checking, the state explosion problem is still one of the main obstacles of automatic verification. The next section explains another problem which lies in the discrepancy between what can be verified and what is actually used for implementing reactive programs.

1.3 Models and Programs

Besides the complexity problems of manual and automatic verification, one of the main problems in the employment of verification techniques for real systems is the fact that the system must be specified in the verification language, which is usually not the implementation language of the system. Figure 1.2 explains this discrepancy by showing the current possibilities of automatic verification. Most automatic verification methods work on a representation of the system which is usually much more abstract than the representation of the same system as a program in a common programming language (v. fig. 1.2 left). Many different languages were developed for describing such abstract models, e.g. the so-called *synchronous languages* Esterel [BC85, BdS91], Signal [BG90, BGJ91], Lustre [CPHP87, CPHR91] or Statecharts [Har87, HN96, HP96]. While this is on the one hand an advantage as to verification because the simplicity of the abstract model allows an easier verification, it constitutes a serious problem on the other hand.

One problem is that sometimes certain aspects of the real system cannot be described adequately in the abstract model. But even if all essential aspects of the system can be described in the abstract model, usually not all aspects of a realization or implementation

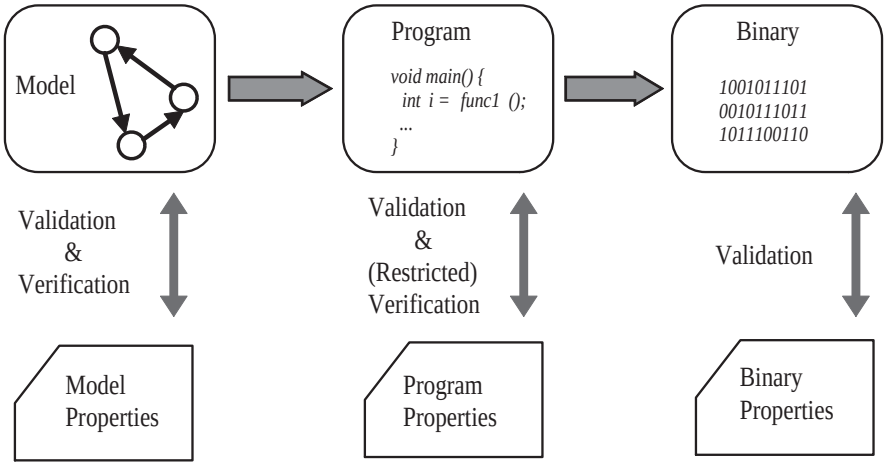


Figure 1.2: **Validation and Verification for different system representations.** For abstract system representations (left) as e.g. finite state machines, automatic validation and verification methods are available. For programs of common programming languages (middle) like C, C++ or Java, in most cases only validation methods like testing are used. However, in recent times software model checkers have been applied that are able to verify the correctness of programs which use certain subsets of available language constructs. For the deployed system including the hardware components and the compiled binary program (right), only testing can be used to validate the correctness of the system.

of the system can be modeled. Therefore, even if properties can be verified to hold in the abstract model, the same properties need not necessarily hold in an implementation of the abstract model. In general, the problem of deriving a correct implementation from an abstract model is an open problem.

Another problem lies in the practical applicability of automatic verification. As mentioned above, to be able to perform an automatic verification one has to have a model in a verification language. However, for most existing systems there exists only the source code of the program realizing the system. If we want to prove something about these systems, one has to manually build an abstract version of the program in the verification language, a process that is often complex, time consuming and error prone. Even if a new system has to be developed and one can start with the design of an abstract model, in most cases several components of already existing systems will be reused, i.e., at least these components must be redesigned in the verification language. This problem is strengthened by the fact that the engineers of the systems do know common programming languages, but seldom have enough knowledge of the verification language in order to be able to build a verification model.

Because of the problems mentioned above, in recent times there has been an increasing effort for building automatic verification methods and tools for real programming languages, an approach which is called *software model checking* (v. fig. 1.2 middle). We will describe this approach in more detail in sect. 2. In this thesis, we also follow this approach by presenting new verification methods for reactive programs written in a variant of a commonly used programming language. While there are extensions regarding the implementation of reactive programs for all common programming paradigms, e.g. specific extensions for functional or logic based programming languages [FJ94, WR95, GR95, Gre97], imperative languages play the dominant role. The reasons for that are diverse: functional and logic based programming languages usually have higher memory requirements and their execution is slower, and due to the broad spreading of imperative languages they usually have a better infrastructure (e.g. compilers for specific hardware platforms or libraries for specific operating systems). In former times, the imperative programming languages *C* and *Ada* were often used for implementation. As object-orientated languages offer better structuring mechanisms for larger systems and simplify the reuse and adaptation of program code, in recent times more and more reactive programs have been implemented using object-oriented languages like *Java* and *C++*. In this thesis, we will focus on SymC++ programs which is a variant of C++ suitable for describing reactive programs.

It is important to note that programs written in common programming languages are more detailed models of the real systems, but nevertheless they are just another form of an abstraction of a real system. This means that verification results obtained by verifying programs need not necessarily hold for the compiled binaries, e.g. if the compiler used to create the binaries works incorrectly for some programs. This restricted validity of verification results is immanent for every verification method, independent of the verification model. Even if we would be able to automatically verify the binary programs (v. fig. 1.2 right), there would still be the need to check the real system to

detect e.g. errors in the underlying hardware. Such a check is usually done by testing the deployed system. However, having a verifiable model is very valuable in detecting many design errors, and it definitively increases the correctness and quality of the implemented system, even if it cannot be fully verified because of its complexity. The next section explains the application of automatic verification methods for systems which are too complex to be verified completely.

1.4 Partial and Heuristic State Space Exploration

As described in sect. 1.2, due to complexity problems a complete state space exploration which is necessary for automatic verification is often infeasible. But even if a full verification cannot be achieved, model checking can still yield valuable results. A useful feature of model checking tools is that in case the property to be verified does not hold the model checker usually generates a counterexample showing an execution sequence of the checked program that violates the property. Therefore, a model checker can also be used as an automatic debugging tool, searching for execution traces that result in errors or certain user defined states. If there exists at least one such state, a complete model checking procedure will eventually find an execution trace to that state. But despite the improvements regarding the size of systems which can be analyzed by model checking, in many cases a complete exploration of the state space is still impossible. However, for finding bugs a complete state space exploration is not necessary because it suffices to explore the part of the state space that contains the desired state.

The left part of fig. 1.3 shows schematically the results obtainable with complete resp. incomplete state space exploration for two different programs P1 and P2 when searching for error states. Since the reachable state space of P1 contains no error states, a complete exploration algorithm would detect this fact provided that enough computational resources are available. Contrary to this, an incomplete exploration algorithm is in general not able to decide that there is no error state reachable in P1. However, as there are error states reachable in P2, both complete and incomplete algorithms are able to find these states. A different aspect of complete resp. incomplete exploration algorithms is shown in the right part of fig. 1.3. Even if we assume that due to restrictions of computational resources a full exploration of the state space of program P3 is impossible and therefore a complete exploration algorithm is unable to find the error state, an incomplete exploration algorithm is still able to find this state. The reason for that is that the number of states which can be explored with given computational resources is in general larger if the exploration algorithm need not to be complete. For instance, an incomplete exploration algorithm need not necessarily store all visited states, which is essential for complete exploration algorithms to ensure termination. The right part of fig. 1.3 also shows another aspect of incomplete exploration algorithms. While for complete exploration algorithms it is in principle not important in which order states are visited, at least as long as at the end of the search all states have been visited, it is of central importance for incomplete exploration algorithms. When there are error

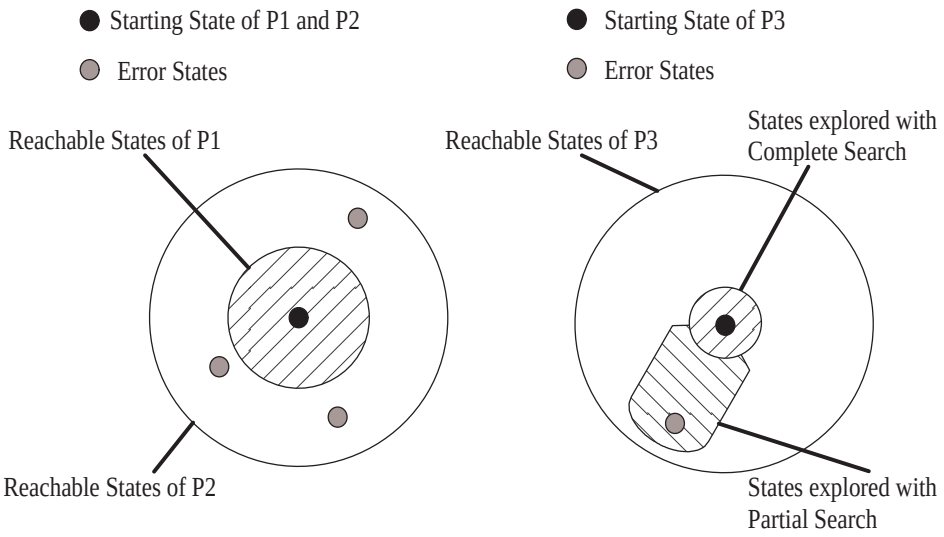


Figure 1.3: **Complete and Partial State Space Exploration.** A complete state space exploration can prove that no error states are reachable in program P1, which is in general not possible with a partial (or incomplete) state space exploration (left). Contrary to this, both complete and partial state space explorations are able to find the reachable error states in P2 (left). However, as partial exploration algorithms can explore a larger number of states than complete exploration algorithms within limited resources, partial exploration algorithms can find error states in large state spaces that cannot be found with complete exploration algorithms (right).

states reachable and not all states can be explored, then the search is only successful if the algorithm explores a part of the state space that contains an error state.

The decision which part of the state space is explored is made by the underlying exploration algorithm and in most cases statically determined, i.e., the order in which different states are explored is fixed regardless the given property. Search algorithms that do not take into account information regarding the search goal are so-called *uninformed search algorithms*. Two prominent examples of such exploration algorithms are *breadth-first search* (BFS) and *depth-first search* (DFS). Roughly speaking, in BFS, after exploring the starting states of the system, all states which are reachable in one step are explored first, then all states reachable in two steps and so on. Contrary to this, in DFS, before the siblings of the current state and all states reachable from them are explored, first all states reachable from the current state are explored. When searching for potential bugs, the advantage of using breadth-first search is that it finds counterexamples of minimal length which are, due to their conciseness, easy to analyze by users. The disadvantage is that in breadth-first search the number of states grows exponentially with the search depth, which means that it is hard or impossible to find property violating states with

larger distance to the start states. In contrast to this, DFS can find property violating states with larger distance to the start states. The disadvantage of DFS is that it yields long counterexamples when it finds a property violating state, and sometimes DFS has to explore much of the state space before reaching a property violating state with a short distance to the start states.

In contrast to uninformed search algorithms, *informed* or *heuristic* search algorithms take into account information about the search goal to direct the search into regions of the state space where a property violating state is supposed. The information is obtained by applying a so-called *heuristic function* h to each visited state. Given a state s, the value $h(s)$ estimates the distance from s to a state fulfilling the search goal. If h provides a good estimation of the real distance to a goal state, heuristic search is able to find a path even in state spaces where uninformed search algorithms fail, or it finds paths much faster. However, crucial for the effectiveness of heuristic search algorithms is an informative heuristic function.

1.5 Organization of the Thesis

After giving a short introduction to reactive systems and their validation and verification procedures, reactive programs and the application of partial and heuristic state space exploration for automatically debugging them, we now formulate the main aims and the further structuring of the thesis. Primary aim of the thesis is the development of an efficient state space exploration procedure for reactive object-oriented programs that allows checking the validity of user given properties. Such a procedure can be used for e.g. automatic debugging of programs or automatic test generation. The further structure of the thesis is as follows:

- Section 2 will give a more detailed overview about current state space exploration techniques. We will analyze the particularities of software model checking and derive several requirements for an efficient state space exploration procedure. In particular, the assets and drawbacks of complete and incomplete methods, the use of symbolic and explicit state representations, and the application of heuristic search and the generation of informative heuristic functions are discussed. As a result of this analysis several design decisions for an efficient state space exploration are formulated. Furthermore, we also give a short overview about other related work.

- In sect. 3, the programming language SymC++ is introduced which will be used throughout this thesis to describe reactive object-oriented programs. After describing the main features of SymC++, e.g. object-orientation and the thread concept, we will introduce a simple language C_{min} that we use throughout the thesis as a formal model for SymC++ programs, and we define a simple logic TL that can be used to define properties of such programs. We explain how SymC++ programs can be translated into C_{min} programs, and we show how we employ SymC++

to perform state space exploration of C++ programs that are generated by an automatic code generator of a commercial UML case tool.

- After having introduced C_{min} programs as a formal model of SymC++ programs, in sect. 4 we define a state space exploration algorithm for C_{min} programs. We start with a very simple explicit-state algorithm that we successively extend with different optimizations. The effectiveness of the optimizations is evaluated experimentally. Later, in sect. 4.2, we define a composite explicit-symbolic state space exploration algorithm, whose effectiveness is also evaluated experimentally.

- As a second component for efficient state space exploration, in sect. 5 an abstraction-based heuristic search procedure is presented. Based on a program P and a property, we generate an abstraction of P for which a complete state space exploration is possible. We use the generated abstract state space as a heuristic to guide the search in the concrete state space towards state that fulfill the given property. Again, we evaluate the effectiveness of the presented heuristic search procedure by means of several experiments.

- At the end, in sect. 6 we will give a summary of the contributions made by this thesis. Moreover, several directions of further work are identified and discussed.

2 Related Work and Design Decisions

In this section, we will give an overview about related work that will allow us to classify the approach we follow in this thesis. Figure 2.1 shows the main problems that arise in the context of state space exploration of reactive, object-oriented programs, in particular C++ programs, and the techniques we apply to overcome these problems. A C++ program has a complicated program control flow due to object-oriented mechanism like inheritance and polymorphism. Furthermore, direct or indirect recursive function calls can create arbitrarily many function invocations during runtime. Using dynamic object creation and pointer variables, a C++ program can create arbitrarily linked structures like lists or graphs that can grow and shrink dynamically. Moreover, reasonable C++

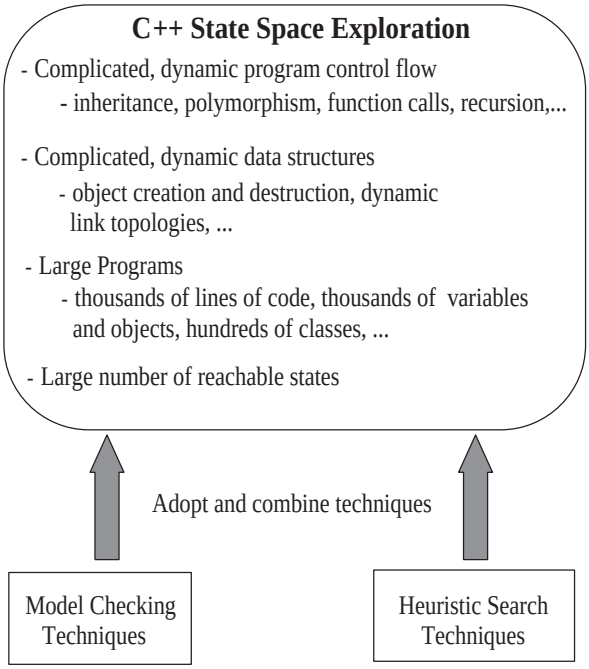


Figure 2.1: **Problems of C++ state space exploration.** The main problems of state space exploration of C++ programs are tackled with techniques from model checking and heuristic search. The borrowed techniques have to be combined and adapted to be effective in the context of C++ state space exploration.

programs are often quite large, having hundreds of classes and thousands of lines of code. Finally, because of e.g. many input values with large domains or concurrently executing threads, the number of reachable states can be tremendously large or even infinite. As can be seen in fig. 2.1, we use techniques both from the area of model checking and heuristic search to be able to cope with the difficulties mentioned above. In the following, we will review related work and techniques used in model checking and heuristic search. After that, in sect. 2.3, we state some observations concerning limitations of these approaches and propose several design decisions in order to realize efficient state space exploration techniques for C++ programs.

2.1 Model Checking

A model checking algorithm decides whether a program P satisfies a particular formula φ. As already mentioned in sect. 1.2, most model checking algorithms perform a kind of graph traversal on the *state graph* of the program, in which vertices correspond to program states and edges between vertices correspond to program transitions between states. An example of such an algorithm is depicted in fig. 2.2, and it can be used to decide if a program satisfies a so-called *invariant* formula φ. A program satisfies an invariant formula φ if every reachable state of the program satisfies φ. To generate all states reachable from the starting state s_0, the algorithm utilizes two sets *Open* and

```
(1)  procedure CheckInvariant
(2)  All ← ∅; Open ← ∅
(3)  Open.insert(s0)
(4)  All.insert(s0)
(5)  while (Open ≠ ∅)
(6)      u ← Open.get();
(7)      if ¬(u satisfies φ)
(8)          return Program violates φ
(9)      foreach successor v of u
(10)         if (v ∉ All)
(11)             Open.insert(v)
(12)             All.insert(v)
(13) endwhile
(14) return Program fulfills φ
(15) end procedure CheckInvariant
```

Figure 2.2: **Invariant Model Checking Algorithm.** The depicted algorithm explores the state space of a program in order to find a program state that violates φ.

All. The set *Open* contains all states that have been generated but which are not yet visited, and the set *All* contains all states that have been generated already. If a state u is extracted from *Open* that violates φ, the algorithm returns that the program violates the invariant φ. Otherwise, all successor states v of u are inserted both into *Open* and *All*. If no state has been found that violates φ and there are no more states in *Open*, the algorithm returns that the program fulfills φ. Although there exist many variants of the algorithm depicted in fig. 2.2 which differ in several aspects, e.g. if individual states or entire sets of states are processed or which logic is used to describe the formula φ, because of its simplicity almost all existing model checking tools provide an algorithm similar to the algorithm shown in fig. 2.2.

One way to classify existing model checkers is to distinguish them by means of the kind of programs they operate on. They can be roughly subdivided into classical model checkers that operate on abstract, finite state machine like input languages, and software model checkers that are aimed to check the correctness of programs in common programming languages. Figure 2.3 shows an overview of some of the current classical and software model checkers. Both classical and software model checkers share a common set of techniques that have been adapted for specific model checkers. Among the classical model checkers one can further distinguish between explicit state model check-

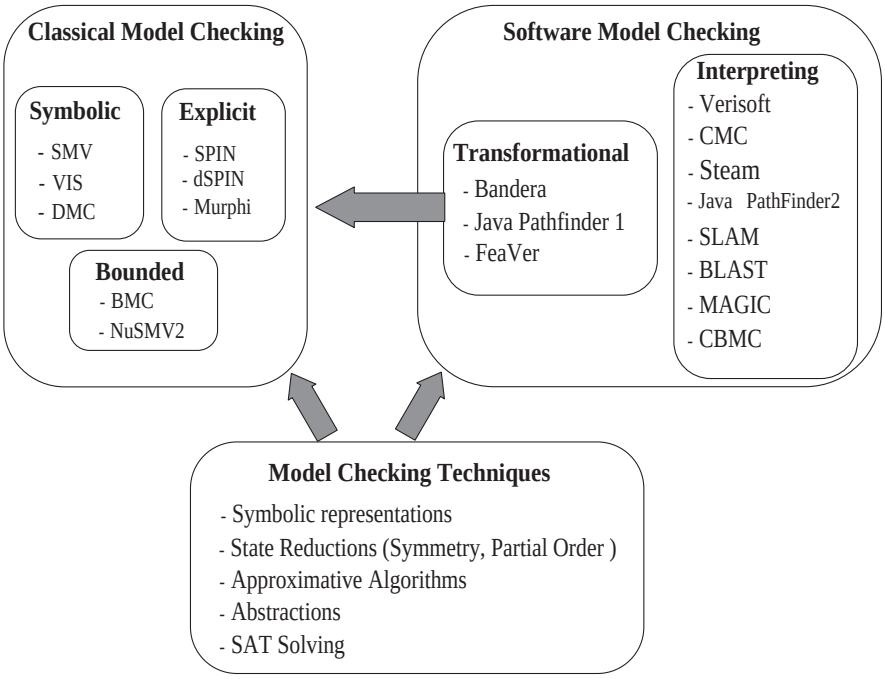


Figure 2.3: **Model Checking tools.** Model checking tools can be roughly subdivided into classical model checkers and software model checkers. They share a common set of model checking techniques.

ers like SPIN [Hol91, Hol03] or its extension dSPIN [IS99], symbolic model checkers like SMV [McM93] or VIS [Gro96], or so-called *bounded model checkers* like BMC [BCZ99] or NuSMV2 [CCG+02]. Explicit model checkers employ an individual representation of a system state. To cope with large state spaces, reduction strategies as partial order reductions, approximative state matching or abstractions are used [Hol03]. Symbolic model checkers utilize symbolic representations like BDDs [Bry86] or linear arithmetic constraints [DP01] to symbolically encode entire sets of states. Similarly to explicit state model checkers, reduction techniques like e.g. abstraction are used to widen the class of models that can effectively be handled by symbolic model checkers. Finally, bounded model checkers [GKvV95, BCZ99] encode the next-state relation of a system as a propositional formula, unroll this to some given finite depth k, and augment it with a corresponding finite unwinding of a temporal formula in order to obtain a propositional satisfiability problem (SAT for short) which is satisfiable if an error trace of length k exists. To solve the satisfiability of the generated propositional formula, modern SAT-solver tools like PROVER [Bor97], SATO [Zha97], GRASP [Sil95] or ZChaff [MMZM01] are used.

In contrast to classical model checkers that operate on an abstract, finite state machine like input languages, software model checkers are aimed to check the correctness of programs in common programming languages. They can roughly be classified into transformational and interpreting approaches. While the transformational approaches take a program in some programming language and translate it into the input language of a classical model checker, the interpreting approaches directly interpret a given program. The Bandera tool set [Ce00] translates Java Source Code into the input language of the model checkers SPIN or SMV. In order to do that, several abstractions have to be performed, e.g. data abstractions. Similarly to Bandera, the Java Pathfinder tool translates Java programs into Promela, the input language of the model checker SPIN. Just like Bandera, several abstractions are performed to be able to translate the Java program. The tool FeaVer [HS99, Hol00], which translates C programs into Promela code, follows the same approach. The Verisoft tool [God97] explores all interleavings of a concurrent C program. In contrast to the transformational model checkers, Verisoft directly executes the compiled C programs. It uses special library functions to be able to control the scheduling of the parallel components of the program. Similar to Verisoft, the model checker CMC [MPC+02, ME04] executes compiled C programs using special library functions. The Java PathFinder2 model checker [BHPV00] uses a specialized Java virtual machine to execute Java bytecode directly. The virtual machine is capable of e.g. exploring different thread interleavings. Similarly, the StEAM model checker [LME04] uses a virtual machine to execute C and C++ programs that have been compiled into so-called *ELF binaries*, which are assembly level representations of such programs. The model checkers SLAM [BR01, BMMR01, TPR01, BR02], BLAST [HJMS02, HJMS03] and MAGIC [CCG+03, CCGS03] can model check C programs, and they work all according to the same principle. Firstly, a C program is abstracted into an abstract representation. After that, a model checking algorithm is performed on the abstract representation. If the model checking procedure proves the validity of a given

property on the abstract representation, than it is also valid for the original C program. Otherwise, if a counterexample is found, it has to be checked if this counterexample is a valid one or spurious. If it is spurious, the whole process is repeated with a more precise abstraction.

2.2 Heuristic Search

Since model checking algorithms are usually aimed to exhaustively search the reachable state space of a program, they normally employ so-called *uninformed search algorithms* that do not take into account information regarding the property to be verified. Two prominent examples of such uninformed search algorithms are *breadth-first search* (BFS) and *depth-first search* (DFS). Roughly speaking, in BFS, after exploring the starting states of the system, all states which are reachable in one step are explored first, then all states reachable in two steps and so on. Contrary to this, in DFS, before the siblings of the current state and all states reachable from them are explored, all states reachable from the current state are explored first. For instance, the algorithm **CheckInvariant** depicted in fig. 2.2 proceeds like BFS if the set *Open* is organized as a queue, and like DFS if *Open* is organized as a stack.

However, when trying to verify a property that does not hold in some states of the reachable state space, the time needed to find a path to such a state, a so-called *counterexample*, is proportional to the number of states that have been explored until a property violating state has been found. Obviously, it is desirable to explore as few

```
(1)   procedure HeuristicSearch
(2)   Closed ← ∅; Open ← ∅
(3)   Open.insert(s_0, h(s_0))
(4)   while (Open ≠ ∅)
(5)       u ← Open.delmin();
(6)       Closed.insert(u)
(7)       if ¬(u satisfies φ)
(8)           return Program violates φ
(9)       foreach successor v of u
(10)          if (v ∉ Closed)
(11)              Open.insert(v, h(v))
(12)  endwhile
(13)  return Program fulfills φ
(14)  end procedure HeuristicSearch
```

Figure 2.4: **Heuristic Search Algorithm.** The algorithm utilizes a heuristic function h to obtain an ordering in which states are extracted from *Open*.

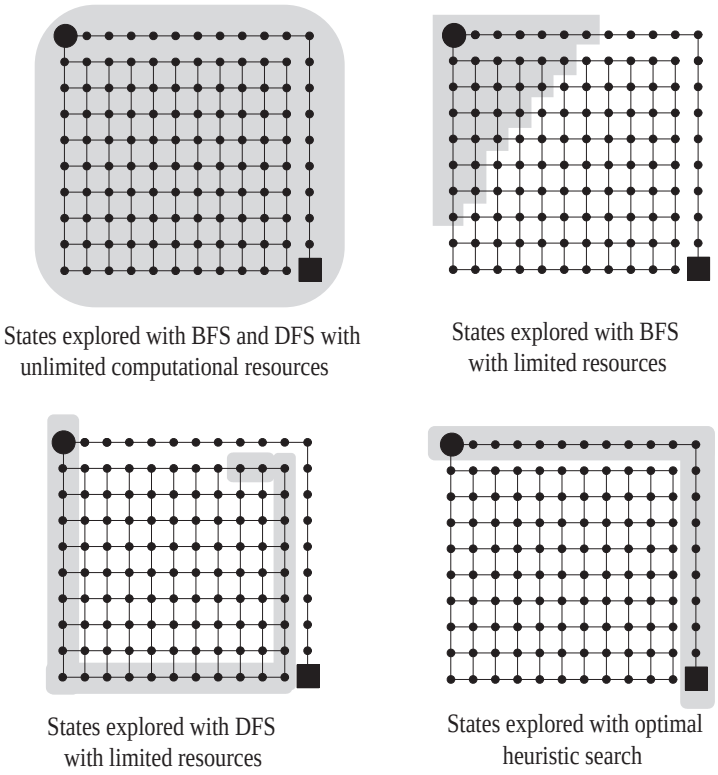


Figure 2.5: **Uninformed and informed search.** In the depicted state graph with the starting state in the top left of the grid and the goal state in the bottom right, both BFS and DFS (assuming that successor states are visited from right to left) have to explore the entire state graph before reaching the goal state if unlimited resources are available (top left). Both BFS (top right) and DFS (bottom left) fail to find the goal state with limited resources. Contrary to this, an optimal heuristic search (bottom right) would only explore a fraction of the state space and would find the goal state even with limited resources.

states as possible before finding a counterexample, since this allows a fast detection of counterexamples. Furthermore, the complete state space often cannot be explored with reasonable computational resources, i.e., it is only possible to explore a fraction of the entire state space. If counterexamples exist in the state space, then it is crucial to explore a part of the state space that contains at least one counterexample. To explore the right part of the state space is the purpose of heuristic search [Pea85]. In contrast to uninformed search algorithms, *informed* or *heuristic* search algorithms take into account information about the search goal to direct the search into regions of the state space where it is likely to find a state that fulfills the search goal. The information is obtained by applying a so-called *heuristic function* h to each visited state. Given

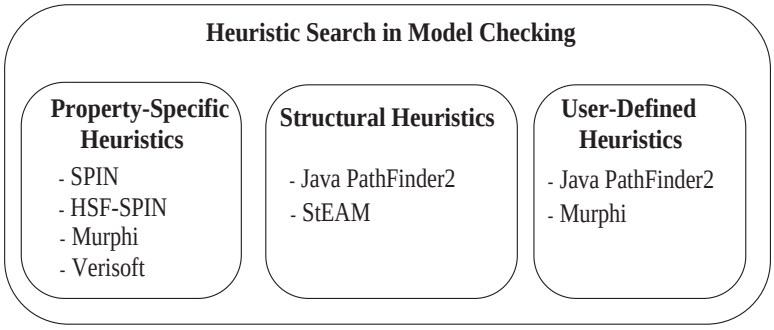


Figure 2.6: **Heuristic Search in Model Checking.** The approaches using heuristic search in model checking can be classified by the type of employed heuristics. Property-specific heuristics exploit information about the property to guide the search. Structural heuristics favor states that cover previously uncovered branches of the program, and user-defined heuristics are hints provided by the user.

a state s, the value $h(s)$ estimates the distance from s to a state fulfilling the search goal. An example of a heuristic search algorithm can be seen in fig. 2.4. In contrast to the algorithm depicted in fig. 2.2, the algorithm shown in fig. 2.4 sorts states s with increasing values $h(s)$ in the set *Open*, i.e., a state s is processed before another state s' if $h(s) < h(s')$. Since states are sorted purely by means of the heuristic function h, the algorithm from fig. 2.4 is called *best first search*. If h provides a good estimation of the real distance to a goal state, heuristic search is able to find a path even in state spaces where uninformed search algorithms fail, or it finds paths much faster. Figure 2.5 shows the effect of heuristic search compared to BFS and DFS. When applied to the depicted state graph with the starting state in the top left and the goal state in the bottom right, both BFS and DFS[1] have to explore the entire state space before finding the goal state. Additionally, if we assume that the available resources are limited and suffice only to explore a limited number of states (30 in fig. 2.5), then both BFS and DFS fail to find the goal state. Contrary to this, performing a heuristic search with an optimal heuristic function would only explore a small fraction of the complete state space, thus allowing to find the goal state quickly even with limited resources. However, crucial for the effectiveness of heuristic search algorithms is an informative heuristic function.

While heuristic search is a common technique in e.g. *artificial intelligence*, it has only been used for classical or software model checking recently. Figure 2.6 shows some of the model checkers that have been used to perform some kind of heuristic search. They can be classified based on the information they try to exploit. Property specific heuristics analyze the error description as the negation of the correctness property. In some cases the underlying methods are only applicable to special kinds of errors, e.g. dead-

[1] We assume that DFS always expands successor states from right to left.

lock detection. In contrast to property specific heuristics, structural heuristics exploit information about already covered resp. uncovered branches of the program, similar to code coverage metrics often used in software testing. The third class of heuristics is formed by user-defined heuristics. User-defined heuristics can be specified e.g. by means of source annotations or necessary conditions that must be met in order to reach the search goal. In [ELL01b, ELL01a] the well known explicit state model checker SPIN has been extended to allow the use of property specific heuristics for heuristic search. With this extension, called HSF-SPIN, it was able to find counterexamples in cases where both BFS and DFS failed. In this work, heuristics evaluating boolean combinations of expressions are used to guide the search towards states fulfilling the compound expressions. Additionally, specific heuristics like maximizing the number of process interleavings tailored for e.g. finding deadlocks are used. Similarly to HSF-SPIN, in [LME04] the model checker StEAM performs heuristic search for deadlocks using deadlock-specific heuristics like e.g. maximizing the number of thread interleavings. Similarly, in [LCL88] heuristics that e.g. maximize the length of message queues are used to guide the search towards potential error states. In [YD98] the model checker Murphi has been extended with heuristic search capabilities, enabling both the application of property specific and user-defined heuristics. For instance, the user can give hints in form of boolean conditions that must be fulfilled in order to reach a certain goal state. In [GK02], genetic algorithms are used to extend the model checker Verisoft with heuristic search capabilities, using a fitness function that favors states with a high number of thread interleavings, trying to guide the search into deadlock states. The model checker Java PathFinder2 is augmented with heuristic search algorithms employing structural heuristics in [GV02b, GV02a]. In this work, structural coverage criteria like branch coverage [Bei90] are used to guide the exploration process. Another approach using structural heuristics is presented in [EM03] where the distance of specific byte code positions of compiled Java programs guide the state space exploration. This approach has also been built on top of the model checker Java PathFinder2.

2.3 Observations and Design Decisions

Summarizing the mentioned related approaches to software model checking one can make the following observations:

- Although the ultimate goal of software model checking is the complete verification of correctness properties of programs, the ability to generate counterexamples often make software model checking approaches attractive for users. In many cases, software model checkers are employed as sophisticated debugging and testing tools allowing to find bugs or to generate particular program traces that can be reused e.g. as test sequences.

- While symbolic model checkers have clear advantages compared to explicit state model checkers when model checking synchronous hardware designs, in the area

of software model checking the situation is more ambiguous. On the one hand, several constructs found in programming language are difficult or impossible to express symbolically. For instance, a BDD based model checker can only represent a finite number of objects. When a program uses dynamic memory allocation it is possible that more than this maximal number of objects are created dynamically. The same holds for constructs like pointer arithmetic or recursive functions, too. Contrary to this, all these constructs can adequately be represented in explicit state model checkers, provided that they support a varying length state representation as e.g. dSpin [IS99] or JavaPathFinder2 [BHPV00]. Additionally, the interleaved execution model of multithreaded programs can easily be represented in explicit state model checkers with an interleaving semantics as e.g. in SPIN [Hol91, Hol03].

- When applied to very large programs, often neither symbolic nor explicit state model checkers are capable of performing a complete state space exploration. However, while it is often easily possible to perform a partial state space exploration using an explicit state model checker, it is difficult or impossible to do the same with a symbolic model checker. The reason for this lies in the fact that symbolic model checkers usually compute symbolic representations for the set of *all* states that are reachable from the predecessor set of states, i.e., they proceed in a breadth-first manner. However, when the symbolic representations become larger than a certain threshold, these image computations cannot be performed within reasonable resources. The threshold varies with the applied symbolic representation. For instance, bounded model checking approaches [GKvV95, BCZ99] using SAT solvers outperforms BDD based model checkers when applied to large models as several comparisons [BCRZ99, Sht89] indicate. The reason for this lies in the fact that SAT solvers can cope with propositional formulas of a size for which equivalent BDD representations are too large to be handled by current BDD tools.

- A crucial drawback of explicit state model checkers is simply the fact that each state is represented explicitly, which implicates that at most a few million states can be explored. This is particularly a problem when model checking embedded software, since normally program variables can take a large range of values, especially when the program reads several values from the environment. For instance, consider a program with just two integer inputs represented using 32 bit. Since the variables serve as inputs, they can take arbitrary values, which means that after selecting input values already 2^{64} distinct states are reachable, a number which cannot be handled by explicit state model checkers like e.g. verisoft, CMC, StEAM or JavaPathFinder.

- Software model checkers like Bandera, PathFinder1 or Feaver that translate a program into the input language of a classical model checker often suffer from the problem that only a restricted subset of programming constructs can be translated. Additionally, in many cases several abstractions are necessary when translating real programs. For instance, dynamic object creation or pointer arithmetic are often

abstracted. These abstractions need considerable human interaction, which means that there is no guarantee that the applied abstractions are sound.

- The software model checkers that work according to the abstraction-refinement scheme like SLAM, BLAST or MAGIC have similar problems as the translating model checkers. Firstly, in many cases some constructs of the program cannot be abstracted adequately. For instance, BLAST and MAGIC cannot handle recursive functions, and SLAM cannot handle concurrent programs. Furthermore, several constructs like e.g. pointer arithmetic are abstracted into uninterpreted functions, i.e., if the correctness of a property depends on such constructs, than this cannot be verified by the model checker. Besides these problems, a major disadvantage of all model checkers relying solely on abstraction-refinement is that they are often only effective if they can prove a property to be correct. If a property does not hold for a program, then they often fail to produce a real counterexample because the applied conservative abstraction often generates spurious counterexamples. Ruling out all spurious transitions in a spurious counterexample often needs many iterations and many new predicates with the effect that the abstraction becomes too large for the model checker.

- Heuristic search can effectively reduce the number of explored states resp. the time needed to find a state with a given property. However, crucial for the effectiveness of heuristic search is an informative heuristic function that provides a good approximation of the real distance to a goal state. However, while the heuristics used in model checkers like e.g. HSF-SPIN or StEAM are effective for certain kinds of error states like deadlocks, they are not effective for more general properties, which need other kinds of heuristics.

Based on these observations, we make some design decisions concerning our approach to state space exploration of C++ programs:

- We concentrate on incomplete state space exploration. Firstly, reasonable programs are as large s.t. it is very unlikely that a complete state space exploration is possible. Furthermore, as mentioned above, software model checkers are often employed as sophisticated debugging and testing tools allowing to find bugs or to generate particular program traces. For this application incomplete state space exploration algorithms are advantageous compared to complete algorithms because they allow the application of approximating algorithms which are faster and more scalable than exact algorithms, i.e., incomplete state space exploration algorithms can explore larger state spaces in shorter time.

- We try to combine the benefits of symbolic and explicit state model checkers by employing a composite explicit-symbolic state representation in our state space exploration tool set. This composite representation enables on the one hand a compact symbolic treatment of large sets of states and on the other hand an

explicit treatment of programming constructs that are difficult to handle symbolically. Furthermore, this representation allows us to apply several approximation algorithms that make state space exploration more efficient.

- We try to combine the benefits of abstraction and heuristic search by defining an abstraction-based heuristic search procedure. Based on a program P and a property, we generate an abstraction of P for which a complete state space exploration is possible. We use the generated abstract state space as a heuristic to guide the search in the concrete state space towards states that fulfill the given property. Since the abstraction-based heuristic takes both the program and the property into account it can give accurate hints even in cases where other heuristics are not informative.

The design decisions mentioned above form the basis of the state space exploration techniques we present in this thesis. However, before we introduce the programming language SymC++ in chapter 3, the next section briefly summarizes other work dealing with validation and verification of software that does not belong to the area of model checking.

2.4 Other Related Work

Besides software model checking, other approaches related to validation and verification of software can roughly be classified into the following categories:

- Manual and Semi-Automated Program Verification: In these approaches, both the program and the properties to be checked are described by means of formulas of a particular logic. Based on the logic's underlying axioms and proof rules one tries to deductively verify the given properties w.r.t the program. As already mentioned in the introduction, early approaches in this area stem from Floyd [Flo67] and Hoare [Hoa69] and have later been extended for various classes of programs, e.g. in [OG76a, OG76b, AFdR80, LG81]. An overview can be found in [AO94, AO97]. In contrast to the model checking approach, manual or semi-automatic approaches do not suffer from the state explosion problem, i.e., they can in principle be used to verify programs of arbitrary size. However, in practice it is seldom possible to verify even moderate sized programs because of the *proof explosion problem*. To verify such programs, thousands or millions of proofs have to be carried out, a task which cannot be done manually. Even when automated theorem provers are used that can realize some proofs automatically as e.g. in [MS95], the decisive proof steps have to be performed manually. Another problem stems from the fact that in case a program cannot proved to be correct usually no counterexample is generated that can provide hints regarding possible errors in the program.

- Static Analysis: Approaches based on static analysis try to deduce information about the behavior of a program without actually executing it. Static analyses are

often based on abstract interpretation [CC77, CC92, NNH99], and are mostly used to find common program errors or coding mistakes. There are various approaches and tools for different languages, among others LCLint [EGHT94, Eva96], a static checker for the programming language C that can be used to detect e.g. unused variables or common coding errors, or JLint [AB01], the Java variant of LCLint. In contrast to software model checking approaches which usually allow checking a large class of properties, static analysis tries to detect specific errors which are often found in programs of a particular programming language. Furthermore, in case an error is detected they usually do not provide an execution trace leading to that error.

- Design by Contract and Runtime Verification: *Design by Contract* [Mey92, Mey97] describes a programming methodology that is based on precise checkable interface specifications for software components. These specifications include pre- and post-conditions of methods or class invariants. Approaches based on Design by Contract differ in the considered programming language and in the way the specifications are used. For instance, in the programming language Eiffel [Mey92] an exception is thrown during runtime if a contract is violated which represents a kind of runtime verification. Similarly, the *Java Modeling Language* (JML) [LBR98, LBR99] realizes design by contract for Java. Several tools utilize JML specifications in different ways. For instance, the runtime assertion checker jmlc [CL02] checks assertions at runtime and reports violations. Another tool is the program checker ESC/Java [FLL$^+$02] which performs an extended form of static analysis. For instance, ESC/Java can statically detect violations of invariants annotated in JML syntax. The rich specifications used in approaches based on design by contract like e.g. JML provide a good way of formally specifying the intended behavior of programs that could also be used by software model checkers or state space exploration tools. However, in our approach we concentrate on very simple specifications since the main aim of the thesis is the development of an efficient state space exploration algorithm. The extension of this algorithm towards richer specifications as provided by e.g. JML is subject of future work and discussed in sect. 6.2.

3 SymC++ and C_{min}

In this chapter, we will give an overview of the programming language SymC++, a variant of the well-known programming language C++ [Str00, ISO03] that is widely used for implementing embedded programs, and the language C_{min} that serves as a formal model of SymC++ programs. SymC++ extends C++ with the two concepts of concurrency and nondeterminism which are needed for the implementation of embedded programs. However, while SymC++ with all its features like inheritance etc. offers powerful and elegant ways for realizing also complex programs, it is far too complicated to explain the concepts and methods we develop for efficient state space explorations of such programs. Therefore, in sect. 3.2 we introduce a much simpler language C_{min} that we will use throughout the thesis as an underlying formal model for explaining the developed concepts and methods. Although C_{min} is much simpler than SymC++ it is powerful enough to adequately capture the semantics of the language constructs available in SymC++. However, since SymC++ programs are much more concise and readable than C_{min} program, we will use SymC++ for describing sample programs.

In the following, we first give a brief overview about the language SymC++ s.t. the reader will be able to understand the sample programs we use. After that, in sect. 3.2 we introduce the language C_{min} that will be used as a formal model for SymC++ programs, and in sect. 3.3 we describe a simple logic TL that can be used to formulate properties about C_{min} programs. In sect. 3.4 we explain how SymC++ programs can be translated into C_{min} programs. Finally, as concrete examples of embedded programs, in sect. 3.5 we consider programs that have been generated from a widely used UML case tool named Rhapsody [HG97, GHP02]. Since these programs make use of many features of C++, we will use these programs for experimental evaluations of the methods we develop in the chapters 4 and 5.

3.1 Overview of SymC++

SymC++ is a variant of the well-known programming language C++. As already mentioned, C++ supports numerous features and programming constructs, e.g.

- Object-oriented programming constructs like classes, objects, methods, inheritance and polymorphism.

- Imperative language constructs like functions or different kinds of loops.

- Language constructs for generic programming like template classes and template functions.
- Language constructs for exception handling.

As one can see, C++ is a very manifold language and accordingly rather complicated, which is also made clear by the fact that the official ISO standard for C++ [ISO03] contains 757 pages. Therefore, we cannot give here a complete description of C++ resp. SymC++. Contrary to this, in the following we will give a brief review of the central language constructs of C++, and after that we will describe the extensions and limitations of SymC++ in sect. 3.1.2. Later, in sect. 3.4, we will sketch how we can translate SymC++ programs into the simple language C_{min} introduced in sect. 3.2, that we will use throughout this thesis as a formal model to explain the state space exploration techniques we develop in the chapters 4 and 5.

3.1.1 C++

A C++ program consists of one or more class definitions, variable definitions and function definitions. C++ defines a set of predefined types as e.g. `char`, `int` or `float`. Using these types one can define variables like `int i`, or other types, e.g. pointer types like `int*` or `int**`. Furthermore, one can define *class types* that are simply called *classes*. A class is a collection of variable definitions and function definitions. For instance, consider the following small program:

```
class C {
  int i;
  C(int k) { i = k; }
  int f() { return i; }
};

void main() {
  C* c1 = new C(1);
  C* c2 = new C(2);
  int i1 = c1->f();
  int i2 = c2->f();
}
```

This program defines a class `C` that has a so-called *member variable* `i` of type `int`. Furthermore, `C` defines two functions `C()` and `f()`. The function `C()` is called a *constructor* and is invoked whenever an object of class `C` is created, as it is the case in the function `main` by using the `new` operator that creates new objects dynamically. Each object of a class has its own member variables, i.e., in the program above two objects are created that both have a member variable `i`. The function `f()` is called a *member function* of `C` since it can only be invoked on a specific object of class `C`. For instance, in the program above there are two invocations of `f()`, one for the object `c1` points at and one for the

object c2 points at. Within a member function of a class one can access all member variables of the object on which the function has been called. Classes can inherit variable and function definitions from other classes, as in the following program:

```
class A { int a; A() {a=0;}; virtual int f() { return a; } };

class B : A
{ int b; B() {b=1;}; virtual int f() { return b;} };

class C : A
{ int c; C() {c=2;}; virtual int f() { return c;} };

void main() {
  int i;
  B* b = new B(); C* c = new C;
  A* a = (A*)b;
  i = a->f();
  a = (A*)c;
  i = a->f();
}
```

This program defines a class A with a so-called *virtual* member function f(). Furthermore, two classes B and C are defined that both inherited from A, i.e., both B and C also contain all member variables that are defined in A. The classes B and C are called *subclasses* of A, and A is called a *superclass* of C. Since an object of a subclass inherits all member variables and functions of its superclasses, whenever a certain language construct expects an object of the superclass, then always an object of any of its subclasses can be used instead, a mechanism which is called *polymorphism*. For instance, in the main function of the program above an object of class B and an object of class C are created. After that, a variable a is defined as a pointer to the superclass A, and the pointer to the B-object b is assigned to a. Now, the following call a->f() of the virtual member function f() results in the call of the function B::f(), since the variable a points at a B-object. After that, the pointer to a C-object c is assigned to a, and the following call of a->f() results in the call of C::f(), since the variable a now points at a C-object. In contrast to e.g. Java, in C++ it is allowed that a class can have several superclasses:

```
class A { int a; A() {a=0;}; int f() { return a; } };

class B { int b; B() {b=1;}; int g() { return b; } };

class C : A,B { int c; C() {c=2;}; int h() { return c; } };

void main() { C* c = new C(); int i = c->f()+c->g()+c->h(); }
```

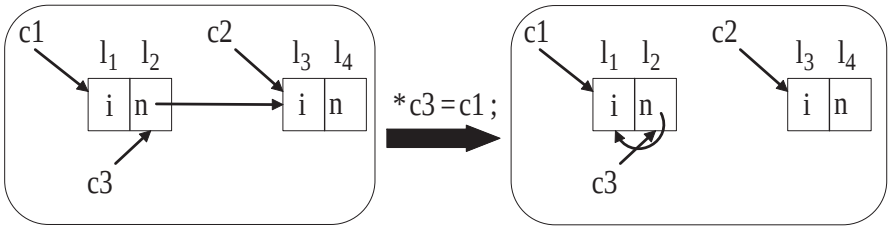


Figure 3.1: **Pointers in C++.** When executing the statement `*c3=c1` in the left configuration, in the successor configuration on the right the location l_2 points at l_1.

In this program, the class `C` inherits both from `A` and `B`, i.e., `C` contains all member variables and member functions of `A` and `B`, and thus a `C`-object can be used whenever a `A`-object or `B`-object is expected.

Besides object-oriented language constructs, C++ programs often make use of pointers and dynamic object creation and destruction. The memory is an ordered sequence of memory locations $l_0, l_1, \cdots$, and each variable used in program has a certain *address*, i.e., each variable `v` is located at a particular address l_i. If `v` is a variable, then `&v` denotes the address where `v` is stored in memory. Each pointer variable can be *dereferenced* with the `*`-operator, i.e., if p points at an object stored at l_i, then `*p` yields the value stored at l_i. Pointer variables can be used to indirectly change the values of other variables or to dynamically create new objects if necessary. For instance, consider the following program:

```
class C { int i; C* n; C() {i=0; n=0;} };

void main() {
  C* c1 = new C(); C* c2 = new C();
  c1->n = c2; C3** c3 = &(c1->n);
  *c3 = c1;
}
```

This program defines a class `C` which has two member variables `i` and `n`, whereas `i` denotes an integer variable and `n` denotes a pointer variable of type `C*`. In the `main` function, two objects of class `C` are created, and pointer to these objects are assigned to the variables `c1` resp. `c2`. Furthermore, the member variable `n` of the object to which `c1` points at is assigned the value of `c2`. Additionally, the pointer variable `c3` is assigned the address of the variable `n` of the object `c1` points at. After executing the statements of the first two lines of the `main` function, a memory configuration as depicted in the left of fig. 3.1 has been created. The variable `c1` points at l_1, the beginning of the first `C`-object, and the variable `c2` points at l_3, the beginning of the second `C`-object. Furthermore, the variable `c3` points at l_2, the address of the member variable `n` of the first `C`-object. Now, if we execute the statement `*c3=c1`, then we change the value of the location l_2 to l_1,

since `c3` points at l_2. As a result, the memory configuration shown in the right of fig. 3.1 has been reached. In C++, pointers are closely related to arrays. The reason for this lies in the fact that array elements are stored in consecutive memory locations. For instance, consider the following program:

```
int a[100];
int sum = 0;

void main() {
 for (int j=0; j < 100; j++) {
   a[j] = j;
   sum = sum + a[j];
 }
}
```

In this program, an array `a` of 100 integer variables is declared. The first element of this array is the element `a[0]` stored at location l_i, and the last element of this array, `a[99]`, is stored at the location l_{i+99}. With the index operator `[]` one can access every element of the array. In C++, there is no range check for arrays, i.e., one can also write `a[102]` or `a[-5]`. The reason for this lies in the fact that C++ treats arrays similar as pointers. To evaluate an expression like `a[i]`, C++ performs the following steps:

- Get the address of the first element of the array `a`, i.e., compute the address $l_k =$`&a[0]`.
- Compute the value of `i`, i.e., $x =$`i`.
- Compute the address of the location where the element `a[i]` is stored by computing l_{k+x}, i.e., add x to the address l_k.
- Get the value at location l_{k+x}.

Against this background, an alternative formulation of the above program could be as follows:

```
int a[100];
int sum = 0;

void main() {
  int* p = &a[0];
  for (int j=0; j < 100; j++) {
    sum = sum + *p;
    p = p + 1;
  }
}
```

In this program, we use a pointer variable `p` to access the individual elements of the array `a`. The direct manipulation of `p` in the expression `p = p + 1` is called *pointer arithmetic* and frequently used in C++. If `p` is a pointer of type `t*` and points at an element `a[i]` of an array of type `t`, i.e., `p=&a[i]`, then `p+x` points at `a[i+x]`. The use of pointers and pointer arithmetic allows very efficient implementations, but it is also very error-prone, since if `a[i+x]` exceeds the array bounds, then the program might crash or other variables are accidentally changed. Besides the addition of a pointer and an integer value, a second pointer arithmetic operation is the *difference* of two pointers `p2-p1`. The difference `p2-p1` of two pointers `p1=&a[i]` and `p2=&a[i+k]` yields the integer value `k`. As already mentioned, because of the complexity of C++ we can explain here only the C++ constructs that are needed to understand the sample programs we use in this thesis. For a complete description of C++ the reader is referred to [Str00, ISO03].

3.1.2 SymC++ Extensions and Limitations

Although C++ supports many language constructs, it lacks of language constructs for the explicit treatment of concurrency. For implementing concurrent programs with C++, one often uses specific low-level routines of the underlying operating system. Therefore, to provide a uniform set of functions dealing with concurrency, SymC++ extends C++ with a thread model similar to the thread model used in Java [GJSB00]. Technically, we define a new base class `Thread` as follows:

```
class Thread {
  Thread();
  void Start();
  virtual void Run();
  void await(int* p);
};
```

The class `Thread` represents the base class for all threads in a program. If one wants to create a new thread one first has to define a new subclass which inherits from `Thread`. In the subclass, one has to implement the virtual function `Run` that represents the main loop of the thread. After defining a new subclass of `Thread`, one can define arbitrary many objects of this class. However, all thread objects must be declared statically, i.e., it is not allowed to create new threads during runtime. In the beginning of the program, an internal thread is started that executes the global `main` function. Within the `main` function, other threads can be started by calling the `Start` member function of the corresponding thread objects. For synchronization, the function `await(int* p)` can be used. The functionality of this function depends on the value `*p`, i.e., the value of the variable `p` points at. If `*p>0`, then the function assigns 0 to the variable `p` points at, i.e., it executes the assignment `*p=0`, and returns. Otherwise, the calling thread is blocked until eventually the condition `*p>0` evaluates to true. All the actions during the invocation of the `await` function cannot be interrupted by other threads, i.e., a call to the `await` function is atomic.

Besides the support for concurrency, SymC++ also provides explicit support for nondeterminism by defining a global function `symcpp_nondet(int a, int b)`. If this function is called, it returns an arbitrary value in the interval `[a,b]` if `a<=b` holds, otherwise it returns 0. To demonstrate the use of threads and nondeterminism, we realize a small program that consists of two threads, a sensor thread and an actuator thread. The sensor thread reads input values from a sensor and stores them in a buffer variable. The actuator thread doubles the stored value and writes it to an output.

```
int sem; int buf; int out;

class Sensor : Thread {
  virtual void Run() {
    while (1) {
      await(&sem);
      buf = symcpp_nondet(0,1000);
      sem = 1;
    }
  }
};

class Actuator : Thread {
  virtual void Run() {
    while (1) {
      await(&sem);
      out = 2 * buf;
      sem = 1;
    }
  };
};

Sensor    sensor;
Actuator  actuator;

void main() {
 sem = 1;
 sensor.Start();
 acuator.Start();
}
```

In this program, two classes `Sensor` and `Actuator` are defined that both inherit from the base class `Thread`, i.e., objects of these classes represent separate threads. In the main loop of the `Run()` function of the class `Sensor`, with the call `await(&sem)` it is guaranteed that the other thread cannot access the shared variable `buf` when the sensor thread reads a new value into `buf`, i.e., the variable `sem` acts like a semaphore that

guarantees the mutual exclusive access to `buf`. To model the reading of a sensor value, we nondeterministically select a value from the interval `[0,1000]` by calling the function `symcpp_nondet(0,1000)`, and store this value in the variable `buf`. Finally, we assign to the variable `sem` the value 1, which allows the other thread to read the variable `buf` again. The `Run()` function of the class `Actuator` works similarly, with the difference that it writes a new value to the variable `out`. In the global `main` function, at first the semaphore variable `sem` is assigned the value 1. After that, both threads are started, i.e., they begin to execute their nonterminating `Run()` methods.

Besides the described extensions, SymC++ currently has several restrictions of the supported language features of C++. The most important restrictions are the following:

- In contrast to C++, which supports several signed and unsigned integer types like `short`, `unsigned short` or `long`, in SymC++ all these types are currently interpreted as signed integers. Furthermore, no floating point types are supported.
- In C++ it is possible to allocate, access and deallocate memory on a byte-level. For instance, in C++ it is allowed to allocate an array of bytes and use this memory to store an object of another type if the size of this object is not larger than the size of the array. Such a low-level memory handling is not supported in SymC++. Contrary to this, in SymC++ new objects can only be created using the typed `new` operator.
- Exception handling and function pointers are not supported.

As already mentioned, we do not want to give a complete description of all language features of C++ resp. SymC++ here but rather a brief overview s.t. the reader is able to understand the sample programs we use to illustrate some of the difficulties that arise in the context of state space exploration of such programs. Appendix A contains a complete description of the C++ constructs that are currently unsupported in SymC++. In the next section, we will introduce the simple language C_{min} that serves as a formal model for SymC++ programs.

3.2 C_{min}

While a programming language like SymC++ with all its features like inheritance etc. offers powerful and elegant ways for realizing also complex programs, it is far too complicated to be handled formally directly. Therefore, we introduce an intermediate language which is on the one hand expressive enough to capture the semantics of SymC++ but on the other hand simple enough to be handled more formally. We will introduce the language C_{min} that serves as such an intermediate language. The following example should give a short overview about the language and its constructs.

Example

```
struct Sensor {
  int value;
}

struct Controller {
  int state;
}

 Controller* c1 = 0;
 Sensor* s1 = 0;
 Sensor* s2 = 0;
 int started = 0;
 int polled = 0;
 int* p = 0;

thread(1) {
  1: new(c1) 2: new(s1) 3: new(s2) 4: started := 1; p := &polled;
  5: await(poll = 0,poll := 1) 6: await(poll = 0,skip)
  7: c1->state := s1->value+s2->value > 10 ? 1 : 0
  8: jump(true,5,8);
  }
thread(2) {
  9: await(started = 1, skip) 10: await(poll = 1, skip)
  11:[s1->value:=nd,s2->value:=nd]
  12: *p := 0; 13: jump(true,10,13)
 }
```

This sample program is very short and simple, but nevertheless it can be used to explain some of the features of C_{min} programs. Firstly, a C_{min} program contains two parts, a declaration part and an action part. The declaration part can contain arbitrary many *structure declarations* and *variable declarations*. A structure declaration describes a new type with a new name together with the types and names of the components of the structure. For instance, in our sample program there are two structure declarations `Sensor` and `Controller`. A variable declaration introduces a new variable with a certain type. As in C++, besides simple types like integer or boolean, a C_{min} program can also have so-called *pointer types*. In contrast to variables of simple type like integer that can hold integer values, variables of pointer types hold *addresses* or *locations*. Figure 3.2 shows the connection between variables and locations. The memory of a program is an infinite sequence of locations $\{l_1, l_2, \cdots\}$, and each declared variable has an associated fixed location. For instance, fig. 3.2 shows that the variable `started` is located at location l_1 and the variable `s1` is located at location l_4. The initial value of a variable of type `int` or `bool` can be any value in the domain of that type and is provided by an explicit

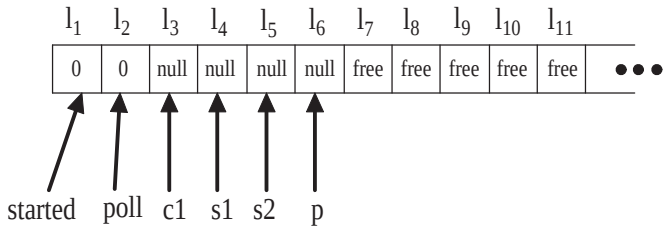


Figure 3.2: **Starting memory configuration.** Each static variable of a C_{min} program is mapped to a particular location. The initial values stored at these locations are determined by explicit initial values for integer and boolean variables. Variables of pointer types always have the initial value *null* (or 0).

initialization when the variable is declared, e.g. `int started=0;`. The initial value of variables of pointer type is always *null* (or 0 which is just a shortcut for *null* when used for pointer types) representing the fact that a pointer variable is currently not pointing at a valid location, therefore the values at location l_3, l_4, l_5 and l_6 are *null*.

The value of a pointer variable will change when a *new*-statement as `new(c1)` in the sample program is executed. The effect of a new statement is that formerly $free$ locations are *allocated* to represent a new object of the type of the given variable, i.e., for each member variable of the allocated type one location is used to store values for the associated member variable. Furthermore, the value of the pointer variable used in the new statement changes to the first allocated location. Figure 3.3 shows the memory configuration after execution of the statements $1-4$ of thread 1. Besides the allocation of new objects, the value of a pointer variable can also change by simply assigning other values to it. For instance, by using the *address*-operator `&` the statement `p:=&polled` stores the location of the variable `polled` in p, and a statement like `s1:=s2` would change `s1` to point at l_9 instead of l_8. By using the *dereference*-operator `*` as in the statement `*p:=0` of thread 2, a new value can be stored in the location the dereferenced variable points at. For pointers with structure type like `c1`, the selection operator `->` can be used to access individual member variables of that structure. For instance, statement 7 of thread 1 accesses the member variables `c1->state`, `s1->value` and `s2->value`. Simple (non-pointer) variables are handled as usual, i.e., an assignment to a simple variable like `started:=1` changes the value stored in the location (l_1 in fig. 3.3) associated with this variable.

To adequately represent concurrency, a C_{min} program allows the definition of arbitrary many *threads* that are distinguished by their associated number. Our sample program defines two threads , thread 1 and thread 2. Thread 1 firstly creates a `Controller` object and two `Sensor` objects. After that, in a never-ending loop it polls the two sensors, and dependent on the values of the sensors it switches the `state` member variable of the controller object. When two or more threads access common variables it is important to synchronize these accesses. In a C_{min} program, an `await` statement can be used for

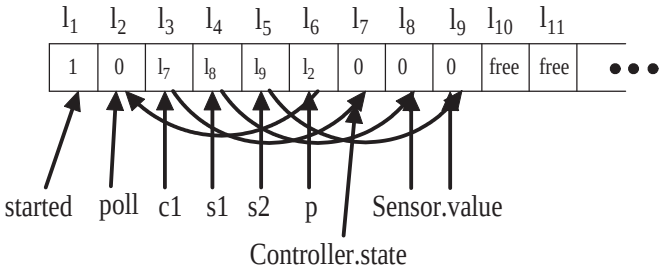


Figure 3.3: **Memory configuration after initialization**. After executing the statements $1-4$ of thread 1, 3 new objects have been created and allocated at the locations l_7, l_8, and l_9, and the values stored at the locations l_3, l_4, l_5 and l_6 have been changed.

that purpose. A statement like `await(poll=0,poll:=1)` can only be executed when the guard `poll=0` evaluates to true, therefore a thread is blocked at that statement as long as the guard is not true. When the guard is true, the statement can be executed, i.e., the assignment `poll:=1` will be executed. With this mechanism it is possible to synchronize accesses to shared variables from different threads.

Another aspect of reactive programs, the reading of input values from the environment, can be expressed in a C_{min} program by using nondeterministic expressions. The meaning of an expression `nd` is the nondeterministic choice of one value from the domain `int`, i.e., after execution of a statement like `s1->value:=nd` the variable `s1->value` can have an arbitrary integer value. Besides assignment-, new- and await-statements there is also the `jump`-statement. In a C_{min} program, each statement is labeled by a unique number. The meaning of a jump statement like `jump(true,5,8)` is that depending on the condition in the jump statement (`true` in this case) the control moves either to statement 5 or to statement 8. With jump statements, all kinds of control-flow constructs such as if-then-else, while loops or switch-case statements can be realized.

After this short introduction into the programming language C_{min} and its constructs by means of an example, in the following we will introduce C_{min} programs formally.

Definition of C_{min}

A C_{min} program P is a tuple

$$P = (Structs(P), Vars(P), init(P), Threads(P)),$$

with the following components:

- $Structs(P) = \{C_1, \cdots, C_k\}$ is a set of so-called *structures*. Each structure C_i defines a sequence of so-called *member variables* $MemberVars(C_i) = \langle m_1, \cdots, m_l \rangle$. The set of all member variables of all structures is denoted with $MemberVars$.

- $Vars(P) = \{v_1, \cdots, v_m\}$ is a set of variables.
- $init(P) = Vars(P) \to D$ is an *initialization function* that assigns to each variable $v \in Vars(P)$ an initial value $init(v)$ from the domain D. The concrete set D will be defined later when we define the semantics of expressions.
- $Threads(P) = \{T_1, \cdots, T_n\}$ is a set of *threads*. The detailed structure of threads will be explained later.

Based on a given C_{min} program P we define the set of types induced by the program. We distinguish between the basic types *int* and *bool*, the set $Structs(P)$ of structures and the set $PTypes$ of so-called *pointer types*, which are inductively defined as follows:

- $int* \in PTypes$ and $bool* \in PTypes$.
- if $C \in Structs(P)$ then $C* \in PTypes$.
- if $t \in PTypes$ then $t* \in PTypes$.

The set of all types is defined as $Types = \{int, bool\} \cup Structs(P) \cup PTypes$. Furthermore, the set $CTypes = \{C* \mid C \in Structs(P)\} \subset PTypes$ denotes the set of all structure pointer types, and we assume a function

$$type : Vars(P) \cup MemberVars \to Types$$

that defines the type $type(v)$ for every variable $v \in Vars(P) \cup MemberVars$. As a restriction, we assume that $type(v) \notin Structs(P)$ for every $v \in Vars$, i.e., it is only allowed to declare variables of structure pointer types, but not structured variables directly. To distinguish between variables that are pointer to structures and other variables we define the set $CVars = \{v \in Vars(P) \mid type(v) \in CTypes\}$ of pointer variables with a structure pointer type. Furthermore, we define the sets

$$IVars = \{v \in Vars(P) \mid type(v) = int\} \text{ and } BVars = \{v \in Vars(P) \mid type(v) = bool\}$$

of all integer- resp. boolean variables, and we define $BIVars = BVars \cup IVars$. For the definition of the semantics we need the two functions

$$\begin{aligned} size &: Types \to \mathbb{N} \\ size(t) &= \begin{cases} 1 & \text{if } t \in Types \setminus Structs(P) \\ \sum_{i=1}^{n} size(type(v_i)) & \text{if } t \equiv C \in Structs(P) \text{ and } C = \langle v_1, \cdots, v_n \rangle \end{cases} \end{aligned}$$

and

$$\begin{aligned} off &: Types \times MemberVars \to \mathbb{N} \\ off(t, v_k) &= \begin{cases} \sum_{i=1}^{k-1} size(type(v_i)) & \text{if } t \equiv C \text{ and } C = \langle v_1, \cdots, v_k, \cdots, v_n \rangle \\ 0 & \text{otherwise} \end{cases} \end{aligned}$$

The function $size$ yields the number of locations $size(t)$ that are needed to store an object of type t, and off yields the offset $off(C, v_k)$ of the member variable v_k within the structure C.

Typed Expressions In the following we define *typed expressions*, i.e., we define the syntax of expressions and their type which is determined by the syntactical components of the expressions. If t is a type, then Exp_t denotes the set of expressions of type t. We distinguish between expressions that can occur on the left side of assignments and other expressions. Expressions that can occur on the left side of assignments are called l-expressions, all other expressions are simply called expressions. L-Expressions are defined as follows:

- (Simple variable):
  if $v \in Vars(P)$ and $type(v) = t$ then $v \in LExp_t$.
- (Referenced location of simple variable):
  if $v \in Vars(P)$ and $type(v) = t*$ then $*v \in LExp_t$.
- (Member variable):
  if $v \in CVars$, $type(v) = C*$, m is a member variable of C and $type(m) = t$ then $v \rightarrow m \in LExp_t$.

The set of all L-expressions is $LExp = \bigcup_{t \in Types} LExp_t$. As expressions can contain constants, we define the sets

$$int_c = \{min, \cdots, -1, 0, 1, \cdots, max\} \quad \text{resp.} \quad bool_c = \{false, true\}$$

of integer constants resp. boolean constants. Additionally, we assume the special constant $fail_c$, and for pointer types we define the set $null_c = \{null_t \mid t \in PTypes\}$, whereby each constant $null_t$ denotes an invalid pointer of type t. With this, we can define the set $Const$ of all constants as

$$Const = int_c \cup bool_c \cup \{fail_c\} \cup null_c.$$

In the following, as a shortcut we often simply write 0 to denote either the integer constant 0, an invalid pointer value $null_t \in null_c$ or the boolean constant $false$. Additionally, we define the sets of arithmetic operations $AOp = \{+, -, *, div, mod\}$, arithmetic relations $ARel = \{<, <=, =, <>, >=, >\}$, pointer relations $PRel = \{==, <>\}$ and boolean operations $BOp = \{and, or, not\}$, and we define

$$Op = AOp \cup ARel \cup PRel \cup BOp.$$

Expressions are defined as follows:

- if $c \in Const$ and $type(c) = t$ then $c \in Exp_t$.
- (Nondeterministic choice): $nd \in Exp_{int}$.
- if $e \in LExp_t$ then $e \in Exp_t$.
- (Location of simple variable):
  if $v \in Vars(P)$ and $type(v) = t$ then $\&v \in Exp_{t*}$.

- (Location of member variable):
  if $v \in CVars$, $type(v) = C*$, m is a member variable of C and $type(m) = t$ then $\&v \to m \in Exp_{t*}$.
- (Arithmetic expressions):
  if $e_1, e_2 \in Exp_{int}$ and $op \in AOp$ then $e_1\, op\, e_2 \in Exp_{int}$.
- (Conditional expressions):
  if $e \in Exp_{bool}, e_1, e_2 \in Exp_t$ then $e?e_1 : e_2 \in Exp_t$.
- (Arithmetic relations):
  if $e_1, e_2 \in Exp_{int}$ and $op \in ARel$ then $e_1\, op\, e_2 \in Exp_{bool}$.
- (Pointer relations):
  if $e_1, e_2 \in Exp_{t*}$ and $op \in PRel$ then $e_1\, op\, e_2 \in Exp_{bool}$.
- (Boolean expressions):
  if $e_1, e_2 \in Exp_{bool}$ then not e_1, e_1 and e_2, e_1 or $e_2 \in Exp_{bool}$.

The set of all expressions is $Exp = \bigcup_{t \in Types} Exp_t$. Additionally, we introduce the function $adr : LExp \to Exp$ that is defined as

$$adr(e) = \begin{cases} v & \text{if } e \equiv *v \\ \&v & \text{if } e \equiv v \\ \&v \to m & \text{if } e \equiv v \to m \end{cases}$$

We will use the function adr to define the semantics of assignments.

Semantics of expressions For the definition of the semantics of expressions we will use the following semantic domains:

- A finite interval of integers $D_{int} = \{min, \ldots, -1, 0, 1, \ldots, max\}$.
- The boolean values $D_{bool} = \{true, false\}$.
- The special values $free$ and $fail$.
- A countable infinite set of *locations* $D_{loc} = \{l_1, l_2, \ldots\}$.

We use the special value $free$ to be able to distinguish between used and unused locations, and the special value $fail$ to indicate an erroneous computation. In our setting, a state will be a mapping from the set of locations to values. As values can be integers, booleans or locations, we define $D = D_{int} \cup D_{bool} \cup D_{loc} \cup \{free\}$ and $\sigma : D_{loc} \to D$. The set of all such states is denoted by Σ. Furthermore, we define $\Sigma_f = \Sigma \cup \{fail\}$. To be able to handle dynamic object creation and destruction, we assume a type-based partitioning of the set of locations, i.e., we assume to have partial functions $tls : D_{loc} \to Structs(P)$ and $tl : D_{loc} \to Types \setminus Structs(P)$ that satisfy the following criteria for all $l \in D_{loc}$:

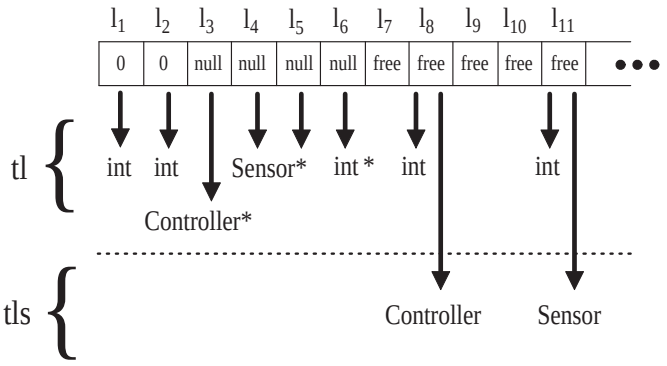


Figure 3.4: **Type-based partitioning functions.** The functions tls and tl are examples of valid type-based partitioning functions.

1. $\forall t \in Structs(P)\, \exists\, l \in D_{loc} : tls(l) = t.$
2. $tls(l) = t \Rightarrow \exists\, l' > l :\ tls(l') = t.$
3. $tls(l) = t \Rightarrow \forall l' \in \{l+1, \cdots, l + size(t) - 1\} :\ tls(l')$ is undefined.
4. $\forall t \in Types \setminus Structs(P)\, \exists\, l \in D_{loc} : tl(l) = t.$
5. $tl(l) = t \Rightarrow \exists\, l' > l : tl(l') = t.$
6. $tls(l) = C \Rightarrow \forall v \in MemberVars(C) : l' = l + off(C, v) \Rightarrow tl(l') = type(v).$

1 and 2 resp. 4 and 5 ensure that there are indeed infinite many locations for every type occurring in the program. Furthermore, 3 ensures that locations of structured types are non-overlapping, and 6 assures that the types of the locations of the member variables of an object of type C correspond to the types of the member variables. Figure 3.4 shows the beginning of suitable functions tls and tl for our sample program. By using the functions tl resp. tls we avoid to store additional type information in the memory which will ease the definition of the semantics of dynamic object creation, i.e., whenever a particular memory location l is used ($\sigma(l) \neq free$), then its type its determined by the functions tl resp. tls.

The semantics of expressions is in general a set of values because of possible nondeterministic expressions. To be able to also use the locations of declared (static) variables, we assume that each declared variable of a program P has a fixed location that is determined by an injective function $varloc : Vars(P) \to D_{loc}$. For elements $s_1, s_2 \in \{free, fail\}$ and every $d \in D$ and $op \in Op$ we define

$$d\ op\, s_1 = fail, s_1\ op\ d = fail, s_1\ op\ s_2 = fail, not\ s_1 = fail.$$

With this, the semantics of an expression $e \in Exp$ is a mapping

$$[\![e]\!] : \Sigma_f \to \mathcal{P}(D \cup fail),$$

and is defined as follows:

- if $e \equiv c \in Const$ then $[\![e]\!](\sigma) = \{c\}$
- (Nondeterministic choice): if $e \equiv nd$ then $[\![e]\!](\sigma) = \{d \mid d \in D_{int}\}$
- (Simple variable): if $e \equiv v$ then $[\![e]\!](\sigma) = \sigma(varloc(v))$
- (Dereferenced simple variable): if $e \equiv *v$ then
  $$[\![e]\!](\sigma) = \begin{cases} \sigma(\sigma(varloc(v))) & \text{if } \sigma(varloc(v)) \neq 0 \wedge \sigma(\sigma(varloc(v))) \neq free \\ fail & \text{otherwise} \end{cases}$$
- (Member variable): if $e \equiv v \rightarrow m$ then
  $$[\![e]\!](\sigma) = \begin{cases} \sigma(\sigma(varloc(v)) + off(type(v), m)) & \text{if } \sigma(varloc(v)) \neq 0 \\ & \wedge \sigma(\sigma(varloc(v))) \neq free \\ fail & \text{otherwise} \end{cases}$$
- (Location of simple variable): if $e \equiv \&v$ then $[\![e]\!](\sigma) = varloc(v)$
- (Location of member variable): if $e \equiv \&v \rightarrow m$ then
  $$[\![e]\!]_l(\sigma) = \begin{cases} \sigma(varloc(v)) + off(type(v), m) & \text{if } \sigma(varloc(v)) \neq 0 \\ fail & \text{otherwise} \end{cases}$$
- (Arithmetic expressions): if $e \equiv e_1\ op\ e_2, op \in AOp$, then
  $[\![e]\!](\sigma) = \{d_1\ op\ d_2 \mid d_1 \in [\![e_1]\!](\sigma), d_2 \in [\![e_2]\!](\sigma)\}$
- (Conditional expressions): if $e \equiv e_1?e_2 : e_3$, then
  $$[\![e]\!](\sigma) = \begin{cases} & \{d \mid d \in [\![e_2]\!] \wedge true \in [\![e_1]\!](\sigma)\} \\ \cup & \{d \mid d \in [\![e_3]\!] \wedge false \in [\![e_1]\!](\sigma)\} \end{cases}$$
- (Arithmetic relations): if $e \equiv e_1\ op\ e_2, op \in ARel$, then
  $[\![e]\!](\sigma) = \{d_1\ op\ d_2 \mid d_1 \in [\![e_1]\!](\sigma), d_2 \in [\![e_2]\!](\sigma)\}$
- (Pointer relations): if $e \equiv e_1\ op\ e_2, op \in PRel$, then
  $[\![e]\!](\sigma) = \{d_1\ op\ d_2 \mid d_1 \in [\![e_1]\!](\sigma), d_2 \in [\![e_2]\!](\sigma)\}$
- (Boolean expressions):

$$\begin{aligned} [\![e_1 \text{ and } e_2]\!](\sigma) &= \{d_1 \wedge d_2 \mid d_1 \in [\![e_1]\!](\sigma), d_2 \in [\![e_2]\!](\sigma)\} \\ [\![e_1 \text{ or } e_2]\!](\sigma) &= \{d_1 \vee d_2 \mid d_1 \in [\![e_1]\!](\sigma), d_2 \in [\![e_2]\!](\sigma)\} \\ [\![\text{not } e_1]\!](\sigma) &= \{\neg d \mid d \in [\![e_1]\!](\sigma)\} \end{aligned}$$

Updates of states To describe the semantics of assignments, we need the notion of an *update* of a state σ, written as $\sigma[l := d]$, where $l, l' \in D_{loc}$ and $d \in D$, and it is defined as

$$\sigma[l := d](l') = \begin{cases} d & \text{if } l = l' \\ \sigma(l') & \text{otherwise.} \end{cases}$$

Furthermore, we define $\sigma[l := fail] = \sigma[fail := d] = fail$, and we extend this definition to a set of values with

$$\sigma[l := \{d_1, \cdots, d_n\}] = \{\sigma[l := d] \mid d \in \{d_1, \cdots, d_n\}\}.$$

Based on this, we define a sequence of updates with

$$\sigma[l_1 := d_1, l_2 := d_2, \cdots, l_n := d_n] = (\sigma[l_1 := d_1])[l_2 := d_2, \cdots, l_n := d_n].$$

Threads As mentioned before, a C_{min} program contains a set of threads

$$Threads(P) = \{t_1, \cdots, t_n\}.$$

A thread $t_i = (i, stms_i)$ consists of an identifier $i \in \{1, \cdots, n\}$ and a sequence of labeled statements

$$stms_i = \langle (k_i^1, stm_i^1), \cdots, (k_i^{j_i}, stm_i^{j_i}) \rangle, k_i^l \in \mathbb{N}.$$

The set of labels of a thread with identifier i is denoted by

$$Labels(i) = \{k_i^l \mid k_i^l \in \{k_i^1, \cdots, k_i^{j_i}\}\},$$

and we assume that

$$i_1 \neq i_2 \Rightarrow Labels(i_1) \cap Labels(i_2) = \emptyset.$$

With $first_i$ we denote the label of the first statement of thread i, and for a label k_i^l, $stm(k_i^l)$ describes the statement labeled with k_i^l. The set of all labels of all threads is denoted by $Labels$, and we use a function

$$\begin{aligned} next &: Labels \to Labels \\ next(k_i^l) &= \begin{cases} k_i^{l+1} & \text{if } l < j_i \\ k_i^l & \text{otherwise} \end{cases} \end{aligned}$$

that yields for every except the last label of a thread the subsequent label, and for the last label simply again the last label. For the definition of statements we need the notions of assignment and concurrent assignment. An assignment $e_l := e$ sets the value of the expression $e \in Exp$ to the location of the expression $e_l \in LExp$. A concurrent assignment $[e_l^1 := e_1, \cdots, e_l^n := e_n]$ describes the execution of all assignments $e_l^i := e_i$ in one transition. Based on the notion of concurrent assignments we can now define the set of possible statements STM_i for a thread i as follows:

- (Concurrent assignment):
  if $stm \equiv [e_l^1 := e_1, \cdots, e_l^n := e_n]$ is a concurrent assignment then $stm \in STM_i$
- (Synchronization):
  if $e \in Exp_{bool}$ and $[e_l^1 := e_1, \cdots, e_l^n := e_n]$ is a concurrent assignment, then
  $\texttt{await}(e, [e_l^1 := e_1, \cdots, e_l^n := e_n]) \in STM_i$
- (Branching):
  if $e \in Exp_{bool}$ and $k_1, k_2 \in Labels(i)$ then $\texttt{jump}(e, k_1, k_2) \in STM_i$
- (Object creation):
  if $v \in Vars(P)$ and $type(v) \in PTypes$ then $\texttt{new}(v) \in STM_i$
- (Object destruction):
  if $v \in Vars(P)$ and $type(v) \in PTypes$ then $\texttt{delete}(v) \in STM_i$

The set STM of all statements of a program is defined as $STM = \bigcup_{i=0}^{n} STM_i$.

Semantics With each thread $t_i = (i, stms_i)$ we associate a so-called program counter $pc_i \in Labels_i$. A *thread configuration* tc of a program is a tuple

$$tc = (pc, \sigma), \sigma \in \Sigma_f.$$

A *thread transition* $(pc, \sigma) \rightarrow (pc', \sigma')$ between two configurations describes one computation step of one thread corresponding to one statement, and is defined as follows:

- (Concurrent assignment):
$$(pc, \sigma) \rightarrow (pc', \sigma') \Leftrightarrow \begin{cases} stm(pc) = [e_l^1 := e_1, \cdots, e_l^n := e_n] \\ \wedge\, pc' = next(pc) \\ \wedge\, \sigma' \in \sigma[[\![adr(e_l^1)]\!](\sigma) := [\![e_1]\!](\sigma), \cdots, [\![adr(e_l^n)]\!](\sigma) := [\![e_n]\!](\sigma)] \end{cases}$$
- (Synchronization):
$$(pc, \sigma) \rightarrow (pc', \sigma') \Leftrightarrow \begin{cases} stm(pc) = \texttt{await}(e, [e_l^1 := e_1, \cdots, e_l^n := e_n]) \\ \wedge\, pc' = next(pc) \ \wedge\ true \in [\![e]\!](\sigma) \\ \wedge\, \sigma' \in \sigma[[\![adr(e_l^1)]\!](\sigma) := [\![e_1]\!](\sigma), \cdots, [\![adr(e_l^n)]\!](\sigma) := [\![e_n]\!](\sigma)] \end{cases}$$
- (Branching):
$$(pc, \sigma) \rightarrow (pc', \sigma') \Leftrightarrow \begin{cases} stm(pc) = \texttt{jump}(e, lab_1, lab_2) \ \wedge\ \sigma' = \sigma \ \wedge \\ ((true \in [\![e]\!](\sigma) \wedge pc' = lab_1) \vee (false \in [\![e]\!](\sigma) \wedge pc' = lab_2)) \end{cases}$$
- (Object creation):
$$(pc, \sigma) \rightarrow (pc', \sigma') \Leftrightarrow \begin{cases} stm(pc) = \texttt{new}(v) \wedge pc' = next(pc) \\ \wedge\, \exists l \in D_{loc} : ((type(v) = C* \wedge tls(l) = C \wedge \sigma(l) = free) \vee \\ (type(v) = t* \notin CTypes \wedge tl(l) = t) \wedge \sigma(l) = free)) \\ \wedge\, \sigma' = \sigma[[\![adr(v)]\!](\sigma) := l, l := 0, \cdots, l + size(t) - 1 := 0] \end{cases}$$

- (Object destruction):

$$(pc, \sigma) \rightarrow (pc', \sigma') \Leftrightarrow \begin{cases} stm(pc) = \texttt{delete}(v) \wedge pc' = next(pc) \\ \wedge \sigma' = \sigma[[\![v]\!](\sigma) := free, \cdots, \\ \qquad\qquad [\![v]\!](\sigma) + size(type(*v)) - 1 := free] \end{cases}$$

Furthermore, for each transition $(pc, \sigma) \rightarrow (pc', \sigma')$ we require that $\sigma \neq fail$, i.e., there is no transition from a failure state. Based on the transition relation for single threads we can now define the possible transition of a whole program. A *configuration* of a program P is a tuple

$$c = (pc_1, \cdots, pc_n, \sigma).$$

The *start configuration* of a program is $(first_1, \cdots, first_n, \sigma_0)$, whereby σ_0 fulfills the conditions

- $\forall v \in Vars(P) : \sigma_0(varloc(v)) = init(v)$.
- $\forall l \in D_{loc} : l \notin Ran(varloc) \Rightarrow \sigma_0(l) = free$.

A *transition* $(pc_1, \cdots, pc_n, \sigma) \rightarrow (pc'_1, \cdots, pc'_n, \sigma')$ describes one computation step of the whole program, and it is defined as

$$(pc_1, \cdots, pc_k, \cdots, pc_n, \sigma) \rightarrow (pc_1, \cdots, pc'_k, \cdots, pc_n, \sigma') \Leftrightarrow (pc_k, \sigma) \rightarrow (pc'_k, \sigma'),$$

i.e., the transitions of the whole program are all possible interleavings of the transitions of all threads. A *run* r of a program P is a sequence of configurations $r = \langle c_0, c_1, \cdots \rangle$ s.t. c_0 is the starting configuration of P and $\forall i \geq 0 : c_i \rightarrow c_{i+1}$ is a transition of P. With $Conf(r)$ we denote the set of all configurations occurring in r. We denote the set of all runs of a program P with $Runs(P)$, and we define

$$Conf(P) = \bigcup_{r \in Runs(P)} Conf(r)$$

and

$$States(P) = \{\sigma \in \Sigma_f \mid (pc_1, \cdots, pc_n, \sigma) \in Conf(P)\}.$$

In the following, when it is not necessary to distinguish the components of a configuration c, we often simply say that c is a *state* of P.

3.3 Properties of C_{min} programs

After having defined the possible runs of a program, we will now define a formalism to specify properties of programs. Given a program P, in the sequel we use $\sigma(r, i)$ to denote state i in a run $r = \langle c_0, c_1, \cdots \rangle$ of P. With $BExp(V)$ resp. $IExp(V)$ we denote the set of all boolean resp. integer expressions only containing variables $v \in V$, and we

define $BIExp(V) = BExp(V) \cup IExp(V)$. Given an expression $e \in BExp(BIVars)$, for a configuration c_i of a run r we define

$$c_i \vDash e \Leftrightarrow [\![e]\!](\sigma(r,i)) = true.$$

Additionally, we define $c_i \nvDash e \Leftrightarrow \neg(c_i \vDash e)$. For the specification of properties, we will use a restricted form of temporal logic TL. Given a program P, the syntax of TL is defined as follows:

- if $q \in BExp(BIVars)$ then q is a state formula.
- if q is a state formula, then $EFq \in TL$.

The semantics of TL-formulas w.r.t. a program P can be given in terms of the runs of P. Intuitively, the notation $P \vDash q$ means that program P fulfills formula q. Given a program P, the relation $\vDash$ is defined as

$$P \vDash EFq \Leftrightarrow \exists r = \langle c_0, c_1, \cdots \rangle \in Runs(P), \exists i \geq 0 : c_i \vDash q$$

The intuitive meaning of a formula EFq w.r.t. a program P is that eventually an execution sequence of the program P will lead to a program configuration where q evaluates to *true*. Formulas of the form EFq are often called *reachability* formulas because they express that a configuration must be reachable in which q holds, and a run

$$r = \langle c_0, c_1, \cdots, c_i \rangle$$

with $c_i \vDash q$ is called a *witness* for EFq. In contrast to other temporal logics like e.g. *linear time temporal logics* (LTL), TL allows only rather restricted specifications. However, if it is not possible to specify property of interest in TL directly, one can often annotate the program with additional statements and variables such that the property is expressible as a TL-formula speaking over the newly introduced variables. Given a formula φ in some logic, the additional statements to the original program are often called an *observer* for φ. The introduction of observers has been described for different logics and computation models, e.g. in [DL02]. Using such an observer, one often can use a simple reachability algorithm to decide if a program fulfills a particular formula φ. This approach also has the advantage that the application of a reachability algorithm to decide the reachability of a particular state of the observer for φ is often much more efficient than the application of a more complex algorithm which is needed to decide the validity of φ directly. However, in this thesis we will not further investigate the problem of richer specification languages and the construction of observers. This is subject of future work and will be discussed in sect. 6.2.

3.4 Translating SymC++ to C_{min}

In this section, we will explain how SymC++ programs can be translated into C_{min} programs. The translation process can be roughly subdivided into three phases: In

the first phase, all object-oriented constructs are translated into equivalent imperative constructs using additional variables, and expressions using arrays are translated into equivalent expressions using pointer arithmetic. In the second phase, function calls and returns are replaced by equivalent constructs using dynamic object creation and destruction, and pointer arithmetic operations are replaced by operations on linked lists. Finally, in the third phase, C_{min} threads are created from the intermediate program. The first phase consists of the following 3 steps:

1. All template classes and template functions are expanded into real classes and functions by instantiating the template definitions with the actual template parameters.

2. All member variables, all statically defined variables and all function local variables of a class type or array type are transformed into pointer variables with a corresponding pointer type. According to that, all array accesses are transformed into equivalent expressions using pointer arithmetic. Furthermore, all implicitly called initialization and destruction functions are made explicit, i.e., implicit constructor and destructor calls for member variables or base classes. For instance, a class definition like

```
class A { ... }
class B { ... };
class C : B {
  A a;
  int d[10];
  int i;
  C() { i=0; }
  int f() { A a1; d[i] = 5; ...; return 1; }
  ...
};
```

   is transformed into a class definition

```
class A { ... };
class B { ... };
class C : B {
  A* a;
  int* d;
  int i;
  C() { B(); a = new A(); d = new int[10]; i=0; }
  int f() { A* a1 = new A(); *(d+i)=5; ...; delete a1; return 1; }
  ...
};
```

3. All classes are transformed into corresponding structures. Each class definition is replaced by a structure definition with the same name and variables. All member functions of a class are replaced by normal functions with the same parameters and the same function bodies, but with an additional `this` parameter. According to that, all accesses to member variables are transformed into accesses using the `this` parameter. For correctly representing the inheritance relation and virtual function calls we need some additional variables in each structure. Let

$$Classes = \{C_1, \cdots, C_n\}$$

denote the set of all classes of a SymC++ program, let $id(C_i) = i \in \{1, \cdots, n\}$ denote a unique identifier for a class C_i and let *inherit* denote the inheritance relation between classes, i.e., $inherit(A, B)$ iff B inherits from A. For a class C, with

$$pre(C) = \{C' \in Classes \mid inherit(C', C)\}$$

and

$$post(C) = \{C' \in Classes \mid inherit(C, C')\}$$

we denote the set of direct predecessors resp. direct successors of C in the inheritance relation, and with $pre^*(C)$ resp. $post^*(C)$ we denote the transitive closure of *pre* resp. *post*. To each structure corresponding to a class C we add for each $C' \in pre(C) \cup post(C)$ a variable $\_C'$ of type $C'*$. Furthermore, we introduce a variable *cid* of type `int` into each structure that serves as a class identifier. Figure 3.5 shows a SymC++ class hierarchy (left) and the corresponding C_{min} structure definitions (right). To adequately represent the construction of a new object of a class, for every class C we define a new function *init_C* that creates a a new object of the corresponding structure. In this function, also new objects of all structures corresponding to classes in $pre^*(C)$ are created, and the pointers $\_C'$ are set according to the inheritance relation. Furthermore, *init_C* sets all *cid* variables of all classes in $pre^*(C)$ to the class index $id(C)$. The pointer variables $\_C'$ are used to correctly represent type casts, and the *cid* variables are used to select the right function in case of virtual function calls. A type cast `(C1*)c2` of a variable `c2` of type `C2` into a variable of type `C1` is transformed into the expression `c2->_C1`, as it is done in the `main` function of fig. 3.5. A virtual function call like `c->f()` on the variable `c` of type `C*` is transformed into a conditional expression that tests the value of the `cid` variable of the structure corresponding to `C`. Depending on the `cid` variable, that contains the index of the real class of the object on which the virtual function is called, the right function to call is selected. For instance, in the `main` function of fig. 3.5, the virtual function call `a->f()` is translated into the conditional expression

```
a->cid=1 ? A::f(a) : C::f(a->_C);
```

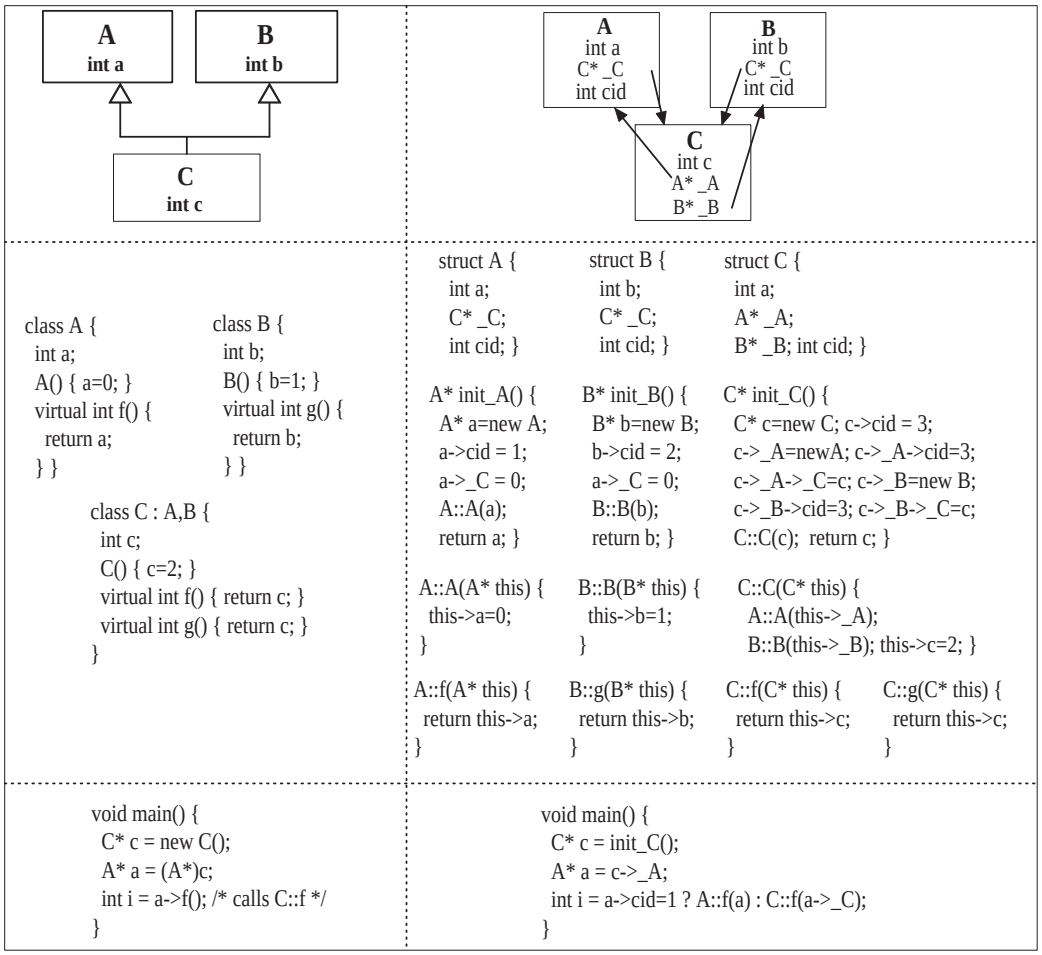


Figure 3.5: **Translating class hierarchies.** A SymC++ class hierarchy consisting of three class A, B and C (left) and the corresponding C_{min} structure definitions (right).

If `a` points at a real `A`-object, then `a->cid` would have the value 1 and thus the function `A::f(a)` would be called. Since in the example shown in fig. 3.5 `a` points at the `A`-part of a `C`-object, `a->cid` has the value 3 and thus `C::f(a->_C)` is called. As one can see, the use of the `cid` variables guarantees the selection of the right function in case of virtual function calls.

After these 3 steps have been performed, the SymC++ program does not contain any classes, arrays or implicit function calls any more, i.e., all object-oriented constructs are transformed into usual imperative constructs, and all array constructs are transformed into constructs using pointer arithmetic. However, as C_{min} programs do not support function calls and pointer arithmetic directly, the translation steps 4, 5, 6 and 7 transform

pointer arithmetic and function calls into equivalent constructs available in C_{min}.

4. Using fresh variables, all expressions are transformed s.t. function calls and pointer arithmetic only occur as single expressions or as a single expression on the right side of an assignment. For instance, an expression like

```
int* f() {...}; int g(int* p) { ... };
*f() = g(f()+1) + 23;
```

   is transformed into the following sequence of expressions:

```
int* f() {...};
int g(int* p) { ... };
int* tmp1 = f();
int* tmp2 = f();
int* tmp3 = tmp2 + 1;
int  tmp4 = g(tmp3);
*tmp1 = tmp4 + 23;
```

   Additionally, all occurrences of the function call `symcpp_nondet` are translated into corresponding nondeterministic expressions in C_{min}, i.e., a call

```
x = symcpp_nondet(0,100);
```

   is translated into

```
tmp = nd;
x = tmp >= 0 && tmp <= 100 ? tmp : 0;
```

   Furthermore, all `new` and `delete` statements are transformed such that they only operate on simple variables. Again, fresh variables are introduced if necessary.

5. All types `t` that are used in pointer arithmetic expressions are transformed into corresponding structure types `t_struct` with additional variables `prev` and `next` of type `t_struct*`. These additional variables are used to realize an array of elements as a double linked list. When a new array is created by a call of the `new[]` operator for arrays, e.g. `C* c = new C[10]`, this call is translated into a sequence of calls of the single `new` operator, and the elements are linked via the `prev` and `next` variables of the corresponding structure type. Furthermore, each pointer arithmetic expression is translated into a traversal of the double linked list representing the underlying array. For instance, the following program

```
struct C { int i; }
void f() {
  C* c = new C[10];
  C* c2 = c+5;
}
```

is translated as follows:

```
struct C_struct { int i; C_struct* prev; C_struct* next; }
void f() {
  C_struct* c = new C;
  C_struct* tmp = c;
  int k;
  for (k=1; k < 10; k++) {
    tmp->next = new C;
    tmp->next->prev = tmp;
    tmp = tmp->next;
  }
  C_struct* c2 = c;
  k=1;
  while (k != 5) {
    c2 = c2->next;
    k = k + 1;
  }
}
```

6. All control-dependent statements, i.e., `while`-, `do-while`-, `for`- and `switch`-statements, are transformed into equivalent statement using only `if`-statements and `goto`-statements. For instance, the loop

```
while (i < 10) {
  i = i + 1;
}
```

is transformed into the loop

```
L0: if (i < 10) goto L2;
L1: goto L3;
L2: i = i + 1;
    goto L0;
L3: ...
```

7. Let $\langle f_1, \cdots, f_m \rangle$ an arbitrary linearization of the set of all functions $\{f_1, \cdots, f_m\}$. We create a new function `prog()` that contains the sequence of statements of all

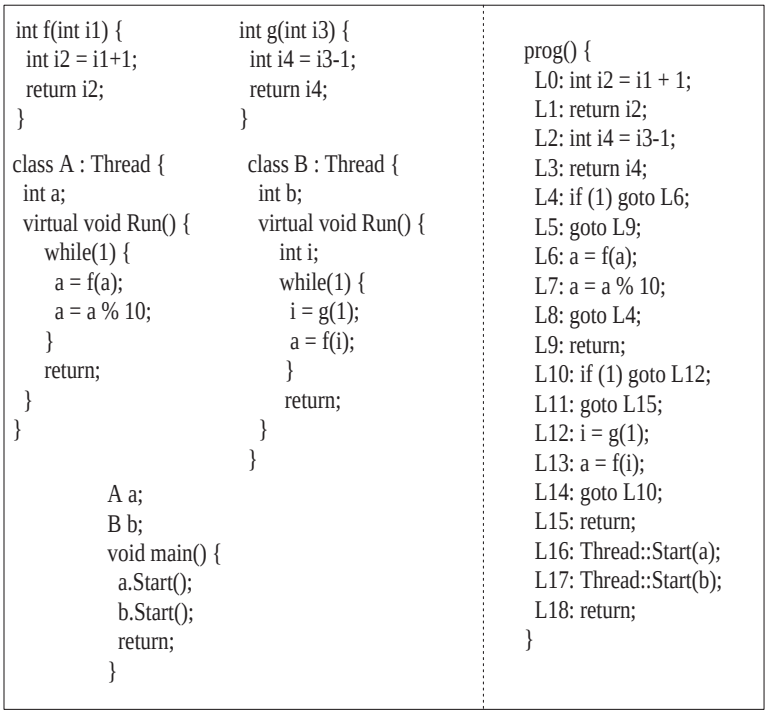


Figure 3.6: **Linearized Functions.** The function `prog` (right) contains a linearized sequence of all statements of all functions of the SymC++ program (left). However, there are still functions calls and return statements in the linearized sequence.

functions according to the given linearization, and we provide each single statement with a unique label. For instance, fig. 3.6. shows the function `prog` for a small SymC++ program. Now, to remove function calls and returns from this sequence we create for each function `f` a corresponding structure `f_params` that contains the following variables:

- A variable for each parameter of the function.
- A variable for each local variable of the function.
- A variable for the return value of the function.
- Two variables `prev` and `next` of type `f_params*`.
- A variable `call` of type `int`.

For instance, for the function

```
C* f(D* d,int i) {
  C* c; ... }
```

the corresponding structure `f_params` is defined as follows:

```
struct f_params {
  D* d;
  int i;
  C* c;
  C* ret;
  f_params *prev,*next;
  int call;
}
```

We will use these structures as *stack frames* during function execution. When a function `f` is called, a new `f_params` object is created. The parameter values of the function call are stored in the corresponding variables of the `f_params` object. Furthermore, we assume that each location has a unique identifier of type `int`, and we store the identifier of the location of the call in the `call`-variable of the `f_params` object. The value of the `call`-variable will be used at the return statement inside the called function to determine the location to which the call has to return. However, during runtime there can exist more than one active invocation of a function due to direct or indirect recursive function calls. Therefore, we organize `f_params` objects as double-linked lists, whereas the variables `next` and `prev` can be used to navigate through these lists. For each structure `f_params` we define a variable `f_p` of type `f_params*`. The variable `f_p` always points at the latest `f_params` object. When a function `f` is called, firstly it is checked if `f_p` already points at a `f_params` object. In such a case a new `f_params` object is created and assigned to `f_p->next`. Furthermore, `f_p->next->prev` gets the value of `f_p` and `f_p` gets the value of `f_p->next`, i.e., `f_p` now points at the newly created `f_params` object. Otherwise, if `f_p` is not pointing at a `f_params` object, a new `f_params` object is created and is simply assigned to `f_p`. During execution of the function `f`, all accesses to parameters and local variables are performed w.r.t. `f_p`, i.e., an access to a local variable `i` in a function `f` is performed via `f_p->i`. When a return statement of a function `f` is reached, we copy the return value into the `ret` variable of the `f_params` object `f_p` is currently pointing at. Furthermore, depending on the value of the `call` variable, a goto to the statement directly after the corresponding call statement is executed. After returning to the statement after the call statement, the return value is copied from `f_p->ret`, `f_p` is set back to `f_p->prev`, and the latest `f_params` object is deleted. As an example, consider the function call depicted in fig. 3.7. For the function `f` the structure `f_params` and the pointer variable `f_p` are created. The function call at L0 is translated to the sequence of statements between L0a and L0k. There, a new `f_param` object is created and assigned to `f_p`. Furthermore, the parameter value 2 is stored in `f_p->i2`, and the index of the location of the call, L0a, is stored in `f_p->call`. After initializing the elements of `f_p`, the goto statement at location L0h jumps to

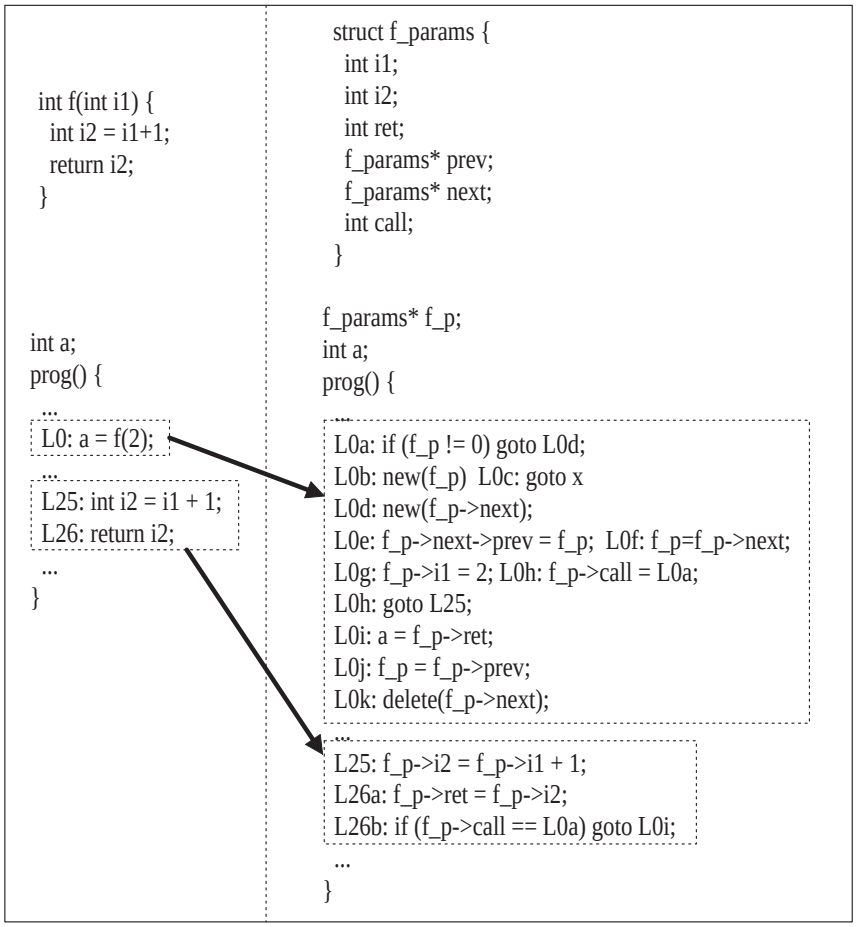


Figure 3.7: **Translating function calls and returns.** For the function `f` the structure `f_params` and the pointer variable `f_p` are created. The function call at L0 is translated to the sequence of statements between L0a and L0k. The statements L26 and L27 belonging to the function `f` are translated to the statements L25, L26a and L26b.

the first statement of the function f at location L25. The access to the parameter `i1` and the local variable `i2` of function `f` is realized through `f_p->i1` resp. `f_p->i2`. When reaching the return statement, at location L26a the computed return value is stored in `f_p->ret`. After that, at location L26b the value of `f_p->call` is tested. If the current call was made at L0a, i.e., if `f_p->call == L0a`, the goto statement branches back to location L0i. There, the return value is copied from `f_p->ret` into the global variable `a`. After that, `f_p` is reset to the previously active `f_params` object, and the `f_params` object `f_p` is pointing at is deleted.

A special case is a return statement in the `main` function resp. in a `Run` function.

Since these functions cannot return to any other functions, the return statements of these functions are translated into a goto statement of the form `Lk: goto Lk`, i.e., whenever the control of a thread reaches such a return statement, the thread stays forever at the label `Lk`.

After the steps 4, 5, 6 and 7 have been performed, we have created a set of structure and variable definitions as well as a `prog` function that contains all statements of all functions. The generated structure and variable definitions are already valid C_{min} definitions. Furthermore, each single statement of the `prog` function can directly be translated into corresponding C_{min} statements. However, each thread of the original SymC++ program has its own call stack, i.e. own copies of local variables. Furthermore, at the beginning of a SymC++ program only the default thread executing the global main function is active. To represent this behavior in the generated C_{min} program, we have to perform two additional transformation steps:

8. Let $\{t_1, \cdots, t_n\}$ be the set of all thread objects of the original SymC++ program. For each thread object t_i we create a new function `thread(i)`, and for the `main` function we create a new function `thread(0)`. For each function `thread(i)` we create own copies of all `f_p` variables denoted as `f_p_i`. Additionally, for each function `thread(i)` we create copies of all statements of the `prog` function replacing the variables `f_p` by the corresponding thread-local variables `f_p_i`. Furthermore, at the beginning of each function `thread(i)` we add an additional goto statement branching to the first statement of the corresponding `Run` function of t_i resp. to the first statement of the `main` function.

9. Since at the beginning of a SymC++ program only the default thread executing the global main function is active, i.e. `thread(0)`, we create for each function `thread(i)` with `i>0` a new integer variable `start_i` that initially has the value 0. At the beginning of each function `thread(i)` we add an additional statement `await(start_i=1)`.

After the transformation steps 8 and 9 have been performed, the resulting program has already become very similar to a C_{min} program, as one can see in fig. 3.8. The function `thread(0)` represents the thread executing the `main` function. Therefore, the first statement in `thread(0)` is a goto statement branching to the first statement of the translated `main` function. The function `thread(1)` represents the thread belonging to the `A` object. Since this thread is not active at the beginning, the first statement of `thread(1)` is an await statement waiting that the condition `started_1==1` becomes true. When this condition becomes true and the await statement can be executed, the subsequent goto statement jumps to the first statement of the translated `A::Run` function. The function `thread(2)` is build up analogously.

Now, the last task we have to realize is the translation of the statements of the functions `thread(i)` into valid C_{min} syntax. This task can be performed straightforward:

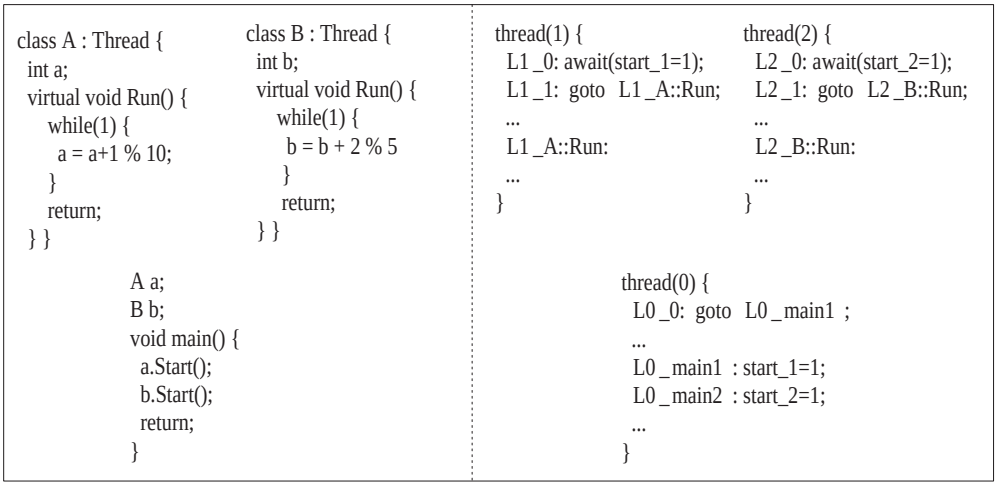


Figure 3.8: **Translating threads.** The thread objects `a` and `b` are translated into C_{min} threads `thread(1)` and `thread(2)`. Additionally, the function `main` is translated into the C_{min} thread `thread(0)`.

- Each assignment $e_l := e$ of an expression e to a location e_l is translated into a C_{min} concurrent assignment $[e_l := e]$.

- Each await statement `await`(e) is translated into a C_{min} await statement `await`$(*e > 0, [*e := 0])$.

- Each if statement `Li: if(e) goto Lk` is translated into a C_{min} jump statement `jump(e,Lk,Li+1)`. Furthermore, each simple goto statement `Li: goto L` is translated into a jump statement `jump(true,L,Li+1)`.

- All new statements resp. all delete statements are already valid C_{min} statements.

After this last step, starting from a SymC++ program we have created an equivalent C_{min} program. Since C++ lacks of a formal semantics we cannot give here a formal proof of the equivalence of the two programs. However, for the evaluation of the state space exploration methods we have tested several SymC++ programs, and for none of these programs we encounter a program run of the corresponding C_{min} program that was not a real run of the original program, which experimentally validates the correctness of the presented translation. In the next section, we give an overview of the SymC++ programs we used for evaluating the state space exploration methods that we present in the chapters 4 and 5.

3.5 Test Programs

In this section, we will give an overview of the SymC++ programs we use to evaluate the state space exploration methods that we present in the chapters 4 and 5. We use programs that are generated from the automatic code generator of the commercial UML case tool Rhapsody [HG97, GHP02]. These programs are ideal candidates for an evaluation, since they show several important characteristics:

- They make intensive use of object-oriented features like inheritance and polymorphism.
- They contain dynamic object creation and destruction, and also recursive function calls.
- They are reactive, i.e., they have cyclic behavior and react to stimuli coming from the environment.
- They contain concurrently running threads.

Therefore, by using such programs we cover a broad range of available SymC++ constructs. In the next section, we will shortly describe Rhapsody UML Models, and we show how we can apply our SymC++ framework to the automatically generated C++ code of these models. After that, in sect. 3.5.2, we give a brief overview about the individual test programs.

3.5.1 Rhapsody UML Models

The Unified Modelling Language (UML) has become accepted as the de facto standard notation for the design of object-oriented software systems. UML contains graphical languages that on the one hand can represent the static structure of a system, e.g. class diagrams, as well as languages that define the dynamic behavior of systems, e.g. state machine diagrams. The class diagrams are used to specify static aspects of a model, e.g. attributes of classes, inheritance relationships, associations and so on. State machines can be used to describe the *reactive* behavior of classes, i.e. how an instance of this class reacts to signal events or call events coming from the environment or from other objects. Furthermore, a so-called *action language* is used to describe all kinds of actions in the systems, e.g. the action part of state machine transitions or the effect of operations of classes. Figure 3.9 shows a very simple class diagram and two state machines. Both class A and class B are so-called *reactive classes*, a notion which in UML is defined as a class which behavior is described by a state machine. An instance of a reactive class, a *reactive object*, can react to events or operation calls. Besides reactive objects there exists also so-called *active objects* which are objects with an own thread of control, and which are usually used to manage a set of reactive objects. Each active object is equipped with an event queue which stores all incoming events. The event processing is divided into so-called *run-to-completion* (RTC) steps. In one RTC step, an event is

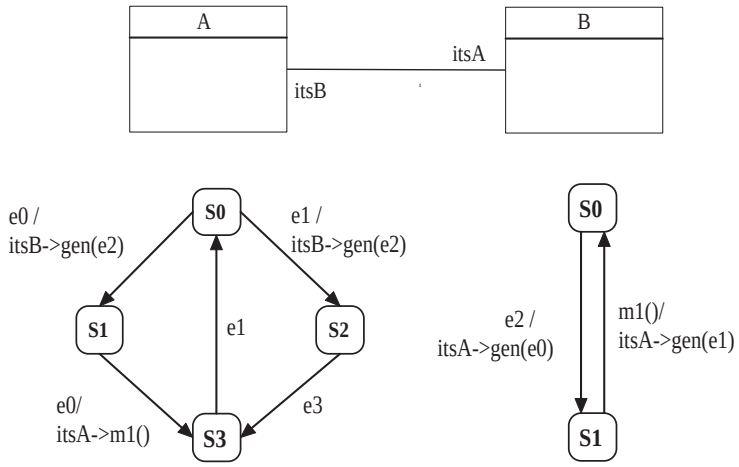
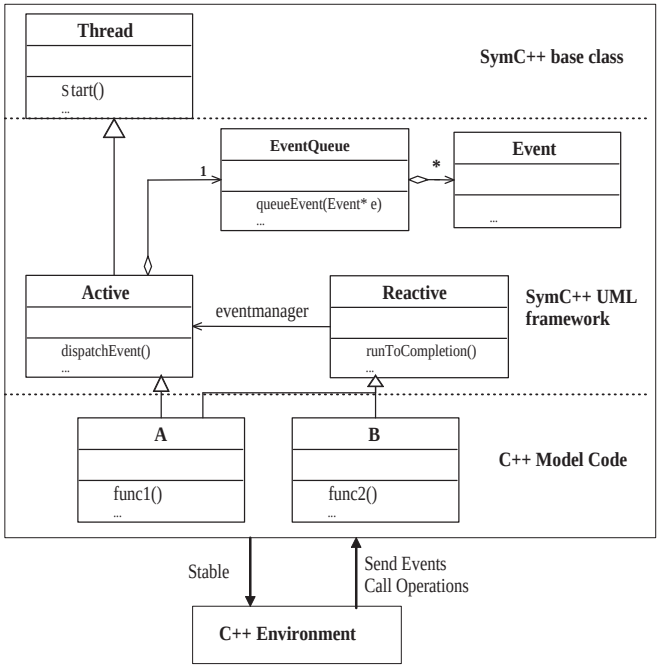


Figure 3.9: **UML Model.** The UML classes A and B are *reactive* as their behavior is defined by UML state machines.

first dispatched from the queue. Dispatching this event enables some transitions in the state machine of the reactive object the event has been sent to, and depending on the current state configuration and values of attributes of the receiving object, a maximal set of non-conflicting transitions will be taken, i.e. the state configuration changes and the actions associated to the taken transitions are executed. After that, so-called *completion* events will be dispatched until the reactive object is in a stable configuration, i.e. the object cannot take any transitions without dispatching a new event (not a completion event) from the event queue. For example, when an instance of class A is in state s0 and it dispatches an event `e0` from the queue then it will change its state to `s1`, and when taking the transition it will send the event `e2` (via the framework function `gen`) along the association `itsB` (which is implemented as a pointer to an object of type `B`). In contrast to asynchronous event sending an object of class A can also communicate synchronously with an object of class B, e.g. when A takes the transition from `s1` to `s3` by calling the method `m1` of B. For a more complete overview about UML see e.g. [Gro04].

The commercial UML case tool Rhapsody [HG97, GHP02] allows the development of so-called *executable* UML models. The name executable stems from the fact that an automatic code generator can generate C++ code from these models. The C++ code is generated according to the UML semantics of the model, and it is divided into model specific code that reflects e.g. the behavior of a particular state machine, and model independent code that is shared between different Rhapsody UML models. The model independent code forms a set of so-called *framework classes* which encapsulate the semantics of certain UML constructs, e.g. the run-to-completion semantics of state machines. The framework classes depend on operating system specific functions to implement e.g. multiple threads. Therefore, we built our own version of these framework




```
void Active::Run() {
  Event* e;
  while (true) {
    if (queue.size() > 0)
    {
      e = queue.top();
      dispatchEvent(e);
    }
  }
}
```

Figure 3.10: **SymC++ UML framework.** Based on the SymC++ class `Thread`, UML framework classes are realized. The `Run` method of the class `Active` iteratively checks its event queue and processes received events via the method `dispatchEvent`.

classes in SymC++. Figure 3.10 shows how we represent the semantics of Rhapsody UML models in our framework. For example, the class *Active* inherits from the SymC++ base class *Thread*. Each thread has a method *Run* which contains the main loop of the thread. *Active's* implementation of the run method (v. fig. 3.10 right) is a never ending loop which continuously checks the event queue associated to this object, and if there is an event in the queue, it is dispatched, i.e. a new RTC step with this event is initiated. To allow also reactive classes without an own thread of control there is the class *Reactive*. Every class which inherits from Reactive has to implement the method `runToCompletion`, which must perform the event processing together with possible completion events. An active class with a statechart can therefore be achieved by inheriting both from Active and Reactive, as it is shown in fig. 3.10 for classes A and B.

In addition to the actual SymC++ code realizing the behavior of the model, we generate a so-called *environment thread* that sends events to objects of the model resp. calls operations and selects input values of parameters. The decision what event or operation is called and which parameter values are used is made nondeterministically, i.e., the generated code for the environment thread simulates the later environment of the embedded program. Additionally, the environment thread is synchronized with

the model threads such that it sends events and calls operations only when the model is *stable*, i.e., when no model thread can proceed any further without a new external stimulus. After generating a new stimulus, the environment thread again waits until the model is stable again. During the processing of the external stimulus, more than one model thread can be active. However, the threads can interleave only at certain points during the computation. Since each model thread belongs to an active object, and since RTC steps are atomic, different threads can only interleave after a RTC step has completed.

3.5.2 Overview of test programs

In this section, we shortly review the individual test model resp. test programs we use for evaluating the state space exploration methods that we present in the chapters 4 and 5:

- **PBX:** The PBX models the behavior of a telephone system consisting of 4 telephones. Each telephone realizes several features, e.g. a quick dial operation. The generated C++ code contains 841 classes and 39340 lines of code. The large number of classes and lines of code stem from the fact that several libraries of the windows operating system are used. In appendix B we give a more detailed overview about this program.

- **SMS:** The SMS models the behavior of a stores management system of an aircraft. The aircraft has several stores that can store different types of loadings. Based on some characteristics of the individual loadings the SMS controls several important parameters of the aircraft. The generated C++ code contains 131 classes and 12295 lines of code.

- **Dishwasher:** The Dishwasher is a model of a dishwasher with several washing programs. For instance, a quick washing program finishes faster than the normal program, but cleans not as thorough as the normal program. The generated C++ code contains 94 classes and 6798 lines of code.

- **CANBus:** The CANBus models the behavior of an implementation of a CANBUS bus protocol. The model contains different parts of a system connected with a bus, e.g. a receiver, a transmitter and a bus controller. The generated C++ code contains 103 classes and 7454 lines of code.

- **ARCS:** The ARCS models the behavior of an automated rail cars system of an airport. Rail cars carry passengers from one terminal to other terminals. During driving the rail cars steadily control the distance to other rail cars and automatically accelerate resp. brake if necessary. The generated C++ code contains 171 classes and 12930 lines of code.

- **Elevator:** The Elevator models the behavior of two elevators in a high rise. A controller arrange the floors where the elevators halt next if an elevator is requested

at different floors at the same time. The generated C++ code contains 98 classes and 6360 lines of code.

- **Pacemaker:** The Pacemaker models the behavior of a computer controlled pacemaker. A controller steadily checks the heart rate of the person wearing the pacemaker, and emits electrical impulses according to that heart rate. The generated C++ code contains 102 classes and 9778 lines of code.
- **HomeHeating:** The HomeHeating models the behavior of a computer controlled heating installation. The user can program a desired temperature, and a controller regulates several radiators in order to establish this temperature. The generated C++ code contains 86 classes and 6885 lines of code.
- **HomeAlarm:** The HomeAlarm models the behavior of an alarm plant. The user can define a PIN with which one can turn on and off the alarm plant. If the alarm plant is enabled, a controller steadily checks the rooms for movements or noise. The generated C++ code contains 96 classes and 7180 lines of code.
- **TCU:** The TCU is a model of mobile telephone control unit in a car. The mobile phone can be used to make or receive phone calls. Furthermore, in case of a breakdown the nearest breakdown service is called automatically, and in case of a crash an emergency call is made. The generated C++ code contains 134 classes and 19147 lines of code.

4 Explicit-Symbolic State Space Exploration

In this chapter, we will define an efficient state space exploration algorithm for C_{min} programs that can be used to find witnesses for formulas EFq. As described in chapter 2, a state space exploration algorithm has to address several issues in order to be efficient:

1. Complexity of the state: In contrast to state space exploration or model checking tools that operate on a more abstract level, a C_{min} program can handle arbitrary dynamic structures. For instance, in a C_{min} program one can use data structures like lists, sets or graphs that cannot be used in verification languages that do not support dynamic memory. Therefore, an efficient treatment of dynamic memory allocation has to be found.

2. Size of programs and state: Although it is in principle possible to build large models or programs in any language, programs in general purpose programming languages like C++ are usually much larger and more detailed than models build in more abstract verification languages. Furthermore, such programs often handle large amount of data. As a consequence, the amount of memory needed to represent a state of a C++ program is usually much larger than the amount of memory needed to represent a state in more abstract verification languages.

3. Number of possible states: An embedded program usually has a certain interface, i.e., a set of functions or methods that are called by the environment as reactions to specific events occurring in the embedded system. These functions or methods usually have several parameters that represent e.g. input values read by some sensors. Due to the possible large domain of these variables, allowing arbitrary input values for these variables results in a tremendous number of possible states. For instance, choosing an arbitrary value for a normal integer variable i with domain $\{-2^{31}+1, \cdots, 2^{31}\}$ in a state σ leads to 2^{32} different successor states.

4. Concurrency: In a program with n concurrent threads each having m distinct states, the global state space can have up to m^n global states. This state explosion due to the interleaving of asynchronously executing threads is present in all languages allowing concurrency, i.e., also in more abstract verification languages, and it has always been a major obstacle when performing a state space exploration. As described in sect. 2, partial order reduction is a successful technique to alleviate this problem.

```
(1)   procedure Explore
(2)   All ← ∅; Open ← ∅
(3)   Open.insert(s_0)
(4)   All.insert(s_0)
(5)   while (Open ≠ ∅)
(6)      u ← Open.get();
(7)      if (eval(u, q) = true)
(8)         return path(u)
(9)      foreach v ∈ next(u)
(10)        if (v ∉ All)
(11)           Open.insert(v)
(12)           All.insert(v)
(13)  end procedure Explore
```

Figure 4.1: **General state expanding exploration algorithm.** The depicted algorithm explores the state space of a C_{min} program in order to find a witness of a formula EFq.

In this chapter, we will address in particular the problems 1, 2 and 3. We will not directly address problem 4 in this chapter, but the heuristic state space exploration procedure presented in chapter 5 can be used to alleviate the state explosion due to concurrency. As already mentioned in sect. 2.1, partial order techniques can sometimes successfully alleviate the effects of concurrency. However, in this thesis we do not further investigate the use of partial order reductions.

The basis of our state space exploration algorithm is the general state expanding search algorithm depicted in fig. 4.1. The algorithm utilizes two sets *Open* and *All*. The set *Open* contains all states that have been generated but which are not yet visited. Contrary to this, the set *All* contains all states that have been generated already. In the beginning, the starting state s_0 is inserted into *Open*. In the main loop of the algorithm, states are extracted from *Open* by calling the operation *Open.get*. Given a formula EFq, if q evaluates to true in a state s then the path from s_0 to s is a witness for EFq. According to that, in line 7 of the algorithm the current state u is evaluated and the path from s_0 to the current state is returned (line 8) if q is true in u. As long as no state is found that fulfills q, all direct successor states $v \in next(u)$ of the current state u are generated. If a successor state v has not been generated before, i.e. v is not in *All*, then it is inserted into *Open*. This procedure is repeated until either a state s is found with $eval(s, q) = true$ or all states have been visited, i.e., $Open = \emptyset$ at line 5.
For the efficiency of the general state expanding search algorithm depicted in fig. 4.1, the following aspects are of particular importance:

- The representation of a state and the realization of the function *next*. Since the function *next* and also the sets *Open* and *All* have to support operations on states,

e.g. the operation $All.insert$, the representation of a state inevitable has impact on the realization of these operations.

- The organization of All. As mentioned above, for realistic programs the number of states that have to be explored by the exploration algorithm can be tremendously large. Therefore, the set All should be realized s.t. a large number of states can be stored. To avoid revisiting states, before inserting a newly generated state s into $Open$ it is checked if already $s \in All$. The efficiency of this operation is crucial for the efficiency of the search algorithm. Furthermore, since states are stored both in $Open$ and All, to avoid redundancy one typically stores only a link or number in $Open$ instead of the complete state. As a consequence, when a state is extracted from $Open$ it must be possible to restore this state from the content stored in All in order to be able to compute the functions $eval$ and $next$. Additionally, since in case a state s is found with $eval(s, q) = true$ the algorithm should return a path from the starting state to s as a witness for EFq, it must be possible to construct such a path from the content stored in All.

- The organization of $Open$. The operation $Open.get$ controls the order in which states are generated and visited. Therefore, the realization of this operation has a crucial impact on the effectiveness of the exploration algorithm. Furthermore, the order in which states are extracted from $Open$ also has implications for the organization of All. For instance, if states can be extracted from $Open$ in an arbitrary order it must be possible to restore an arbitrary state from the content stored in All.

In the sequel, we will discuss and evaluate different realizations for the aspects in the list above. With regard to the last point, the organization of $Open$, in this chapter we will only consider the uninformed search algorithms breadth-first search and depth-first search. When $Open$ is implemented as a stack, algorithm 4.1 performs a depth-first search. Contrary to this, if $Open$ is implemented as a queue, algorithm 4.1 performs a breadth-first search. In section 5, where we present an abstraction-based heuristic search procedure, we will evaluate the usage of informed search algorithms. As a starting point, in the next section we will present an exploration algorithm based on an explicit representation of individual states, and we discuss and evaluate how the operations listed above can be realized efficiently w.r.t. the explicit state representation. After that, in sect. 4.2, we extend the techniques developed in sect. 4.1 to work not only for individual states but also for entire sets of states which are represented by a composite explicit-symbolic representation.

4.1 Explicit State Representation

Beginning with the starting configuration, the state space exploration algorithm shown in fig. 4.1 generates successor configurations from already visited configurations. The generation of successor configurations is repeated until either a witness for a formula EFq

is found or all reachable configurations have already been visited. A configuration is a tuple $(pc_1, \cdots, pc_n, \sigma)$ consisting of the program counters of all threads and a state σ that maps the infinite set of locations to values from D. However, to handle configurations algorithmically we need a finite representation for a configuration. Given a state $\sigma \in \Sigma$, a *state vector* $v(\sigma) \in D^*$ is a sequence

$$v(\sigma) = (\sigma(l_1), \cdots, \sigma(l_k)) \quad \text{s.t.} \quad \sigma(l_k) \neq free \ \wedge \forall i > k : \sigma(i) = free,$$

i.e., $v(\sigma)$ denotes the finite sequence of values from $\sigma(l_1)$ to $\sigma(l_k)$ whereby l_k is the largest location of σ that is not mapped to $free$. Figure 4.2 shows two states and their corresponding state vectors. As can be seen in fig. 4.2, state vectors vary in length, but their length is always finite. Given a configuration $c = (pc_1, \cdots, pc_n, \sigma)$, a configuration vector $v(c) \in Labels^n \times D^*$ is a sequence

$$v(c) = (pc_1, \cdots, pc_n, v(\sigma)),$$

i.e., a configuration vector is simply a state vector extended by a prefix that describes the current valuation of the program counters. Since a configuration vector is finite, it can be used as a data structure to represent configurations in our state space exploration algorithm shown in fig. 4.1. After defining configuration vectors, we can now define how the function *next* and the two sets *Open* and *All* operate on configurations vectors. Given a configuration vector $v(c)$, the function $next : Conf(P) \rightarrow \mathcal{P}(Conf(P))$ with

$$next(v(c)) = \{v(c') \mid c \rightarrow c'\}$$

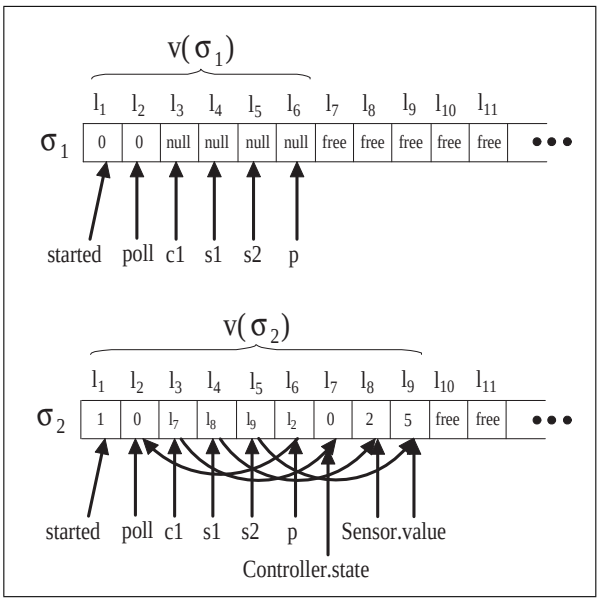


Figure 4.2: **Explicit State Vectors.** Two states σ_1 and σ_2 and their corresponding state vectors $v(\sigma_1)$ and $v(\sigma_2)$.

computes the set of configuration vectors that are successors of the current configuration vector. To allow fast membership checking $v(c) \in All$ of a newly generated configuration vector $v(c)$, the set All of already generated configuration vectors is organized as a hashing table. In a hashing table of configuration vectors with m slots, the index of the slot to store a configuration vector $v(c)$ is determined by a *hashing function*

$$h : Labels^n \times D^* \rightarrow \{0, \cdots, m-1\}.$$

We do not define h in more detail here, but we assume that h distributes uniformly over the range $\{0, \cdots, m-1\}$. Since $|dom(h)| > |ran(h)|$, it is clear that in general h cannot be injective. Therefore, we can have two configuration vectors $v(c_1) \neq v(c_2)$ with $h(v(c_1)) = h(v(c_2))$, a situation that is called *hash collision*. One strategy for dealing with hash collisions is *linear probing* that we will use because of its simplicity. In linear probing, in case of a hash collision at position $i = h(v(c))$, we successively test the positions

$$(i+1)\ mod\ m, (i+2)\ mod\ m, \cdots$$

for the membership of $v(c)$, until either at position $(i+k)\ mod\ m$ we find the element $v(c)$ or we find a free slot in the hashing table. However, if all m slots of the hashing table are occupied, then we cannot store newly generated states any more. In theory, in such a case one could extend h to a larger range $\{0, \cdots, m'-1\}$ with $m' > m$. However, in practice one can choose m such that most of the available randomly accessible memory will be reserved for the hash table. When all entries of the hashing table are filled, and we try to increase the number of entries of the hashing table in order to store more states, then the operating system has to swap parts of the hashing table into secondary memory. The effect of holding parts of the hashing table into secondary memory is a drastic performance decrease, since the entries of the hashing table are addressed randomly, i.e., the probability that an entry of the hashing table that resides on a secondary memory device is accessed is very high. Therefore, when the size of the hashing table becomes larger than the available random access memory, then most of the time will be consumed by memory swapping. In practice this means that if a state space exploration is not successful until all available random access memory is exhausted, then one cannot simply allocate more memory for the hashing table because of the described deceleration of the exploration process. There are two different approaches to deal with such a situation. Either one can simply stop the exploration process, knowing that neither the search has found a witness for a formula EFq nor it has been complete. Alternatively, one can only hold the states in memory which belongs to the current execution path and forget all states belonging to other paths. While such an approach would still be complete in theory, in practice this would also result in a drastic performance decrease, because no duplicate states would be detected that lie on different execution paths. Therefore, in practice one prefers the first alternative, and we also follow this approach.

When the set All is organized as a hashing table, then we can use the computed hash value $h(v(c))$ as an identifier for the configuration vector $v(c)$ in the set $Open$, avoiding a redundant storage of $v(c)$. Figure 4.3 shows a typical situation during the execution

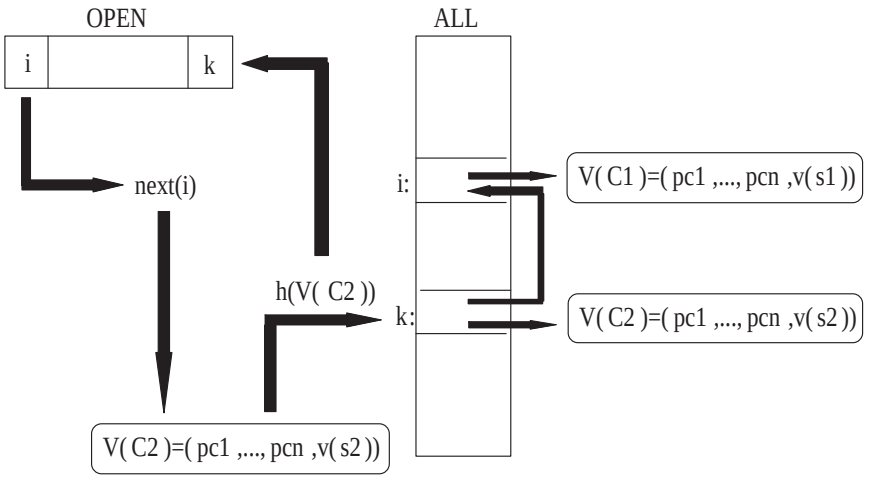


Figure 4.3: **Organization of *All*.** Using an appropriate hashing function h, the set *All* can be organized as a hashing table storing configuration vectors $v(c)$. The set *Open* can use the index $h(c)$ to identify a configuration c.

of algorithm 4.1. Since the index i is extracted from *Open* (line 6 of algorithm 4.1), the next configuration vector to expand (line 9 of algorithm 4.1) is the configuration vector stored at position i in the hashing table representing the set *All*. Since at position i the configuration vector $v(c1)$ is stored, $next(v(c1))$ computes all successor configurations vectors of $v(c1)$, among other things $v(c2)$. To check if $v(c2) \in All$ (line 10 of algorithm 4.1) we compute the index $k = h(v(c2))$. If $v(c2)$ is already in *All*, then we do not insert $v(c2)$ into *All* and we do not insert k in *Open*, since we have done this already before. If $v(c2) \notin All$, then we insert $v(c2)$ into *All* at position k (line 12 of algorithm 4.1) and also insert index k into *Open* (line 11 of algorithm 4.1). Furthermore, if the search goal is EFq, and sometimes later we will find a configuration vector $v(c)$ that fulfills q, we want to return a path from the starting configuration to c as a witness for the search goal EFq (line 8 of algorithm 4.1). In order to reconstruct the path, for each configuration vector we also store a predecessor link that points at the direct predecessor of the configuration vector. In fig. 4.3, since $v(c1)$ is a predecessor of $v(c2)$, in addition to $v(c2)$ we store at position k a predecessor link pointing at position i.

The algorithm depicted in fig. 4.1 concretized with the described treatment of configurations, the successor function *next* and the sets *Open* and *All* together form a complete implementation of a state space exploration algorithm of C_{min} program w.r.t formulas EFq. Because of its simplicity we will call the described solution the *naive* or *simple* approach. In the next sections, we analyze some of the deficiencies of the naive approach and describe some improvements. After that, in sect. 4.1.4 we will analyze the runtime behavior of the naive approach compared to the improved approaches by means of several experiments.

4.1.1 Dynamic Object Creation and Symmetries

As described in the previous section, one purpose of having the set *All* of already generated configuration vectors is the possibility to detect configuration vectors that have been generated before, so-called *duplicates*. However, in a C_{min} program objects can be allocated and deallocated dynamically, which can result in configurations that differ only in the locations where the objects are allocated and the values of the pointer variables. As an example, consider the following program fragment:

```
class C { int x; };
class D { int x; };
C* c1;
D* d1;

void main() {
 L0: if (symcpp_nondet(0,1)) {
       L1:
       c1 = new C(); c1->x = 1;
       d1 = new D(); d1->x = 2;
     }
     else {
       L2:
       d1 = new D(); d1->x = 2;
       c1 = new C(); c1->x = 1;
     }
 L3:
 ...
}
```

When the execution of the program reaches label L0, due to the nondeterminism in the condition of the if-statement two successor configurations are possible, one at which the program counter points at the statement at L1 and one at which the program counter points at the statement at L2. When executing both paths until label L3 is reached, and if newly allocated objects are mapped to the smallest free location, then we would get the states shown in fig. 4.4. The states shown on the left side of fig. 4.4 belong to the execution sequence

```
c1 = new C; c1->x=1;
d1 = new D; d1->x=2;
```

and the states shown on the right side belong to the execution sequence

```
d1 = new D; d1->x=2;
c1 = new C; c1->x=1;
```

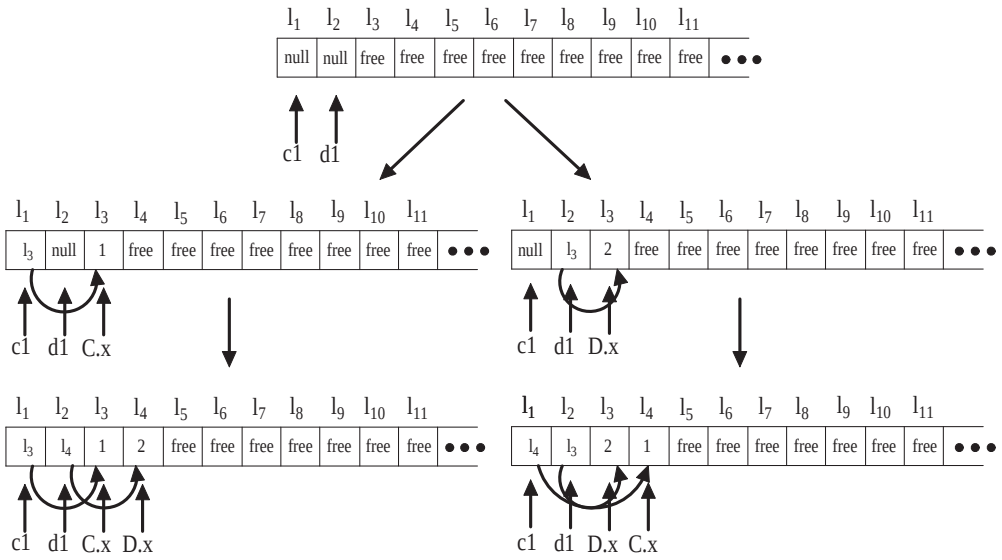


Figure 4.4: **Symmetric configurations.** The two configurations at the bottom are symmetric since they differ only in the locations where the objects are allocated and the values of the pointer variables $c1$ and $c2$

As one can see, the two execution sequences lead to different states, since e.g. location l_3 has the value 1 in the left state and the value 2 in the right state. However, the semantics of C_{min} programs does not depend on the concrete locations used to store objects. Looking at the values reachable from program variables, then we observe that in both states $c1$ points at an object that has a member variable x with value 1 and $d1$ points at an object that has a member variable x with value 2, i.e., the two states shown at the bottom of fig. 4.4 differ only in the positions of the objects, they are *symmetric*. To define symmetry for states formally we need the notion of a *permutation*. Given $n_0 \in \mathbb{N}$, a permutation π is a bijection from the set of natural numbers $\{1, \cdots, n_0\}$ to itself. For a location $l_i \in D_{loc}$ and a permutation π we define $\pi(l_i) = l_{\pi(i)}$, and for all $d \in D_{int} \cup D_{bool}$ we define $\pi(d) = d$, i.e., values from $D_{int} \cup D_{bool}$ are not changed by π. The following definition formalizes the notion of symmetric states.

Definition 4.1 *Let P be a C_{min} program and σ, σ' states of P. We say that two states σ and σ' are symmetric (in symbols $\sigma \equiv \sigma'$) iff there exists a permutation π s.t. $\forall i \geq 0$ it holds*

1. *$l_i \in Ran(varloc) \Rightarrow \pi(i) = i$, i.e., all locations belonging to static variables are not permuted*

2. *$\pi(\sigma(l_i)) = \sigma'(l_{\pi(i)})$, i.e., the permuted value of location l_i in state σ is equal to the value of location $l_{\pi(i)}$ in state σ'.*

Furthermore, we say that two configurations $c_1 = (pc_1, \cdots, pc_n, \sigma)$ *and* $c_2 = (pc'_1, \cdots, pc'_n, \sigma')$ *are symmetric (in symbols* $c_1 \equiv c_2$*) if* $pc_i = pc'_i$ *for all* $i \in \{1, \cdots, n\}$ *and* $\sigma \equiv \sigma'$ *holds.*

We state now some important properties of symmetric states. Firstly, the value of an expression $e \in Exp(P)$ is equivalent in symmetric states. Secondly, if two states are symmetric, when changing both the value of a location in one state and the value of the permuted location in the other state, then the resulting states are also symmetric. Additionally, we state that successors of symmetric states are also symmetric.

Lemma 4.2 *Let* P *be a* C_{min} *program,* $\sigma, \sigma' \in \Sigma$ *be two states with* $\sigma \equiv \sigma'$*,* $e \in Exp(P)$ *be an expression,* $l_i \in D_{loc}$ *a location and* $d \in D$ *an arbitrary value. Let further* c_1 *and* c_1^π *configurations of* P *s.t.* $c_1 \equiv c_1^\pi$*. Then the following holds:*

1. $\pi([\![e]\!](\sigma)) = [\![e]\!](\sigma')$.
2. $\sigma[l_i := d] \equiv \sigma'[l_{\pi(i)} := \pi(d)]$.
3. $c_1 \rightarrow c_2 \Rightarrow \exists c_2^\pi : c_1^\pi \rightarrow c_2^\pi \ \wedge \ c_2 \equiv c_2^\pi$.

Proof:

1. (By structural induction):
   - if $e \equiv c \in Const$ then $\pi([\![e]\!](\sigma)) = \pi(c) = c = [\![e]\!](\sigma')$
   - if $e \equiv v$ then $\pi([\![e]\!](\sigma)) = \pi(\sigma(varloc(v))) = \sigma'(\pi(varloc(v)))$. Since $varloc(v) \in Ran(varloc)$, from 1 of def. 4.1 we know that $\pi(varloc(v)) = varloc(v)$, thus we have $\sigma'(\pi(varloc(v))) = \sigma'(varloc(v)) = [\![e]\!](\sigma')$. The cases
     $$e \equiv *v, \quad e \equiv v \rightarrow m, \quad e \equiv \&v, \quad \text{and } e \equiv \&v \rightarrow m$$
     can be shown similarly.
   - (Nondeterministic choice): if $e \equiv nd$ then
     $\pi([\![e]\!](\sigma)) = \pi(\{d \,|\, d \in D_{int}\}) = \{d \,|\, d \in D_{int}\} = [\![e]\!](\sigma')$
   - (Arithmetic expressions): Suppose the lemma holds for expressions e_1 and e_2. If $e \equiv e_1 \; op \; e_2, op \in AOp$, then
     $$\begin{aligned} \pi([\![e]\!](\sigma)) &= \pi(\{d_1 \; op \; d_2 \,|\, d_1 \in [\![e_1]\!], d_2 \in [\![e_2]\!]\}) \\ &= \{d_1 \; op \; d_2 \,|\, d_1 \in \pi([\![e_1]\!](\sigma)), d_2 \in \pi([\![e_2]\!](\sigma))\} \\ &= \{d_1 \; op \; d_2 \,|\, d_1 \in [\![e_1]\!](\sigma')), d_2 \in [\![e_2]\!](\sigma')\} \\ &= [\![e]\!](\sigma') \end{aligned}$$
     The cases
     $$e \equiv e_1 ? e_2 : e_3, \quad e \equiv e_1 \; op \; e_2, op \in ARel, \quad e \equiv e_1 \; op \; e_2, op \in PRel,$$
     $$e_1 \text{ and } e_2, \quad e_1 \text{ or } e_2, \quad \text{not } e_1$$
     can be shown similarly.

2. If $l_k \neq l_i$ we have

$$\pi(\sigma[l_i := d](l_k)) = \pi(\sigma(l_k)) = \sigma'(l_{\pi(k)}) = \sigma'[l_{\pi(i)} := \pi(d)](l_{\pi(k)}).$$

   Otherwise, if $l_k = l_i$ we have

$$\pi(\sigma[l_i := d](l_k)) = \pi(d) = \sigma'[l_{\pi(i)} := \pi(d)](l_{\pi(k)}).$$

3. Let $c_1 = (pc_1, \cdots, pc_i, \cdots, pc_n, \sigma)$ and $c_1^\pi = (pc_1, \cdots, pc_i, \cdots, pc_n, \sigma^\pi)$ be two symmetric configurations related by the permutation π. Let further $c_2 = (pc_1, \cdots, pc_i', \cdots, pc_n, \sigma')$ be a successor configuration of c_1, i.e., thread i performs a transition $(pc_i, \sigma) \rightarrow (pc_i', \sigma')$. We distinguish the following cases:

   - Concurrent assignment $stm(pc_i) = [e_l^1 := e_1, \cdots, e_l^n := e_n]$. Let

$$\sigma' \in \sigma[[\![adr(e_l^1)]\!](\sigma) := [\![e_1]\!](\sigma), \cdots, [\![adr(e_l^n)]\!](\sigma) := [\![e_n]\!](\sigma)]$$

     be an arbitrary successor state of σ. Since $c_1 \equiv c_1^\pi$, from 1 and 2 of lemma 4.2 we can conclude that

$$\begin{array}{lcl} \forall \sigma_i & \in & \sigma[[\![adr(e_l^1)]\!](\sigma) := [\![e_1]\!](\sigma), \cdots, [\![adr(e_l^n)]\!](\sigma) := [\![e_n]\!](\sigma)] \\ \exists \sigma_k & \in & \sigma^\pi[[\![adr(e_l^1)]\!](\sigma^\pi) := [\![e_1]\!](\sigma^\pi), \cdots, [\![adr(e_l^n)]\!](\sigma^\pi) := [\![e_n]\!](\sigma^\pi)] : \sigma_i \equiv \sigma_k, \end{array}$$

     i.e., we can choose $\sigma^{\pi'}$ s.t. $\sigma' \equiv \sigma^{\pi'}$. Since furthermore

$$c_2^\pi = (pc_1, \cdots, pc_i', \cdots, pc_n, \sigma^{\pi'})$$

     is a successor of c_1^π we can conclude that $c_2 \equiv c_2^\pi$.
   - Synchronization $stm(pc_i) = \texttt{await}(e, [e_l^1 := e_1, \cdots, e_l^n := e_n])$. Let

$$\sigma' \in \sigma[[\![adr(e_l^1)]\!](\sigma) := [\![e_1]\!](\sigma), \cdots, [\![adr(e_l^n)]\!](\sigma) := [\![e_n]\!](\sigma)]$$

     be an arbitrary successor state of σ. With 1 of lemma 4.2 it is clear that

$$true \in [\![e]\!](\sigma) \Leftrightarrow true \in [\![e]\!](\sigma^\pi),$$

     i.e., there is also a transition possible from c_1^π. Furthermore, since $c_1 \equiv c_1^\pi$, from 1 and 2 of lemma 4.2 we can conclude that

$$\begin{array}{lcl} \forall \sigma_i & \in & \sigma[[\![adr(e_l^1)]\!](\sigma) := [\![e_1]\!](\sigma), \cdots, [\![adr(e_l^n)]\!](\sigma) := [\![e_n]\!](\sigma)] \\ \exists \sigma_k & \in & \sigma^\pi[[\![adr(e_l^1)]\!](\sigma^\pi) := [\![e_1]\!](\sigma^\pi), \cdots, [\![adr(e_l^n)]\!](\sigma^\pi) := [\![e_n]\!](\sigma^\pi)] : \sigma_i \equiv \sigma_k, \end{array}$$

     i.e., we can choose $\sigma^{\pi'}$ s.t. $\sigma' \equiv \sigma^{\pi'}$. Thus, since

$$c_2^\pi = (pc_1, \cdots, pc_i', \cdots, pc_n, \sigma^{\pi'})$$

     is a successor of c_1^π we can conclude that $c_2 \equiv c_2^\pi$.

- Branching $stm(pc_i) = \texttt{jump}(e, lab_1, lab_2)$. With 1 of lemma 4.2 it is clear that

$$[\![e]\!](\sigma) = [\![e]\!](\sigma^\pi).$$

  Therefore, if $true \in [\![e]\!](\sigma) \wedge pc_i' = lab_1$, then also $true \in [\![e]\!](\sigma^\pi) \wedge pc_i' = lab_1$, and if $false \in [\![e]\!](\sigma) \wedge pc_i' = lab_2$, then also $false \in [\![e]\!](\sigma^\pi) \wedge pc_i' = lab_2$. Thus we have $c_2 \equiv c_2^\pi$.

- Object creation $stm(pc_i) = \texttt{new}(v)$. Let l_j be the first location where the newly created object has been allocated in state σ, i.e.,

$$\sigma' = \sigma[[\![adr(v)]\!](\sigma) := l_j, l_j := 0, \cdots, l_j + size(t) - 1 := 0].$$

  Analogously, let l_m be the first location where the newly created object has been allocated in state σ^π, i.e.,

$$\sigma^{\pi'} = \sigma^\pi[[\![adr(v)]\!](\sigma^\pi) := l_m, l_m := 0, \cdots, l_m + size(t) - 1 := 0].$$

  The permutation π' that is defined as

$$\pi'(i) = \begin{cases} m + i - j & \text{if } i \in \{j, \cdots, j + size(t) - 1\} \\ \pi(i) & \text{otherwise} \end{cases}$$

  fulfills the requirements from def. 4.1, therefore we have $c_2 \equiv c_2^\pi$.

- Object destruction $stm(pc_i) = \texttt{delete}(v)$. From the definition of the semantics we know that

$$\sigma' = \sigma[[\![v]\!](\sigma) := free, \cdots, [\![v]\!](\sigma) + size(type(*v)) - 1 := free]$$

  and

$$\sigma^{\pi'} = \sigma^\pi[[\![v]\!](\sigma^\pi) := free, \cdots, [\![v]\!](\sigma^\pi) + size(type(*v)) - 1 := free].$$

  Since $c_1 \equiv c_1^\pi$, from 1 and 2 of lemma 4.2 we can directly conclude that $\sigma' \equiv \sigma^{\pi'}$ and also $c_2 \equiv c_2^\pi$.

□

By using the fact that the set of permutations together with functional composition and identity forms a group it can be shown that $\equiv$ is an equivalence relation on configurations (v. e.g. [CGP99]). The equivalence class, also known as the *orbit*, of a configuration c is denoted by $[c]$, i.e., for each $c' \equiv c$ we have $c' \in [c]$. We now define a so-called *canonization function* which computes for each equivalence class $[c]$ a unique representative.

Definition 4.3 *Let P be a C_{min} program and $\equiv$ a symmetry relation as defined in definition 4.1. A function $n : Conf(P) \to Conf(P)$ is called a canonization function for $\equiv$ iff for all $c, c' \in Conf(P)$ it holds*

$$c \equiv c' \Leftrightarrow n(c) = n(c').$$

Given a canonization function n and a program P, we can define the set of *canonized runs* of P as follows:

Definition 4.4 *Let P be a C_{min} program, $\equiv$ a symmetry relation as defined in definition 4.1 and n a canonization function. The set $NRuns(P)$ of canonized runs is defined as*

$$NRuns(P) = \left\{ \langle n(c_0), n(c_1), \cdots \rangle \;\middle|\; \begin{array}{c} c_0 \text{ is starting configuration of } P \\ \wedge \forall i \geq 0 : n(c_i) \rightarrow c_{i+1} \end{array} \right\}$$

Each canonized run starts with the canonized starting configuration. Furthermore, when there is a transition $n(c) \rightarrow c'$ possible in P, then we use $n(c')$ as the successor configuration of $n(c)$ instead of c'. The following theorem states the important result that for each run $r \in Runs(P)$ we have a symmetric run $r' \in NRuns(P)$ and vice versa.

Theorem 4.5 *Let P be a C_{min} program and $\equiv$ a symmetry relation as defined in definition 4.1. Then the following holds:*

$$r = \langle c_0, c_1, \cdots \rangle \in Runs(P) \Leftrightarrow r' = \langle n(c_0), n(c_1), \cdots \rangle \in NRuns(P)$$

Proof:

- "$\Rightarrow$": By induction on the length of runs. Let $r = \langle c_0 \rangle \in Runs(P)$. Since c_0 is the starting configuration of P, from definition 4.4 we can conclude that $r' = \langle n(c_0) \rangle \in NRuns(P)$. Now suppose the theorem holds for all runs up to length k, and let $r = \langle c_0, \cdots, c_k, c_{k+1} \rangle \in Runs(P)$ be a run of length $k+1$. Since the theorem holds for all runs up to length k we know that $r' = \langle n(c_0), \cdots, n(c_k) \rangle \in NRuns(P)$. Since $c_k \equiv n(c_k)$, from 3 of lemma 4.2 we know that if $c_k \rightarrow c_{k+1}$ then there exists a $c'_{k+1} \equiv c_{k+1}$ s.t. $n(c_k) \rightarrow c'_{k+1}$. Since $c'_{k+1} \equiv c_{k+1}$ we know that $n(c'_{k+1}) = n(c_{k+1})$, and therefore $r' = \langle n(c_0), \cdots, n(c_{k+1}) \rangle \in NRuns(P)$.
- "$\Leftarrow$":

  By induction on the length of runs. Let $r' = \langle n(c_0) \rangle \in NRuns(P)$. Since $n(c_0)$ is the canonized starting configuration of P we know from definition 4.4 that c_0 is the starting configuration of P, and therefore it holds that $r = \langle c_0 \rangle \in Runs(P)$. Now suppose the theorem holds for all runs up to length k, and let $r' = \langle n(c_0), \cdots, n(c_k), n(c_{k+1}) \rangle \in NRuns(P)$ be a run of length $k+1$. Since the theorem holds for all runs up to length k we know that $r = \langle c_0, \cdots, c_k \rangle \in Runs(P)$. Since $n(c_k) \equiv c_k$, from 3 of lemma 4.2 we know that if $n(c_k) \rightarrow n(c_{k+1})$ then there exists a $c'_{k+1} \equiv n(c_{k+1})$ s.t. $c_k \rightarrow c'_{k+1}$. Since $c'_{k+1} \equiv n(c_{k+1})$ we know that $n(c'_{k+1}) = n(c_{k+1})$, and therefore $r = \langle c_0, \cdots, c_k, c_{k+1} \rangle \in Runs(P)$.

□

Since from 1 of lemma 4.2 we know that the value of expressions is equivalent in symmetric configurations, and since theorem 4.5 holds, to check whether $P \vDash EFq$ it suffices to

```
(1)  procedure Explore
(2)  All ← ∅;
(3)  Open ← ∅
(4)  Open.insert(n(s_0))
(5)  while (Open ≠ ∅)
(6)      u ← Open.get();
(7)      if (eval(u, q) = true)
(8)          return path(u)
(9)      foreach v ∈ next(u)
(10)         if (n(v) ∉ All)
(11)             Open.insert(n(v))
(12)             All.insert(n(v))
(13) end procedure Explore
```

Figure 4.5: **Exploring canonical states.** The depicted exploration algorithm uses a canonization function n to canonize configurations before they are inserted into *All* and *Closed*.

explore only canonized runs instead of all runs. The slightly modified search algorithm depicted in fig. 4.5 utilizes a canonization function n. Instead of a configuration c, the algorithm always inserts a canonized configuration $n(c)$ into *Open* and *All*. Additionally, for checking if a newly generated configuration c has already been generated before (line 10 of algorithm 4.5) the canonized configuration $n(c)$ is used. The advantage of algorithm 4.5 compared to the simple algorithm is that in general algorithm 4.5 explores fewer configurations than the simple algorithm. The reason for this lies in the fact that a canonized configuration is a unique representative of all configurations that are symmetric to each other. For instance, consider the following program which is slightly modified compared to the program from fig. 4.4:

```
class C { C* n; };
class D { D* n; }
C *c1,*cl1;
D *d1,*dl1;

int i=0;int N=100;
```

```
void main() {
 L0: while (i < N) {
       if (symcpp_nondet(0,1)) {
         L1:
         if (i=0) {
           c1 = new C(); c1->n = 0; cl1 = c1;
           d1 = new D(); d1->n = 0; dl1 = d1; }
         else {
           cl1->n = new C(); cl1 = cl1->n; cl1->n = 0;
           dl1->n = new D(); dl1 = dl1->n; dl1->n = 0; }
       }
       else {
         L2:
         if (i=0) {
           d1 = new D(); d1->n = 0; dl1 = d1;
           c1 = new C(); c1->n = 0; cl1 = c1; }
         else {
           dl1->n = new D(); dl1 = dl1->n; dl1->n = 0;
           cl1->n = new C(); cl1 = cl1->n; cl1->n = 0; }
       }
       i = i + 1;
     }
 L3: ...
}
```

The program creates two single linked lists of objects of class C resp. class D. The variables $c1$ resp. $d1$ point at the roots of the lists, and the variables $cl1$ resp. $dl1$ point at the last elements. In each iteration of the while loop it is nondeterministically chosen which list is extended first. Therefore, the simple algorithm would generate 2^N different configurations that reaches label L3, as can be seen in fig. 4.6. However, algorithm 4.5 that canonizes configurations with a canonization function n would only generate a single configuration that reaches label L3, as can be seen in fig. 4.7. Although the order of the object creations is chosen nondeterministically, by canonizing generated configurations in each loop iteration a duplicated configuration is detected. Therefore, only a single configuration would be generated that reaches label L3.

A prerequisite for applying algorithm 4.5 is a suitable canonization function n. Unfortunately, it has been shown that the general problem of finding a canonical representation for every element in the same orbit is at least as hard as testing graph isomorphism, for which no polynomial-time solution is known to exist [CFJ93]. In [Ios01] the more specific problem of finding a canonical heap representation for heap-manipulating programs without explicit object destruction was considered. To compute a canonical representative, after each program transition a complete depth-first reachability analysis of the

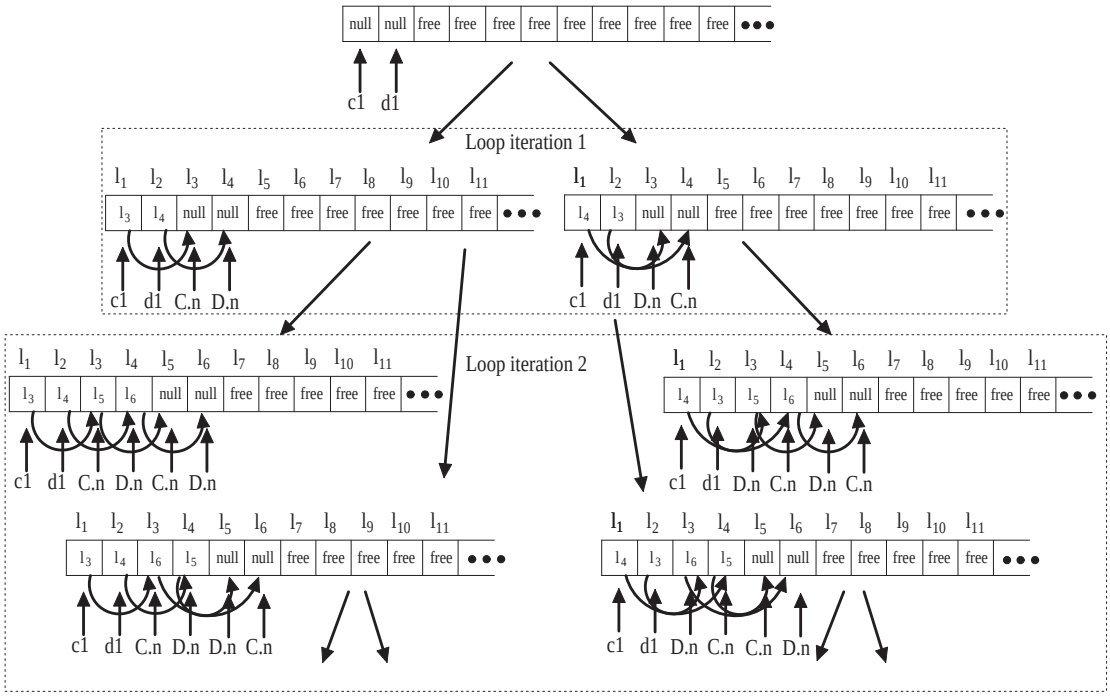


Figure 4.6: **Exponential number of symmetric states.** In each loop iteration, two new objects are created. Since the order of the object creations is chosen nondeterministically, after N loop iterations there are 2^N different configurations reachable.

heap objects has to be performed whose complexity grows linear with the number of heap objects. Clearly, such a complex computation after each program transition drastically slows down the exploration process. To avoid such complex computations that slow down the exploration process, we consider an *approximative canonization function* n which can be computed very efficiently in constant time but which is in general not optimal, i.e., there can be symmetric configurations c, c' s.t. $n(c) \neq n(c')$. The basic idea for the computation of n is to memorize the type of the object that has been allocated at a particular location. During the exploration process, whenever a location l is used the first time, we memorize the type T of the object that has been allocated at l, and for the rest of the exploration process at l only objects of type T can be allocated. More formally, for each type T we maintain a list $loc(T) = \langle l_{i_1}, l_{i_2}, \cdots, l_{i_n} \rangle$ of locations that already have been used for storing objects of type T. At the beginning of the exploration process all these lists are empty. If a new object o of type T has to be created, the list $loc(T)$ is searched for a free location. If there is a free location $l_{i_k} \in loc(T)$, then this location is used for storing o. If no such location is found in $loc(T)$, then a new location l_{new} is used to store o. This new location must not occur in any list so far, i.e., it must hold that $\forall t \in Types(P) : l_{new} \notin loc(t)$. This can be achieved by maintaining a

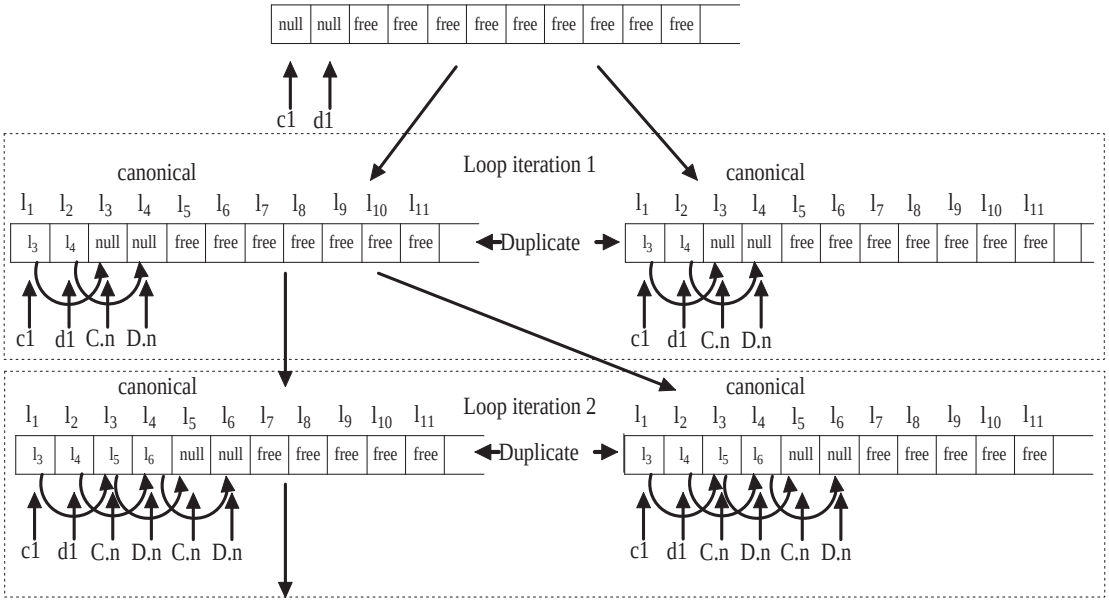


Figure 4.7: **Duplicate detection of symmetric states.** In each loop iteration, two new objects are created. Although the order of the object creations is chosen nondeterministically, by normalizing generated configurations in each loop iteration a duplicated configuration is detected.

variable l_{max} that contains the largest location that has been used in all configurations. Additionally, the location l_{new} is inserted into $loc(T)$. The algorithm in fig. 4.8 realizes the described allocation scheme. It is easy to see that the described treatment of object creations yields optimal symmetry reductions for the examples shown in fig. 4.4 and fig. 4.6, since the placement of an object depends only on the number of already generated objects with the same type and not on the particular program path that has been executed. However, there are programs for which our allocation scheme will not lead to a canonical representation of symmetric states, as can be seen in the following example:

```
class C {int x}; C* c1; C* c2;

void main() {
  if (symcpp_nondet(0,1)) {
    c1 = new C; c2 = new C;
    delete c1; c1 = 0;
  }
  else { c2 = new C; }
  L3:
  ...
}
```

```
(1)  procedure New(Type t)
(2)  if (∃ free location l ∈ loc(t))
(3)     return l
(4)  else
(5)     l = l_max;
(6)     loc(t).insert(l);
(7)     l_max = l_max + size(t);
(8)     return l
(9)  end procedure New
```

Figure 4.8: **Approximative canonization by type-based object placement.** By using the depicted algorithm for calculating a new free location for storing an object of type t, a fast approximative canonization is realized.

Both execution paths that reaches label L3 lead to configurations that are symmetric, since at L3 in both configurations c2 points at an object of class C and c1 points at 0. Unfortunately, our allocation scheme would yield two different configurations at L3, since the location of the object pointed at by c2 is different in both configurations. However, in practice our algorithm is able to detect many symmetric configurations, and the time overhead is almost non-existent due to the simplicity of the algorithm. The experimental results in sect. 4.1.4 will provide evidence on the effectiveness of our algorithm.

4.1.2 Approximated Duplicate Detection

As described in sect. 4.1, to safely detect state duplicates we have to store complete states in the set *All* of already generated states. However, when searching for a witness of a formula EFq, storing complete states can be counterproductive. As already mentioned, in practice the limiting factor of the number of states that can be stored is the available randomly accessible memory, i.e., there exists a maximum number m of states that can be stored. However, for realistic programs the number of reachable states n is often far larger than m, i.e. $n \ggg m$. Thus, for realistic programs a complete state space exploration will seldom be possible.

One way to increase the number of states that can be stored is to use an *approximated* duplicate detection instead of an exact one. Techniques for approximated duplicate detection often fall into the category of *probabilistic methods* because there is a certain probability with which different states are wrongly assumed to be duplicates. The first who applied such a method in the context of state space exploration was Holzmann who invented a technique called *bitstate hashing* [Hol87, Hol91]. Instead of maintaining a conventional table in which complete states are stored, bitstate hashing only maintains a huge table of bits which are initially set to zero. When a state is inserted into the table, one or more hash values are calculated from the state and the bits corresponding to these

hash values are set to one. When the algorithm examines a newly generated state and finds that the bits corresponding to this state are already set to one, it assumes that it has visited the newly generated state already. Clearly, this assumption can be incorrect, since with a certain probability two different states will have the same hash values. In such a case, the newly generated state and potential successors are not explored. Another method called *hash compaction* [WL93] was introduced by Wolper and Leroy and later refined by Stern and Dill [SD95a, SD95b]. Like bitstate hashing, hash compaction aims at reducing the memory requirements for the state table. However, hash compaction stores a compressed state in a conventional table instead of setting bits corresponding to hash values.

As already mentioned in sect. 4.1, we do not want to restrict the possible search order in which states can be explored. Since in bitstate hashing no predecessor information can be stored in the hashing table, it can usually only be used in combination with depth-first search, storing the path to the current state on the depth-first stack. Therefore, we adopt the method of hash compaction, since in addition to the calculated hash value we have to store additional information in the table if we want to be able to explore the state space in an arbitrary order. Firstly, as in the simple approach, for each state stored in the table we store a predecessor link pointing at the entry in the table where the predecessor state has been stored. For a configuration vector $v(c)$, $pred(v(c)) = v(c')$ denotes the predecessor configuration vector of $v(c)$. Secondly, if c' is a successor configuration of c, then we store the content of all memory locations of c' that are different compared to c. Formally, if $v(c = \langle pc_1, \cdots, pc_n, \sigma\rangle)$ and $v(c' = \langle pc'_1, \cdots, pc'_n, \sigma'\rangle)$ are two configuration vectors, then $d(v(c), v(c')) \in (N \times Labels)^* \times (D_{loc} \times D)^*$ with

$$\begin{array}{rcl} d(v(c), v(c')) & = & \langle (j_1, pc'_{j_1}), \cdots, (j_k, pc'_{j_k}), (l_{i_1}, \sigma'(l_{i_1})), \cdots, (l_{i_l}, \sigma'(l_{i_l}))\rangle \\ & \text{s.t.} & j \in \{j_1, \cdots, j_k\} \Leftrightarrow pc_j \neq pc'_j \\ & \wedge & i \in \{i_1, \cdots, i_l\} \Leftrightarrow \sigma(l_i) \neq \sigma'(l_i) \end{array}$$

is the *difference* of $v(c)$ and $v(c')$, i.e., $d(v(c), v(c'))$ contains all program counters and memory locations from $v(c')$ that are different compared to $v(c)$. To restore an arbitrary configuration vector $v(c_i)$, one can now proceed as follows:

- Create the path $\langle v(c_0), v(c_1), \cdots, v(c_i)\rangle$ s.t. for all $j \in \{1, \cdots, i\}$ it holds that $pred(v(c_j)) = v(c_{j-1})$, i.e., we use the predecessor link to reconstruct the path from the starting configuration vector $v(c_0)$ to $v(c_i)$.
- Beginning with the starting configuration, successively apply the stored differences $\langle d(v(c_0), v(c_1)), \cdots, d(v(c_{i-1}), v(c_i))\rangle$ to restore $v(c_i)$.

However, the described method can become inefficient if we have to restore states with a large path length. In such situations it can be beneficial if we can reconstruct a state from any other state. For instance, consider the situation depicted on the left side of fig. 4.9. Suppose the last state that has been expanded is state y, and its successors have been generated. Now, due to the search order, the next state to expand is state

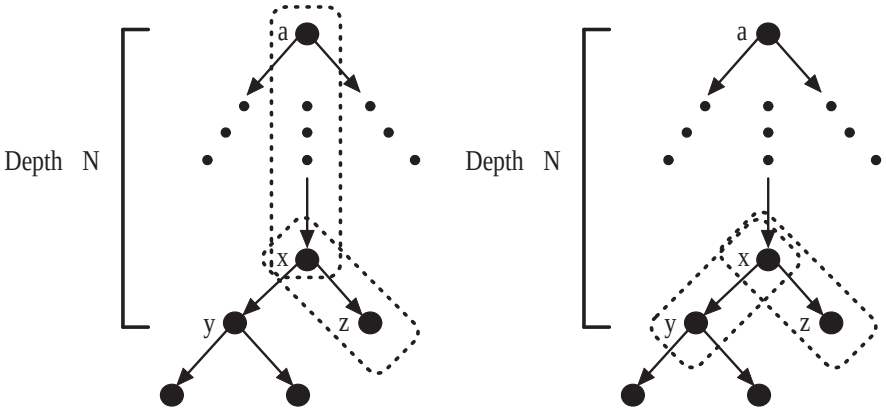


Figure 4.9: **Reconstructing configurations using symmetric configuration differences.** For reconstructing states using only the difference $d(v(c_1), v(c_2))$ of a configuration vector and its predecessor, all states lying on the path from a to z must be reconstructed (left). When using the symmetric difference $sd(v(c_1), v(c_2))$, starting from y we can directly reconstruct x and then z (right).

z, i.e., we have to reconstruct z in order to be able to compute $next(z)$. If for each state we have only stored the difference to the predecessor, then we must restore z by traversing the whole path from the starting state a to z. If the path length N is large, the time needed to reconstruct states can dominate the overall exploration time. Therefore, instead of storing the difference $d(v(c), v(c'))$ at the entry for $v(c')$, we store the *symmetric difference* $sd(v(c), v(c'))$ at $v(c')$, which is defined as

$$\begin{aligned} sd(v(c), v(c')) \quad &\in \quad (N \times Labels \times Labels)^* \times (D_{loc} \times D \times D)^* \\ sd(v(c), v(c')) \quad &= \quad \langle (j_1, pc_{j_1}, pc'_{j_1}), \cdots, (j_k, pc_{j_k}, pc'_{j_k}), \\ &\qquad (l_{i_1}, \sigma(l_{i_1}), \sigma'(l_{i_1})), \cdots, (l_{i_l}, \sigma(l_{i_l}), \sigma'(l_{i_l}))\rangle \\ &\text{s.t.} \quad j \in \{j_1, \cdots, j_k\} \Leftrightarrow pc_j \neq pc'_j \\ &\wedge \quad i \in \{i_1, \cdots, i_l\} \Leftrightarrow \sigma(l_i) \neq \sigma'(l_i). \end{aligned}$$

The symmetric difference $sd(v(c), v(c'))$ of two configuration vectors contains both the program counters and memory locations of $v(c)$ and also the program counters and memory locations of $v(c')$ that are different from each other. Therefore, the symmetric difference $sd(v(c), v(c'))$ is $\frac{1}{3}$ larger than $d(v(c), v(c'))$. However, the moderate increase in the amount of memory needed for storing one state allows a much faster restoring of states as one can see on the right side of fig. 4.9. If y is the last state that has been expanded, and the next state to expand is z, then from y we can reconstruct x and from x we can reconstruct z, avoiding the time consuming reconstruction of all states lying on the path from a to z. Situations similar to the one depicted in fig. 4.9 occur quite

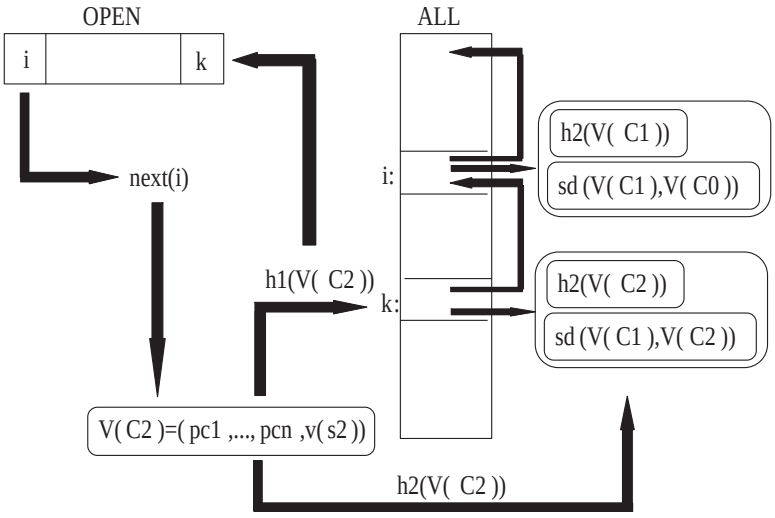


Figure 4.10: **Organization of *All* using configuration differences.** The set *All* organized as a hashing table. Instead of storing complete states, only hash signatures computed by a second hash function $h2$ are stored together with the symmetric difference $sd(v(c_1), v(c_2))$ between a configuration vector $v(c_2)$ and its predecessor $v(c_1)$.

often during state space exploration, especially in breadth-first search, where the next state to expand is almost always a neighbor state.

The overall organization of the set *All* when storing only hash signatures and differences between states is shown in fig. 4.10. Since the index i is extracted from *Open*, the next configuration vector to expand is the configuration vector stored at position i in the hashing table representing the set *All*. The configuration is reconstructed using the information in the hashing table, and its successors $next(v(c1))$ are generated, among other things $v(c2)$. To check if $v(c2) \in All$, we compute the index $k = h1(v(c2))$ and the hashing signature $h2(v(c2))$. If at position k there is already an entry with the signature $h2(v(c2))$, we assume that $v(c2)$ has already been generated before, therefore we do not insert $v(c2)$ into *Open* and *All* again. Otherwise, we insert at position k the hashing signature $h2(v(c2))$ and the symmetric difference $sd(v(c1), v(c2))$, and we set the predecessor link to i. To get an impression of the memory savings that are possible using the described approach, consider the following part of a program:

```
int N=1000; int a[N];
void main() {
 for(int i=0; i<1000; i++) {
   a[i] = 0;
   ...
}
```

The program declares an array of N integers. If one integer is 4 byte large, then the memory needed to represent one configuration is around $4N$ bytes. If $N = 10^4$ and if 256 MByte are available for the hashing table, then at most 6400 states can be explored if we store complete states. Contrary to this, when using the storing scheme based on hash compaction, the memory needed to store one state is almost constant, because in most cases two consecutive configurations differ only in one program counter and one memory location. In addition to the changed program counter and memory location[1] we need 4 byte for the predecessor link and 4 byte for the hashing signature, i.e., to store one state we need around $6 \cdot 4 = 24$ byte. Thus we can store around 10^7 states in the hashing table, i.e., more than 10^3 times more states than if we store complete states. The experimental results in sect. 4.1.4 will provide evidence on the effectiveness of the presented approximated duplicate detection.

4.1.3 State Storage Reduction

While in the previous section we have presented a method that aims at reducing the memory needed to store one state, in this section we will describe an approach that aims at reducing the number of states to store. To illustrate the idea of the reduction consider the following part of a program:

```
void main() {
   ...
   L0: x = 0; y = 0; z = 0;
   L1: x = symcpp_nondet(0,1);
   L2: x = x + 1;
   L3: y = 2*x + y + 1;
   L4: z = z - y;
   L5: y = symcpp_nondet(0,1);
   ...
}
```

When a configuration reaches label L2, our state space exploration algorithm would generate and store successively 3 configurations, one after executing the statement at L2, one after executing the statement at L3 and one after executing the statement at L4. However, if between L2 and L4 only one thread is active and if the search goal does not contain x, y and z, then it is not necessary to store the intermediate states at L2, L3 and L4. The left side of fig. 4.11 shows a part of the state graph of the above program. The state a corresponds to a configuration whose next statement to execute is the statement at L1. Because of the nondeterminism of this statement, two successor states $b1$ and $c1$ are generated. The states $b1$, $b2$ and $b3$ resp. $c1$, $c2$ and $c3$ correspond to the states that are reached after executing the statements at L2, L3 and L4. Finally, the states b and c correspond to states whose next statement to execute is the statement at L5. If

[1] We assume that a program counter and a memory location can be represented using 4 bytes.

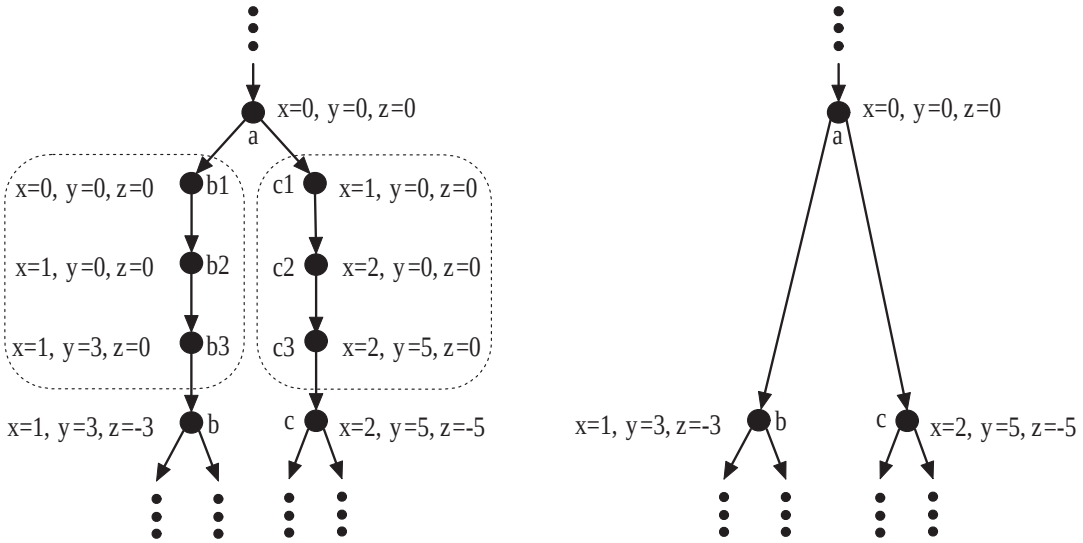


Figure 4.11: **Intermediate states.** The states $b1$, $b2$ and $b3$ resp. $c1$, $c2$ and $c3$ need not be stored if the search goal does not depend on x, y or z, because they all have only a single successor state. Therefore, it would be sufficient to store only states b and c.

the search goal does not contain variables that can be changed by the statements at L2, L3 and L4, instead of storing the states $b1$, $b2$ and $b3$ resp. $c1$, $c2$ and $c3$, it is sufficient to store only the states a, b and c and the difference between these states. The reason for this is simple: if state a does not fulfill the search goal, then none of the states $b1$, $b2$ and $b3$ resp. $c1$, $c2$ and $c3$ can fulfill the search goal. Furthermore, if we later find another path that leads to one of the states $b1$, $b2$ and $b3$ resp. $c1$, $c2$ and $c3$, then we will detect a duplicate of state b resp. state c, because there is no nondeterminism on the path between $b1$ and b resp. $c1$ and c.

We can achieve the reduction described above by defining a new successor function $next'$ that computes the set of successor configuration vectors $next'(v(c))$ for a configuration vector $v(c)$. For two configuration vectors $v(c), v(c')$, let $change(v(c), v(c'))$ denote the set of locations that are different in $v(c)$ and $v(c')$, and let EFq be the search goal. A possible definition of $next'$ could be

$$next'(v(c)) = \left\{ v(c_i^{n_i}) \;\middle|\; \begin{array}{l} next(v(c)) = \{v(c_1^1), \cdots, v(c_n^1)\} \\ \wedge \forall i \in \{1, \cdots, n\} : \forall j \in \{1, \cdots, n_i - 1\} : \\ \quad next(v(c_i^j)) = \{v(c_i^{j+1})\} \\ \quad \wedge\ change(v(c_i^j), next(v(c_i^{j+1}))) \cap varloc(Vars(q)) = \emptyset \\ \wedge \forall i \in \{1, \cdots, n\} : |next(v(c_i^{n_i}))| > 1 \end{array} \right\}$$

However, a serious problem of the above definition is that an algorithm computing the function $next'$ can be nonterminating since there can be infinite paths

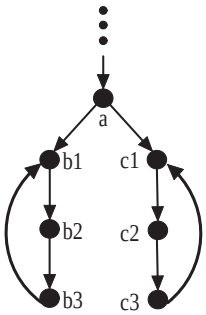


Figure 4.12: **Cycles in the state graph without exits.** Two cycles $b_1 \to b_2 \to b_3 \to b1$ and $c_1 \to c_2 \to c_3 \to c_1$ without exits.

$v(c_1), next(v(c_1)), \cdots, next(v(c_n)) = v(c_1), \cdots$, i.e., paths that contain loops. For instance, consider the following part of a program:

```
void main() {
   ...
   L0: x = symcpp_nondet(0,1);
   L1: y = 1;
   L2: z = 2;
   L3: goto L1;
   ...
}
```

Figure 4.12 shows a part of the state graph of the above program. Since at label L3 there is a goto statement, there exists e.g. the infinite path $b1 \to b2 \to b3 \to b1 \to ...$, i.e., an algorithm computing $next'$ would be nonterminating since for no state b_i the condition $|next(b_i)| > 1$ is true. To avoid nonterminating behavior, we modify the definition of $next'$ s.t. no such infinite sequences of configurations can occur during the computation of $next'$. Since the number of statements of a program is finite, an infinite path executes at least one statement infinitely many times. Since the statements are totally ordered, to execute a statement infinitely many times there must be at least one transition $c \to c'$ with $c = (pc_1, \cdots, pc_i, \cdots, pc_n, \sigma)$, $c' = (pc_1, \cdots, pc_i', \cdots, \sigma')$ s.t. $pc_i' \leq pc_i$. We call such a transition a *backjump*. Based on this, we can now define $next'$ as follows:

$$next'(v(c)) = \left\{ v(c_i^{n_i}) \;\middle|\; \begin{array}{l} next(v(c)) = \{v(c_1^1), \cdots, v(c_n^1)\} \\ \wedge \forall i \in \{1, \cdots, n\} : \forall j \in \{1, \cdots, n_i - 1\} : \\ \quad next(v(c_i^j)) = \{v(c_i^{j+1})\} \\ \quad \wedge\ change(v(c_i^j), next(v(c_i^{j+1}))) \cap varloc(Vars(q)) = \emptyset \\ \wedge \forall i \in \{1, \cdots, n\} : \\ \quad (|next(v(c_i^{n_i}))| > 1) \ \vee\ (c_i^{n_i - 1} \to c_i^{n_i} \text{ is a backjump}) \end{array} \right\}$$

When using the function $next'$ instead of $next$, much fewer states are generated and stored. For instance, for the program whose unreduced state graph is shown on the left side of fig. 4.11, using $next'$ instead of $next$ would produce only the states depicted on the right side of fig. 4.11. Furthermore, $next'$ always terminates due to the specified treatment of backjumps. However, there is still the potential to optimize the function $next'$ further. The reason for this lies in the fact that most loops in programs are terminating. For instance, consider the following part of a program:

```
int a[1000]; int i;
void main() {
   ...
   L0: i = symcpp_nondet(0,1);
   L1: for (i=0; i < 1000; i++) {
   L2:  a[i] = i; }
   L3: i = symcpp_nondet(0,1);
   ...
}
```

The part of the state graph representing the loop of the above program is depicted on the left side of fig. 4.13. Using the function $next'$, only the black states are generated and stored, the intermediate grey states are omitted, i.e., the function $next'$ reduces the number of generated states only from $2N$ to N, since there are N backjumps in the depicted sequence. However, since the sequence of all $2N$ states is completely deterministic and contains no cycle, for this sequence it would be sufficient to store only the last state at label L3, since this state has more than one successor state. To achieve this, we can change $next'$ in the following way: We introduce a number $maxtrans \in \mathbb{N}$ that denotes the maximal number of internal transitions the function $next'$ is allowed to compute. If a deterministic sequence is longer than $maxtrans$, then this sequence is interrupted at the next backjump transition:

$$next'(v(c)) = \left\{ v(c_i^{n_i}) \;\middle|\; \begin{array}{l} next(v(c)) = \{v(c_1^1), \cdots, v(c_n^1)\} \\ \wedge \forall i \in \{1, \cdots, n\} : \forall j \in \{1, \cdots, n_i - 1\} : \\ \quad next(v(c_i^j)) = \{v(c_i^{j+1})\} \\ \quad \wedge\ change(v(c_i^j), next(v(c_i^{j+1}))) \cap varloc(Vars(q)) = \emptyset \\ \wedge \forall i \in \{1, \cdots, n\} : \\ \quad (|next(v(c_i^{n_i}))| > 1) \\ \quad \vee\ (n_i \geq maxtrans\ \wedge\ c_i^{n_i - 1} \rightarrow c_i^{n_i} \text{ is a backjump}) \end{array} \right\}$$

Contrary to the previous definition of $next'$, with this definition a deterministic sequence of transitions is only interrupted when it is longer than $maxtrans$. Therefore, for the program of fig. 4.13, if we choose $maxtrans$ s.t. $maxtrans > N$, then only the state at label L3 will be generated, as can be seen on the right side of fig. 4.13. Additionally, by restricting the maximal number of internal transitions that can be computed by $next'$, $next'$ is guaranteed to terminate. The only drawback of the above definition is that

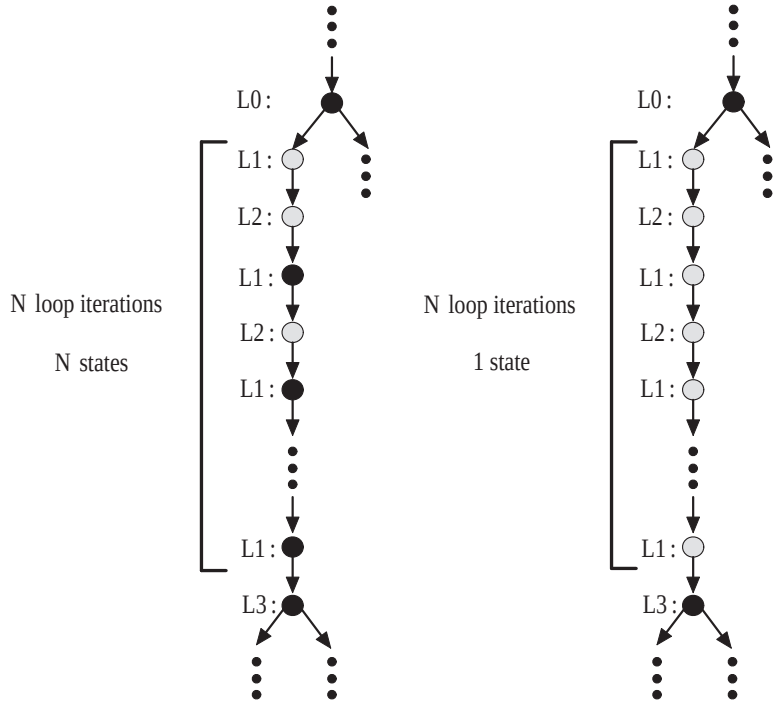


Figure 4.13: **Next reductions using backjumps and counters.** If deterministic state sequences are interrupted after each backjump, for the depicted state sequence N states would be stored (left). Contrary to this, interrupting deterministic state sequences only after $maxtrans$ transitions, only 1 state would be stored (if $maxtrans > 2N$).

$next'$ computes for every nonterminating loop at least $maxtrans$ transitions, i.e., the computation time of $next'$ can be larger for nonterminating loops than if we generate a new state for each backjump. However, since non-terminating loops are very seldom, in practice we choose large values for $maxtrans$. The experimental results in sect. 4.1.4 will provide evidence on the effectiveness of the presented treatment of the successor function $next$.

4.1.4 Experimental Results

To evaluate the effectiveness of the described optimizations presented in the previous sections, we apply them to our collection of test programs. For each of the test programs we define a reachable property. Furthermore, we restricted the possible values of input variables s.t. for each program we were able to compute a witness with the simple algorithm described in sect. 4.1. Allowing large domains of input variables is a problem that is treated separately in sect. 4.2. For each program, we measure the number of generated

states, the average number of bytes stored per state, and the overall search time using both DFS with a depth limit and BFS. We perform these measures with 5 different configurations: The simple algorithm from sect. 4.1 (Simple), the algorithm using the canonization function from sect. 4.1.1 (HS), the algorithm using the approximated duplicate detection from sect. 4.1.2 (AD), the algorithm using the state storage reduction from sect. 4.1.3 (SSR), and a configuration which applies all these optimizations together (All).

PBX	Simple	HS	AD	SSR	All
#states (BFS)	217000	96000	217000	2440	1230
#states (DFS)	243000	114000	242000	2490	1260
bytes per state (BFS)	920	925	24	920	42
bytes per state (DFS)	934	946	24	934	42
time (BFS)	206	98	23	11	4
time (DFS)	238	116	25	12	5

Figure 4.14: **Results for PBX.**

SMS	Simple	HS	AD	SSR	All
#states (BFS)	67000	48000	67000	930	760
#states (DFS)	62000	47000	62000	916	750
bytes per state (BFS)	2930	2942	24	2930	56
bytes per state (DFS)	3015	3022	24	3015	56
time (BFS)	228	172	29	9	2
time (DFS)	214	170	27	8	2

Figure 4.15: **Results for SMS**

Dishwasher	Simple	HS	AD	SSR	All
#states (BFS)	181000	97000	181000	1800	1060
#states (DFS)	193000	101000	193000	1910	1100
bytes per state (BFS)	891	894	24	891	38
bytes per state (DFS)	902	906	24	902	38
time (BFS)	192	99	20	10	2
time (DFS)	208	105	21	11	2

Figure 4.16: **Results for Dishwasher**

CANBus	Simple	HS	AD	SSR	All
#states (BFS)	95000	82000	95000	1120	1070
#states (DFS)	72000	67000	72000	1030	995
bytes per state (BFS)	1812	1812	24	1812	48
bytes per state (DFS)	1828	1832	24	1828	48
time (BFS)	242	212	26	8	3
time (DFS)	220	206	23	7	3

Figure 4.17: **Results for CANBus**

ARCS	Simple	HS	AD	SSR	All
#states (BFS)	103000	91000	103000	970	960
#states (DFS)	123000	116000	123000	1090	1070
bytes per state (BFS)	2130	2142	24	2130	52
bytes per state (DFS)	2268	2292	24	2268	52
time (BFS)	248	229	29	8	2
time (DFS)	282	273	32	10	3

Figure 4.18: **Results for ARCS**

Elevator	Simple	HS	AD	SSR	All
#states (BFS)	227000	182000	227000	2100	2010
#states (DFS)	224000	181000	224000	2080	1990
bytes per state (BFS)	765	765	24	765	38
bytes per state (DFS)	771	771	24	771	38
time (BFS)	248	229	29	8	2
time (DFS)	282	273	32	10	3

Figure 4.19: **Results for Elevator**

Pacemaker	Simple	HS	AD	SSR	All
#states (BFS)	156000	142000	156000	1350	1330
#states (DFS)	169000	158000	158000	1410	1400
bytes per state (BFS)	1290	1298	24	1290	56
bytes per state (DFS)	1312	1316	24	1312	56
time (BFS)	254	240	32	10	3
time (DFS)	275	267	33	10	3

Figure 4.20: **Results for Pacemaker**

HomeHeating	Simple	HS	AD	SSR	All
#states (BFS)	173000	144000	173000	1670	1490
#states (DFS)	171000	143000	171000	1660	1480
bytes per state (BFS)	826	834	24	826	42
bytes per state (DFS)	844	852	24	844	42
time (BFS)	190	167	23	9	2
time (DFS)	182	166	22	9	2

Figure 4.21: **Results for HomeHeating**

HomeAlarm	Simple	HS	AD	SSR	All
#states (BFS)	144000	141000	144000	830	820
#states (DFS)	129000	128000	129000	770	760
bytes per state (BFS)	730	732	24	730	38
bytes per state (DFS)	733	734	24	733	38
time (BFS)	152	150	19	8	1
time (DFS)	144	143	17	7	1

Figure 4.22: **Results for HomeAlarm**

TCU	Simple	HS	AD	SSR	All
#states (BFS)	125000	117000	125000	930	910
#states (DFS)	129000	123000	129000	970	960
bytes per state (BFS)	1410	1414	24	1410	60
bytes per state (DFS)	1425	1428	24	1425	60
time (BFS)	217	210	27	7	2
time (DFS)	231	226	29	8	2

Figure 4.23: **Results for TCU**

In summary one can say that each optimization we have presented improves the performance of the state space exploration algorithm, more or less independent from a specific search order (BFS or DFS) or a specific test program. However, the magnitude of performance improvements differ for the different optimizations. In the best case, state canonization for detecting symmetric states halves the number of generated states (PBX). Much larger performance improvements are realized by the state storage reduction. When state storage reduction is applied, only 0.5% (HomeAlarm) to 1.3% (SMS) of the number of states are generated compared to the number of states generated by the simple algorithm. Applying approximated duplicate detection reduces the consumed memory per states to 1.8% (SMS) to 5% (HomeAlarm) of the memory needed by the simple algorithm. The experiments show that the savings gained with approximated duplicate detection grow with the unreduced state size. Furthermore, both approximated duplicate detection and state storage reduction yields considerable reductions of the exploration time. However, the greatest reductions are gained when applying all optimizations together, which shows that the presented optimizations are more or less orthogonal, i.e., applying one optimization does not have a negative influence for the other optimizations.

Nevertheless, as described above, to be able to successfully find witnesses of formulas EFq for the test programs required that we reduce the domains of input variables manually, a task that can be complicated, requires knowledge of the programs and which is error-prone. Therefore, in the next section we will present an alternative explicit-symbolic state representation that allows us to successfully perform state space explorations without these manual tasks.

4.2 Explicit-Symbolic State Representation

A major source of complexity when performing a state space exploration of programs are the normally large domains of numerical variables. As an example, consider the following program:

```
int min(int x1, int x2, int x3) { ... }
int x1,x2,x3,y;
int error=0;

void main() {
  L1: x1 = symcpp_nondet(-50000,50000);
  L2: x2 = symcpp_nondet(-50000,50000);
  L3: x3 = symcpp_nondet(-50000,50000);
  L4: y = min(x1,x2,x3);

  // check result
  if (y == x1) error = (x2 < x1) || (x3 < x1);
  else if (y == x2) error = (x1 < x2) || (x3 < x2);
  else if (y == x3) error = (x1 < x3) || (x2 < x3);
  else error = 1;
  // end check result
}
```

The program implements a function `min` that should compute the minimum of three integers $x1$, $x2$ and $x3$. In the main function we want to check if the implemented function `min` works correctly when $x1, x2, x3 \in [-5 \cdot 10^4, 5 \cdot 10^4]$. Therefore, we choose nondeterministically values for $x1$, $x2$ and $x3$ in the range $[-5 \cdot 10^4, 5 \cdot 10^4]$, compute the value `y=min(x1,x2,x3)` and check if y is indeed the minimum of $x1$, $x2$ and $x3$. If y is not the minimum, then the variable `error` is set to 1. Therefore, a witness for the formula $EF\ error = 1$ will contain values for $x1$, $x2$ and $x3$ for which the function `min` works incorrectly. Unfortunately, even this small program has a very large state space of more than 10^{15} states, since each of the variables x_i can have one of 10^5 distinct values. If the function `min` works incorrectly for a single combination of values $x1$,$x2$ and $x3$, due to the tremendous number of states of the program it is very unlikely that this combination of values can be found within reasonable time.

To cope with this problem, in this section we will define a composite explicit-symbolic state representation that allows us to handle entire sets of states rather than only individual states as with the pure explicit representation. As described in sect. 2.1, a symbolic representation of all states in a set can be much more compact than an individual representation of all states in the set. Unfortunately, symbolic representations also have serious drawbacks compared to explicit representations. For instance, it is sometimes hard or impossible to represent certain programming constructs fully symbolically, as it is the case for e.g. pointers and dynamic object creation. Furthermore, when a symbolic representation becomes too large, then often necessary symbolic computations using these representations cannot be performed efficiently any more. Against this background we decided to use a composite explicit-symbolic state representation, trying to combine the succinctness of symbolic representations with the flexibility of explicit representations. More precisely, our approach works as follows:

- We use arithmetic and boolean expressions as symbolic representations for the (possibly very large) set of concrete values fulfilling these expressions, i.e., the sets of possible values of numeric or boolean variables are represented by symbolic expressions.
- Since reasonable programs can easily contain hundreds or thousands of variables, e.g. if large arrays are used, we represent only some of these variables symbolically, thus preventing too large symbolic representations that cannot be handled efficiently any more. The variables which are handled symbolically are those that directly or indirectly depend on variables that occur in nondeterministic assignments `x := nd`. Depending on the computations performed on these variables, the symbolically represented part of the program variables can grow and shrink in different states.
- Pointer variables are always represented explicitly, which allows a simple treatment of mechanisms like pointer aliasing, dynamic object creation or object destruction.

As mentioned above, we will use expressions containing variables from a set I as a symbolic representation for the (possibly very large) set of concrete values fulfilling these expressions. Given an infinite set $I = \{I_1, I_2, \cdots\}$ of variables with domain D_{int}, with $Exp(I)$ we denote the set of expressions that can be build using constants from int_c, arithmetic operations from AOp, arithmetic relations from $ARel$ and boolean operations from BOp. For instance,

$$27, \quad 2 + I_1, \quad \text{or} \quad (I_2 + 25 * I_5/3 > 2 * I_4) \; and \; (I_6 == I_3/2)$$

are expressions from $Exp(I)$. Furthermore, with $BExp(I) \subset Exp(I)$ we denote the set of boolean expressions over variables in I. A valuation $val : I \to D_{int}$ is a mapping from variables to values, and the set of all valuations is denoted with VAL. The semantics of an expression $e \in Exp(I)$ is a function $[\![e]\!] : VAL \to (D_{int} \cup D_{bool})$ with the usual meaning. For instance, if $val(I_1) = 4$, then $[\![3+I_1]\!](val) = 7$. Furthermore, for a boolean expression $b \in BExp(I)$,

$$[\![b]\!] = \{val \in VAL \mid [\![b]\!](val) = true\}$$

denotes the set of all valuations val s.t. the semantics of b evaluates to $true$. Additionally, we say that a boolean expression $b \in BExp(I)$ is *satisfiable* if $[\![b]\!] \neq \emptyset$. As defined in sect. 3.2, a state σ is a mapping from the set of locations D_{loc} to the set of values D. However, in the composite state representation we are dealing with so-called *symbolic states*. A symbolic state $s = (\hat{\sigma}, pcond)$ is a tuple consisting of a mapping

$$\hat{\sigma} : D_{loc} \to D \cup Exp(I)$$

that assigns to each location either an element from $Exp(I)$ or a value $d \in D$, and a so-called *path condition* $pcond \in BExp(I)$. With $\hat{\Sigma}$ we denote the set of all symbolic

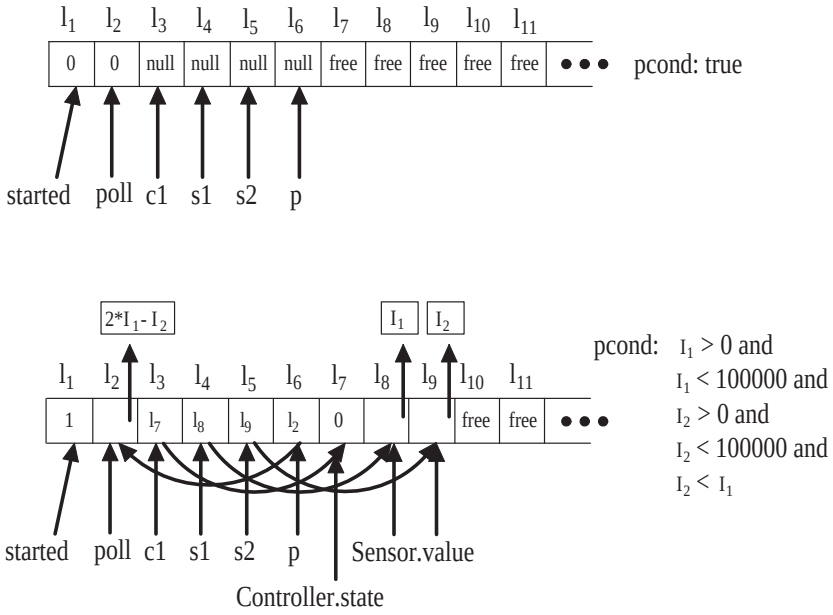


Figure 4.24: **Symbolic states.** The upper symbolic state does not map any location to a symbolic expression, and the path condition is simply true. Contrary to that, the lower symbolic state maps l_8 to to the expression I_1, l_9 to I_2 and l_2 to $2 * I_1 - I_2$. Furthermore, for this state the path condition is $I_1 > 0 \wedge I_1 < 100000 \wedge I_2 > 0 \wedge I_2 < 100000 \wedge I_2 < I_1$.

states, and we define $\hat{\Sigma}_f = \hat{\Sigma} \cup \{fail\}$. Figure 4.24 shows an example of a symbolic state. While the upper symbolic state does not map any location to a symbolic expression and has simply *true* as the path condition, the lower symbolic state maps l_8 to the expression I_1, l_9 to I_2 and l_2 to $2 * I_1 - I_2$, and it has the path condition

$$I_1 > 0 \wedge I_1 < 100000 \wedge I_2 > 0 \wedge I_2 < 100000 \wedge I_2 < I_1.$$

Intuitively, the lower symbolic state represents all those states $\sigma \in \Sigma$ in which all variables occurring in expressions $e \in Ran(\hat{\sigma})$ are instantiated with values s.t. the associated path condition evaluates to *true*. In this example, the depicted symbolic state represents more than $9 * 10^9$ simple states, since $99999 * 99998$ combinations of values for I_1 and I_2 fulfill the given path condition.
Given a symbolic state $s = (\hat{\sigma}, pcond)$ and a valuation $val \in VAL$, $\hat{\sigma}[val] \in \Sigma$ denotes a state with

$$\hat{\sigma}[val](l) = \begin{cases} \hat{\sigma}(l) & \text{if } \hat{\sigma(l)} \in D \\ [\![\hat{\sigma}(l)]\!](val) & \text{otherwise} \end{cases}$$

For instance, for the symbolic state $s = (\hat{\sigma}, pcond)$ depicted in fig. 4.24 and a valuation val with $val(I_1) = 2$ and $val(I_2) = 3$, $\hat{\sigma}[val]$ denotes a state with $\hat{\sigma}[val](l_8) = 2$,

$\hat{\sigma}[val](l_9) = 3$, $\hat{\sigma}[val](l_2) = 2 * 2 - 3 = 1$ and $\hat{\sigma}[val](l) = \hat{\sigma}(l)$ for all $l \in D_{loc} \setminus \{l_2, l_8, l_9\}$. Furthermore, for a symbolic state $s = (\hat{\sigma}, pcond)$, with $[\![s]\!]$ we denote the set of states that is defined as

$$[\![s]\!] = \{\hat{\sigma}[val] \mid val \in [\![pcond]\!]\},$$

i.e., $[\![s]\!]$ denotes the set of all states in Σ that we get if we evaluate all expressions assigned to locations in $\hat{\sigma}$ under all valuations val s.t. the path condition $pcond$ evaluates to true.

In sect. 3.2 we have defined the semantics of expressions and statements of C_{min} programs w.r.t. states $\sigma \in \Sigma$. We now define an alternative *symbolic semantics* that operates on symbolic states directly. To this end, we assume for every operation $op \in AOp \cup ARel \cup BOp$ a corresponding *symbolic operation* op_s that syntactically constructs compound expressions from its operands, i.e., given two expressions $e_1, e_2 \in Exp(I)$, $e_1\ op_s\ e_2$ denotes the compound expression '$e_1\ op\ e_2$'. Given an expression $e \in Exp$ and a symbolic state $s = (\hat{\sigma}, pcond) \in \hat{\Sigma}$, the symbolic semantics is a mapping

$$[\![e]\!]_s : \hat{\Sigma} \to D \cup Exp(I) \cup fail.$$

For expressions e of kind

$$e \equiv c, \quad e \equiv fail, \quad e \in LExp, \quad e \equiv \&v, \quad e \equiv \&v \to m, \quad e \equiv e_1\ op\ e_2, op \in PRel$$

the symbolic semantics is defined analogously to the semantics in sect. 3.2. The other cases are defined as follows:

- (Nondeterministic choice): if $e \equiv nd$ then
  $[\![e]\!]_s(s) = I_k$, whereby $I_k \in I$ and $I_k \notin Vars(s)$.

- (Arithmetic expressions): if $e \equiv e_1\ op\ e_2, op \in AOp$, then
  $$[\![e]\!]_s(s) = \begin{cases} [\![e_1]\!]_s(s)\ op_s\ [\![e_2]\!]_s(s) & \text{if } \bigvee_{i=1}^{2}([\![e_i]\!]_s(s) \in Exp(I)) \\ [\![e_1]\!]_s(s)\ op\ [\![e_2]\!]_s(s) & \text{otherwise} \end{cases}$$

- (Conditional expressions): if $e \equiv e_1?e_2 : e_3$, then
  $$[\![e]\!]_s(s) = \begin{cases} [\![e_1]\!]_s(s)\ ?_s\ [\![e_2]\!]_s(s)\ :_s\ [\![e_3]\!]_s(s) & \text{if } \bigvee_{i=1}^{3}([\![e_i]\!]_s(s) \in Exp(I)) \\ [\![e_1]\!]_s(s)\ ?\ [\![e_2]\!]_s(s)\ :\ [\![e_3]\!]_s(s) & \text{otherwise} \end{cases}$$

- (Arithmetic relations): if $e \equiv e_1\ op\ e_2, op \in ARel$, then
  $$[\![e]\!]_s(s) = \begin{cases} [\![e_1]\!]_s(s)\ op_s\ [\![e_2]\!]_s(s) & \text{if } \bigvee_{i=1}^{2}([\![e_i]\!]_s(s) \in Exp(I)) \\ [\![e_1]\!]_s(s)\ op\ [\![e_2]\!]_s(s) & \text{otherwise} \end{cases}$$

- (Boolean expressions):

$$[\![e_1 \text{ and } e_2]\!]_s(s) = \begin{cases} [\![e_1]\!]_s(s) \; and_s \; [\![e_2]\!]_s(s) & \text{if } \bigvee_{i=1}^{2}([\![e_i]\!]_s(s) \in Exp(I)) \\ [\![e_1]\!]_s(s) \; \wedge \; [\![e_2]\!]_s(s) & \text{otherwise} \end{cases}$$

$$[\![e_1 \text{ or } e_2]\!]_s(s) = \begin{cases} [\![e_1]\!]_s(s) \; or_s \; [\![e_2]\!]_s(s) & \text{if } \bigvee_{i=1}^{2}([\![e_i]\!]_s(s) \in Exp(I)) \\ [\![e_1]\!]_s(s) \; \vee \; [\![e_2]\!]_s(s) & \text{otherwise} \end{cases}$$

$$[\![\text{not } e_1]\!]_s(s) = \begin{cases} not_s[\![e_1]\!]_s(s) & \text{if } [\![e_1]\!]_s(s) \in Exp(I) \\ \neg[\![e_1]\!]_s(s) & \text{otherwise} \end{cases}$$

In contrast to the semantics of expressions defined in sect. 3.2 which is in general a set of values, the symbolic semantics either computes a value from D or $fail$ or an expression from $Exp(I)$. For instance, consider the lower symbolic state $s = (\hat{\sigma}, pcond)$ depicted in fig. 4.24 and the expression $e \equiv poll > s1 \rightarrow value + s2 \rightarrow value$. Then

$$[\![poll > s1 \rightarrow value + s2 \rightarrow value]\!]_s(s) = (2 * I_1 - I_2) > (I_1 + I_2),$$

since $\hat{\sigma}(l_2) = 2 * I_1 - I_2$, $s1 \rightarrow value$ points at l_8, $s2 \rightarrow value$ points at l_9, $\hat{\sigma}(l_8) = I_1$ and $\hat{\sigma}(l_9) = I_2$. Furthermore, since

$$pcond = I_1 > 0 \wedge I_1 < 100000 \wedge I_2 > 0 \wedge I_2 < 100000 \wedge I_2 < I_1,$$

we can find both valuations val_1 and val_2 of I_1 and I_2 s.t. $[\![pcond]\!](val_1) = true$, $[\![pcond]\!](val_2) = true$, $[\![e]\!](\hat{\sigma}[val_1]) = true$ and $[\![e]\!](\hat{\sigma}[val_2]) = false$, for instance with $val_1(I_1) = val_2(I_1) = 3$, $val_1(I_2) = 1$ and $val_2(I_2) = 2$. Therefore, the symbolic semantics of the expression e in the symbolic state s yields an expression which is a symbolic representation for the set {true,false}.

After defining the symbolic semantics of expressions, we can now define the symbolic semantics of statements. Updates of symbolic states are defined analogously to updates of simple states. Furthermore, a symbolic thread configuration is a tuple

$$tc = (pc, s = (\hat{\sigma}, pcond)).$$

A symbolic thread transition $(pc, s) \rightarrow (pc', s')$ with $s = (\hat{\sigma}, pcond)$ and $s' = (\hat{\sigma}', pcond')$ between two symbolic thread configurations describes a computation step of one thread corresponding to one statement. For statements stm of kind

$$stm \equiv new(v) \quad \text{or} \quad stm \equiv delete(v),$$

the symbolic thread transition is defined analogously to simple thread transitions. The other cases are defined as follows:

- (Concurrent assignment):

$$(pc, s) \rightarrow (pc', s') \Leftrightarrow \begin{cases} stm(pc) = [e_l^1 := e_1, \cdots, e_l^n := e_n] \\ \wedge\, pc' = next(pc) \;\wedge pcond' = pcond \\ \wedge\, \hat{\sigma}' = \hat{\sigma}[[\![adr(e_l^1)]\!]_s(s) := [\![e_1]\!]_s(s), \cdots, [\![adr(e_l^n)]\!]_s(s) := [\![e_n]\!]_s(s)] \end{cases}$$

- (Synchronization):

$$(pc,s) \rightarrow (pc',s') \Leftrightarrow \begin{cases} stm(pc) = \texttt{await}(e,[e_l^1 := e_1, \cdots, e_l^n := e_n]) \\ \wedge\ [\![pcond \wedge [\![e]\!]_s(s)]\!] \neq \emptyset \\ \wedge\ pc' = next(pc)\ \wedge pcond' = pcond \wedge [\![e]\!]_s(s) \\ \wedge\ \hat{\sigma}' = \hat{\sigma}[[\![adr(e_l^1)]\!]_s(s) := [\![e_1]\!]_s(s), \cdots, [\![adr(e_l^n)]\!]_s(s) := [\![e_n]\!]_s(s)] \end{cases}$$

- (Branching):

$$(pc,s) \rightarrow (pc',s') \Leftrightarrow \begin{cases} stm(pc) = \texttt{jump}(e, lab_1, lab_2)\ \wedge\ \hat{\sigma}' = \hat{\sigma}\ \wedge \\ \quad ((\![pcond \wedge [\![e]\!]_s(s)]\!] \neq \emptyset \\ \quad \wedge\ pc' = lab_1\ \wedge pcond' = pcond \wedge [\![e]\!]_s(s)) \\ \vee\ ([\![pcond \wedge \neg[\![e]\!]_s(s)]\!] \neq \emptyset \\ \quad \wedge\ pc' = lab_2\ \wedge\ pcond' = pcond \wedge \neg[\![e]\!]_s(s))) \end{cases}$$

Analogously to sect. 3.2, a symbolic configuration of a program P is a tuple

$$c_s = (pc_1, \cdots, pc_n, s),$$

and $Conf_s(P)$ denotes the set of all symbolic configurations of a program P. The starting configuration is

$$c_s^0 = (first_1, \cdots, first_n, s_0 = (\hat{\sigma_0}, true)),$$

whereby $\hat{\sigma_0}$ has to fulfill the same conditions as in the normal semantics, i.e.

- $\forall v \in Vars(P) : \hat{\sigma}_0(varloc(v)) = init(v)$
- $\forall l \in D_{loc} : l \notin Ran(varloc) \Rightarrow \hat{\sigma}_0(l) = free$

. A transition $(pc_1, \cdots, pc_n, s) \rightarrow (pc_1', \cdots, pc_n', s')$ is defined as

$$(pc_1, \cdots, pc_k, \cdots, pc_n, s) \rightarrow (pc_1, \cdots, pc_k', \cdots, pc_n, s') \Leftrightarrow (pc_k, s) \rightarrow (pc_k', s').$$

Given a symbolic configuration $c_s = (pc_1, \cdots, pc_n, s)$,

$$[\![c_s]\!] = \{(pc_1, \cdots, pc_n, \sigma) \mid \sigma \in [\![s]\!]\}$$

denotes the set of simple configurations belonging to the symbolic configuration c_s. A symbolic run $r_s = \langle c_s^0, c_s^1, \cdots \rangle$ is a sequence of symbolic configurations, $Runs_s(P)$ denotes the set of all symbolic runs of a program P, and

$$[\![r_s]\!] = \{\langle c_0, c_1, \cdots \rangle \mid \forall i : c_i \in [\![c_s^i]\!]\}$$

denotes the set of all simple runs belonging to the symbolic run r_s. Given an expression $e \in BExp(BIVars)$, for a symbolic configuration $c_s^i = (pc_1, \cdots, pc_n, s_i = (\hat{\sigma_i}, pcond))$ of a symbolic run r_s we define

$$c_s^i \vDash_s e \Leftrightarrow true \in [\![[\![e]\!]_s(s_i)]\!].$$

Additionally, we define $c_s^i \not\models_s e \Leftrightarrow \neg(c_s^i \models e)$. Given a formula $EFq \in TL$, we define the relation $\models_s$ as follows:

$$P \models_s EFq \Leftrightarrow \exists r_s = \langle c_s^0, c_s^1, \cdots \rangle \in Runs_s(P), \exists i \geq 0 : c_s^i \models q.$$

After defining symbolic configurations and runs, one can now ask for the relationship between normal runs $r \in Runs(P)$ and symbolic runs $r_s \in Runs_s(P)$. The following lemma states that the two semantics are equivalent in the sense that for each simple run one can find a corresponding symbolic run s.t. each configuration c^i of the simple run is an element of the set of configurations $[\![c_s^i]\!]$ that are instantiations of the corresponding symbolic configuration c_s^i, and vice versa.

Lemma 4.6 *Let P be a C_{min} program. Then the following holds:*

1. $r \in Runs(P) \Rightarrow \exists r_s \in Runs_s(P) : r \in [\![r_s]\!]$.
2. $r_s \in Runs_s(P) \Rightarrow \exists r \in Runs(P) : r \in [\![r_s]\!]$.

Proof:
1 (by induction): Let $r = \langle c^0, c^1, \cdots, c^i \rangle \in Runs(P)$ be a run of P. We show that there exists a corresponding symbolic run $r_s = \langle c_s^0, c_s^1, \cdots, c_s^i \rangle \in Runs_s(P)$ s.t. $r \in [\![r_s]\!]$.

- Case $i = 0$. The starting configuration of P is

  $$c^0 = (first_1, \cdots, first_n, \sigma_0),$$

  and the symbolic starting configuration is

  $$c_s^0 = (first_1, \cdots, first_n, s_0 = (\hat{\sigma_0}, true)).$$

  Since the starting conditions for σ_0 and $\hat{\sigma_0}$ are the same, we have $\forall l \in D_{loc} : \sigma_0(l) = \hat{\sigma}_0(l)$, and therefore $c^0 \in [\![c_s^0]\!]$.

- Case $i + 1$: Let $r = \langle c^0, \cdots, c^i = (pc_1, \cdots, pc_n, \sigma_i) \rangle$ be a simple run of P and $r_s = \langle c_s^0, \cdots, c_s^i = (pc_1, \cdots, pc_n, (\hat{\sigma}_i, pcond_i)) \rangle$ a corresponding symbolic run. A successor configuration $c^{i+1} = (pc_1, \cdots, pc'_k, \cdots, pc_n, \sigma_{i+1})$ of c^i can be reached by one of the following cases:

  - $stm(pc_k) = [e_l^1 := e_1, \cdots, e_l^n := e_n]$. From the definition of the semantics it follows that

    $$\sigma_{i+1} \in \sigma_i[[\![adr(e_l^1)]\!](\sigma_i) := [\![e_1]\!](\sigma_i), \cdots, [\![adr(e_l^n)]\!](\sigma_i) := [\![e_n]\!](\sigma_i)]$$

    and $s_{i+1} = (\hat{\sigma}_{i+1}, pcond_{i+1})$ with

    $$\hat{\sigma}_{i+1} = \hat{\sigma}_i[[\![adr(e_l^1)]\!]_s(s_i) := [\![e_1]\!]_s(s_i), \cdots, [\![adr(e_l^n)]\!]_s(s_i) := [\![e_n]\!]_s(s_i)].$$

Because of the definition of expressions and because $\sigma_i \in [\![s_i]\!]$, for all $k \in \{1, \cdots, n\}$ we have

$$[\![adr(e_l^k)]\!](\sigma_i) = [\![adr(e_l^k)]\!]_s(s_i)$$

and

$$[\![e_k]\!](\sigma_i) \subseteq \{[\![e_k]\!]_s(s_i)(val) \mid val \in [\![pcond]\!]\}.$$

Furthermore we have $pcond_{i+1} = pcond_i$, and therefore $\sigma_{i+1} \in [\![s_{i+1}]\!]$ and $c^{i+1} \in [\![c_s^{i+1}]\!]$.

– $stm(pc_k) = \texttt{await}(e, [e_l^1 := e_1, \cdots, e_l^n := e_n])$. Since

$$[\![e]\!](\sigma_i) \subseteq \{[\![e]\!]_s(s_i)(val) \mid val \in [\![pcond]\!]\},$$

also from c_s^i we can execute the await-statement. Furthermore, from the definition of the semantics we know that

$$\sigma_{i+1} \in \sigma_i[[\![adr(e_l^1)]\!](\sigma_i) := [\![e_1]\!](\sigma_i), \cdots, [\![adr(e_l^n)]\!](\sigma_i) := [\![e_n]\!](\sigma_i)]$$

and $s_{i+1} = (\hat{\sigma}_{i+1}, pcond_{i+1})$ with

$$\hat{\sigma}_{i+1} = \hat{\sigma}_i[[\![adr(e_l^1)]\!]_s(s_i) := [\![e_1]\!]_s(s_i), \cdots, [\![adr(e_l^n)]\!]_s(s_i) := [\![e_n]\!]_s(s_i)].$$

Additionally, from the definition of expressions and because $\sigma_i \in [\![s_i]\!]$, we know that for all $k \in \{1, \cdots, n\}$

$$[\![adr(e_l^k)]\!](\sigma_i) = [\![adr(e_l^k)]\!]_s(s_i)$$

and

$$[\![e_k]\!](\sigma_i) \subseteq \{[\![e_k]\!]_s(s_i)(val) \mid val \in [\![pcond]\!]\}$$

holds. Moreover, we know that $pcond_{i+1} = pcond_i \wedge [\![e]\!]_s(s)$, from which we can conclude that $\sigma_{i+1} \in [\![s_{i+1}]\!]$ and $c^{i+1} \in [\![c_s^{i+1}]\!]$.

– $stm(pc_k) = \texttt{jump}(e, lab_1, lab_2)$. From the definition of expressions and because $\sigma_i \in [\![s_i]\!]$, we know that

$$[\![e]\!](\sigma_i) \subseteq \{[\![e]\!]_s(s_i)(val) \mid val \in [\![pcond]\!]\}.$$

If now $[\![e]\!](\sigma_i) = true$ and $pc'_k = lab_1$, we know that also $true \in \{[\![e]\!]_s(s_i)(val) \mid val \in [\![pcond]\!]\}$ and therefore $c^{i+1} \in [\![c_s^{i+1}]\!]$. Otherwise, if $[\![e]\!](\sigma_i) = false$ then also $false \in \{[\![e]\!]_s(s_i)(val) \mid val \in [\![pcond]\!]\}$ and therefore $c^{i+1} \in [\![c_s^{i+1}]\!]$.

– $stm(pc_k) = new(v)$ or $stm(pc_k) = delete(v)$: From the definition of the semantics we know that both σ_{i+1} resp. $\hat{\sigma}_{i+1}$ differ from σ_i resp. $\hat{\sigma}_i$ only in the newly allocated locations resp. deallocated locations. Since the values of these locations are the same both in σ_{i+1} and $\hat{\sigma}_{i+1}$ it follows immediately that $c^{i+1} \in [\![c_s^{i+1}]\!]$.

2 (by induction): Let $r_s = \langle c_s^0, c_s^1, \cdots, c_s^i \rangle \in Runs_s(P)$ be a symbolic run of P.

- Case $i = 0$. The symbolic starting configuration of P is

$$c_s^0 = (first_1, \cdots, first_n, s_0 = (\hat{\sigma_0}, true)),$$

and the starting configuration of P is

$$c^0 = (first_1, \cdots, first_n, \sigma_0).$$

Since the starting conditions for σ_0 and $\hat{\sigma_0}$ are the same, we have $\forall l \in D_{loc} : \sigma_0(l) = \hat{\sigma}_0(l)$, and therefore $c^0 \in [\![c_s^0]\!]$.

- Case $i+1$: Let $r = \langle c_s^0, \cdots, c_s^i = (pc_1, \cdots, pc_n, s_i = (\hat{\sigma}_i, pcond_i)) \rangle$ be a symbolic run and $r = \langle c^0, \cdots, c^i = (pc_1, \cdots, pc_n, \sigma_i) \rangle$ be a corresponding simple run. A successor configuration $c_s^{i+1} = (pc_1, \cdots, pc_k', \cdots, pc_n, s_{i+1} = (\hat{\sigma}_{i+1}, pcond_{i+1}))$ of c_s^i can be reached by one of the following cases:

  - $stm(pc_k) = [e_l^1 := e_1, \cdots, e_l^n := e_n]$. From the definition of the semantics it follows that $s_{i+1} = (\hat{\sigma}_{i+1}, pcond_{i+1})$ with

$$\hat{\sigma}_{i+1} = \hat{\sigma}_i[[\![adr(e_l^1)]\!]_s(s_i) := [\![e_1]\!]_s(s_i), \cdots, [\![adr(e_l^n)]\!]_s(s_i) := [\![e_n]\!]_s(s_i)].$$

    Now consider an arbitrary $\sigma_{i+1} \in [\![s_{i+1}]\!]$. Then there exists a valuation $val \in [\![pcond_{i+1}]\!]$ s.t. $\sigma_{i+1} = s[val]$. Since $pcond_{i+1} = pcond_i$ it follows that $val \in [\![pcond_i]\!]$. With $\sigma_i = s_i[val]$ we have

$$\sigma_{i+1} \in \sigma_i[[\![adr(e_l^1)]\!](\sigma_i) := [\![e_1]\!](\sigma_i), \cdots, [\![adr(e_l^n)]\!](\sigma_i) := [\![e_n]\!](\sigma_i)],$$

    i.e., there exists a transition $c^i \rightarrow c^{i+1}$ with $c^i \in [\![c_s^i]\!]$ and $c^{i+1} \in [\![c_s^{i+1}]\!]$.

  - $stm(pc_k) = \texttt{await}(e, [e_l^1 := e_1, \cdots, e_l^n := e_n])$. From the definition of the semantics it follows that $s_{i+1} = (\hat{\sigma}_{i+1}, pcond_{i+1})$ with

$$\hat{\sigma}_{i+1} = \hat{\sigma}_i[[\![adr(e_l^1)]\!]_s(s_i) := [\![e_1]\!]_s(s_i), \cdots, [\![adr(e_l^n)]\!]_s(s_i) := [\![e_n]\!]_s(s_i)]$$

    and $pcond_{i+1} = pcond_i \wedge [\![e]\!]_s(s)$. Now consider an arbitrary $\sigma_{i+1} \in [\![s_{i+1}]\!]$. Then there exists a valuation $val \in [\![pcond_{i+1}]\!]$ s.t. $\sigma_{i+1} = s[val]$. Since $pcond_i \wedge [\![e]\!]_s(s) \Rightarrow pcond_i$ it follows that $val \in [\![pcond_i]\!]$. With $\sigma_i = s_i[val]$ we have

$$\sigma_{i+1} \in \sigma_i[[\![adr(e_l^1)]\!](\sigma_i) := [\![e_1]\!](\sigma_i), \cdots, [\![adr(e_l^n)]\!](\sigma_i) := [\![e_n]\!](\sigma_i)],$$

    and we know also that $[\![e]\!](\sigma_i) = true$, i.e., there exists a transition $c^i \rightarrow c^{i+1}$ with $c^i \in [\![c_s^i]\!]$ and $c^{i+1} \in [\![c_s^{i+1}]\!]$.

- $stm(pc_k) = \texttt{jump}(e, lab_1, lab_2)$. We distinguish two cases: Either we have $pc'_k = lab_1$ and $pcond_{i+1} = pcond \wedge [\![e]\!]_s(s)$. Consider an arbitrary $\sigma_{i+1} \in [\![s_{i+1}]\!]$. Then there exists a valuation $val \in [\![pcond_{i+1}]\!]$ s.t. $\sigma_{i+1} = s[val]$. Since $pcond_i \wedge [\![e]\!]_s(s) \Rightarrow pcond_i$ it follows that $val \in [\![pcond_i]\!]$. With $\sigma_i = s_i[val]$ we have $[\![e]\!](\sigma_i) = true$ and therefore also a transition $c^i \rightarrow c^{i+1}$ with $c^i \in [\![c^i_s]\!]$ and $c^{i+1} \in [\![c^{i+1}_s]\!]$. The same argument holds for the case $pc'_k = lab_2$ and $pcond_{i+1} = pcond \wedge \neg[\![e]\!]_s(s)$.
- $stm(pc_k) = new(v)$ or $stm(pc_k) = delete(v)$: From the definition of the semantics we know that both σ_{i+1} resp. $\hat{\sigma}_{i+1}$ differ from σ_i resp. $\hat{\sigma}_i$ only in the newly allocated locations resp. deallocated locations. Since the values of these locations are the same both in σ_{i+1} and $\hat{\sigma}_{i+1}$ it follows immediately that $c^{i+1} \in [\![c^{i+1}_s]\!]$.

□

After we have proven that for each simple run one can find a corresponding symbolic run and vice versa, we can now prove the equivalence of $\vDash$ and $\vDash_s$.

Theorem 4.7 *Let P be a C_{min} program and $EFq \in TL$ be a formula. Then the following holds:*

$$P \vDash EFq \Leftrightarrow P \vDash_s EFq.$$

Proof:
$\Rightarrow$: Let $r = \langle c_0, \cdots, c_k \rangle \in Runs(P)$ be a run of P s.t. $c_k \vDash EFq$. From lemma 4.6 we know that there exists a symbolic run $r_s = \langle c^0_s, \cdots, c^k_s \rangle \in Runs_s(P)$ s.t. $c_k \in [\![c^k_s]\!]$. Hence it follows that $c^k_s \vDash_s EFq$. The other direction proceeds analogously. □

Since theorem 4.7 holds, as an alternative to the state space exploration algorithm from sect. 4.1 which is based on the original semantics we can also define an exploration algorithm that is based on the symbolic semantics. The advantage of using the symbolic semantics is that the number of symbolic states that have to be generated to reach a certain search goal is in general much smaller than the number of simple states. However, to be able to apply the state space exploration algorithm shown in fig. 4.1 using the symbolic semantics, we have to define symbolic configuration vectors, the symbolic successor function $next_s$ and the treatment of the set All of already generated states. Given a symbolic state $s = (\hat{\sigma}, pcond)$, a *symbolic state vector* $v(s)$

$$v(s) \in (D \cup Exp(I))^* \times BExp(I)$$
$$v(s) = (\hat{\sigma}(l_1), \cdots, \hat{\sigma}(l_k), pcond) \quad \text{s.t.} \quad \hat{\sigma}(l_k) \neq free \ \wedge \forall i > k : \hat{\sigma(i)} = free,$$

is a sequence denoting the finite sequence of values from $\hat{\sigma}(l_1)$ to $\hat{\sigma}(l_k)$ whereby l_k is the largest location of $\hat{\sigma}$ that is not mapped to $free$. Given a symbolic configuration $c = (pc_1, \cdots, pc_n, s)$, a *symbolic configuration vector* $v(c)$

$$v(c) \in Labels^n \times (D \cup Exp(I))^* \times BExp(I)$$
$$v(c) = (pc_1, \cdots, pc_n, v(s)),$$

is simply a symbolic state vector extended by a prefix that describes the current valuation of the program counters. Since a symbolic configuration vector is finite, it can be used as a data structure to represent symbolic configurations in our state space exploration algorithm shown in fig. 4.1. After defining symbolic configuration vectors, we can now define how the symbolic successor function $next_s$ and the two sets *Open* and *All* operate on symbolic configurations vectors. Given a symbolic configuration vector $v(c)$, the function $next_s : Conf_s(P) \to \mathcal{P}(Conf_s(P))$ with

$$next_s(v(c)) = \{v(c') \mid c \to_s c'\}$$

computes the set of symbolic configuration vectors that are successors of the current symbolic configuration vector. However, to check if there is a transition from c to c' for the cases

- $stm(pc) = \texttt{await}(e, [e_l^1 := e_1, \cdots, e_l^n := e_n])$
- $stm(pc) = \texttt{jump}(e, lab_1, lab_2)$

requires to check $[\![pcond \wedge [\![e]\!]_s(s)]\!] \neq \emptyset$ resp. $[\![pcond \wedge \neg[\![e]\!]_s(s)]\!] \neq \emptyset$, i.e., we have to check if $[\![pcond \wedge [\![e]\!]_s(s)]\!]$ resp. $[\![pcond \wedge \neg[\![e]\!]_s(s)]\!]$ are satisfiable. If $[\![e]\!]_s(s) \in D$, i.e., the expression evaluates to an explicit value $d \in \{true, false\}$, the check is trivial, since the path condition *pcond* is always satisfiable. Otherwise, if $[\![e]\!]_s(s) \in BExp(I)$, since all variables in e have finite domains, the satisfiability problem is decidable, because we can simply evaluate e under all possible valuations of the occurring variables. While there is no general algorithm that can solve arbitrary arithmetic and boolean expressions over finite domains efficiently, for certain subsets of expressions efficient solvers have been implemented. In our implementation, we use a hierarchy of different solving techniques when checking the satisfiability of a symbolic expression. Firstly, we apply a solver that can solve systems of linear integer constraints. Whenever this solver fails to prove or disprove the satisfiability of a boolean expression b, a simple random solver tries to solve b by selecting random values for the variables occurring in b several times. When also the random solver cannot solve b, then an exhaustive enumerative solver definitely proves or disproves the satisfiability of b. In our test models, almost all expressions could be solved efficiently already with the linear constraint solver. This does not mean that there occur only linear expressions in these programs. As mentioned above, as long as $[\![e]\!]_s(s) \in D$, the satisfiability check is trivial, even for nonlinear expressions. However, if there are many expressions e in a program s.t. $[\![e]\!]_s(s) \in BExp(I)$ contains nonlinear expressions, it might be necessary to apply other constraint solving techniques that are better tailored for such cases, since otherwise the time needed for constraint solving will dominate the overall exploration time.

After defining the symbolic successor function $next_s$, we now have to explain how the set *All* of already generated symbolic configuration vectors can be organized s.t. the membership testing of a newly generated configuration vector $v(c) \in All$ can be realized efficiently. Since a symbolic configuration vector $v(c)$ is a symbolic representation for

the set

$$[\![v(c)]\!] = \{v(c') | c' \in [\![c]\!]\}$$

of simple configurations, the membership test $v(c) \in All$ is in fact a subset check of $[\![v(c)]\!] \subseteq [\![All]\!]$, i.e., we have to check if each simple configuration contained in $[\![v(c)]\!]$ is already contained in $\bigcup_{v(c) \in All} [\![v(c)]\!]$. To realize such a precise membership test, we firstly have to define a formula representation $f(v(s))$ of a symbolic state vector $v(s)$ as follows:

$$f((\hat{\sigma}(l_1), \cdots, \hat{\sigma}(l_k), pcond) = \left(\bigwedge_{i=1}^{k} l_i = \hat{\sigma}(l_i) \wedge pcond \right).$$

For a symbolic configuration vector $v(c)$ with $c = (pc_1, \cdots, pc_n, s)$ we define $f(v(c)) = f(s)$. With this, we could organize the set All as depicted in fig. 4.25. Since all program counters have explicit values, we can use the sequence of program counters to compute the index $h((pc_1, \cdots, pc_n))$ of the slot of the hashing table where all states whose program counters have values equal to $pc_1, \cdots, pc_n$ are stored. Suppose we want to test the membership of a newly generated configuration vector $v(c' = (pc'_1, \cdots, pc'_n, s))$. Firstly, we compute the index $h((pc'_1, \cdots, pc'_n))$ of the corresponding slot in the hashing table. Each slot of the hashing table contains 4 different entries. The first entry is the sequence $(pc_1, \cdots, pc_n)$ of program counters that identifies a slot. If $(pc'_1, \cdots, pc'_n) \neq (pc_1, \cdots, pc_n)$, a hash conflict has occurred, and a new index i' will be computed by e.g. linear probing. The second entry is the disjunction of formula representations $f(v(c_1), \cdots, f(v(c_m))$ of all configurations vectors that have been generated so far whose program counters equals $pc_1, \cdots, pc_n$. To check if all possible values of all locations of c' are already covered by $c_1, \cdots, c_m$ we have to evaluate the formula

$$F(v(c')) := \forall I_1, \cdots, I_k \exists I'_1, \cdots, I'_l : f(v(c')) \wedge \left(\bigvee_{i=1}^{m} f(v(c_i)) \right),$$

whereby $\{I_1, \cdots, I_k\} = Vars(f(v(c'))$ and $\{I'_1, \cdots, I'_l\} = \bigcup_{i=1}^{m} Vars(f(v(c_i)))$. If $[\![F(v(c'))]\!] = true$, then all possible values of all locations of $v(c')$ are already possible in at least one of the configuration vectors $v(c_1), \cdots, v(c_m)$, i.e., it holds that $v(c') \in All$ and thus we can discard $v(c')$. Otherwise, if $[\![F(v(c'))]\!] = false$, there exists at least one location of $v(c')$ that can hold a value which is not possible in all configuration vectors $v(c_1), \cdots, v(c_m)$, thus we add $v(c')$ to the set of configuration vectors already stored at this slot. The third entry contains the symmetric differences $sd(v(c_1), pred(v(c_1))), \cdots, sd(v(c_m), pred(v(c_m)))$ of all configuration vectors $v(c_i)$ and their predecessors $pred(v(c_i))$, and the fourth entry contains the indices of predecessors of all configuration vectors $v(c_i)$. Together the third and fourth entry are needed to restore a configuration vector $v(c_i)$.

While the above described organization of All allows precise membership tests, a serious problem of this solution are the high memory requirements for maintaining the hashing table and the enormous complexity of the membership test. In practice, a state space exploration procedure treating All as described above would be very inefficient,

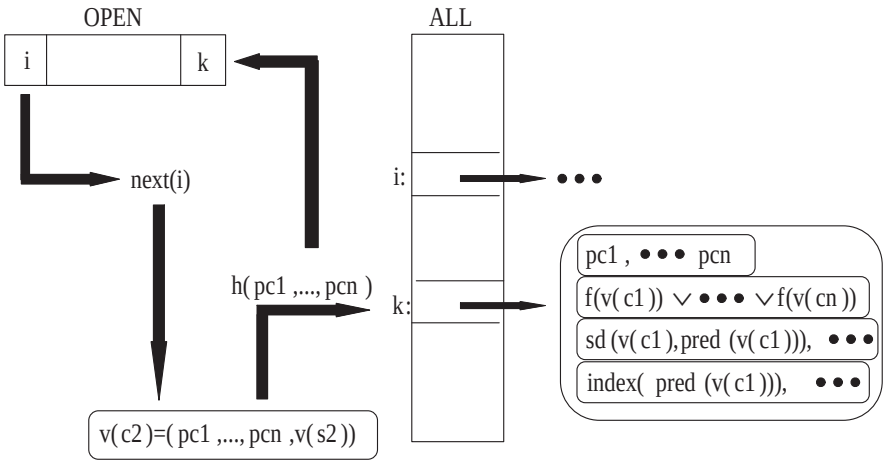


Figure 4.25: **Organization of *All* with symbolic states.** For a precise membership testing of symbolic states, in the hashing table representing the set *All* formula representations of symbolic configuration vectors have to be stored.

since only a few states could be stored and the complexity of the membership test would slow down the exploration process heavily. Therefore, we propose a solution that allows a fast approximate duplicate detection for composite explicit-symbolic states, similar to the approximate duplicate detection for purely explicit states described in sect. 4.1.2. There, we used two hashing functions $h1$ and $h2$ that compute both the index $h1(v(c))$ of the slot in the hashing table where an purely explicit configuration vector $v(c)$ has to be stored and a hashing signature $h2(v(c))$ that serves as a fingerprint to identify $v(c)$. According to that, we define two hashing functions $hs1$ resp. $hs2$ that compute hashing values for explicit-symbolic configuration vectors $v(c)$. Given a value $v \in D \cup Exp(I)$, with $ex(v) \in D$ we denote an explicit value that is defined as

$$ex(v) = \begin{cases} d_{ex} & \text{if } v \in Exp(I) \\ v & \text{otherwise,} \end{cases}$$

whereas $d_{ex} \in D$ denotes an arbitrary but fixed element from D. Given a symbolic configuration vector $v(c = (pc_1, \cdots, pc_n, s = (\hat{\sigma}, pcond)))$, with $ex(v(c))$ we denote an explicit configuration vector that is defined as

$$\begin{aligned} &ex((pc_1, \cdots, pc_n, \hat{\sigma}(l_1), \cdots, \hat{\sigma}(l_k), pcond)) \\ &\quad = (pc_1, \cdots, pc_n, ex(\hat{\sigma}(l_1)), \cdots, ex(\hat{\sigma}(l_k))), \end{aligned}$$

i.e., all symbolic values occurring in $v(c)$ are replaced by the explicit value d_{ex} in $ex(v(c)$, and the path condition *pcond* is simply dropped. With this we can define $hs1$ resp. $hs2$

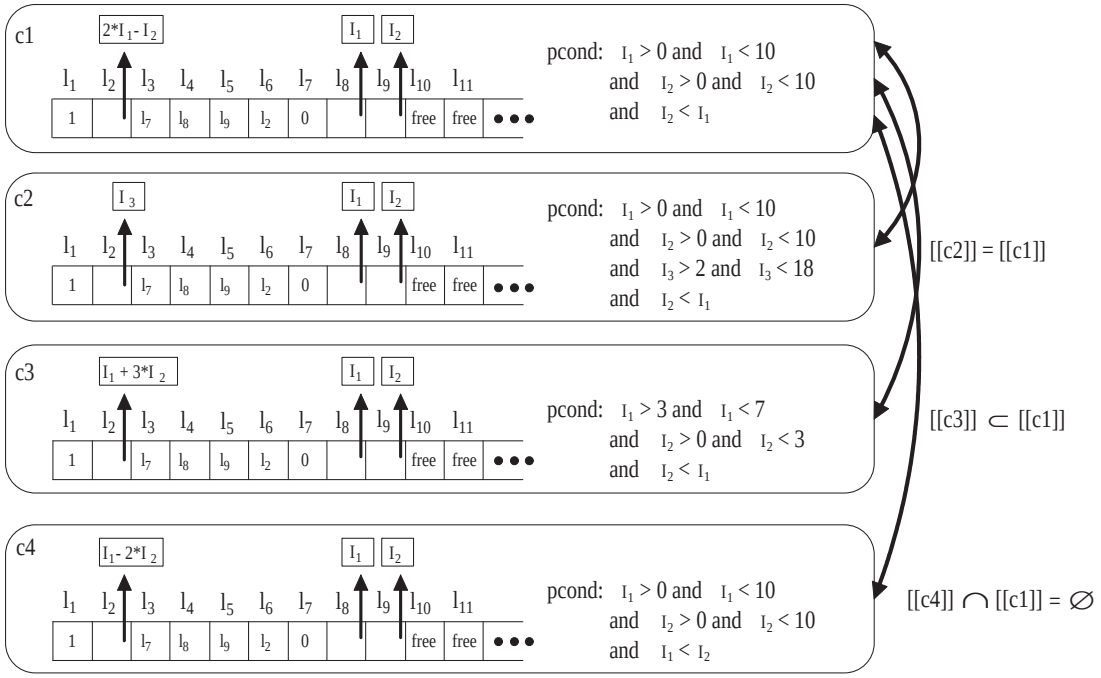


Figure 4.26: **Approximated symbolic duplicate detection.** Due to the definition of $hs1$ resp $hs2$, during state space exploration we assume that the configuration vectors $v(c1)$, $v(c2)$, $v(c3)$ and $v(c4)$ are duplicates.

as

$$\begin{aligned} hs1(v(c)) &= h1(ex(v(c))) \\ hs2(v(c)) &= h2(ex(v(c))), \end{aligned}$$

i.e., $hs1$ resp. $hs2$ simply apply the hashing functions $h1$ resp. $h2$ on the explicit configuration vector $ex(v(c))$. The idea behind the definition of $hs1$ resp. $hs2$ is the following: Often the larger part of a configuration is represented by explicit values from D. Since all explicit values of two symbolic configuration vectors $v(c_1)$ and $v(c_2)$ remain unchanged in $ex(v(c_1))$ and $ex(v(c_2))$, and since all symbolically represented locations of $v(c_1)$ and $v(c_2)$ are changed to d_{ex} in $ex(v(c_1))$ and $ex(v(c_2))$, if all explicitly represented locations of $v(c_1)$ are also explicitly represented in $v(c_2)$, and if the values of corresponding locations are identical, then we have $hs1(v(c_1)) = hs1(v(c_2))$ resp. $hs2(v(c_1)) = hs2(v(c_2))$, i.e., we assume that $v(c_1)$ and $v(c_2)$ are identical. In other words, we assume that $v(c_1)$ and $v(c_2)$ are identical if:

- All explicitly represented locations of $v(c_1)$ are also explicitly represented in $v(c_2)$ and vice versa.
- All explicitly represented locations of $v(c_1)$ and $v(c_2)$ have identical values.

- All symbolically represented locations of $v(c_1)$ are also symbolically represented in $v(c_2)$ and vice versa.

For instance, consider the symbolic configuration vectors c_1, c_2, c_3 and c_4 depicted in fig. 4.26. It is easy to see that $[\![c_2]\!] = [\![c_1]\!]$, $[\![c_3]\!] \subset [\![c_1]\!]$ and $[\![c_4]\!] \cap [\![c_1]\!] = \emptyset$. Suppose c_1 has already been stored in the hashing table representing the set *All*, and we would successively test c_2, c_3 and c_4 for membership in *All*. Due to the definition of $hs1$ we have $hs1(v(c_1)) = hs1(v(c_2)) = hs1(v(c_3)) = hs1(v(c_4))$, and the same holds for $hs2$, i.e., we assume that c_2, c_3 and c_4 are duplicates of c_1, and thus we do not explore them any further. While this is safe for c_2 resp. c_3, discarding c_4 leads to a pruning of the reachable state space. Such an unsafe pruning is the price we have to pay for the very efficient membership test via $hs1$ and $hs2$. However, the experimental results in sect. 4.2.1 will show that the usage of $hs1$ and $hs2$ for membership testing rarely produces an unsafe pruning.

Using the functions $hs1$ resp. $hs2$ we can organize the set *All* as depicted in fig. 4.27. The symmetric difference of two symbolic configuration vectors $v(c_1)$ and $v(c_2)$ can be defined as it has been done for purely explicit states in sect. 4.1.2, with the addition that $sd(v(c_1), v(c_2))$ also contains the difference of the path conditions of $v(c_1)$ and $v(c_2)$. To store a newly generated configuration vector $v(c_2)$ that is a successor of $v(c_1)$, we compute the index $k = hs1(v(c_2))$, and store the fingerprint $hs2(v(c_2))$, the symmetric difference

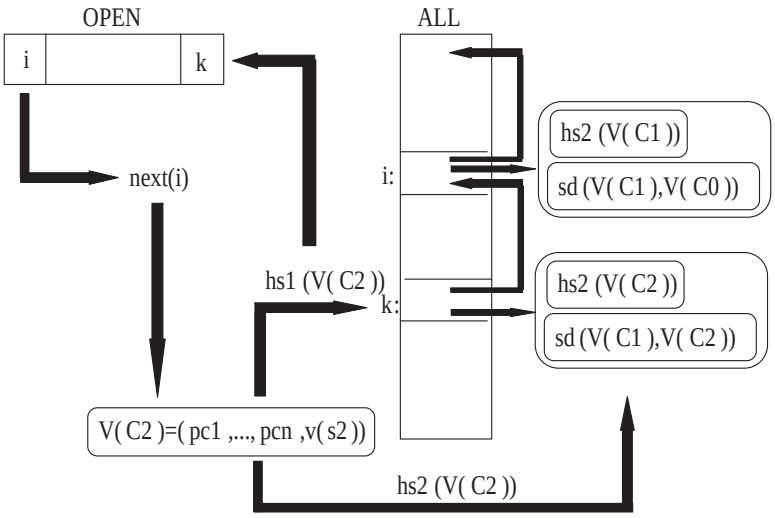


Figure 4.27: **Organization of *All* with approximated symbolic duplicate detection.** The organization of *All* as a hashing table that stores only signatures $hs2(v(c))$ of symbolic configuration vectors and symmetric differences $sd(v(c_1), v(c_2))$. Using the hashing function $hs1$ resp. $hs2$, storing states is very memory efficient, and approximated membership testing can be performed almost as fast as for purely explicit states.

$sd(v(c_1), v(c_2))$ and a predecessor link to the index of $v(c_1)$ at the slot with index k. Using such an organization, storing symbolic states and performing duplicate detection can be realized as efficiently as for purely explicit states. Furthermore, the detection of symmetric states described in sect. 4.1.1 as well as the state storage reduction described in sect. 4.1.3 are fully compatible with the explicit-symbolic state representation, thus we can apply these reductions also when using the explicit-symbolic state representation without any modifications.

4.2.1 Experimental Results

To evaluate the effectiveness of the presented explicit-symbolic state representation, we performed a series of experiments with our test programs. For each of the test programs we define a reachable property. We build 4 configurations of each program that differ in the range of input values, allowing 8, 16, 32 and 2^{32} different values for input variables. For the ranges 8, 16 and 32 we perform a state space exploration with the explicit state representation using all optimizations described in the previous sections (All-8, All-16, All-32). We stopped the exploration process if more than 10^6 states have been generated. For the ranges 32 and 2^{32} we perform a state space exploration using the explicit-symbolic state representation, also using all optimizations (All-Sym-32 and All-Sym-2^{32}). Since BFS and DFS performed similarly in the experiments in sect. 4.1.4, here we concentrate on BFS only. The results are shown below.

PBX	All-8	All-16	All-32	All-Sym-32	All-Sym-2^{32}
#states	63000	$> 10^6$	$> 10^6$	890	880
time	103	-	-	4	4

Figure 4.28: Results for PBX

SMS	All-8	All-16	All-32	All-Sym-32	All-Sym-2^{32}
#states	189000	$> 10^6$	$> 10^6$	1720	1820
time	327	-	-	9	10

Figure 4.29: Results for SMS

Dishwasher	All-8	All-16	All-32	All-Sym-32	All-Sym-2^{32}
#states	39000	572000	$> 10^6$	760	760
time	49	781	-	3	3

Figure 4.30: Results for Dishwasher

CANBus	All-8	All-16	All-32	All-Sym-32	All-Sym-2^{32}
#states	27000	381000	$> 10^6$	630	660
time	31	637	-	3	3

Figure 4.31: Results for CANBus

ARCS	All-8	All-16	All-32	All-Sym-32	All-Sym-2^{32}
#states	67000	$> 10^6$	$> 10^6$	950	890
time	114	-	-	5	4

Figure 4.32: Results for ARCS

Elevator	All-8	All-16	All-32	All-Sym-32	All-Sym-2^{32}
#states	31000	297000	$> 10^6$	1130	1130
time	47	493	-	5	5

Figure 4.33: Results for Elevator

Pacemaker	All-8	All-16	All-32	All-Sym-32	All-Sym-2^{32}
#states	75000	$> 10^6$	$> 10^6$	1330	1410
time	126	-	-	7	8

Figure 4.34: Results for Pacemaker

HomeHeating	All-8	All-16	All-32	All-Sym-32	All-Sym-2^{32}
#states	52000	449000	$> 10^6$	920	920
time	86	710	-	5	5

Figure 4.35: Results for HomeHeating

HomeAlarm	All-8	All-16	All-32	All-Sym-32	All-Sym-2^{32}
#states	47000	$> 10^6$	$> 10^6$	1080	990
time	94	-	-	5	5

Figure 4.36: Results for HomeAlarm

TCU	All-8	All-16	All-32	All-Sym-32	All-Sym-2^{32}
#states	74000	$> 10^6$	$> 10^6$	1470	1470
time	125	-	-	8	8

Figure 4.37: Results for TCU

As one can see, the usage of the composite explicit-symbolic state representation leads to considerable performance improvements for all tested programs. While none of the state space explorations of the All-32 configurations were successful, all explorations using the explicit-symbolic state representation with an input range of 32 resp. 2^{32} were successful. Furthermore, the number of generated states as well as the exploration time for the configurations All-Sym-32 resp. All-Sym-2^{32} are approximately equal, with slight fluctuations. This indicates that the number of generated states and therefore the exploration time is are more or less independent from the ranges of input variables in the tested programs.

A second set of experiments have been performed to evaluate how many false duplicates are produced by the presented approximated explicit-symbolic duplicate detection. To determine the number of wrong duplicates, we have implemented a version of the exploration algorithm that writes the complete configuration vectors that are assumed to be duplicates by the approximated duplicate detection into a file. After the exploration algorithm terminates, in a subsequent phase we checked if the configuration vectors stored in the files are real duplicates by applying an exhaustive solver. Because of the high time requirements of the exhaustive solver we perform these experiments for 5 programs only. For each program we measured the number of explored explicit-symbolic states, the number of duplicates detected by the approximated duplicate detection, and the number of false duplicates. The results are shown below.

	PBX	Dishwasher	Elevator	Pacemaker	HomeAlarm
#states	12700	9800	15300	17100	12600
# detected duplicates	2640	1250	2180	3690	2030
# false duplicates	1	1	2	5	0

Figure 4.38: Number of false duplicates

As can be seen in fig. 4.38, although the presented approximated duplicate detection for composite explicit-symbolic states seems to be rather inaccurate, when applied to real programs it only produces very few false duplicates. Additionally, since in reactive programs there exist in most cases many execution paths that reach a particular program state, there is a good chance that the exploration algorithm finds at least one of these paths even if wrong duplicates are detected.

4.3 Summary of results

The experiments carried out show that the optimizations developed in this section successfully improves the performance of a state space exploration procedure for C_{min} programs. The optimizations presented in sect. 4.1 considerably reduce the number of generated states as well as the memory consumption per state. However, these optimizations are not enough to be able to cope with programs that have input variables with large domains. As the experiments show, using the composite explicit-symbolic state representation developed in sect. 4.2 allows us to perform successful state space explorations also for programs with large input ranges. However, this might be traced back to the fact that the solver used for solving the symbolic expressions occurring in the programs could solve almost all expressions very efficiently. For programs that create symbolic expressions during state space exploration that cannot be solved efficiently with this solver we might need a different solver to be able to perform a successful state space exploration.

4.4 Related Work

We already described related work in chapter 2, thus we concentrate here on work which is similar to our approach. A preliminary version of the described composite explicit-symbolic state representation has been presented in [Let03]. Concerning the presented symmetry reductions in sect. 4.1.1, a similar approach has been proposed in [LV01]. Contrary to our approach, in [LV01] the instruction that created an object together with the number of times the instruction has been executed is used to identify a particular memory location. However, e.g. for the programs shown in fig. 4.4 or 4.6 such an approach would not detect the symmetric states. In [Ios01] a complete depth-first reachability analysis of the heap objects is used to compute a canonical state representation. The advantage of that approach is that it detects all symmetric states. However, the disadvantage lies in the fact that the computation of the canonical state representation is very complex, and it has to be performed after each program transition which results in a drastic deceleration of the exploration process. While it is standard to have a kind of approximated duplicate detection in explicit state model checkers ([Hol87, Hol91, SD95a]) similar to the approximated duplicate detection presented in sect. 4.1.2, to our knowledge the presented symmetric difference to cut down the memory requirements for storing states is new. Concerning the state storage reduction, a similar approach has been presented in [BLP03] in the context of model checking timed automata. However, to our knowledge no one has applied the presented state storage reductions in the context of software verification. In [KPV03] an extension of the model checker JavaPathFinder has been proposed that uses symbolic execution to generate a symbolic execution tree. In contrast to our approach, they do not perform duplicate detection as in our approach with the approximated duplicate detection from sect. 4.2. Another approach using symbolic execution has been presented in [CDGP01] that addresses the problem of verifying programs

in the language Safer-C which is a subset of the C programming language. Contrary to our approach, they neither have a composite explicit-symbolic state representation nor they address the problem of duplicate detection.

5 Heuristic State Space Exploration

In this section, we will apply heuristic search techniques to state space exploration of C_{min} programs w.r.t. a formula EFq. As described in sect. 2.2, a heuristic search algorithm takes into account additional information to direct the search into regions of the state space where it is likely to find a state that fulfills q. The additional information is obtained by applying a heuristic function h to states s yielding a value $h(s)$ that estimates the distance from s to a state satisfying q. By ordering the exploration of states by means of increasing heuristic values, heuristic search can effectively reduce the number of states which has to be explored when searching for states fulfilling q. The reduction of the number of explored states can shorten the exploration time, and in some applications goal states can be found where uninformed search algorithms fail due to time and space restrictions.

Crucial for the effectiveness of heuristic search algorithms is an informative heuristic function h. However, when searching for witnesses of formulas EFq regarding a program P, the question is how to obtain such a function h. Since we want to deal with arbitrary programs P and arbitrary formulas EFq, it is clear that the function h must take both P and q into account. Therefore, we propose a process which can be roughly separated into the following steps:

- Given a program P and a formula EFq, *generate* a function $h(P, q)$ that estimates the distance of an arbitrary state of P to a state of P that fulfills q.
- Perform a heuristic state space exploration of P w.r.t. EFq using the function $h(P, q)$ as a heuristic to guide the search to states fulfilling q.

For the first step, however, the question remains how to generate such a heuristic $h(P, q)$ that is specific for a particular program P and formula q. The central idea is that, starting from P and q, we can generate a new program P^a which is an *abstraction* of P w.r.t. q. Roughly speaking, P^a is called an abstraction of P w.r.t. q if for every run

$$r = \langle c_0, c_1, \cdots \rangle \in Runs(P)$$

there exists an abstract run

$$r^a = \langle c_0^a, c_1^a, \cdots \rangle \in Runs(P^a)$$

such that the truth-value of q in a configuration c_i is preserved in the corresponding abstract configuration c_i^a. Through a *collapsing* of many (sometimes infinite many) concrete states into one abstract state we can make the state space of P^a finite state

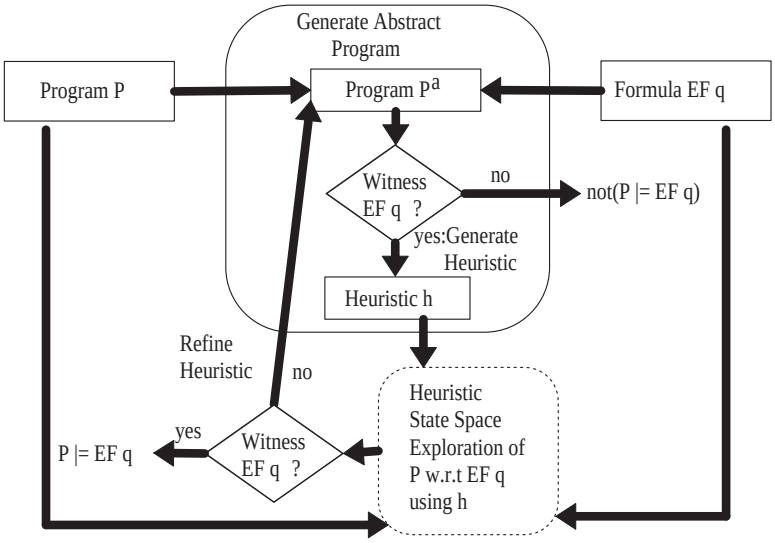


Figure 5.1: **Abstraction-based heuristic search procedure.** Based on a program P and a property EFq, an abstraction P^a of P w.r.t. EFq is generated. Based on the generated abstraction a heuristic function h is created that is used for a heuristic state space exploration of the original program P.

and reasonable small so that it is possible to perform a complete state space exploration of P^a w.r.t. q. Such a complete state space exploration immediately yields a decision procedure for the problem $P^a \models EFq$. As said before, every concrete run of P has a corresponding abstract run in P^a, but through the construction of P^a there can be so-called *spurious* abstract runs

$$r^a = \langle c_0^a, c_1^a, \cdots, c_i^a \rangle \in Runs(P^a)$$

which do not have corresponding concrete runs. Therefore, from $P^a \not\models EFq$ we can safely conclude $P \not\models EFq$, but from $P^a \models EFq$ we cannot directly conclude $P \models EFq$. However, since

$$P \models EFq \Rightarrow P^a \models EFq,$$

i.e., for each concrete witness $r = \langle c_0, \cdots, c_i \rangle$ there is also a corresponding abstract witness $r^a = \langle c_0^a, \cdots, c_i^a \rangle$, we can use the abstract witnesses as a guide to search for concrete witnesses. More precisely, using such an abstraction P^a we can set up an exploration process as depicted in fig. 5.1. The process passes through the following steps:

1. Given a program P and a formula EFq, generate a finite state program P^a that is an abstraction of P w.r.t. EFq.

2. Build the complete state space graph of P^a and find all states in the graph that fulfill q. If no abstract state fulfills q, terminate and return $P \not\models EFq$.

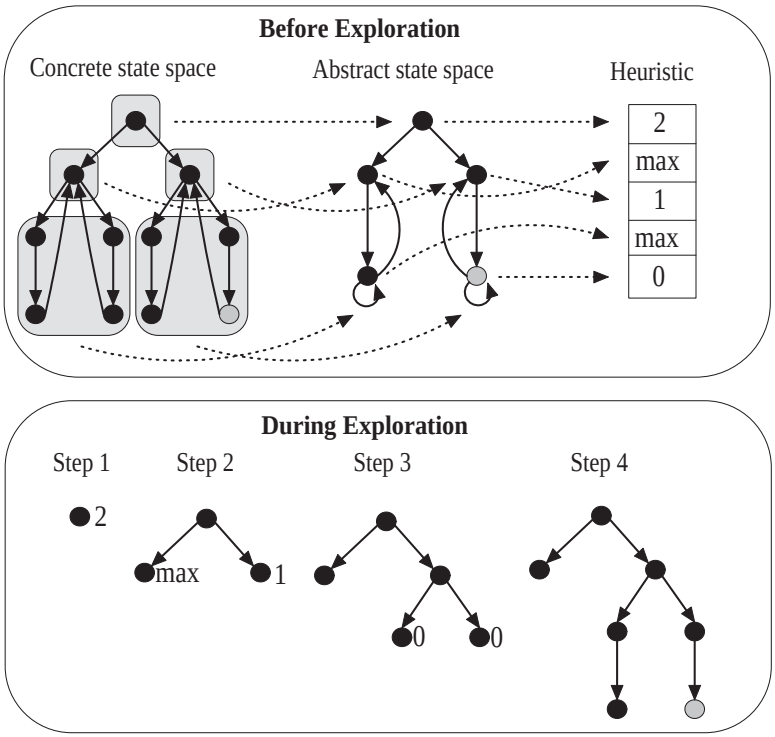


Figure 5.2: **Abstractions as heuristic functions.** The concrete state space (top left), the corresponding abstract state space (top middle) and the computed heuristic values for the abstract states (top right). During state space exploration of the concrete state space, the heuristic values of the corresponding abstract states are utilized as heuristic values for the concrete states.

3. Create a heuristic that contains for each abstract state the distance in terms of transitions to the nearest abstract state that fulfills q.

4. Perform a heuristic state space exploration of P w.r.t. EFq using the generated heuristic. For a concrete configuration c, use the value of the corresponding abstract configuration c^a stored in the heuristic as the heuristic value for c.

5. After timeout or when the (incomplete) search terminates, refine the heuristic and go back to step 1.

As an example, consider the situation depicted in fig. 5.2. According to step 1 and 2 in the procedure described above, before exploring the concrete state space (fig. 5.2 left), a corresponding abstract state space is constructed. In the abstract state space, one or more concrete states are summarized into one abstract state, and for all transitions between two concrete states there is a transition between the corresponding abstract states. Therefore, every path in the concrete state space has a corresponding path in

the abstract state space. Then, in step 3, based on the abstract state space a heuristic function is generated that yields for every abstract state the minimal transition distance to an abstract target state. After the heuristic function has been generated, in step 4 the heuristic exploration of the concrete state space can be performed. For every concrete state the corresponding abstract state is computed, and the value for the abstract state provided by the heuristic is taken as the heuristic value for the concrete state. In the bottom of fig. 5.2 one can see different stages of the exploration process, where successively larger parts of the state space are constructed. However, due to the heuristic exploration process (the heuristic values are depicted for the leaf states of the currently created part of the state space), only a fraction of the state space has to be constructed before a target state is found.

Given a program P and a formula EFq, the first task in the proposed procedure is the generation of a suitable program P^a that is an abstraction of P w.r.t. EFq. In the next section, we will show how to create such an abstract program P^a. After that, in sect. 5.2 we show how we utilize the generated abstraction for computing a heuristic function that will be used for heuristic state space exploration of the original program.

5.1 Abstraction of C_{min} programs

In a C_{min} program, both boolean variables and integer variables have finite domains. However, since in a C_{min} program objects can be dynamically created, the range of pointer variables is in general infinite. Therefore, when trying to find a program P^a that is both an abstraction of a program P w.r.t. EFq and that has only finite many states, one first has to find a suitable abstraction for object creation. Correspondingly, we will build P^a in a two-step process: in the first step, we will create a finite state program P^f that is an abstraction of P w.r.t. EFq. P^f will use both boolean and integer variables with their usual domain, but pointer variables will be replaced by restricted integer variables, and also the number of objects will be restricted to a finite domain. Although the reachable state space of P^f is finite, it can nevertheless be very large. To be able to perform a complete state space exploration, we further abstract P^f into a more abstract program P^a that uses only boolean variables. In the next section, we will introduce a restricted subset of C_{min} that is sufficient to describe finite state programs.

5.1.1 C_{fin}

The syntax of C_{fin} is defined as the syntax of C_{min} in sect. 3.2 but with some limitations. All declared variables must be of type `int` or `bool`, i.e., a C_{fin} program contains no structures and no pointer variables. Furthermore, in expressions the address-operator `&`, the dereference-operator `*` and the selection-operator `->` are not allowed. Additionally, the `new`-statement and the `delete`-statement are not allowed. As in sect. 3.2, with $Vars(P^f)$ we denote the set of all declared variables, and the function $type^f : Vars(P^f) \to Types$ yields the type $type^f(v)$ for every variable $v \in Vars(P^f)$. An assignment $v := e$ stores

the value of the expression $e \in Exp$ in the variable $v \in Vars(P^f)$. A concurrent assignment $[v_1 := e_1, \cdots, v_n := e_n]$ describes the execution of all assignments $v_i := e_i$ in one transition. Furthermore, as in sect. 3.2, with Exp we denote the set of all expressions, STM_i denotes the set of all statements of thread i, and $STM = \bigcup_{i=0}^{n} STM_i$.

Semantics Since in a C_{fin} program there are no pointer variables and no dynamic object creation, we can define a simpler semantics than the semantics presented in sect. 3.2. For the definition of the semantics of expressions we will use the semantic domains D_{int} and D_{bool} as defined in sect. 3.2, and we define $D^f = D_{int} \cup D_{bool} \cup \{free\}$. However, in contrast to C_{min} programs, where a state is a mapping from locations to values, a state of a C_{fin} program is a mapping $\sigma^f : Vars(P^f) \to D^f$ from variables to values, and the set of such states is denoted by Σ^f. As in sect. 3.2, we also use a special state $fail$, and we define $\Sigma^f_f = \Sigma^f \cup \{fail\}$. With this, the semantics of an expression e is a mapping

$$[\![e]\!] : \Sigma^f_f \to \mathcal{P}(D^f \cup \{fail\}),$$

and it is defined as follows:

- if $e \equiv c \in Const$ then $[\![e]\!](\sigma^f) = \{c\}$
- if $e \equiv v \in Vars$ then $[\![e]\!](\sigma^f) = \sigma^f(v)$
- (Nondeterministic choice): if $e \equiv nd$ then $[\![e]\!](\sigma^f) = \{d \mid d \in D_{int}\}$
- (Arithmetic expressions): if $e \equiv e_1\ op\ e_2, op \in AOp$, then $[\![e]\!](\sigma^f) = \{d_1\ op\ d_2 \mid d_1 \in [\![e_1]\!](\sigma^f), d_2 \in [\![e_2]\!](\sigma^f)\}$
- (Conditional expressions): if $e \equiv e_1 ? e_2 : e_3$, then
$$[\![e]\!](\sigma^f) = \begin{cases} \{d \mid d \in [\![e_2]\!](\sigma^f) \wedge true \in [\![e_1]\!](\sigma^f)\} \\ \cup \quad \{d \mid d \in [\![e_3]\!](\sigma^f) \wedge false \in [\![e_1]\!](\sigma^f)\} \end{cases}$$
- (Arithmetic relations): if $e \equiv e_1\ op\ e_2, op \in ARel$, then $[\![e]\!](\sigma^f) = \{d_1\ op\ d_2 \mid d_1 \in [\![e_1]\!](\sigma^f), d_2 \in [\![e_2]\!](\sigma^f)\}$
- (Boolean expressions):

$$\begin{aligned} [\![e_1 \text{ and } e_2]\!](\sigma^f) &= \{d_1 \wedge d_2 \mid d_1 \in [\![e_1]\!](\sigma^f), d_2 \in [\![e_2]\!](\sigma^f)\} \\ [\![e_1 \text{ or } e_2]\!](\sigma^f) &= \{d_1 \vee d_2 \mid d_1 \in [\![e_1]\!](\sigma^f), d_2 \in [\![e_2]\!](\sigma^f)\} \\ [\![\text{not } e_1]\!](\sigma^f) &= \{\neg d \mid d \in [\![e_1]\!](\sigma^f)\} \end{aligned}$$

As in sect. 3.2, to describe the semantics of assignments we need the notion of an *update* of a state σ^f, written as $\sigma^f[v := d]$, where $v, v' \in Vars(P^f)$ and $d \in D^f$, and it is defined as

$$\sigma^f[v := d](v') = \begin{cases} d & \text{if } v = v' \\ \sigma^f(v') & \text{otherwise.} \end{cases}$$

Furthermore, we define $\sigma^f[v := fail] = \sigma^f[fail := d] = fail$, and we extend this definition to a set of values with

$$\sigma^f[v := \{d_1, \cdots, d_n\}] = \{\sigma^f[v := d] \,|\, d \in \{d_1, \cdots, d_n\}\}.$$

Based on this, we define a sequence of updates with

$$\sigma^f[v_1 := d_1, v_2 := d_2, \cdots, v_n := d_n] = (\sigma^f[v_1 := d_1])[v_2 := d_2, \cdots, v_n := d_n].$$

As in sect. 3.2, with each thread $t_i = (i, stms_i)$ we associate a program counter $pc_i \in Labels_i$. A *thread configuration* tc of a program is a tuple

$$tc = (pc, \sigma^f), \sigma^f \in \Sigma_f^f.$$

A *thread transition* $(pc, \sigma^f) \rightarrow (pc', \sigma^{f'})$ between two configurations describes one computation step of one thread corresponding to one statement, and is defined as follows:

- (Concurrent assignment):
$$(pc, \sigma^f) \rightarrow (pc', \sigma^{f'}) \Leftrightarrow \begin{cases} stm(pc) = [v_1 := e_1, \cdots, v_n := e_n] \\ \wedge\, pc' = next(pc) \\ \wedge\, \sigma^{f'} \in \sigma^f[v_1 := [\![e_1]\!](\sigma^f), \cdots, v_n := [\![e_n]\!](\sigma^f)] \end{cases}$$
- (Synchronization):
$$(pc, \sigma^f) \rightarrow (pc', \sigma^{f'}) \Leftrightarrow \begin{cases} stm(pc) = \texttt{await}(e, [v_1 := e_1, \cdots, v_n := e_n]) \\ \wedge\, pc' = next(pc) \;\wedge\; true \in [\![e]\!](\sigma^f) \\ \wedge\, \sigma^{f'} \in \sigma^f[v_1 := [\![e_1]\!](\sigma^f), \cdots, v_n := [\![e_n]\!](\sigma^f)] \end{cases}$$
- (Branching):
$$(pc, \sigma^f) \rightarrow (pc', \sigma^{f'}) \Leftrightarrow \begin{cases} stm(pc) = \texttt{jump}(e, lab_1, lab_2) \;\wedge\; \sigma^{f'} = \sigma^f \;\wedge \\ ((true \in [\![e]\!](\sigma^f) \wedge pc' = lab_1) \vee (false \in [\![e]\!](\sigma^f) \wedge pc' = lab_2)) \end{cases}$$

Based on the transition relation for single threads we can now define the possible transition of a whole program. A *configuration* of a program P^f is a tuple

$$c^f = (pc_1, \cdots, pc_n, \sigma^f).$$

The *starting configuration* of a program P^f is

$$c_0^f = (first_1, \cdots, first_n, \sigma_0^f),$$

whereby whereby σ_0^f fulfills the condition

$$\forall v \in Vars(P^f) \,:\, \sigma_0^f(v) = init(v).$$

A *transition* $(pc_1, \cdots, pc_n, \sigma^f) \rightarrow (pc_1', \cdots, pc_n', \sigma^{f'})$ describes one computation step of the whole program, and it is defined as

$$(pc_1, \cdots, pc_k, \cdots, pc_n, \sigma^f) \rightarrow (pc_1, \cdots, pc_k', \cdots, pc_n, \sigma^{f'}) \Leftrightarrow (pc_k, \sigma^f) \rightarrow (pc_k', \sigma^{f'}),$$

i.e., the transitions of the whole program are all possible interleavings of the transitions of all threads. A *run* r^f of a program P^f is a sequence of configurations $r^f = \langle c_0^f, c_1^f, \cdots \rangle$ s.t. c_0^f is the starting configuration of P^f and $\forall i \geq 0 : c_i^f \rightarrow c_{i+1}^f$ is a transition of P^f. With $Conf(r^f)$ we denote the set of all configurations occurring in r^f. We denote the set of all runs of a program P^f with $Runs(P^f)$, and we define

$$Conf(P^f) = \bigcup_{r^f \in Runs(P^f)} Conf(r^f)$$

and

$$States(P^f) = \{\sigma^f \in \Sigma_f^f \,|\, (pc_1, \cdots, pc_n, \sigma^f) \in Conf(P^f)\}.$$

For the specification of properties, we can use TL as defined in sect. 3.3 without any restrictions since formulas $EFq \in TL$ only contain boolean and integer variables. Therefore, for a configuration c^f and a formula $EFq \in TL$ we define $c^f \vDash EFq$ as in sect. 3.3. Additionally, for a C_{fin} program P^f and a formula $EFq \in TL$, we also define the relation $P^f \vDash EFq$ as in sect. 3.3.

5.1.2 Relation between C_{min} and C_{fin}

As mentioned before, our goal is to generate, for a given C_{min} program P and a formula EFq, a C_{fin} program P^f that is an abstraction of P w.r.t. EFq. The following definition formalizes the notion of abstraction.

Definition 5.1 *Let P be a C_{min} program, P^f be a C_{fin} program and let $EFq \in TL$ be a formula. We say P^f is an abstraction of P w.r.t. EFq iff for all runs*

$$r = \langle c_0, c_1, \cdots \rangle \in Runs(P)$$

there exists a run

$$r^f = \langle c_0^f, c_1^f, \cdots \rangle \in Runs(P^f)$$

s.t. $\forall i \geq 0 : c_i \models q \Leftrightarrow c_i^f \models q$ holds.

Based on the definition of abstraction we can state the following lemma.

Lemma 5.2 *Let P be a C_{min} program, $EFq \in TL$ a formula and P^f be a C_{fin} program that is an abstraction of P w.r.t. EFq. Then $P \vDash EFq \Rightarrow P^f \vDash EFq$.*

Proof:
Assume $P \vDash EFq$. Then there exists a run $r = \langle c_0, \cdots, c_i \rangle \in Runs(P)$ of P with $c_i \vDash q$. Because P^f is an abstraction of P there also exists a run $r^f = \langle c_0^f, \cdots, c_i^f \rangle \in Runs(P^f)$ with $c_i^f \vDash q$, and therefore we have $P^f \vDash EFq$. □

One way to show that a program P^f is an abstraction of a program P is to show that P^f can *simulate* P. Roughly speaking, a program P^f simulates a program P if for all

runs $r = \langle c_0, c_1, \cdots \rangle \in Runs(P)$ there exist a run $r^f = \langle c_0^f, c_1^f, \cdots \rangle \in Runs(P^f)$ s.t. the values of all variables $v \in Vars(P) \cap Vars(P^f)$ are equal or equivalent in some sense. However, a state σ of a C_{min} program is a mapping from the infinite set of locations D_{loc} to values, whereas a state σ^f of a C_{fin} program is a mapping from variables to values. Furthermore, a value in a C_{min} program can either be an integer or boolean or a location, whereas a value in a C_{fin} program can only be an integer or boolean. Therefore, before we can define a simulation relation between a C_{min} program P and a C_{fin} program P^f, we have to define an appropriate relation which relates states $\sigma \in \Sigma$ with states $\sigma^f \in \Sigma^f$. To define such a relation $\sim_\pi$ appropriately, we will use an injective, partial function

$$\pi : D_{loc} \rightarrow Vars(P^f)$$

that maps locations used in a state $\sigma \in \Sigma$ to variables of P^f. Given a state $\sigma \in \Sigma$, a state $\sigma^f \in \Sigma^f$ for which $\sigma \sim_\pi \sigma^f$ should hold must fulfill several conditions. Firstly, a location belonging to a variable $v \in Vars(P) \cap Vars(P^f)$ must be mapped to the same variable $v \in Vars(P^f)$ by π. Secondly, every location $l \in D_{loc}$ whose type is not a pointer type and that is in the domain of π must have the same value both in σ and σ^f. Together these two conditions ensure that the value for an expression $e \in Exp(P) \cap Exp(P^f)$ is equivalent in σ and σ^f. Additionally, for all locations l in the domain of π that have a pointer type we require a certain kind of consistency between $\sigma(l)$ and $\sigma^f(\pi(l))$. We assume that each variable $v \in Vars(P^f)$ has a unique index given by the function

$$index : Vars(P^f) \rightarrow \{1, \cdots, |Vars(P^f)|\}$$

that yields for any $v \in Vars(P^f)$ a unique index $index(v) \in \{1, \cdots, |Vars(P^f)|\}$. Additionally, we define $index(0) = 0$, and we define the constant $top \equiv |Vars(P^f)| + 1$. The meaning of the constant *top* is following: since a C_{fin} program P^f has only finite many variables, we cannot map all locations of a state σ to variables of P^f. When a location l is mapped by π to a variable $v \in Vars(P^f)$, and the value of l is another location l' that is not in the domain of π, then we use the constant *top* as the corresponding value for v, indicating that v points at something that is not represented in P^f. For two states σ, σ^f with $\sigma \sim_\pi \sigma^f$ we require that for all locations l that have a pointer type the following holds: if $v = \pi(l)$ and $v' = \pi(\sigma(l))$ then $\sigma^f(v) = index(v')$, i.e., if l points at l', then the value of v is the index of v'. Furthermore, if $\sigma(l) \notin Dom(\pi)$ then $\sigma^f(v) = top$, i.e., if l points at some location l' that is not in the domain of π, then the value of v is *top*. The following definition summarizes the above reflections.

Definition 5.3 *Let P be a C_{min} program, let*

$$tl : D_{loc} \rightarrow Types \setminus Structs(P)$$

be a type-based partitioning function as defined in sect. 3.2 and let $\sigma \in \Sigma$. Let P^f be a C_{fin}program and

$$index : Vars(P^f) \rightarrow \{1, \cdots, |Vars(P^f)|\}$$

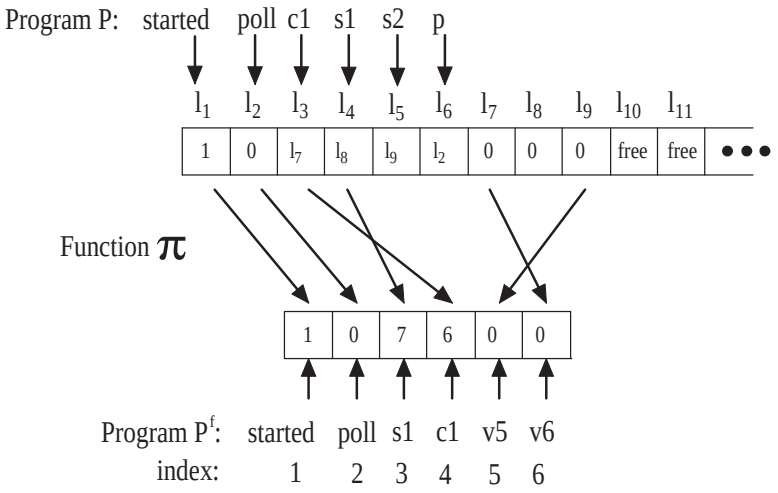


Figure 5.3: **π-corresponding states.** The function π maps locations from a state of a program P to variables of a program P^f. Since all conditions of definition 5.3 are fulfilled, the two states are π-corresponding.

be an index function and $\sigma^f \in \Sigma^f$. *Let further* $\pi : D_{loc} \rightarrow Vars(P^f)$ *be a partial, injective function. We say* σ *and* σ^f *are* π*-corresponding (in symbols:* $\sigma \sim_\pi \sigma^f$*) iff* $\forall l \in D_{loc}$ *the following holds:*

1. $(v \in Vars(P) \cap Vars(P^f) \wedge l = varloc(v)) \Rightarrow \pi(l) = v.$
2. $(tl(l) \in \{bool, int\} \wedge v = \pi(l)) \Rightarrow \sigma(l) = \sigma^f(v).$
3. $(tl(l) \notin \{bool, int\} \wedge v = \pi(l) \wedge v' = \pi(\sigma(l))) \Rightarrow \sigma^f(v) = index(v').$
4. $(tl(l) \notin \{bool, int\} \wedge v = \pi(l) \wedge \sigma(l) \notin Dom(\pi)) \Rightarrow \sigma^f(v) = top.$

As an example of two π-corresponding states, consider the situation shown in fig. 5.3. According to 1 in definition 5.3, since both in P and P^f we have the variables *started*, *poll*, c_1 and s_1, π maps these variables in P to their corresponding variables in P^f. Furthermore, according to 2 the values of the variables *started* and *poll* are the same both in P and in P^f, because they are both of type *int*. The pointer variable $c1$ located at location l_3 points at l_7 and π maps l_7 to the variable v_6 which has the index 6. Therefore, according to 3 in definition 5.3, the value of c_1 in P^f is 6, and according to 2 the variable v_6 has the same value as location l_7. The pointer variable $s1$ located at location l_4 points at l_8. Since l_8 is not in the domain of π, according to 5 in definition 5.3 the value of s_1 in P^f is $top = 6 + 1 = 7$.

Based on definition 5.3, we define π-correspondence for configurations and runs. Given two configurations $c = (pc_1, \cdots, pc_n, \sigma)$ and $c^f = (pc_1^f, \cdots, pc_m^f, \sigma^f)$, we say c and c^f are

π-corresponding if

$$n = m \wedge \forall i \in \{1, \cdots, n\} : pc_i = pc_i^f \wedge \sigma \sim_\pi \sigma^f,$$

i.e., two configurations are π-corresponding if all program counters are equal and σ and σ^f are π-corresponding. Additionally, two runs $r = \langle c_0, c_1, \cdots \rangle$ and $r^f = \langle c_0^f, c_1^f, \cdots \rangle$ are π-corresponding if $\forall i \geq 0 : c_i \sim_\pi c_i^f$ holds. As for states, we write $c \sim_\pi c^f$ resp. $r \sim_\pi r^f$ if c and c^f resp. r and r^f are π-corresponding. The following lemma states two important properties about the value of a variable $v \in Vars(P) \cap Vars(P^f)$ in π-corresponding states σ and σ^f. If $type(v) \in \{int, bool\}$, then the value of v is equal in both states. On the other hand, if $type(v) \notin \{int, bool\}$ and therefore $type^f(v) = int$, then either the index of the variable that is the image of the value of v in σ is equal to the value of v in σ^f or the value of v in σ^f is the constant *top*.

Lemma 5.4 *Let P be a C_{min} program, P^f be a C_{fin} program and $v \in Vars(P) \cap Vars(P^f)$. Let further $\sigma \in \Sigma$ and $\sigma^f \in \Sigma^f$ with $\sigma \sim_\pi \sigma^f$. Then the following holds:*

1. *$type(v) \in \{int, bool\} \Rightarrow [\![v]\!](\sigma) = [\![v]\!](\sigma^f)$.*
2. *$type(v) \notin \{int, bool\} \Rightarrow index((\pi([\![v]\!](\sigma))) = [\![v]\!](\sigma^f)) \vee [\![v]\!](\sigma^f) = top$.*

Proof:

1. Let $l = varloc(v)$. Since $\sigma \sim_\pi \sigma^f$, from definition 5.3 we get

   $$1.\forall l \in D_{loc} : (v \in Vars(P) \cap Vars(P^f) \wedge l = varloc(v)) \Rightarrow \pi(l) = v$$

   and

   $$2.\forall l \in D_{loc} : (tl(l) \in \{bool, int\} \wedge v = \pi(l)) \Rightarrow \sigma(l) = \sigma^f(v),$$

   thus $[\![v]\!](\sigma) = \sigma(l) = \sigma^f(\pi(l)) = \sigma^f(v) = [\![v]\!](\sigma^f)$.

2. Let $l = varloc(v)$. Since $\sigma \sim_\pi \sigma^f$, from definition 5.3 we get

   $$1.(v \in Vars(P) \cap Vars(P^f) \wedge l = varloc(v)) \Rightarrow \pi(l) = v.$$

   We have to distinguish between two cases: either $\sigma(l) \in Dom(\pi)$. Then from 3 of definition 5.3 we can conclude that

   $$index(\pi([\![v]\!](\sigma))) = index(\pi(\sigma(l))) = index(v') = \sigma^f(v) = [\![v]\!](\sigma^f).$$

   On the other hand, in the case $\sigma(l) \notin Dom(\pi)$ with 4 of definition 5.3 we can conclude that $top = \sigma^f(v) = [\![v]\!](\sigma^f)$.

□

The following lemma states that the values $[\![e]\!](\sigma)$ and $[\![e]\!](\sigma^f)$ for an expression $e \in Exp(P) \cap Exp(P^f)$ that uses only variables v with $type(v) \in \{bool, int\}$ are equal for all states σ and σ^f which are π-corresponding.

Lemma 5.5 *Let P be a C_{min}program, P^f be a C_{fin}program and $e \in Exp(P) \cap Exp(P^f)$ an expression. Let further $\sigma \in \Sigma$ and $\sigma^f \in \Sigma^f$ with $\sigma \sim_\pi \sigma^f$. Then $[\![e]\!](\sigma) = [\![e]\!](\sigma^f)$.*

Proof:
(By structural induction): Let $e \in Exp(P) \cap Exp(P^f)$. The case $e \equiv c \in Const$ is clear. For the case $e \equiv v \in Vars(P) \cap Vars(P^f)$ with lemma 5.4 we get $[\![v]\!](\sigma) = [\![v]\!](\sigma^f)$. The case $e \equiv nd$ is also clear since the definition of the semantics $[\![nd]\!](\sigma)$ resp. $[\![nd]\!](\sigma^f)$ is identical.
Induction: The assumption holds for expressions e_1,e_2,e_3. Now consider the following cases:

- (Arithmetic expressions): if $e \equiv e_1\ op\ e_2, op \in AOp$, then

$$\begin{aligned} [\![e]\!](\sigma) &= \{d_1\ op\ d_2 \mid d_1 \in [\![e_1]\!](\sigma), d_2 \in [\![e_2]\!](\sigma)\} \\ &= \{d_1\ op\ d_2 \mid d_1 \in [\![e_1]\!](\sigma^f), d_2 \in [\![e_2]\!](\sigma^f)\} \\ &= [\![e]\!](\sigma^f) \end{aligned}$$

- (Conditional expressions): if $e \equiv e_1?e_2 : e_3$, then

$$\begin{aligned} [\![e]\!](\sigma) &= \begin{cases} & \{d \mid d \in [\![e_2]\!](\sigma) \wedge true \in [\![e_1]\!](\sigma)\} \\ \cup & \{d \mid d \in [\![e_3]\!](\sigma) \wedge false \in [\![e_1]\!](\sigma)\} \end{cases} \\ &= \begin{cases} & \{d \mid d \in [\![e_2]\!](\sigma^f) \wedge true \in [\![e_1]\!](\sigma^f)\} \\ \cup & \{d \mid d \in [\![e_3]\!](\sigma^f) \wedge false \in [\![e_1]\!](\sigma^f)\} \end{cases} \\ &= [\![e]\!](\sigma^f) \end{aligned}$$

- (Arithmetic relations): if $e \equiv e_1\ op\ e_2, op \in ARel$, then

$$\begin{aligned} [\![e]\!](\sigma) &= \{d_1\ op\ d_2 \mid d_1 \in [\![e_1]\!](\sigma), d_2 \in [\![e_2]\!](\sigma)\} \\ &= \{d_1\ op\ d_2 \mid d_1 \in [\![e_1]\!](\sigma^f), d_2 \in [\![e_2]\!](\sigma^f)\} \\ &= [\![e]\!](\sigma^f) \end{aligned}$$

- (Boolean expressions):

$$\begin{aligned} [\![e_1 \text{ and } e_2]\!](\sigma) &= \{d_1 \wedge d_2 \mid d_1 \in [\![e_1]\!](\sigma), d_2 \in [\![e_2]\!](\sigma)\} \\ &= \{d_1 \wedge d_2 \mid d_1 \in [\![e_1]\!](\sigma^f), d_2 \in [\![e_2]\!](\sigma^f)\} \\ &= [\![e]\!](\sigma^f). \end{aligned}$$

  The cases $e \equiv e_1$ or e_2 and $e \equiv$ not e_1 can be shown analogously.

□

Based on the definition of π-correspondence of runs we can now define what it means that a program P^f simulates a program P.

Definition 5.6 *Let P be a C_{min} program and P^f be a C_{fin} program. We say P^f simulates P iff holds:*

$$\forall r \in Runs(P)\, \exists\, r^f \in Runs(P^f) : r \sim_\pi r^f$$

Given a C_{min} program P and a formula EFq, when we can construct a C_{fin} program P^f that contains at least the variables $v \in Vars(q)$ and that is a simulation of P, then P^f is an abstraction of P w.r.t. EFq, which is stated in the following lemma.

Lemma 5.7 *Let P be a C_{min} program and P^f be a C_{fin} program that is a simulation of P. Let further $EFq \in TL$ be a formula with $Vars(q) \subseteq Vars(P) \cap Vars(P^f)$. Then P^f is an abstraction of P w.r.t. EFq.*

Proof:
Since P^f simulates P and $Vars(q) \subseteq Vars(P) \cap Vars(P^f)$, with lemma 5.5 we conclude that for all runs

$$r = \langle c_0, c_1, \cdots \rangle \in Runs(P)$$

there exists a run

$$r^f = \langle c_0^f, c_1^f, \cdots \rangle \in Runs(P^f)$$

s.t. $\forall i \geq 0 : [\![q]\!](\sigma^i) = [\![q]\!](\sigma_i^f)$ holds. Therefore we have $\forall i \geq 0 : c_i \models q \Leftrightarrow c_i^f \models q$, from which by definition 5.1 we can conclude that P^f is an abstraction of P. □

In the next section, we will show how to construct, for a given C_{min} program P and a formula EFq, a C_{fin} program P^f that simulates P.

5.1.3 Translating C_{min} to C_{fin}

In this section we will define a *transformation function*

$$trans : C_{min} \times (Types \to \mathbb{N}) \to C_{fin}$$

that will perform a syntactical transformation of P into a C_{fin} program P^f s.t. P^f will be a simulation of P. Besides a C_{min} program P, the transformation function has an additional parameter, a so-called memory configuration $m : Types(P) \to \mathbb{N}$, that specifies for each type $t \in Types(P)$ the number $m(t)$ of objects of type t which can dynamically be allocated in the transformed program. Given a C_{min} program

$$P = (Structs(P), Vars(P), init(P), Threads(P)),$$

and a memory configuration m, $trans(P, m) = P^f$ will be a C_{fin} program

$$P^f = (Vars(P^f), init(P^f), Threads(P^f)).$$

The set $Vars(P^f)$ of variables will be subdivided into two disjunctive sets. The first set will contain so-called static variables and the second set so-called dynamic variables. Before we explain the use of static and dynamic variables we will give some definitions.

Definition 5.8 *Given a C_{min} program P and a memory configuration m, $SVars(P)$ denotes the set of variables*

$$SVars(P) = \{v | v \in Vars(P)\}$$

$DVars(P, m)$ denotes the set of variables that is defined as

$$DVars(P,m) \quad = \quad \begin{cases} & \{t[j] | \, 0 < j \leq m(t) \wedge t \notin Structs(P)\} \\ \cup & \{t.m[j] | 0 < j \leq m(t) \wedge t \in Structs(P) \wedge m \in MemberVars(t)\} \end{cases}$$

and $Vars(P, m) = SVars(P) \cup DVars(P, m)$. The type of the variables in $Vars(P, m)$ is given by the function $type^f(v)$ and defined as

$$type^f(v) = \begin{cases} bool & if \quad (v \in SVars(P) \wedge type(v) = bool) \\ & \quad \vee ((v \equiv t[j] \vee v \equiv t.m[j]) \wedge t = bool) \\ int & otherwise \end{cases}$$

For instance, suppose P is a C_{min} program that defines a structure C that has three member variables x, y and z of type *bool*. Furthermore, there is a integer variables $i2$ defined in P. Moreover, assume m is a memory configuration with $m(int) = 3$ and $m(C) = 2$. Then $SVars(P) = \{i2\}$ and $DVars(P, m)$ contains three integer variables $\{int[1], int[2], int[3]\}$, and 6 boolean variables $\{C.x[1], C.y[1], C.z[1], C.x[2], C.y[2], C.z[2]\}$.

To illustrate the intended use of static variables in $SVars(P)$ and dynamic variables in $DVars(P, m)$, consider the situation depicted in fig. 5.4. The top left of fig. 5.4 shows a state σ of a C_{min} program P that uses the variables $V = \{started, poll, c1, s1, s2, p\}$. At the bottom left one can see a state σ_f of a program P^f, that also uses all variables in V, i.e., $SVars(P) = V$. Furthermore, P^f also has two dynamic variables $int[1]$ and $int[2]$. Since all conditions from definition 5.3 are fulfilled, the states σ and σ^f are π-corresponding. However, the question is how to get again a π-corresponding state $\sigma^{f'}$ when P executes a statement and changes its state from σ to σ'. For instance, P can execute a `new`-statement like `new(p)`. The top right of fig. 5.4 shows a possible successor state σ' of P after execution of this statement. The effect of the new statement is that l_6 now points at the newly allocated location l_{11}, and according to that l_{11} is initialized to 0. To emulate the effect of this statement in σ^f, we can use the dynamic variables in $DVars(P, m)$. Since the dynamic variable $int[2]$ is unused in state σ^f, we can change π s.t. $\pi(l_{11}) = int[2]$, thus $int[2]$ represents the location l_{11} in $\sigma^{f'}$. According to the change of l_6, in $\sigma^{f'}$ we have to change the value of p to 8, since $index(int[2]) = 8$. Additionally, according to the change of l_{11}, in $\sigma^{f'}$ we have to change the value of $int[2]$ to 0. After these changes, again all conditions of definition 5.3 are fulfilled, therefore σ' and $\sigma^{f'}$ are π-corresponding. To achieve the behavior described above, the statements of P must be transformed into statements of P^f appropriately. For our example in fig. 5.4, the statement

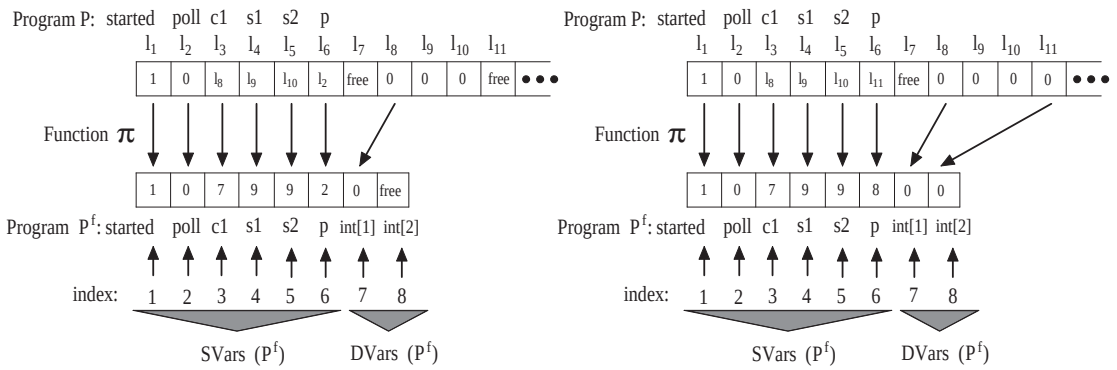


Figure 5.4: **π-correspondence and dynamic object creation.** The configuration of the program P (top left) is π-corresponding to the configuration of program P^f (bottom left). Executing the statement `new(p)` in the depicted configuration of P yields the successor configuration depicted in the top right. To get again a π-corresponding state for P^f, the `new` statement must be transformed in P^f s.t. its effect can be represented using the dynamic variables $int[1]$ resp. $int[2]$.

```
new(p)
```

has to be transformed into the statements

```
p:=(int[1]=free ? 7 : (int[2] = free ? 8 : 9))
[ int[1]:= p=7 ? 0 : int[1], int[2]:= p=8 ? 0 : int[2] ]
```

In the assignment representing the transformed `new`-statement of type t we successively test all variables in $m(t)$ if they are used in the current state. If there is at least one free variable, then we return the index of the found variable. If there is no free variable, then we return the constant $top = 9$. Additionally, in the following assignment we set the newly allocated variable to 0.

The purpose of the constant top can be demonstrated when trying to dereference a variable having this value. As mentioned before, when in a state σ^f the variable v has the value $\sigma^f(v) = top$, then v is pointing at a location l that is not represented in σ^f, i.e., $l \notin Dom(\pi)$, and therefore we do not know which value is stored in the referenced location. For instance, suppose the program P in fig. 5.4 is in state σ' and the next statements to be executed are:

```
new(p)
started:=*p
```

Since in $\sigma^{f'}$ there is no free dynamic variable, the transformed `new`-statement (v. above) yields $p = 9$. The transformation of the assignment

```
started:=*p"
```

yields the statement

```
started:= p=7 ? int[1] : (p=8 ? int[2] : (p=0 ? fail : nd)).
```

If $p = 9$, then we do not know anything about the value of the variable p is pointing at, thus in the transformed statement we have to select nondeterministically a value by executing the nondeterministic expression nd. This additional nondeterminism leads to states in the transformed program P^f that are not possible int the original program P. For instance, after execution of the assignment above, the transformed program can be in a state with $started = 1$. In contrast to this, the original program can only be in a state with $started = 0$. However, since we want to construct a program that simulates the original program, the additional nondeterminism is unavoidable. In the following, we will define the transformation of statements formally. We begin with the transformation of expressions. To this end, given a C_{min} program P, a type $t \in Types(P)$ and a memory configuration m, $V(t)$ denotes the set

$$V(t) = \begin{cases} & \{v \in SVars(P) | type(v) = t\} \\ \cup & \{v \in DVars(P,m) | v \equiv t[j]\} \\ \cup & \{v \in DVars(P,m) | v \equiv t.m[j] \wedge offset(t,m) = 0\} \\ \cup & \{v \in DVars(P,m) | v \equiv t1.m[j] \wedge type(t1.m) = t\}, \end{cases}$$

i.e., $V(t)$ denotes the set of all variables in $Vars(P,m)$ whose type in P is t. Additionally, for a variable $t.m[j] \in DVars(P,m)$ we define

$$MV(t.m[j]) = \{t.m'[j] \,|\, m' \in MemberVars(t)\}$$

i.e., $MV(t.m[j])$ denotes the set of variables $v' \in DVars(P,m)$ that belong to the same object instance j of the structured type t. Given a set $V \subseteq Vars(P,m)$, the set $I(V) = \{index(v) | v \in V\}$ denotes the set of all indices of variables in V. Furthermore, given a type $t \in Types(P)$ we define

$$ND(t) = \begin{cases} nd = 0?false : true & \text{if } t = bool \\ nd & \text{if } t = int \\ nd = 0?0 : nd = 0?i_1 : \cdots & \\ \cdots nd = 0?i_n : top & \text{if } t \notin \{bool, int\} \wedge i_j \in I(V(t)), \end{cases}$$

i.e., $ND(t)$ nondeterministically selects one element of the domain of t. For the types $bool$ resp. int we select a value from $\{false, true\}$ resp. $\{min, max\}$, and for all structure and pointer types we select an index of a variable with the referenced type. With this, we can define the transformation of expressions as follows:

Definition 5.9 *Let P be a C_{min} program and m a memory configuration. Let further $e \in Exp(P)$ an expression. Then $trans(e)$ denotes an expression that is defined as follows:*

- *if* $e \equiv c \in Const$ *then* $trans(e) = e$.
- *(Simple variable): if* $e \equiv v$ *then* $trans(e) = e$.
- *(Referenced location of simple variable): Let* $e \equiv *v$, $t = type(e)$ *and* $V(t) = \{v_1, \cdots, v_n\}$. *Then* $trans(e)$

$$v = index(v_1)?v_1 : \cdots v = index(v_n)?v_n : v = top?ND(t) : fail.$$

- *(Member variable): Let* $e \equiv v \rightarrow m_k$, $t = type(v)$ *and* $V(t) = \{t.m_1[1], \cdots, t.m_1[n]\}$. *Then* $trans(e)$

$$\begin{aligned} &v = index(t.m_1[1])?t.m_k[1] : \cdots \\ &\cdots v = index(t.m_1[n])?t.m_k[n] : v = top?ND(type(v \rightarrow m_k)) : fail. \end{aligned}$$

- *(Location of simple variable): If* $e \equiv \&v$ *then* $trans(e) = index(v)$.
- *(Location of member variable): Let* $e \equiv \&v \rightarrow m$, $t = type(v)$ *and* $V(t) = \{t.m_1[1], \cdots, t.m_1[n]\}$. *Then* $trans(e)$

$$\begin{aligned} &v = index(t.m_1[1])?index(t.m_k[1]) : \cdots \\ &\cdots v = index(t.m_1[n])?index(t.m_k[n]) : v = top?ND(type(v \rightarrow m_k)) : fail \end{aligned}$$

- *(Nondeterministic choice): if* $e \equiv nd$ *then* $trans(e) = e$.
- *(Arithmetic expressions): if* $e \equiv e_1\ op\ e_2, op \in AOp$, *then* $trans(e) = trans(e_1)\ op\ trans(e_2)$.
- *(Conditional expressions): if* $e \equiv e_1?e_2 : e_3$, *then* $trans(e) = trans(e_1)\ ?\ trans(e_2)\ :\ trans(e_3)$.
- *(Arithmetic relations): if* $e \equiv e_1\ op\ e_2, op \in ARel$, *then* $trans(e) = trans(e_1)\ op\ trans(e_2)$.
- *(Pointer relations): if* $e \equiv e_1\ op\ e_2, op \in PRel$, *then*

$$\begin{aligned} trans(e) \quad = \quad & (trans(e_1) = top\ and\ trans(e_2) = top)?ND(bool) \\ & : (trans(e_1)\ op\ trans(e_2)). \end{aligned}$$

- *(Boolean expressions):*

$$\begin{aligned} trans(e_1\ and\ e_2) &= trans(e_1)\ and\ trans(e_2). \\ trans(e_1\ or\ e_2) &= trans(e_1)\ or\ trans(e_2). \\ trans(not\ e_1) &= not\ trans(e_1). \end{aligned}$$

The following lemma describes the relationship between the semantics of an expression $e \in Exp(P)$ in a state σ and its transformed expression $trans(e)$ in a state σ^f that is π-corresponding to σ.

Lemma 5.10 *Let P be a C_{min} program, m a memory configuration and $e \in Exp(P)$ an expression. Let further P^f be a C_{fin} program and $trans(e) \in Exp(P^f)$. If $\sigma \in \Sigma$ and $\sigma^f \in \Sigma^f$ are π-corresponding, then the following holds:*

1. $type(e) \in \{bool, int\} \Rightarrow [\![e]\!](\sigma) \subseteq [\![trans(e)]\!](\sigma^f)$.
2. $type(e) \notin \{bool, int\} \Rightarrow I(\pi([\![e]\!](\sigma) \cap Dom(\pi))) \subseteq [\![trans(e)]\!](\sigma^f)$.
3. $type(e) \notin \{bool, int\} \Rightarrow [\![e]\!](\sigma) \not\subseteq Dom(\pi) \Rightarrow top \in [\![trans(e)]\!](\sigma^f)$.

Proof:
(By structural induction): Let $e \in Exp(P)$.
Base cases:

- Case $e \equiv c \in Const$ is clear since $trans(c) = c$.
- Case $e \equiv v \equiv trans(v)$. Let $l = varloc(v)$. We distinguish two cases.
  - If $type(v) \in \{bool, int\}$ then 2 and 3 are true. Additionally, with 1 of lemma 5.4 we get $[\![e]\!](\sigma) = [\![trans(e)]\!](\sigma^f)$ which proves 1.
  - If $type(v) \notin \{bool, int\}$ then 1 is true. We distinguish two cases:
    - If $\sigma(l) \in Dom(\pi)$ with 2 of lemma 5.4 we get 2 and 3.
    - If $\sigma(l) \notin Dom(\pi)$ again with 2 of lemma 5.4 we get 2 and 3.
- Case $e \equiv *v$ and $trans(e) \equiv$

  $$v = index(v_1)?v_1 : \cdots v = index(v_n)?v_n : v = top?ND(t) : fail.$$

  Let $l = varloc(v)$ and $l' = \sigma(l)$.
  - If $type(*v) \in \{bool, int\}$ then 2 and 3 are true. Since $[\![*v]\!](\sigma) = \sigma(l')$ we distinguish the cases $l' \in Dom(\pi)$ and $l' \notin Dom(\pi)$. If $l' \in Dom(\pi)$, then from definition 5.3 we get $\sigma^f(v) = index(v_j)$ and $\sigma^f(v_j) = \sigma(l')$, therefore from the construction of $trans(e)$ we can conclude 1. If $l' \notin Dom(\pi)$, then from definition 5.3 we get $\sigma^f(v) = top$, therefore from the construction of $trans(e)$ we can conclude 1.
  - If $type(*v) \notin \{bool, int\}$ then 1 is true. We distinguish the cases $l' \in Dom(\pi)$ and $l' \notin Dom(\pi)$. If $l' \in Dom(\pi)$, then from definition 5.3 we get $\sigma^f(v) = index(v_j)$. If $\sigma(l') \in Dom(\pi)$ then from 3 of definition 5.3 we can conclude 2 and 3. If $\sigma(l') \notin Dom(\pi)$ then from 4 of definition 5.3 we can conclude 2 and 3. The case $l' \notin Dom(\pi)$ can be shown analogous.

The cases $e \equiv v \to m_k$, $e \equiv \&v$ and $e \equiv \&v \to m$ can be shown similarly.

- (Nondeterministic choice): if $e \equiv nd$ then $trans(e) = e$. Therefore, 1, 2 and 3 hold.

Induction: Assume 1, 2 and 3 hold for expressions e_1, e_2 and e_3. Now consider the following cases:

- (Arithmetic expressions): if $e \equiv e_1\, op\, e_2, op \in AOp$, then 2 and 3 hold since $type(e) = int$. Additionally, we have

$$\begin{aligned} [\![e]\!](\sigma) &= \{d_1\ op\ d_2 | d_1 \in [\![e1]\!](\sigma), d_2 \in [\![e_2]\!](\sigma)\} \\ &\subseteq \{d_1\ op\ d_2 | d_1 \in [\![e1]\!](\sigma^f), d_2 \in [\![e_2]\!](\sigma^f)\} \\ &= [\![trans(e_1)\ op\ trans(e_2)]\!](\sigma^f) = [\![trans(e)]\!](\sigma^f). \end{aligned}$$

- (Conditional expressions): if $e \equiv e_1 ? e_2 : e_3$, then

$$trans(e) = trans(e_1) ? trans(e_2) : trans(e_3).$$

Since the assumption holds for e_1 we know that $[\![e_1]\!](\sigma) \subseteq [\![trans(e_1)]\!](\sigma^f)$. Now we can distinguish two cases. Either $type(e) \in \{bool, int\}$, then we have

$$[\![e_2]\!](\sigma) \subseteq [\![e_2]\!](\sigma^f) \text{ and } [\![e_3]\!](\sigma) \subseteq [\![e_3]\!](\sigma^f),$$

and therefore 1, 2 and 3 hold. Otherwise, if $type(e) \notin \{bool, int\}$, we distinguish two cases. Either $[\![e_2]\!](\sigma) \in Dom(\pi)$, then we have

$$(\pi([\![e_2]\!](\sigma) \cap Dom(\pi))) \subseteq [\![trans(e_2)]\!](\sigma^f),$$

which proves 2. Otherwise, if $[\![e_2]\!](\sigma) \notin Dom(\pi)$, we have

$$[\![e_2]\!](\sigma) \not\subseteq Dom(\pi) \Rightarrow top \in [\![trans(e_2)]\!](\sigma^f),$$

which proves 3. The same arguments holds for e_3.

- (Arithmetic relations): if $e \equiv e_1\, op\, e_2, op \in ARel$, then 2 and 3 hold since $type(e) = bool$. Additionally, we have

$$\begin{aligned} [\![e]\!](\sigma) &= \{d_1\ op\ d_2 | d_1 \in [\![e1]\!](\sigma), d_2 \in [\![e_2]\!](\sigma)\} \\ &\subseteq \{d_1\ op\ d_2 | d_1 \in [\![e1]\!](\sigma^f), d_2 \in [\![e_2]\!](\sigma^f)\} \\ &= [\![trans(e_1)\ op\ trans(e_2)]\!](\sigma^f) = [\![trans(e)]\!](\sigma^f). \end{aligned}$$

The cases $e_1\ op\ e_2, op \in PRel$, $e_1\ and\ e_2$, $e_1\ or\ e_2$ and $not\ e_1$ can be shown similarly.

□

After describing the transformation of expressions, we can now define the transformation of assignments.

Definition 5.11 *Let P be a C_{min} program and $e_l := e$ an assignment of P. Then $trans(e_l := e)$ is defined as*

- *Case 1: If $e_l \equiv v$ then $trans(v := e) = v := trans(e)$.*
- *Case 2: Let $e_l \equiv *v$ and $t = type(v)$ and $V(t) = \{v_1, \cdots, v_n\}$. Then*

$$\begin{aligned} trans(*v := e) \quad = \quad & [v_1 := v = index(v_1)?trans(e) : (v = 0?fail : v_1), \cdots \\ & \cdots v_n := v = index(v_n)?trans(e) : (v = 0?fail : v_n)]. \end{aligned}$$

- *Case 3: Let $e_l \equiv v \rightarrow m_k$ and $type(v) = t$ and $V(t) = \{t.m_1[1], \cdots, t.m_1[n]\}$. Then*

$$\begin{aligned} trans(v \rightarrow m := e) \quad = \quad & [t.m_k[1] := v = index(t.m_1[1])?trans(e) \\ & \quad : (v = 0?fail : t.m_k[1]), \cdots \\ & \cdots t.m_k[n] := v = index(t.m_1[n])?trans(e) \\ & \quad : (v = 0?fail : t.m_k[n])]. \end{aligned}$$

Based on the transformation of assignments, we now prove that, given two π-corresponding states σ, σ^f and an assignment $e_l := e$, for every state σ' in the set of updated states $\sigma[[\![adr(e_l)]\!](\sigma) := [\![e]\!](\sigma)]$ there exist a π-corresponding state $\sigma^{f'}$ in the set of updated states $\sigma^f[trans(e_l := e)]$. Before we prove this fact we state some simple relationships between the updated states of two π-corresponding states σ and σ^f.

Lemma 5.12 *Let P be a C_{min} program, P^f be a C_{fin} program, $\sigma \in \Sigma$ and $\sigma^f \in \Sigma^f$ with $\sigma \sim_\pi \sigma^f$. Let further $l_1, l_2, l_3, l_4 \in D_{loc}$ with $l_1, l_2 \in Dom(\pi)$, $l_3, l_4 \notin Dom(\pi)$ and $d \in D^f$. Then it holds:*

1. $\sigma[l_1 := d] \sim_\pi \sigma^f[\pi(l_1) := d]$
2. $\sigma[l_1 := l_2] \sim_\pi \sigma^f[\pi(l_1) := index(\pi(l_2))]$
3. $\sigma[l_1 := l_3] \sim_\pi \sigma^f[\pi(l_1) := top]$
4. $\sigma[l_3 := d] \sim_\pi \sigma^f$
5. $\sigma[l_3 := l_1] \sim_\pi \sigma^f$
6. $\sigma[l_3 := l_4] \sim_\pi \sigma^f$

Proof:

1. Since $l_1 \in Dom(\pi)$ we have $v = \pi(l_1) \in Vars(P^f)$. $\sigma[l_1 := d]$ differs from σ only at location l_1, and $\sigma^f[\pi(l_1) := d]$ differs from σ^f only at variable v. Since $\sigma[l_1 := d](l_1) = d = \sigma^f[\pi(l_1) := d](v)$, all conditions of definition 5.3 are fulfilled and therefore $\sigma[l_1 := d] \sim_\pi \sigma^f[\pi(l_1) := d]$ holds.

The other cases can be shown similarly. □

Lemma 5.13 *Let P be a C_{min} program, m a memory configuration and $e_l := e$ an assignment. Let further P^f be a C_{fin} program. If $\sigma \in \Sigma$ and $\sigma^f \in \Sigma^f$ are π-corresponding, then the following holds:*

$$\forall \sigma' \in \sigma[[\![adr(e_l)]\!](\sigma) := [\![e]\!](\sigma)] \quad \exists \sigma^{f'} \in \sigma^f[trans(e_l := e)] : \sigma' \sim_\pi \sigma^{f'}.$$

Proof:
Let $l \in [\![adr(e_l)]\!](\sigma)$, $d \in [\![e]\!](\sigma)$ and $\sigma' = \sigma[l := d]$. We distinguish the following cases:

- Case $e_l \equiv v$. Then $l \in Dom(\pi)$ and there exists a $v \in Vars(P, m)$ with $\pi(l) = v$ and $trans(v := e) \equiv v := trans(e)$. We distinguish the following cases:
  - $type(d) \in \{bool, int\}$, then with 1 of lemma 5.10 we get $d \in [\![trans(e)]\!](\sigma^f)$, therefore $\sigma^{f'} = \sigma^f[v := d] \in \sigma^f[trans(e_l := e)]$, and with 1 of lemma 5.12 we get $\sigma \sim_\pi \sigma^{f'}$.
  - $type(d) \notin \{bool, int\}$. If $d \in Dom(\pi)$ then with 2 of lemma 5.10 we get $d \in [\![trans(e)]\!](\sigma^f)$, therefore $\sigma^{f'} = \sigma^f[v := d] \in \sigma^f[trans(e_l := e)]$, and with 2 of lemma 5.12 we get $\sigma \sim_\pi \sigma^{f'}$. If otherwise $d \notin Dom(\pi)$, then with 3 of lemma 5.10 we get $top \in [\![trans(e)]\!](\sigma^f)$, therefore $\sigma^{f'} = \sigma^f[v := top] \in \sigma^f[trans(e_l := e)]$, and with 3 of lemma 5.12 we get $\sigma \sim_\pi \sigma^{f'}$.

The cases $e \equiv *v$ and $e \equiv v \to m_k$ can be shown similarly. □

After defining the transformation of expressions and assignments, we can now define the transformation of statements.

Definition 5.14 *Let P be a C_{min} program and $stm \in STM(P)$ be a statement of P. Then $trans(stm)$ is defined as follows:*

- *(Concurrent assignment):*
  if $stm \equiv [e_l^1 := e_1, \cdots, e_l^n := e_n]$ is a concurrent assignment, then

$$trans(stm) = [trans(e_l^1 := e_1), \cdots, trans(e_l^n := e_n)].$$

- *(Synchronization):*
  if $stm \equiv await(e, [e_l^1 := e_1, \cdots, e_l^n := e_n]))$ is an await statement,then

$$trans(stm) = await(trans(e), trans([e_l^1 := e_1, \cdots, e_l^n := e_n])).$$

- *(Branching):*
  if $stm \equiv jump(e, k_1, k_2)$ is a jump statement, then

$$trans(stm) = jump(trans(e), k_1, k_2).$$

- *(Object creation):*
  Case $stm \equiv new(v)$*: Let* $t = type(*v)$*,* $V(t) = \{v_1, \cdots, v_n\}$ *and* $MV(v_j) = \{v_j^1, \cdots, v_j^{m_j}\}$ *then*

$$\begin{aligned} trans(stm) \;=\; & [v := v_1 = free?index(v_1) : \cdots v_n = free?index(v_n) : top, \\ & v_1^1 := v_1 = free?0 : v_1^1, \cdots, v_1^{m_1} := v_1 = free?0 : v_1^{m_1}, \\ & \cdots, v_n^1 := v_n = free?0 : v_n^1, \cdots, v_n^{m_n} := v_n = free?0 : v_n^{m_n}]. \end{aligned}$$

- *(Object destruction):*
  Case $stm \equiv delete(v)$*: Let* $t = type(*v)$*,* $V(t) = \{v_1, \cdots, v_n\}$ *and* $MV(v_j) = \{v_j^1, \cdots, v_j^{m_j}\}$*, then*

$$\begin{aligned} trans(stm) \;=\; & [v_1^1 := v = index(v_1)?free : v_1^1, \cdots, \\ & \quad v_1^{m_1} := v = index(v_1)?free : v_1^{m_1}, \\ & v_n^1 := v = index(v_n)?free : v_n^1, \cdots, \\ & \quad v_n^{m_n} := v = index(v_n)?free : v_n^{m_n}, v := 0]. \end{aligned}$$

Given a labeled statement (l, stm), we define $trans((l, stm)) = (l, trans(stm))$. Additionally, for a sequence of labeled statements $\langle (l_1, stm_1), \cdots, (l_n, stm_n)\rangle$ we define

$$trans(\langle (l_1, stm_1), \cdots, (l_n, stm_n)\rangle) = \langle trans((l_1, stm_1)), \cdots, trans((l_n, stm_n)\rangle$$

With this preliminaries, we can now define the abstract program P^f formally.

Definition 5.15 *Given a* C_{min} *program*

$$P = (Structs(P), Vars(P), init(P), Threads(P))$$

and a memory configuration m*, we define the* C_{fin} *program*

$$P^f(m) = (Vars(P^f(m)), init(P^f(m)), Threads(P^f(m)))$$

s.t.

- $Vars(P^f(m)) = Vars(P, m)$
- $init(P^f(m))$ *is an initialization function that is defined as*

$$init(P^f(m))(v) = \begin{cases} init(v) & \text{if } v \in Vars(P) \\ free & \text{otherwise} \end{cases}$$

- $Threads(P^f(m)) = \{(1, stms_1^f), \cdots, (n, stms_n^f)\}$ *s.t.*

$$\forall i \in \{1, \cdots, n\} : stms_i^f = trans(stms_i).$$

After we have defined the transformation of a C_{min} program P into the C_{fin} program $P^f(m)$ formally, we will now prove that $P^f(m)$ is indeed a simulation of P.

Theorem 5.16 *Let P be a C_{min} program and m a memory configuration. Then $P^f(m)$ is a simulation of P.*

Proof:
By induction. We have to show that

$$\forall r \in Runs(P) \, \exists \, r^f \in Runs(P^f) : r \sim_\pi r^f.$$

Let $r = \langle c_0, \cdots, c_i \rangle \in Runs(P)$. We construct a corresponding run $r^f = \langle c_0^f, \cdots, c_i^f \rangle \in Runs(P^f(m))$ s.t. $r \sim_\pi r^f$.

- Case $i = 0$: $c_0 = (start_1, \cdots, start_n, \sigma_0)$ is the starting configuration of P, and $c_0^f = (start_1, \cdots, start_n, \sigma_0^f)$ is the starting configuration of $P^f(m)$. From the definition of starting configurations we know that

  - $\forall v \in Vars(P) : \sigma_0(varloc(v)) = init(v)$
  - $\forall l \in D_{loc} : l \notin Ran(varloc) \Rightarrow \sigma_0(l) = free$

  hold. Furthermore we know that

  $$\forall v \in Vars(P^f) \, : \, \sigma_0^f(v) = init(v)$$

  holds. Now we choose π s.t. $\pi(l) = v \Leftrightarrow l = varloc(v)$ holds. With such a π all conditions of def. 5.3 are fulfilled:

  1. From the definition of π it follows directly that

     $$(v \in Vars(P) \cap Vars(P^f) \wedge l = varloc(v)) \Rightarrow \pi(l) = v,$$

     which proves 1 of def. 5.3.
  2. From the definition of σ_0, σ^f and π we know that

     $$\forall v \in BIVars(P) \; \exists \; l \in D_{loc} : v = \pi(l)) \wedge \sigma(l) = \sigma^f(v)$$

     holds. Additionally, we know that $\pi(l) = v \Leftrightarrow l = varloc(v)$. From this we can directly conclude

     $$(tl(l) \in \{bool, int\} \wedge v = \pi(l)) \Rightarrow \sigma(l) = \sigma^f(v),$$

     which proves 2 of def. 5.3.

3. From the definition of σ_0, σ^f and π we know that

$$\forall v \in Vars(P) \setminus BIVars(P) \; \exists \; l \in D_{loc} : v = \pi(l)) \wedge \sigma(l) = 0 = \sigma^f(v)$$

holds. Additionally, we know that $\pi(l) = v \Leftrightarrow l = varloc(v)$. Since $index(0) = 0$, we can conclude

$$(tl(l) \notin \{bool, int\} \wedge v = \pi(l) \wedge v' = \pi(\sigma(l))) \Rightarrow \sigma^f(v) = index(v'),$$

which proves 3 of def. 5.3.

4. From the definition of σ_0, σ^f and π we know that

$$\forall v \in Vars(P) \setminus BIVars(P) \; \exists \; l \in D_{loc} : v = \pi(l)) \wedge \sigma(l) = 0 = \sigma^f(v)$$

holds. Since all pointer variables $v \in Vars(P) \setminus BIVars(P)$ all already in the domain of π, i.e.,

$$v \in Vars(P) \setminus BIVars(P) \Rightarrow v \in Dom(\pi)$$

holds, it follows directly that

$$(tl(l) \notin \{bool, int\} \wedge v = \pi(l) \wedge \sigma(l) \notin Dom(\pi)) \Rightarrow \sigma^f(v) = top$$

holds, which proves 4 of def. 5.3.

Since all conditions of definition 5.3 are fulfilled, it follows that $\sigma_0 \sim_\pi \sigma_0^f$ and therefore $c_0 \sim_\pi c_0^f$.

- Case $i + 1$: Assume the theorem holds for all runs up to length i, i.e., $\forall j \in \{0, \cdots, i\} : c_i \sim_\pi c_i^f$. From $c_i = (pc_1, \cdots, pc_n, \sigma_i)$ we can reach a successor configuration $c_{i+1} = (pc_1, \cdots, pc_k', \cdots, pc_n, \sigma_{i+1})$ by executing a transition $c_i \rightarrow c_{i+1}$ of a thread k of P. Let stm denote the statement at the label corresponding to the program counter pc_k of thread k, and let stm^f denote the statement of the corresponding thread of $P^f(m)$. We show that executing stm^f in configuration c_i^f leads to a configuration c_{i+1}^f with $c_{i+1} \sim_\pi c_{i+1}^f$. We have to distinguish the following cases:

  – (Concurrent assignment):
  if $stm \equiv [e_l^1 := e_1, \cdots, e_l^n := e_n]$ is a concurrent assignment, then

$$\sigma_{i+1} \in \sigma[[\![adr(e_l^1)]\!](\sigma) := [\![e_1]\!](\sigma), \cdots, [\![adr(e_l^n)]\!](\sigma) := [\![e_n]\!](\sigma)]$$

  and

$$stm^f = [trans(e_l^1 := e_1), \cdots, trans(e_l^n := e_n)].$$

  From lemma 5.13 we know that for all assignments $e_l^i := e_i$ it holds that

$$\forall \sigma' \in \sigma[[\![adr(e_l)]\!](\sigma) := [\![e]\!](\sigma)] \quad \exists \sigma^{f'} \in \sigma^f[trans(e_l := e)] : \sigma' \sim_\pi \sigma^{f'}.$$

Therefore, by applying lemma 5.13 n times we get

$$\forall \sigma_{i+1} \in \sigma[[\![adr(e_l^1)]\!](\sigma) := [\![e_1]\!](\sigma), \cdots, [\![adr(e_l^n)]\!](\sigma) := [\![e_n]\!](\sigma)]$$
$$\exists \sigma_{i+1}^f \in \sigma^f[trans(e_l^1 := e_1), \cdots, trans(e_l^n := e_n)] : \sigma_{i+1} \sim_\pi \sigma_{i+1}^f,$$

i.e., we can always choose σ_{i+1}^f s.t. $\sigma_{i+1} \sim_\pi \sigma_{i+1}^f$ holds, and thus we have $c_{i+1} \sim_\pi c_{i+1}^f$.

– (Synchronization):
if $stm \equiv await(e, [e_l^1 := e_1, \cdots, e_l^n := e_n]))$ is an await statement, then

$$true \in [\![e]\!](\sigma_i)$$

and

$$\sigma_{i+1} \in \sigma[[\![adr(e_l^1)]\!](\sigma) := [\![e_1]\!](\sigma), \cdots, [\![adr(e_l^n)]\!](\sigma) := [\![e_n]\!](\sigma)].$$

Additionally, we know that

$$stm^f = await(trans(e), trans([e_l^1 := e_1, \cdots, e_l^n := e_n])).$$

Since $type(e) = bool$, from lemma 5.10 (1) we know that

$$[\![e]\!](\sigma_i) \subseteq [\![trans(e)]\!](\sigma_i^f),$$

thus we can conclude that

$$true \in [\![e]\!](\sigma_i^f).$$

Additionally, from lemma 5.13 we know that for all assignments $e_l^i := e_i$ it holds that

$$\forall \sigma' \in \sigma[[\![adr(e_l)]\!](\sigma) := [\![e]\!](\sigma)] \quad \exists \sigma^{f'} \in \sigma^f[trans(e_l := e)] : \sigma' \sim_\pi \sigma^{f'}.$$

Therefore, by applying lemma 5.13 n times we get

$$\forall \sigma_{i+1} \in \sigma[[\![adr(e_l^1)]\!](\sigma) := [\![e_1]\!](\sigma), \cdots, [\![adr(e_l^n)]\!](\sigma) := [\![e_n]\!](\sigma)]$$
$$\exists \sigma_{i+1}^f \in \sigma^f[trans(e_l^1 := e_1), \cdots, trans(e_l^n := e_n)] : \sigma_{i+1} \sim_\pi \sigma_{i+1}^f,$$

i.e., we can always choose σ_{i+1}^f s.t. $\sigma_{i+1} \sim_\pi \sigma_{i+1}^f$ holds, and thus we have $c_{i+1} \sim_\pi c_{i+1}^f$.

– (Branching):
if $stm \equiv jump(e, k_1, k_2)$ is a jump statement, then

$$true \in [\![e]\!](\sigma_i) \wedge pc' = k_1) \vee (false \in [\![e]\!](\sigma_i) \wedge pc' = k_2))$$

and

$$stm^f = jump(trans(e), k_1, k_2).$$

If $true \in [\![e]\!](\sigma_i)$, from lemma 5.10 (1) we know that in this case also $true \in [\![e]\!](\sigma_i^f)$ and therefore $pc' = k_1 = pc^{f'}$. The same argument holds also for the case $false \in [\![e]\!](\sigma_i)$, thus we can conclude that $c_{i+1} \sim_\pi c_{i+1}^f$.

– (Object creation):
Case $stm \equiv new(v)$: Let $t = type(*v)$, $V(t) = \{v_1, \cdots, v_n\}$ and $MV(v_j) = \{v_j^1, \cdots, v_j^{m_j}\}$. From the definition of the semantics we know that

$$\begin{aligned} \exists l \in D_{loc} \quad : \quad & (t \in CTypes(P) \wedge tls(l) = t \wedge \sigma_i(l) = free) \\ & \vee (t \notin CTypes \wedge tl(l) = t \wedge \sigma_i(l) = free) \end{aligned}$$

and

$$\sigma_{i+1} = \sigma_i[[\![adr(e)]\!](\sigma_i) := l, l := 0, \cdots, l + size(t) - 1 := 0].$$

Additionally, we know that

$$\begin{aligned} stm^f \quad = \quad & [v := v_1 = free?index(v_1) : \cdots v_n = free?index(v_n) : top, \\ & v_1^1 := v_1 = free?0 : v_1^1, \cdots, v_1^{m_1} := v_1 = free?0 : v_1^{m_1}, \\ & \cdots, v_n^1 := v_n = free?0 : v_n^1, \cdots, v_n^{m_n} := v_n = free?0 : v_n^{m_n}]. \end{aligned}$$

We distinguish the following cases:

* $\exists v_j = t.m[j] \in V(t)$ with $\sigma_i^f(t.m[j]) = free$. Then $\sigma_{i+1}(v) = index(t.m[j])$, and for all variables $t.m'[j] \in MV(t.m[j])$ we have $\sigma_{i+1}(t.m'[j]) = 0$. If we change π s.t. $\pi(l) = t.m[j]$ and $\pi(l') = t.m'[j]$ for all locations l' belonging to member variables m' of the object instance created at location l, then all conditions of definition 5.3 are fulfilled. Therefore, it follows that $\sigma_{i+1} \sim_\pi \sigma_{i+1}^f$ and $c_{i+1} \sim_\pi c_{i+1}^f$.
* $\forall v_j \in V(t) : \sigma_i^f(v_j) \neq free$. Then $\sigma_{i+1}(v) = top$, and all conditions of definition 5.3 are fulfilled. Therefore, it follows that $\sigma_{i+1} \sim_\pi \sigma_{i+1}^f$ and $c_{i+1} \sim_\pi c_{i+1}^f$.

– (Object destruction): This case can be shown analogously to the case of object construction.

□

By proving theorem 5.16 we have shown that we can construct for a given C_{min} program P a C_{fin} program $P^f(m)$ that simulates P. Since $P^f(m)$ simulates P and since all variables $v \in Vars(P)$ are also contained in $P^f(m)$, from lemma 5.7 we can conclude that $P^f(m)$ is an abstraction of P w.r.t. any formula $EFq \in TL$. As a consequence, we could now apply an appropriate finite-state model checker to build the complete state space of $P^f(m)$ in order to create the abstraction-based heuristic function. Unfortunately, due to the possible large domains of the variables used in $P^f(m)$, the reachable state space will be much too large for programs of reasonable size. Therefore, in the next section we show how we can further abstract $P^f(m)$ into a more abstract program whose state space will be small enough s.t. we can explore its state space completely.

5.1.4 Further Abstraction

In this section, we will describe how we can further abstract $P^f(m)$ into a program $B(P^f(m), Preds)$ by using a technique known as *predicate abstraction* [GS97, DDP99]. Contrary to $P^f(m)$, the program $B(P^f(m), Preds)$ will have only boolean variables denoted with $B = \{b_1, \cdots, b_n\}$. The number of boolean variables is determined by the set $Preds = \{p_1, \cdots, p_n\}$ of so-called *predicates*. A predicate p is a boolean expression over variables in $P^f(m)$, i.e. $p_i \in BExp(Vars(P^f(m)))$ for $i \in \{1, \cdots, n\}$. Concrete states of the system are mapped to abstract states according to the evaluation of the given predicates. A mapping $\beta : Preds \to B$ maps each predicate p_i to a corresponding boolean variable $\beta(p_i) = b_i$ in B. A concrete state σ^f of $P^f(m)$ corresponds to an abstract state σ^a of $B(P^f(m), Preds)$ if for all b_i the value of b_i in σ^a is equivalent to the value of p_i in σ^f. Given a state $\sigma^f \in States(P^f(m))$, $abs(\sigma^f)$ denotes a state of $B(P^f(m), Preds)$ with

$$\forall i \in \{1, \cdots, n\} : abs(\sigma^f)(\beta(p_i)) = [\![p_i]\!](\sigma^f).$$

For a configuration $c = (pc_1, \cdots, pc_m, \sigma^f)$ of $P^f(m)$, $abs(c)$ denotes a configuration of $B(P^f(m), Preds)$ with

$$abs(c) = (pc_1, \cdots, pc_m, abs(\sigma^f)).$$

We can now define what it means that $B(P^f(m), Preds)$ is an *abstraction* of $P^f(m)$.

Definition 5.17 *Let $P^f(m)$ be a C_{fin} program and $Pred = \{p_1, \cdots, p_n\}$ a set of predicates $p_i \in BExp(Vars(P^f(m))$. Let further $B(P^f(m), Preds)$ be a C_{fin} program with $Vars(B(P^f(m), Preds)) = \{b_1, \cdots, b_n\}$ and $type(b_i) = bool, i \in \{1, \cdots, n\}$. We say $B(P^f(m), Preds)$ is an abstraction of $P^f(m)$ iff*

$$\langle c_0, c_1, \cdots \rangle \in Runs(P^f(m)) \Rightarrow \langle abs(c_o), abs(c_1), \cdots \rangle \in Runs(B(P^f(m), Preds))$$

If we want to check if $P^f(m) \vDash EFq$, and if $B(P^f(m), Preds)$ is an abstraction of $P^f(m)$ and $q \in Preds$, then instead of performing a state space exploration of $P^f(m)$ we can perform an exploration of $B(P^f(m), Preds)$ whose state space is usually much smaller than the state space of $P^f(m)$. Subject to the result of the state space exploration of $B(P^f(m), Preds)$ we can draw conclusions about the behavior of $P^f(m)$.

Lemma 5.18 *Let $P^f(m)$ be a C_{fin} program, $Pred = \{p_1, \cdots, p_n\}$ a set of predicates $p_i \in BExp(Vars(P^f(m))$ and $EFp_k \in TL$ a formula with $p_k \in Preds$. Let further $B(P^f(m), Preds)$ be an abstraction of $P^f(m)$. Then it holds*

$$(P^f(m) \vDash EFp_k) \Rightarrow (B(P^f(m), Preds) \vDash EF\beta(p_k)).$$

Proof:
Let

$$\langle c_0, c_1, \cdots, c_j = (pc_1, \cdots, pc_m, \sigma_j^f) \rangle \in Runs(P^f(m))$$

with $[\![p_k]\!](\sigma_j^f) = true$. Because $B(P^f(m), Preds)$ is an abstraction of $P^f(m)$ we have

$$\langle abs(c_o), abs(c_1), \cdots, abs(c_j) = (pc_1, \cdots, pc_m, abs(\sigma_j^f))\rangle \in Runs(B(P^f(m), Preds)).$$

Since $[\![p_k]\!](\sigma_j^f) = abs(\sigma_j^f)(\beta(p_k))$ we have $B(P^f(m), Preds) \vDash EF\beta(p_k)$. □

When a complete state space exploration of $B(P^f(m), Preds)$ yields $B(P^f(m), Preds) \nvDash \beta(p_k)$, then we can safely conclude $P^f(m) \nvDash p_k$. However, if the result yields $B(P^f(m), Preds) \vDash \beta(p_k)$, then both $P^f(m) \nvDash p_k$ and $P^f(m) \vDash p_k$ are possible. In the latter case we will use the generated state space to build an abstraction-based heuristic which itself can be used for a heuristic state space exploration of the original program P. In the following we will describe how we transform a C_{fin} program $P^f(m)$ into a C_{fin} program $B(P^f(m), Preds)$ that is an abstraction of $P^f(m)$.

For a statement $v := e$ and a predicate p, let $WP(v := e, p)$ denote the *weakest liberal precondition* [Dij76, Gri81] of p with respect to the statement $v := e$. $WP(v := e, p)$ is defined as the weakest predicate whose truth before $v := e$ entails the truth of p afterwards. The standard weakest precondition rule says that $WP(v := e, p)$ is p with all occurrences of v replaced with e, denoted $p[e/v]$. For instance

$$WP(v := v + 2, v < 8) = (v + 2) < 8 = v < 6.$$

Given a statement $v := e$, a set $Preds = \{p_1, \cdots, p_n\}$ of predicates and a predicate $p_j \in Preds$, it can happen that $WP(v := e, p_j) \notin Preds$. In such a case we can *strengthen* the weakest precondition to an expression over the predicates in $Preds$. A *monomial* m is a conjunction $c1 \wedge \cdots \wedge c_n$ where each c_i is one of $\{b_i, \neg b_i\}$. We extend the function β to range also over monomials, i.e.

$$\beta(c_1 \wedge \cdots \wedge c_n) = \beta(c_1) \wedge \cdots \wedge \beta(c_n)$$

and disjunctions of monomials, i.e.

$$\beta(m_1 \vee \cdots \vee m_n) = \beta(m_1) \vee \cdots \vee \beta(m_n).$$

Based on this, we formalize the strengthening of predicates as follows:

Definition 5.19 *Let $Preds = \{p_1, \cdots, p_n\}$ be a set of predicates, e a boolean expression and $B = \{b_1, \cdots, b_n\}$ a set of boolean variables. With $W(e) = m_1 \vee \cdots \vee m_k$ we denote the largest disjunction of monomials m_i such that $\beta^{-1}(m_i) \Rightarrow e$.*

The predicate $\beta^{-1}(W(e))$ represents the weakest predicate over $Preds$ that is stronger than e. For instance, if $Preds = \{v < 8, v = 3\}$ and $B = \{b_1, b_2\}$, then

$$W(v < 6) = (b_1 \wedge b_2) \vee (\neg b_1 \wedge b_2).$$

With the notion of the weakest precondition we can now define the transformation of assignments. Given a set of predicates $Preds = \{p_1, \cdots, p_n\}$, a function β as described above and an assignment $v := e$, $B(v := e)$ is defined as

$$
\begin{array}{l}
[\\
\quad \beta(p_1) := \quad W(WP(v := e, p_1))?true : W(WP(v := e, \neg p_1))?false : ND(bool), \\
\quad \quad \cdots \\
\quad \beta(p_n) := \quad W(WP(v := e, p_n))?true : W(WP(v := e, \neg p_n))?false : ND(bool) \\
]
\end{array}
$$

The translated statement $B(v := e)$ must appropriately update all of the boolean variables that correspond to predicates that are affected by the assignment. If $WP(v := e, p)$ is true, then $\beta(p)$ may be safely set to true. Similarly, if $WP(v := e, \neg p)$ is true, then $\beta(p)$ may be safely set to false. Because the predicate $WP(v := e, p)$ may not be in $Preds$ we need to weaken it to a predicate over expressions in $Preds$ that implies $WP(v := e, p)$, therefore we compute $W(WP(v := e, p))$. Similarly we weaken $WP(v := e, \neg p)$. However, if neither $W(WP(v := e, p))$ nor $W(WP(v := e, \neg p))$ is true, i.e., we cannot determine the value of $\beta(p)$ after execution of $v := e$, then we conservatively update $\beta(p)$ nondeterministically with true or false. Analogously to assignments, for *await*- and *jump*-statements with a guard e we have to find the weakest predicate over expressions in $Preds$ that implies e resp. $\neg e$, i.e., we transform such a guard e into another guard

$$B(e) = W(e)?true : W(\neg e)?false : ND(bool).$$

For instance, if we have a guard $v > 0$ and $Preds = \{v > 2, v = 0\}$, then

$$B(v > 0) = v > 2?true : v = 0?false : ND(bool).$$

Based on this, we can now define the translation of arbitrary statements.

Definition 5.20 *Let $P^f(m)$ be a C_{fin} program and $stm \in STM(P^f(m))$ be a statement of $P^f(m)$. Then $B(stm)$ is defined as follows:*

- *(Concurrent assignment):*
  if $stm \equiv [v_1 := e_1, \cdots, v_n := e_n]$ is a concurrent assignment, then
  $$B(stm) = [B(v_1 := e_1), \cdots, B(v_n := e_n)].$$
- *(Synchronization):*
  if $stm \equiv await(e, [v_1 := e_1, \cdots, v_n := e_n]))$ is an await statement,then
  $$B(stm) = await(B(e), [B(v_1 := e_1), \cdots, B(v_n := e_n)]).$$
- *(Branching):*
  if $stm \equiv jump(e, k_1, k_2)$ is a jump statement, then
  $$B(stm) = jump(B(e), k_1, k_2).$$

Given a statement sequence $stms = \langle(1, stm_1), \cdots, (n, stm_n)\rangle$, $B(stms)$ denotes the statement sequence with $B(stm) = \langle(1, B(stm_1)), \cdots, (n, B(stm_n))\rangle$. Based on this, the following definition formalizes the complete translation of $P^f(m)$ into $B(P^f(m), Preds)$.

Definition 5.21 *Let* $P^f(m) = (Vars(P^f(m)), init(P^f(m)), Threads(P^f(m)))$ *be a* C_{fin} *program with*

$$Threads(P^f(m)) = \{(1, stms_1), \cdots, (n, stms_n)\}.$$

Let further $Pred = \{p_1, \cdots, p_n\}$ *be a set of predicates* $p_i \in BExp(Vars(P^f(m))$. *Then* $B(P^f(m), Preds) = (Vars(B(P^f(m))), init(B(P^f(m))), Threads(B(P^f(m))))$ *is defined as:*

- $Vars(B(P^f(m), Preds) = \{b_1, \cdots, b_n\}$.
- $init(B(P^f(m)))$ *is an initialization function that is defined as follows:*

$$init(B(P^f(m)))(b_i) = \begin{cases} true & \text{if } [\![\beta^{-1}(b_i)]\!](\sigma_0^f) = true \\ false & \text{otherwise,} \end{cases}$$

  whereby σ_0^f *denotes a state with* $\sigma_0^f(v_i) = init(P^f(m))(v_i)$ *for all* $v_i \in Vars(P^f(m))$.

- $Threads(B(P^f(m), Preds)) = \{(1, B(stms_1)), \cdots, (n, B(stms_n))\}$.

After defining $B(P^f(m), Preds)$ formally, we can now prove that it is indeed an abstraction of $P^f(m)$, which is stated in the following theorem.

Theorem 5.22 *Let* $P^f(m)$ *be a* C_{fin} *program and* $Pred = \{p_1, \cdots, p_n\}$ *a set of predicates* $p_i \in BExp(Vars(P^f(m))$. *Then* $B(P^f(m), Preds)$ *is an abstraction of* $P^f(m)$.

Proof:
(By induction). We have to show that

$$\langle c_0^f, c_1^f, \cdots\rangle \in Runs(P^f(m)) \Rightarrow \langle abs(c_0^f), abs(c_1^f), \cdots\rangle \in Runs(B(P^f(m), Preds))$$

Let

$$\langle c_0^f, c_1^f, \cdots, c_j^f = (pc_1, \cdots, pc_m, \sigma_j^f)\rangle \in Runs(P^f(m))$$

be a run of $P^f(m)$. We construct a corresponding abstract run

$$\langle abs(c_0^f), abs(c_1^f), \cdots abs(c_j^f) = (pc_1, \cdots, pc_m, \sigma_j^a)\rangle \in Runs(B(P^f(m), Preds)).$$

- Case $j = 0$: From the definition of the starting configuration $c_0^f = (first_1, \cdots, first_m, \sigma_0^f)$ for $P^f(m)$ we know that

$$\forall v \in Vars(P^f) \,:\, \sigma_0^f(v) = init(v).$$

Additionally, from the definition of the starting configuration $c_0^a = (first_1, \cdots, first_m, \sigma_0^a)$ we know that $\forall b_i \in Vars(B(P^f(m)))$ it holds that

$$init(B(P^f(m)))(b_i) = \begin{cases} true & \text{if } [\![\beta^{-1}(b_i)]\!](\sigma_0^f) = true \\ false & \text{otherwise.} \end{cases}$$

Thus, we have

$$\forall i \in \{1, \cdots, n\} : [\![p_i]\!](\sigma_0^f) = [\![\beta(p_i)]\!](\sigma_0^a),$$

and therefore $\sigma_0^a = abs(\sigma_0^f)$ resp. $c_0^a = abs(c_0^f)$.

- Case $j+1$: Assume we already constructed

$$\langle abs(c_0^f), abs(c_1^f), \cdots, abs(c_j^f) = (pc_1, \cdots, pc_m, \sigma_j^a)\rangle.$$

Suppose that in configuration c_j^f thread i executes a statement stm_j s.t.

$$c_{j+1}^f = (pc_1, \cdots, pc_i', \cdots, pc_m, \sigma_{j+1}^f).$$

We then construct $abs(c_{j+1}^f)$ subject to c_{j+1}^f.

  - Case $stm_j = [v_1 := e_1, \cdots, v_k := e_k]$: Then

$$\sigma_{j+1}^f = \sigma_j^f[v_1 := [\![e_1]\!](\sigma_j^f), \cdots, v_k := [\![e_k]\!](\sigma_j^f)]$$

From definition of $B(P^f(m), Preds)$ we know that

$$B(stm_j) = [B(v_1 := e_1), \cdots, B(v_k := e_k)].$$

Now let p_r be an arbitrary predicate in $Preds$. If no assignment in stm_j changes a variable of p_r, then

$$WP(v_i := e_i, p_r) = p_r \text{ and } WP(v_i := e_i, \neg p_r) = \neg p_r$$

for all $i \in \{1, \cdots, k\}$, and therefore also

$$W(WP(v_i := e_i, p_r)) = p_r \text{ and } W(WP(v_i := e_i, \neg p_r)) = \neg p_r$$

for all $i \in \{1, \cdots, k\}$. Since

$\beta(p_r) := W(WP(v_i := e_i, p_r))?true : W(WP(v_i := e_i, \neg p_r))?false : ND(bool)$

we know that if $[\![p_r]\!](\sigma^f_{j+1}) = true$ then either

$$[\![W(WP(v_i := e_i, p_r))]\!](\sigma^a_j) = true \text{ and therefore } [\![\beta(p_r)]\!](\sigma^a_{j+1}) = true$$

or

$$[\![W(WP(v_i := e_i, p_r))]\!](\sigma^a_j) = false \text{ and } [\![W(WP(v_i := e_i, \neg p_r))]\!](\sigma^a_j) = false,$$

in which case we have $\beta(p_r) := ND(bool)$, so we can choose s.t. $[\![\beta(p_r)]\!](\sigma^a_{j+1}) = true$. The case $[\![p_r]\!](\sigma^f_{j+1}) = false$ proceeds analogously. If on the other hand there is at least one assignment in stm_j that changes a variable of p_r then there is a last assignment $v_i := e_i$ in stm_j that changes a variable of p_r. If $[\![p_r]\!](\sigma^f_{j+1}) = true$ then either

$$[\![W(WP(v_i := e_i, p_r))]\!](\sigma^a_j) = true \text{ and therefore } [\![\beta(p_r)]\!](\sigma^a_{j+1}) = true$$

or

$$[\![W(WP(v_i := e_i, p_r))]\!](\sigma^a_j) = false \text{ and } [\![W(WP(v_i := e_i, \neg p_r))]\!](\sigma^a_j) = false,$$

in which case we have $\beta(p_r) := ND(bool)$, so we can choose s.t. $[\![\beta(p_r)]\!](\sigma^a_{j+1}) = true$. Again, the case $[\![p_r]\!](\sigma^f_{j+1}) = false$ proceeds analogously. Therefore we can conclude that $abs(c^f_j) \rightarrow abs(c^f_{j+1})$.

– Case $stm_j = await(e, [v_1 := e_1, \cdots, v_k := e_k])$: c^f_j can make a transition to c^f_{j+1} only when $[\![e]\!](\sigma^f_j) = true$. Since

$$B(stm_j) = await(B(e), [B(v_1 := e_1), \cdots, B(v_n := e_n)])$$

and

$$B(e) = W(e)?true : W(\neg e)?false : ND(bool),$$

either $[\![W(e)]\!](\sigma^a_j) = true$ and therefore $[\![B(e)]\!](\sigma^a_j) = true$, or $[\![W(e)]\!](\sigma^a_j) = false$ and $[\![W(\neg e)]\!](\sigma^a_j) = false$, in which case we have $B(e) = ND(bool)$, so we can choose s.t. $[\![B(e)]\!](\sigma^f_j) = true$. The updates due to the assignment sequence can be handled as in the proof of the assignment sequence, therefore we can conclude that $abs(c^f_j) \rightarrow abs(c^f_{j+1})$.

– Case $stm_j = jump(e, l_1, l_2)$: If $next(pc_i) = l_1$ then $[\![e]\!](\sigma^f_j) = true$. Since

$$B(stm_j) = jump(B(e), k_1, k_2)$$

and

$$B(e) = W(e)?true : W(\neg e)?false : ND(bool),$$

either $[\![W(e)]\!](\sigma^a_j) = true$ and therefore $[\![B(e)]\!](\sigma^a_j) = true$, or $[\![W(e)]\!](\sigma^a_j) = false$ and $[\![W(\neg e)]\!](\sigma^a_j) = false$, in which case we have $B(e) = ND(bool)$,

so we can choose s.t. $[\![B(e)]\!](\sigma_j^f) = true$. If $[\![e]\!](\sigma_j^f) = false$ and $next(pc_i) = l_2$, then either $[\![W(\neg e)]\!](\sigma_j^a) = true$ and therefore $[\![B(e)]\!](\sigma_j^a) = false$, or $[\![W(e)]\!](\sigma_j^a) = false$ and $[\![W(\neg e)]\!](\sigma_j^a) = false$, in which case we have $B(e) = ND(bool)$, so we can choose s.t. $[\![B(e)]\!](\sigma_j^f) = true$, therefore we can conclude that $abs(c_j^f) \rightarrow abs(c_{j+1}^f)$.

□

After we have formally described how to create, starting with an arbitrary C_{min} program P, a C_{fin} program $B(P^f(m), Preds)$ that is a finite state abstraction of P, in the next section we will demonstrate how we can use $B(P^f(m), Preds)$ to create an appropriate heuristic function h that can be used to guide an exploration process of P to find a witness for a given formula EFq.

5.2 Abstraction-based Heuristic State Space Exploration

Before we explain how we can use $B(P^f(m), Preds)$ to create an heuristic function h that can be used for state space exploration of P, we sum up the overall abstraction process described in the previous sections. Figure 5.5 shows the flow of abstractions we have to perform to construct $B(P^f(m), Preds)$. Starting with a C_{min} program P, the first abstraction yields a finite state C_{fin} program $P^f(m)$. Although the state space of

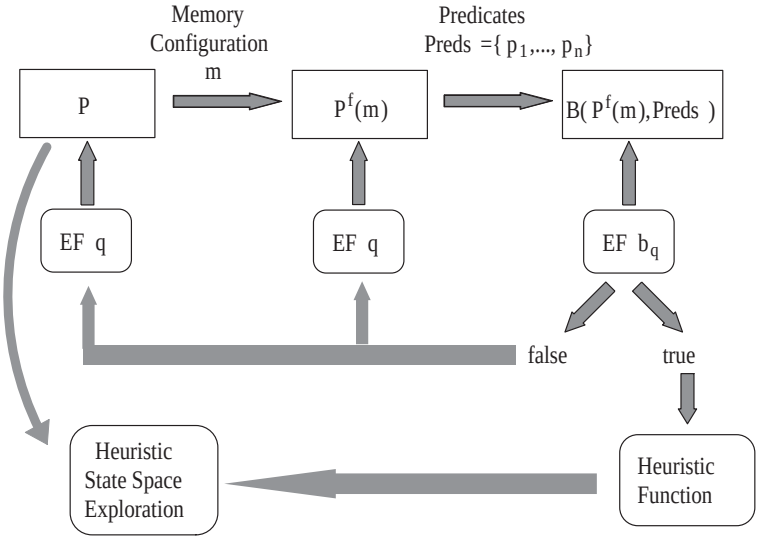


Figure 5.5: **The overall abstraction process**. Starting with P, we first abstract P into $P^f(m)$, and subsequently into $B(P^f(m), Preds)$. A complete state space exploration of $B(P^f(m), Preds)$ yields a heuristic function that we can then use for heuristic state space exploration of P.

$P^f(m)$ is finite, it can still be much too large for an exhaustive state space exploration. Therefore, with another abstraction we get a more abstract program $B(P^f(m), Preds)$. Due to the fact that $B(P^f(m), Preds)$ is more abstract than $P^f(m)$, it is much more likely that we can perform an exhaustive state space exploration of $B(P^f(m), Preds)$. If we assume that the property to be checked is EFq, as the result of such an exhaustive exploration we get $B(P^f(m), Preds) \not\models EFq$ or $B(P^f(m), Preds) \models EFq$. Because $B(P^f(m), Preds)$ is an abstraction of $P^f(m)$ and $P^f(m)$ itself is an abstraction of P, in the case $B(P^f(m), Preds) \not\models EFq$ we can conclude that both $P^f(m) \not\models EFq$ and $P \not\models EFq$.

Unfortunately, in the case $B(P^f(m), Preds) \models EFq$ we cannot conclude either $P^f(m) \models EFq$ or $P \models EFq$, since a witness for $B(P^f(m), Preds) \models EFq$ need not necessarily be a witness for $P^f(m) \models EFq$ or $P \models EFq$. However, if there is at least one witness for $P \models EFq$, then there is also a witness for $P^f(m) \models EFq$ and $B(P^f(m), Preds) \models EFq$. Therefore we can try to use a witness for $B(P^f(m), Preds) \models EFq$ as a hint to search for a witness for $P \models EFq$.

We will exploit the information gained by the exhaustive state space exploration of $B(P^f(m), Preds)$ by means of a so-called *abstraction-based heuristic*. For a property EFq, in the abstraction-based heuristic we store for each state of $B(P^f(m), Preds)$ the shortest path in the state graph of $B(P^f(m), Preds)$ that leads to a configuration c^a with $c^a \models q$. For a sequence $r^a = \langle c_1^a \rightarrow c_2^a \rightarrow \cdots \rightarrow c_n^a \rangle$ of consecutive configurations, let $r^a(i)$ denote the postfix of r^a starting at configuration c_i^a. Furthermore, let $len(r^a) = n$ denote the length of a sequence r^a. The function $dist : Conf(B(P^f(m), Preds) \times TL \rightarrow \mathbb{N}$ with

$$dist(c_i^a, EFq) = \begin{cases} min\{len(r^a(i)) | r^a = \langle c_0^a, \cdots, c_n^a \rangle \in Runs(B(P^f(m), Preds)) \wedge c_n^a \models q\} \\ \infty \text{ otherwise} \end{cases}$$

computes for a configuration c_i^a the minimal number of transitions that must be taken to reach a configuration c_j^a with $c_j^a \models q$. For a property EFq, to create the heuristic we compute the value $dist(c_i^a, EFq)$ for every configuration c_i^a that is reachable in $B(P^f(m), Preds)$. After creating the abstraction-based heuristic, in a subsequent state space exploration of P we will compute a value for a configuration c by means of the heuristic function h. If we encounter a configuration c during the exploration of P, we compute the corresponding abstract configuration c^a of $B(P^f(m), Preds)$ and retrieve the computed value stored for c^a. Let c be a configuration of P, c^a the corresponding abstract configuration of $B(P^f(m), Preds)$ and EFq a property, then we define the heuristic function $h : Conf(P) \times TL \rightarrow \mathbb{N}$ as

$$h(c, EFq) = dist(c^a, EFq).$$

This function h can effectively be computed by the algorithm shown in fig. 5.6. The depicted algorithm is just a little variation of the standard breadth-first algorithm for computing shortest paths. We assume that we have already computed the reachable state space of a program $B(P^f(m), Preds)$ and that we have marked all states that fulfill a property q. Furthermore, we assume that for every state s of $B(P^f(m), Preds)$ we store

```
(1)   procedure CreateHeuristic(q)
(2)     Dist ← ∅; Closed ← ∅;
(3)     ActStates ← {s|s ⊨ q}; d ← 0
(4)     while (ActStates ≠ ∅)
(5)       PredStates ← ∅
(6)       Closed = Closed ∪ ActStates
(7)       foreach v ∈ Actstates
(8)         Dist.insert(v, d)
(9)         if (v.pred ∉ Closed)
(10)          PredStates.insert(v.pred)
(11)      d = d + 1
(12)      ActStates = PredStates
(13)  end procedure CreateHeuristic
(14)
(15)  procedure h(c)
(16)    c^a = abs(c);
(17)    return Dist(c^a);
(18)  end procedure h
```

Figure 5.6: **Generation of the abstraction-based heuristic.** The algorithm `CreateHeuristic` performs a backward traversal of the abstract state space to store for every abstract state a minimal transition distance to the nearest goal state in the set *Dist*. The algorithm `h` maps a concrete state to the corresponding abstract one and returns the value of the abstract state stored in *Dist*.

a predecessor link *s.pred*. Based on this, the algorithm in 5.6 performs a backward traversal starting from the set of states fulfilling q. For each state s encountered during this backward traversal we store in *Dist* the distance d to the nearest state fulfilling q. After having computed the abstraction-based heuristic and therefore defined the heuristic function h, h can now be used in a heuristic search algorithm like the *weighted A** algorithm [Poh70] which is depicted in fig. 5.7. In the weighted A* algorithm, we compute for every configuration c_i a value

$$f(c_i) = g(c_i) + w * h(c_i) = g(c_{i-1}) + 1 + w * h(c_i),$$

whereby $g(c_i)$ denotes the length of the path from the starting configuration c_0 to c_i, i.e., the value $f(c)$ is the weighted sum between between the path length of c_i and the product of the *weighting factor* w and the heuristic estimate $h(c_i)$. In contrast to the simple best-first search algorithm described in sect. 2.2, in the weighted A* algorithm the weighting factor w determines how far the heuristic estimate $h(c_i)$ influences the value $f(c_i)$. For $w = 0$, weighted A* works like BFS, and for $w = 1$ it works like standard A*

```
(1)   procedure WA*
(2)   Closed ← ∅; Open ← ∅
(3)   Open.insert(s_0, h(s_0))
(4)   while (Open ≠ ∅)
(5)      u ← Open.delmin();
(6)      Closed.insert(u)
(7)      if (goal_reach(u))
(8)         return Goal_Reached
(9)      foreach successor v of u
(10)        f'(v)=g(v)+w*h(v) =
(11)           g(u) + 1 + w * h(v)
(12)        if (v ∉ Closed)
(13)           Open.insert(v, f'(v))
(14)     end procedure WA*
```

Figure 5.7: **Weighted A* algorithm**

[HNR68]. For larger values $w > 1$, weighted A* behaves more like best first search.

To illustrate the usage of the abstraction-based heuristic, consider the situation depicted in fig. 5.8. In the top left we see the grid-like state space of a program P under consideration. The big circle in the middle of the grid indicates the starting state of P, and the square in the bottom right of the grid indicates a goal state, i.e., a state of P that fulfills a certain property q. According to our abstraction-based heuristic search procedure, we compute an abstraction $B(P^f(m), Preds)$ and perform a exhaustive state space exploration of $B(P^f(m), Preds)$. In the top right of fig. 5.5 we can see thereby generated abstract state space. In the abstract state space, several states of the original state space are summarized into one abstract state. Additionally, due to the abstraction process there are also two additional abstract goal states, one at the bottom left of the abstract grid and one at the bottom middle. These two abstract goal states are reachable in the abstract state space, but not in the concrete one. However, due to the fact that $B(P^f(m), Preds)$ is an abstraction of P, there is also an abstract counterpart for the only concrete goal state at the bottom right of the abstract grid. After applying algorithm 5.6 for creating the abstraction-based heuristic, the heuristic contains for each abstract state a distance value corresponding to the color table depicted at the right of fig. 5.5. If we perform a state space search of P using BFS, we would generate almost the entire state space before finding the goal state (v. fig. 5.5 bottom left). On the other hand, if we perform a heuristic state space search of P using the weighted A* algorithms with $w = 3$, we would only generate the part of the state space depicted in the bottom right of fig. 5.5. During heuristic search, a state of the concrete state space would only be generated if the distance of the corresponding abstract state to the nearest abstract goal state is less or equal 2. All concrete states whose corresponding abstract states have

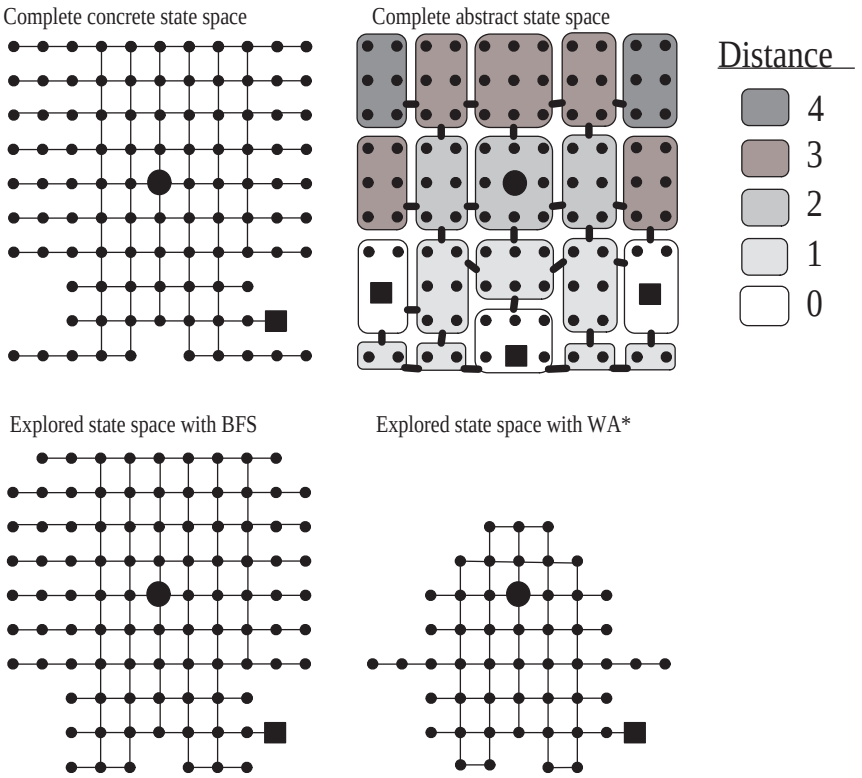


Figure 5.8: **Abstraction-based heuristic state space exploration.** When using BFS to find a goal state (black square), almost the entire state space is explored (bottom left). When using the abstraction based heuristic for heuristic state space exploration, only the fraction of the state graph depicted in the bottom right is explored.

a distance $d > 2$ to an abstract goal state are not generated.

For the example in fig. 5.5, the abstraction-based heuristic search procedure generates around a half of the number of states generated with an uninformed search algorithm like BFS. However, the benefit of applying the abstraction-based heuristic search procedure can sometimes be much higher. For instance, consider the sample program depicted in the top left of fig. 5.9. We assume that $N > 1$ is a predefined constant. In the main loop of the program, the variable x is incremented, and the possible values of the variables y and z increase with every iteration of the loop. In the top right one can see the state space generated by a BFS state space exploration up to search depth 3. Given a search depth d, it is easy to see that BFS has to generate $2^d - 1$ states until a state of depth d has been generated. Suppose a goal state should have the property $x = N \wedge z = 0$. Then BFS would has to generate at least $2^N - 1$ states before a goal state can be found. Since it is seldom possible to generate more than a few million states within a

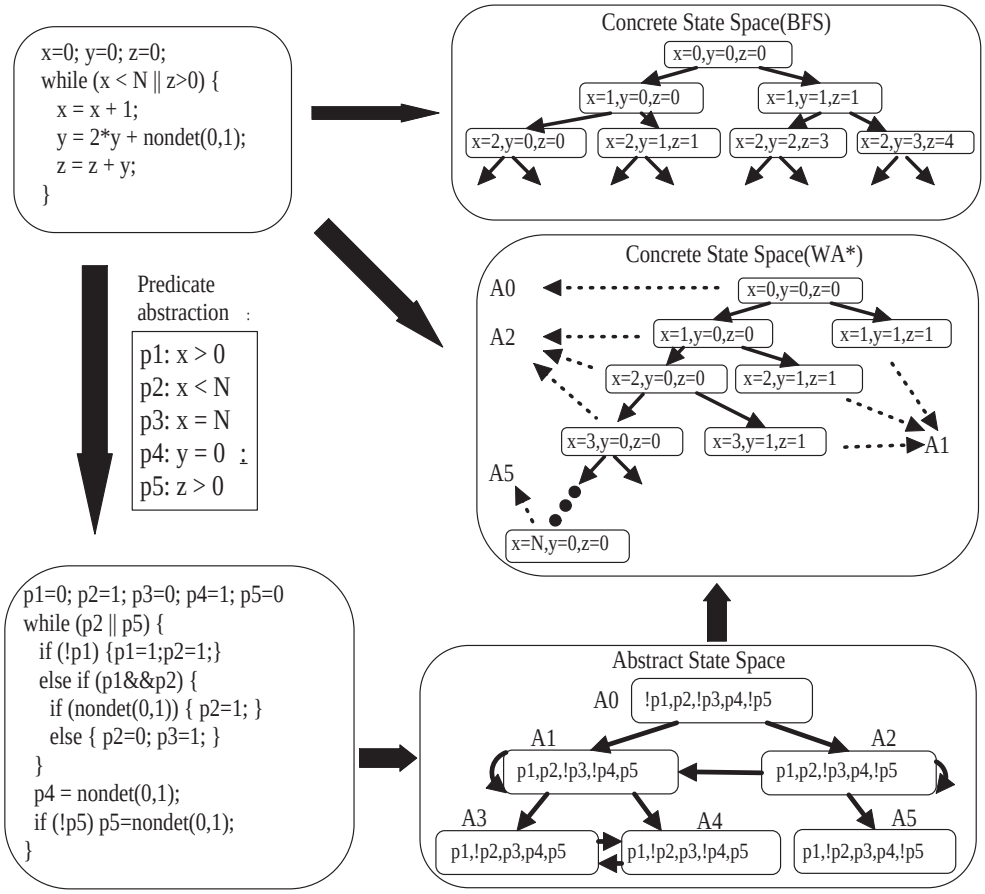


Figure 5.9: **Saving an exponential number of states with heuristic state space exploration.**

reasonable amount of time and space, for values $N > 25$ it will be hard or impossible to find a goal state with BFS. Contrary to this, applying our abstraction-based heuristic search procedure allows us to find goal states for very large N also. According to our procedure, given some abstraction predicates we compute an abstract program depicted in the bottom left of fig. 5.9. We then perform an exhaustive state space exploration of this abstract program retrieving the abstract state space shown in the bottom right of fig. 5.9, where the abstract state $A5$ is the only abstract goal state. After computing the abstraction-based heuristic ($A0 = 2, A1 = \infty, A2 = 1, A3 = \infty, A4 = \infty, A5 = 0$) we can then perform the abstraction-based heuristic search, which is shown in the middle of fig. 5.9. When expanding the starting state we get two successor states, one that is mapped to $A2$ and one that is mapped to $A1$. Since $dist(A2) < dist(A1)$, the next state to expand is the state mapped to $A2$. After expanding this state, we get again two successor states, one belonging to $A2$ and one belonging to $A1$. Therefore, we again

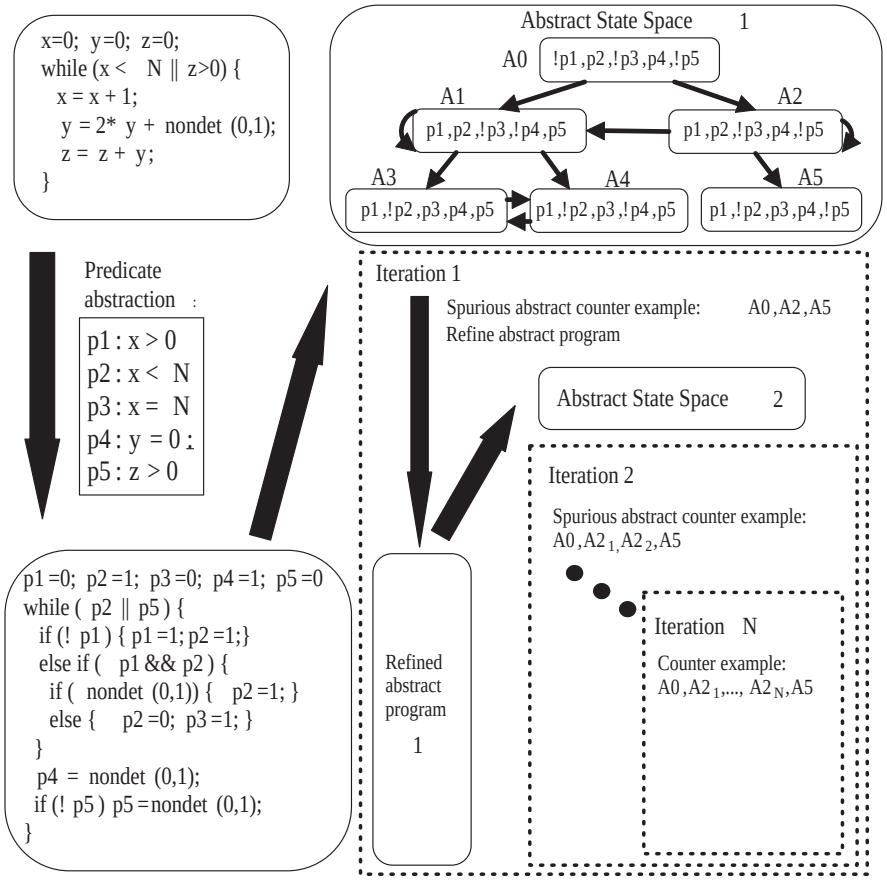


Figure 5.10: **Abstraction-Refinement.** Contrary to the heuristic state space exploration shown in fig. 5.9, model checking approaches based on abstraction-refinement would have to generate N refinements adding altogether N predicates. If N is large, the state space of the abstraction becomes too large to be analyzed completely.

expand the state belonging to $A2$. This procedure will continue until finally we reach the concrete goal state $(x = N, y = 0, z = 0)$ that is mapped to $A5$. Obviously, when using the abstraction-based heuristic search procedure, only $2N$ states are generated before finding a goal state, which allows us to find goal states even for very large $N > 10^6$.

The example in fig. 5.9 demonstrates also the potential benefit of the abstraction-based heuristic search procedure with regard to approaches based on abstraction-refinement. Figure 5.10 on the following page shows the application of the abstraction-refinement approach to our sample program. As described in sect. 2.1, in abstraction-refinement the state space exploration is performed on the abstract state graph. Since model check-

ers usually find shortest counterexamples[1], in our example a model checker would find the abstract path $\langle A0, A2, A5\rangle$. Checking this counterexample would detect that this counterexample is spurious, i.e., there is no corresponding concrete counterexample. Therefore, based on the generated abstract counterexample, the current abstraction is refined by adding one or more predicates that eliminate this abstract counterexample. To rule out the given abstract counterexample, one has to split state $A2$ into at least two new states $A2_1$ and $A2_2$, for example by adding a predicate $x = 1$ to the current set of predicates. Exploring the state graph of the refined abstract program would yield the spurious abstract counterexample $\langle A0, A2_1, A2_2, A5\rangle$. Again, the state $A2_2$ has to be split into two new states, for instance by adding the predicate $x = 2$. However, the abstraction has to be refined until $N - 1$ states $A2_1, \cdots, A2_{N-1}$ can be distinguished in the abstract state graph, whereby each $A2_i$ correspond to a concrete state that fulfills $x = i$. To distinguish all $N - 1$ different values of x means that we have to add at least $\log_2 N - 1$ predicates to the initial set, if we encode the value of x in a binary way, or $N - 1$ predicates if we encode each value with a single predicate. For each iteration, a new abstract program has to be generated, which is often quite time consuming since the computation of the abstract transitions usually requires the usage of decision procedures. Additionally, since the number of predicates grows with each iteration, the time and space needed for the generation of the abstract program and exploration of the abstract state space will also grow. In practice, if the number of predicates used in the abstract program exceeds a certain threshold, model checking the abstract program will be similar complex as model checking the original program.

Nevertheless, refining an abstraction can also be beneficial for our abstraction-based heuristic search procedure. If the computed abstraction is too coarse, i.e., it contains too many spurious transitions, then it may be the case that the abstraction-based heuristic is not informative enough to give good guidance towards real goal states. In such a case, refining the abstraction can improve the accuracy of the abstraction-based heuristic. An abstraction $B(P^f(m), Preds)$ can be refined by adding additional predicates that rule out certain spurious transitions. However, similarly to model-checking approaches based on abstraction-refinement, adding more predicates usually increases the number of states of the abstraction, i.e., care must be taken that the state space of the abstraction remains small enough s.t. it can be completely generated. As already described in sect. 2.1, several model-checking approaches define a fully automated abstraction-refinement scheme where no user-interaction is necessary to create the abstractions or refinements. Contrary to this, in our approach the user has to provide a memory configuration m and the set $Preds$ of abstraction predicates. Automating these manual steps is subject of future work and will be discussed in sect. 6.2. In the next section, we experimentally evaluate the described abstraction-based heuristic exploration procedure.

[1] In our context, the term "counterexample" is just a synonym for a witness of the negation of a property.

5.3 Experimental Results

To evaluate the effectiveness of the abstraction-based heuristic search procedure presented in the previous section, we apply it to our collection of test programs we already used for the evaluation of the explicit-symbolic state space exploration. For each program in the test suite we check a formula stating the reachability of a particular state of an object used in the program. For our abstraction-based heuristic search procedure, we computed two abstractions A1 and A2 s.t. A2 is a refinement of A1. To compute an abstraction for a program P, we firstly define an appropriate memory configuration m so that we are able to compute the finite-state program $P^f(m)$. For the generation of the boolean program $B(P^f(m), Preds)$ we manually create a set of abstraction predicates. The predicates we choose mainly correspond to conditionals in the program that use variables of the object referenced in the formula. For each abstraction, we list the number of predicates (#preds), the number of states (#states) and the time needed to generate the state space of the abstraction. Furthermore, for each program we list the number of generated states (#states) resp. the time needed to find a goal state. We measure the number of generated states and the search time for 3 different configurations: BFS, WA* using the heuristic based on A1 and WA* using the heuristic based on A2. For the heuristic searches we choose a large weighting factor $w = 100$, i.e., the search order is mainly determined by the abstraction-based heuristic function h.

PBX

PBX Abstraction Results	
A1: #preds	15
A1: # states	123519
A1: time (sec)	41
A2: #preds	49
A2: # states	727881
A2: time (sec)	312

PBX Search Results	
BFS (states)	323700
BFS (time)	615
WA*,A1 (states)	46540
WA*,A1 (time)	25
WA*,A2 (states)	46540
WA*,A2 (time)	25

Figure 5.11: Results for PBX

In the PBX program, using the abstraction-based heuristic search significantly decreases the number of generated states from 323700 with BFS to 46540 with WA*. According to that, also the exploration time decreases from 615 to 25 seconds. However, computing and using the more precise abstraction A2 instead of A1 does not pay off for this program, since using A2 in the abstraction-based heuristic does not decrease the number of generated states any more.

SMS

SMS Abstraction Results	
A1: #preds	21
A1: # states	243500
A1: time (sec)	112
A2: #preds	34
A2: # states	566173
A2: time (sec)	289

SMS Search Results	
BFS (states)	489100
BFS (time)	760
WA*,A1 (states)	512000
WA*,A1 (time)	802
WA*,A2 (states)	328000
WA*,A2 (time)	570

Figure 5.12: Results for SMS

In the SMS program (#LOC: 12295, #classes: 131), using A1 for the abstraction-based heuristic search has no positive effect on the number of generated states. With WA* using A1, 512000 states are generated, while BFS only generates 489000 states. This increase in the number of states results due to spurious abstract goal states that leads the heuristic search into the wrong direction. Using the refined abstraction A2 then reduces the number of generated states to 328000 states.

Dishwasher

Dishwasher Abstraction Results	
A1: #preds	12
A1: # states	101980
A1: time (sec)	40
A2: #preds	27
A2: # states	421800
A2: time (sec)	176

Dishwasher Search Results	
BFS (states)	612000
BFS (time)	912
WA*,A1 (states)	29170
WA*,A1 (time)	19
WA*,A2 (states)	29170
WA*,A2 (time)	19

Figure 5.13: Results for Dishwasher

In the Dishwasher program, when using A1 for the abstraction-based heuristic search, the number of generated states decreases from 612000 with BFS to 29170 with WA*. According to that, also the search time decreases from 912 seconds with BFS to 19 seconds with WA*. However, using the more detailed abstraction A2 leads to no improvements w.r.t. generated states or decreased search time.

CANBus

In the CANBus program, the number of generated states using A1 with WA* decreases only moderately from 567100 with BFS to 472080. Using the more precise abstraction A2, the number of generated states again drops moderately down to 425880.

CANBus Abstraction Results	
A1: #preds	16
A1: # states	143212
A1: time (sec)	58
A2: #preds	31
A2: # states	389400
A2: time (sec)	126

CANBus Search Results	
BFS (states)	567100
BFS (time)	780
WA*,A1 (states)	472080
WA*,A1 (time)	643
WA*,A2 (states)	425880
WA*,A2 (time)	591

Figure 5.14: Results for CANBus

ARCS

ARCS Abstraction Results	
A1: #preds	19
A1: # states	209350
A1: time (sec)	104
A2: #preds	35
A2: # states	566920
A2: time (sec)	376

ARCS Search Results	
BFS (states)	471290
BFS (time)	793
WA*,A1 (states)	61800
WA*,A1 (time)	64
WA*,A2 (states)	55610
WA*,A2 (time)	58

Figure 5.15: Results for ARCS

In the ARCS program, using the abstraction-based heuristic search significantly decreases the number of generated states from 471290 with BFS to 61800 with WA*. According to that, also the exploration time decreases from 793 to 64 seconds. Using the more precise abstraction A2 instead of A1 yields another decrease in the number of generated states down to 55610.

Elevator

In the Elevator program using A1 for the abstraction-based heuristic search yields only small savings in the number of generated states. With WA* using A1, 392130 states are

Elevator Abstraction Results	
A1: #preds	12
A1: # states	87230
A1: time (sec)	39
A2: #preds	23
A2: # states	201900
A2: time (sec)	114

Elevator Search Results	
BFS (states)	414610
BFS (time)	721
WA*,A1 (states)	392130
WA*,A1 (time)	688
WA*,A2 (states)	392130
WA*,A2 (time)	688

Figure 5.16: Results for Elevator

generated, while BFS generates 414610 states. Using the refined abstraction A2 has no effect on the number of generated states.

Pacemaker

Pacemaker Abstraction Results	
A1: #preds	17
A1: # states	176210
A1: time (sec)	86
A2: #preds	29
A2: # states	436970
A2: time (sec)	305

Pacemaker Search Results	
BFS (states)	523923
BFS (time)	823
WA*,A1 (states)	425095
WA*,A1 (time)	641
WA*,A2 (states)	391105
WA*,A2 (time)	598

Figure 5.17: Results for Pacemaker

In the Pacemaker program, using A1 for the abstraction-based heuristic search yields only small savings in the number of generated states. With WA* using A1, 425095 states are generated, while BFS generates 523923 states. Using the refined abstraction A2 reduces the number of generated states down to 391105.

Home Heating

In the Home Heating program, when using A1 for the abstraction-based heuristic search, the number of generated states decreases from 470042 with BFS to 35266 with WA*. According to that, also the search time decreases from 772 seconds with BFS to 31 seconds with WA*. However, using the more detailed abstraction A2 leads to no improvements w.r.t. generated states or decreased search time.

Home Heating Abstraction Results	
A1: #preds	14
A1: # states	136226
A1: time (sec)	57
A2: #preds	26
A2: # states	428803
A2: time (sec)	194

HomeHeating Search Results	
BFS (states)	470042
BFS (time)	772
WA*,A1 (states)	35266
WA*,A1 (time)	31
WA*,A2 (states)	35266
WA*,A2 (time)	31

Figure 5.18: Results for Home Heating

Home Alarm

Home Alarm Abstraction Results	
A1: #preds	12
A1: # states	84523
A1: time (sec)	31
A2: #preds	17
A2: # states	177905
A2: time (sec)	72

Home Alarm Search Results	
BFS (states)	536184
BFS (time)	851
WA*,A1 (states)	27412
WA*,A1 (time)	16
WA*,A2 (states)	26826
WA*,A2 (time)	15

Figure 5.19: Results for Home Alarm

In the Home Alarm program, when using A1 for the abstraction-based heuristic search, the number of generated states decreases from 536184 with BFS to 27412 with WA*. According to that, also the search time decreases from 851 seconds with BFS to 16 seconds with WA*. Using the more detailed abstraction A2 leads again reduces the number of generated states and the search time to 26826 states resp. 15 seconds.

TCU

In the TCU program, using A1 for the abstraction-based heuristic search has a negative effect on the number of generated states. With WA* using A1, 678235 states are generated, while BFS only generates 577380 states. This increase in the number of states results due to spurious abstract goal states that leads the heuristic search into the wrong direction. Although using the refined abstraction A2 decreases the number of generated states to 592206, WA* still behaves worse than BFS.

TCU Abstraction Results	
A1: #preds	21
A1: # states	271518
A1: time (sec)	194
A2: #preds	37
A2: # states	642329
A2: time (sec)	622

TCU Search Results	
BFS (states)	577380
BFS (time)	834
WA*,A1 (states)	678235
WA*,A1 (time)	962
WA*,A2 (states)	592206
WA*,A2 (time)	891

Figure 5.20: Results for TCU

5.3.1 Summary of results

Summing up the experimental results, one can say that for the programs PBX, Dishwasher, ARCS, Heating and Alarm the usage of the abstraction-based heuristic search procedure significantly reduces the number of generated states resp. the search time, as one can see in fig. 5.21 and 5.22. For the programs CANBus, Elevator and Pacemaker, the usage of the abstraction-based heuristic search procedure reduces the number of generated states only marginally. Additionally, when looking at the sum of the generation time for the abstraction A2 and the search time for the abstraction-based heuristic search, one can observe that for the programs Elevator and Pacemaker the overall time is higher than if using pure BFS, i.e., although the usage of the abstraction-based heuristic search procedure decreases the number of generated states a little, due to the overhead induced by the generation of the abstraction the overall search time increases. Even worse are the results for the programs SMS and TCU. In both programs, using the abstraction-based heuristic search procedure with abstraction A1 increases the number of generated states compared to BFS. Accordingly, the overall search time is larger than the search time for BFS. Although using A2 instead of A1 decreases the number of generated states a little, the overall search time is still larger than the search time with BFS.

The achieved results shows that an abstraction-based heuristic can sometimes be very effective when searching for particular paths in large state spaces. For several sample programs, using the abstraction-based heuristic search procedure significantly reduces both the number of states to be generated and the needed search time. However, the experiments also affirm the dependency between the quality of the heuristic function and the effectiveness of heuristic search. For some programs, in particular SMS and TCU, the abstraction-based heuristic is not informative enough to reduce the number of generated states or the search time when used in a heuristic search procedure.

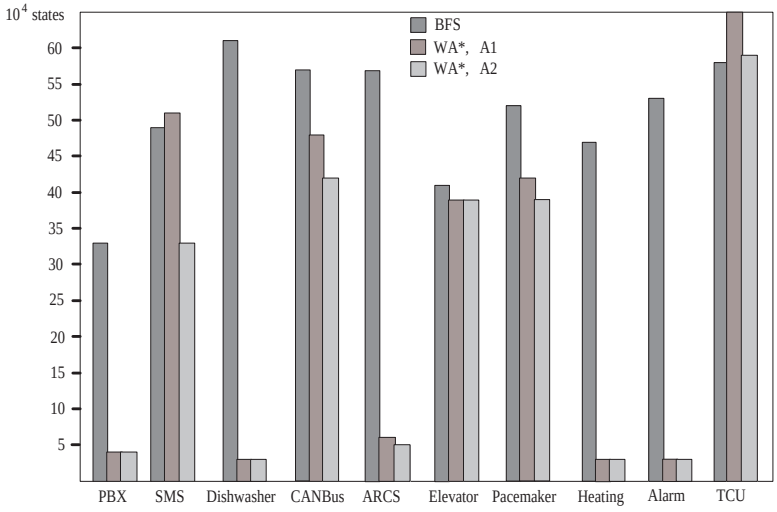


Figure 5.21: **Summary of exploration results.** The bar chart shows the number of generated states for our test programs when using BFS, WA* with abstraction A1 and WA* with abstraction A2.

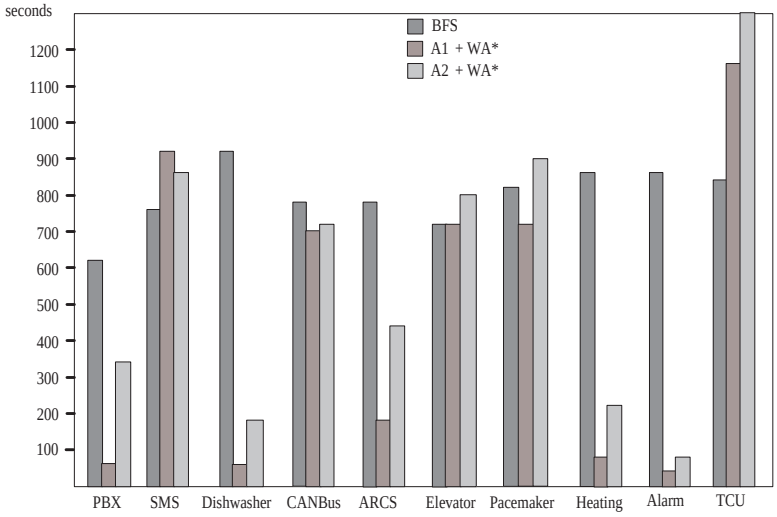


Figure 5.22: **Summary of abstraction and exploration times.** The bar chart shows the search time for our test programs when using BFS resp. the sum of the generation time of the abstractions A1 resp. A2 and the search time using WA* with A1 and A2.

5.4 Related Work

We already described related work in chapter 2, thus we concentrate here on work which is similar to our approach. A preliminary version of the described abstraction-based heuristic state space exploration has been presented in [Let03]. In [ELL01b, ELL01a] the well known explicit state model checker SPIN has been extended to allow the use of property specific heuristics for heuristic search. For an atomic boolean expression like `x=1` the heuristic yields 0 if `x=1` holds and 1 otherwise. For compound boolean expressions the heuristic computes the minimum for disjunctions resp. the maximum for conjunctions. Contrary to our abstraction-based heuristic, the property based heuristic does not take into account information about the underlying program. As a result, a heuristic search towards a state that fulfills `x=1` proceeds like an uninformed BFS. In [GV02b, GV02a] the model checker JavaPathFinder2 is augmented with heuristic search algorithms employing structural heuristics. In this work, structural coverage criteria like branch coverage [Bei90] are used to guide the exploration process. Contrary to our approach, these heuristics do not take into account information about the property searching for. Recently, in [EL04] so-called *abstraction databases* are used for guiding the search towards specified goal states. As in our abstraction-based heuristic state space exploration procedure, an abstraction of the program is computed, and the distances to an abstract goal state in the abstract state space are taken as estimates of the distances to a concrete goal state in the concrete state space. However, in [EL04] programs for the model checker SPIN are considered, and the abstraction are computed by means of the tool α-SPIN [GMMP02] that can be used to perform data abstraction on the considered finite-state programs. In contrast to this, our approach deals with abstraction of programs that in general can have an infinite number of states due to dynamic object creation. Furthermore, the predicate abstraction applied in our approach is more general than the data abstraction in α-SPIN. Another approach that employs abstraction for heuristic search is presented in [QN04]. Similar as in [EL04], data abstraction is used to compute an abstraction with the model checker NuSMV2 [CCG+02]. In contrast to our work, the reachable abstract states are stored symbolically as BDDs, i.e., a BDD b_n is used to represent all abstract states reachable in n steps. During the symbolic forward search of the concrete system, in each step the BDD representing the current frontier states is partitioned w.r.t. the BDDs representing the abstraction. The generated partitions are then explored individually, starting with the partition that was generated by the BDD with the minimal distance to an abstract goal state. The results described in [QN04] show that this guided symbolic search can outperform standard symbolic search on some sample problems. However, a serious drawback of this approach is that in contrast to ordinary symbolic forward search all states reachable in n steps are not represented by one BDD any more but by many different BDDs. In extreme cases, the number of BDDs used to represent all states reachable in n steps grows exponentially with n. Additionally, since BDDs are used both for computing the abstraction and the concrete system, the approach is limited to finite state systems.

6 Conclusion and Future Work

In this final chapter, we briefly summarize the contributions of this thesis, and we identify some directions in which future work can be done.

6.1 Summary of Contributions

The main contribution of this thesis is the development of efficient state space exploration algorithms for embedded C++ programs. Since embedded C++ programs often use specific low-level routines of the underlying operating system to realize e.g. concurrency and synchronization, we introduce the language SymC++ that allows us to handle embedded C++ programs in a uniform manner. As an underlying formal model of SymC++ programs we introduce the language C_{min} and the simple logic TL that allows the specification of reachability properties of C_{min} programs, and we describe how SymC++ programs can be translated into C_{min} programs. Additionally, we show how several embedded C++ programs generated by an automatic code generator of the UML case tool Rhapsody can be handled within the framework of SymC++. Throughout this thesis these programs serve as test programs to evaluate the developed state space exploration algorithms.

Starting with a very simple state space exploration algorithm for C_{min} programs, we successively improve the efficiency of this algorithm by introducing several optimizations. The optimizations concerning symmetric states, the approximated duplicate detection and the state storage reduction can successfully increase the efficiency of the basic state space exploration algorithm, as it is made evident by the performed experiments. The presented composite explicit-symbolic state representation combined with the approximated algorithm for duplicate detection leads to even more drastic performance improvements, allowing the efficient state space exploration of large C++ programs with very large state spaces, which is also validated experimentally.

To gain further performance improvements, in chapter 5 we present a heuristic state space exploration algorithm for C_{min} programs. The employed heuristic function is based on an abstraction of the considered program that is generated prior to the state space exploration. The abstraction-based heuristic function takes distances in the abstract state space as heuristic values for states in the concrete state space. In contrast to the optimizations presented in chapter 4, applying the abstraction-based heuristic state space exploration not always yields performance improvements. However, for several programs significant performance gains have been achieved.

6.2 Future Work

Based on the contributions of this thesis, there are several starting points for future work. Among others, the following directions could be interesting:

- **Richer specification languages**: The logic TL is only suitable for specifying simple reachability properties. As described in sect. 2.4, there are several extensions of common programming languages that allow specifications in a design by contract style, i.e., specification of pre- and postconditions of methods or the specification of class invariants as it is possible in e.g. JML [LBR98, LBR99]. Such specifications provide an easy way of formally specifying the intended behavior of programs, and it would be desirable to have these possibilities in SymC++ programs, too. Having such a specification language, it would be interesting to analyze to what extend the presented state space exploration algorithms can be adapted to a richer specification language. As a first approach, one could try to translate a program P with such a specification S into a new program P' and a formula EFq of TL s.t. $P \models S$ iff $P' \models EFq$. If this is possible, then one can reuse the presented algorithms without any change.

- **Symbolic representations and constraint solving**: The explicit-symbolic state representation presented in sect. 4.2 uses arithmetic constraints to symbolically encode sets of states. As described in sect. 4.2, to find satisfying assignments to the variables of the constraints we follow a three step approach. Firstly, we apply a linear constraint solver. If this solver cannot solve a constraint, a random solver is tried. If this solver also fails to solve the constraint, an exhaustive test of all possible valuations of the variables of the constraint definitely solve the constraint. For all tested programs, the linear constraint solver could already successfully solve almost all occurring constraints. However, this may not be the case for programs that use symbolically represented variables in complicated nonlinear computations, e.g. a program computing an approximation of the sinus function. For efficiently solving such constraints different solvers are needed, and it would be desirable to extend the existing tool set to be able to utilize a dedicated solver for certain kinds of constraints. Besides the application of other constraint solvers, one could also explore the possibility to encode larger parts of a state symbolically than only the variables which are directly or indirectly involved in computations containing the nondeterministic choice `symcpp_nondet`. However, crucial for the efficiency of such an approach is the choice of the right variables to represent symbolically, and care must be taken that the symbolically represented part will not become too large.

- **Fully automatic heuristic state space exploration**: For the abstraction-based heuristic state space exploration algorithm presented in chapter 5, two steps have to be performed manually. Firstly, the user has to provide a memory configuration that specifies for each type the number of objects which can dynamically be allo-

cated, and secondly he has to define the predicates which are used for predicate abstraction. Although the other steps are already automated, it would be interesting to explore the possibilities to automate the abstraction-based heuristic state space exploration completely. As a starting point, one could try to adopt completely automated procedures used in existing model checking approaches based on abstraction-refinement.

- **Enhancement of supported language constructs**: As described in sect. 3.1.2, there are currently some restrictions concerning the C++ language constructs that are supported in SymC++, e.g. unsigned types, bit operations, low-level memory handling or exception handling. It would be desirable to extend SymC++ to support these constructs, too.

Bibliography

[AB01] C. Artho and A. Biere. Applying static analysis to large-scale, multithreaded Java programs. In *ASWEC'01: 13th Australian Software Engineering Conference*, pages 68–75, 2001.

[AFdR80] K.R. Apt, N. Francez, and W.P. de Roever. A proof system for communicating sequential processes. *ACM Transactions on Programming languages and Systems*, 2(3):359–385, 1980.

[AO94] K.R. Apt and E.R. Olderog. *Programmverifikation. Sequentielle, parallele und verteilte Programme.* Springer-Lehrbuch, 1994.

[AO97] K.R. Apt and E.R. Olderog. *Verification of Sequential and Concurrent Programs.* Springer-Texts in Computer Science, 2nd. ed., 1997.

[BC85] G. Berry and L. Cosserat. The Esterel synchronous programming language and its mathematical semantics. In *Seminar on Concurrency*, volume 197 of *Lecture Notes in Computer Science*, pages 389–448. Springer Verlag, 1985.

[BCM90] C. Berthet, O. Coudert, and J.C. Madre. New ideas on symbolic manipulations of finite state machines. In *Conference on Computer Aided Design (ICAAD)*, 1990.

[BCM+92] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. *Information and Computation*, 98(2):142–170, 1992.

[BCMD90] J.R. Burch, E.M. Clarke, K.L. McMillan, and D.L. Dill. Sequential circuit verification using symbolic model checking. In *Design Automation Conference (DAC)*, pages 46–51, Los Alamitos, CA, June 1990. ACM Press.

[BCRZ99] A. Biere, E. Clarke, R. Raimi, and Y. Zhu. Verifying safety properties of a power pc microprocessor using symbolic model checking without BDDs. In *Proc. 11th Conference on Computer Aided Verification (CAV'99)*, Lecture Notes in Computer Science. Springer Verlag, 1999.

[BCZ99] A. Biere, A. Cimatti, and Y. Zhu. Symbolic model checking without BDDs. In *TACAS'99*, volume 1579 of *Lecture Notes in Computer Science*. Springer Verlag, 1999.

[BdS91] F. Boussinot and R. de Simone. The ESTEREL Language. In *Proceedings of IEEE,79(9):1293-1304, September 1991*, 1991.

[Bei90] B. Beizer. *Software Testing Techniques*. Van Nonstrand Reinhold, 1990.

[BG90] A. Benveniste and P. Le Guernic. Hybrid dynamical systems theory and the SIGNAL language. *IEEE Transactions on Automatic Control*, 35(5):535–546, May 1990.

[BGJ91] A. Benveniste, P. Le Guernic, and C. Jacquemot. Synchronous programming with events and relations: the SIGNAL language and its semantics. *Science of Computer Programming*, 16(2):103–149, September 1991.

[BHPV00] G. Brat, K. Havelund, S. Park, and W. Visser. Java PathFinder - Second Generation of a Java Model Checker. In *Proceedings of Post-CAV Workshop on Advances in Verification*, July 2000.

[BLL+95] J. Bengtsson, K.G. Larsen, F. Larsson, P. Petterson, and Wang Yi. Uppaal - a Tool Suite for Automatic Verification of Real-Time Systems. In *Hybrid Systems III - Verification and Control*, volume 1066 of *Lecture Notes in Computer Science*, pages 232–243, 1995.

[BLP03] G. Behrmann, K. Larsen, and R. Pelanek. To store or not to store, 2003.

[BMMR01] T. Ball, R. Majumdar, T. Millstein, and S.K. Rajamani. Automatic Predicate Abstraction of C Programs. In *Proceedings of PLDI'01*, 2001.

[Bor97] A. Boralv. The industrial success of verification tools based on Stalmarck's Method. In *International Conference on Computer-Aided Verification (CAV'97)*, number 1254 in LNCS. Springer Verlag, 1997.

[BR01] T. Ball and S.K. Rajamani. Automatically Validating Temporal Safety Properties of Interfaces. In *Proceedings of the 8th SPIN Workshop*, volume 2057 of *Lecture Notes in Computer Science*, pages 103–122. Springer-Verlag, 2001.

[BR02] T. Ball and S.K. Rajamani. The SLAM Project: Debugging system software via static analysis. In *POPL 02: Principles of Programming Languages*, pages 1–3, January 2002.

[Bry86] R.E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8), 1986.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[CC92] P. Cousot and R. Cousot. Abstract Interpretation Framework. *Journal of Logic and Computation*, 4(2):511–547, August 1992.

[CCG+02] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV 2: An opensource tool for symbolic model checking. In *Com-puter Aided Verification (CAV 2002)*, volume 2404 of *LNCS*, page 359364. Springer, 2002.

[CCG+03] S. Chaki, E.M. Clarke, A. Groce, S. Jha, and H. Veith. Modular Verification of Software Components in C. In *25th International Conference on Software Engineering (ICSE)*, pages 385–395, 2003.

[CCGS03] S. Chaki, E.M. Clarke, A. Groce, and O. Strichman. Predicate Abstraction with Minimum Predicates. In *12th Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME)*, volume 2860 of *LNCS*, pages 19–34, 2003.

[CDGP01] Alberto Coen-Porisini, Giovanni Denaro, Carlo Ghezzi, and Mauro Pezzè. Using symbolic execution for verifying Safety-Critical systems. pages 142–151, 2001.

[CE81] E.M. Clarke and E.A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Logic of Programs: Workshop, Yorktown Heights, NY, LNCS 131*, May 1981.

[Ce00] J. Corbett and et.al. Bandera: Extracting Finite State Models from Java Source Code. In *Proceedings of the 22nd Int. Conf. On Software Engineering*, pages 439–448. IEEE, 2000.

[CES86] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Progr. Lang. and Systems*, 8:244–263, 1986.

[CFJ93] E.M. Clarke, T. Filkorne, and S. Jha. Exploiting Symmetry In Temporal Logic Model Checking. In *Proc. 5th Conference on Computer Aided Verification*, pages 450–462, 1993.

[CGP99] E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking*. MIT Press, 1999.

[CL02] Y. Cheon and G. Leavens. A runtime assertion checker for the java modeling language, 2002.

[CPHP87] P. Caspi, D. Pilaud, N. Halbwachs, and J. Plaice. LUSTRE: A declarative language for programming synchronous systems. In *14th ACM Symp. on Principles of Programming Languages*, January 1987.

[CPHR91] P. Caspi, D. Pilaud, N. Halbwachs, and P. Raymond. The Synchronous Data Flow Programming Language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[DDP99] S. Das, D.L. Dill, and S. Park. Experience with predicate abstraction. In *CAV'00: Computer Aided Verification*, number 1633 in Lecture Notes in Computer Science, pages 160–171. Springer Verlag, 1999.

[Dij76] E. Dijkstra. *A discipline of Programming*. Prentice-Hall, 1976.

[DL02] Henning Dierks and Marc Lettrari. Constructing test automata from graphical real-time requirements. In *FTRTFT '02: Proceedings of the 7th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 433–454, London, UK, 2002. Springer-Verlag.

[DP01] G. Delzanno and A. Podelski. Constraint-based deductive model checking. *International Journal on Software Tools for Technology Transfer*, 3(3):250–270, 2001.

[EGHT94] David Evans, John Guttag, James Horning, and Yang Meng Tan. LCLint: A tool for using specifications to check code. In *Proceedings of the ACM SIGSOFT '94 Symposium on the Foundations of Software Engineering*, pages 87–96, 1994.

[EL04] S. Edelkamp and A.L. Lafuente. Abstraction Databases in Theory and Model Checking Practice. In *ICAPS Workshop on Connecting Planning Theory with Practice*, June 2004.

[ELL01a] S. Edelkamp, A.L. Lafuente, and S. Leue. Directed explicit model checking with HSF-Spin. In *SPIN*, volume 2057 of *LNCS*, pages 57–79, 2001.

[ELL01b] S. Edelkamp, A.L. Lafuente, and S. Leue. Protocol verification with heuristic search. In *AAAI Symposium on Model-based Validation of Intelligence*, 2001.

[EM03] S. Edelkamp and T. Mehler. Byte Code Distance Heuristics and Trail Direction for Model Checking Java Programs. In *Workshop on Model Checking and Artificial Intelligence (MoChart)*, August 2003.

[Eva96] David Evans. Static detection of dynamic memory errors. In *SIGPLAN Conference on Programming Language Design and Implementation (PLDI '96)*, 1996.

[FJ94] S. Finne and S. Jones. Programming Reactive Systems in Haskell. In *Glasgow Functional Programming Workshop, Ayr, Scotland*. Springer-Verlag, 1994.

[FLL+02] C. Flanagan, K. Leino, M. Lillibridge, C. Nelson, J. Saxe, and R. Stata. Extended static checking for java, 2002.

[Flo67] R. Floyd. Assigning meaning to programs. *Symposia in Applied Mathematics: Mathematical Aspects of Computer Science*, 19:19–31, 1967.

[GG88] S. Garland and J. Guttag. Inductive methods for reasoning about abstract data types. In *Symposium on Principles of Programming Languages (POPL)*, pages 219–228, San Diego, California, 1988.

[GHP02] E. Gery, D. Harel, and E. Palachi. Rhapsody: A complete life-cycle model-based development system. In *Proceedings of the Third International Conference on Integrated Formal Methods*, pages 1–10, 2002.

[GJSB00] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Addison Wesley, 2000.

[GK02] P. Godefroid and S. Khurshid. Exploring very large state spaces using Genetic Algorithms. In *Proc. TACAS'02*, pages 1–10, 2002.

[GKvV95] J.F. Groote, J.W.C. Koorn, and S.F.M. van Vlijmen. The Safety Guaranteeing System at station Hoorn-Kersenboogerd. In *Proceedings of the 10th IEEE Conference on Computer Assurance COMPASS 95*, pages 131–150. IEEE, 1995.

[GM93] M. Gordon and T. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge university Press, 1993.

[GMMP02] M. Gallardo, J. Martinez, P. Merino, and E. Pimentel. alpha spin: Extending spin with abstraction. In *Proceedings of the 9th International SPIN Workshop on Model Checking of Software*, pages 254–258, London, UK, 2002. Springer-Verlag.

[God97] P. Godefroid. Model Checking for Programming Languages Using VerifSoft. In *Proceedings POPL'97*, pages 174,186. ACM Press, 1997.

[GR95] S. Gregory and R. Ramirez. Tempo: a declarative concurrent programming language. In *Proceedings of the 12th International Conference on Logic Programming*. MIT Press, 1995.

[Gre97] S. Gregory. A declarative approach to concurrent programming. In *Proceedings of the 9th International Symposium on Pramming Languages, Implementations, Logics, and Programs*. Springer-Verlag, 1997.

[Gri81] D. Gries. *The Science of programming*. Springer-Verlag, 1981.

[Gro96] The VIS Group. VIS : A System for Verification and Synthesis. In *8th international Conference on Computer Aided Verification*, number 1102 in LNCS, 1996. VIS 1.3 is available from the VIS home-page: `http://www-cad.eecs.Berkeley.EDU/~vis`.

[Gro04] Object Management Group. UML 2.0 Superstructure Specification, Oct. 2004.

[GS97] S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In *Proc. of the 9th International Conference on Computer Aided Verification*, number 1254 in Lecture Notes in Computer Science, pages 72–83. Springer, 1997.

[GV02a] A. Groce and W. Visser. Heuristic Model Checking for Java Programs. In *Proceedings of SPIN 2002*, Grenoble, France, 2002.

[GV02b] A. Groce and W. Visser. Model Checking Java Programs using Strucural Heuristics. In *Proceedings of ISSTA 2002*, Rome, Italy, 2002.

[Har87] D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8:231–274, 1987.

[HG97] David Harel and Eran Gery. Executable Object Modeling. *IEEE Computer*, 1997.

[HJMS02] A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy Abstraction. In *ACM SIGPLAN-SIGACT Conference on Principles of Programming Languages*, pages 58–70, 2002.

[HJMS03] A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software Verification with BLAST. In *Model Checking Software (SPIN)*, pages 235–239, 2003.

[HN96] D. Harel and A. Naamad. The STATEMATE semantics of statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4), 1996.

[HNR68] P.E. Hart, N.J. Nilsson, and B. Raphael. A formal basis for heuristic determination of minimum path cost. *IEEE Transactions on Systems Science and Cybernetics*, 4:100–107, 1968.

[Hoa69] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–580, 1969.

[Hol87] G.J. Holzmann. On limits and possibilities of automated protocol analysis. In *Proc. 7th IFIP WG 6.1 Int. Workshop on Protocol Specification, Testing, and Verification*, pages 137–161. North-Holland Publ., Amsterdam, 1987.

[Hol91] G.J. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, Englewood Cliffs, New Jersey, 1991.

[Hol00] G.J. Holzmann. Logic Verification of ANSI-C code with SPIN. In *Model Checking Software (SPIN)*, pages 112–127, 2000.

[Hol03] G.J. Holzmann. *The SPIN Model Checker: primer and reference manual*. Addison-Wesley, Boston MA, 2003.

[HP96] David Harel and Michal Politi. *Modeling Reactive Systems with Statecharts: The STATEMATE Approach*. Part No. D–1100–43. i-Logix Inc., Three Riverside Drive, Andover, MA 01810, June 1996.

[HS99] G.J. Holzmann and M.H. Smith. Software Model Checking: Extracting Verification Models from Source Code. *Journal on Formal Methods for Protocol Engineering and Distributed Systems*, pages 481–497, 1999.

[HU79] J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley, 1979.

[Ios01] R. Iosif. Exploiting Heap Symmetries in Explicit-State Model Checking of Software. In *Proc. 16th IEEE Conference on Automated Software Engineering*, 2001.

[IS99] R. Iosif and R. Sisto. dSPIN: A Dynamic Extension of SPIN. In *Proc. 6th SPIN Workshop*, volume 1680 of *Lecture Notes in Computer Science*, pages 261–276, 1999.

[ISO03] ISO. C++ Standard ISO/IEC 14882:2003, 2003.

[KM94] M. Kaufmann and J. Moore. Design goals for ACL2. Technical Report 101, Computational Logic, Inc., 1994.

[KPV03] S. Khurshid, C. Pasareanu, and W. Visser. Generalized symbolic execution for model checking and testing, 2003.

[LBK01] M. Lettrari, U. Brockmeyer, and J. Klose. UML Validation Suite. In J. Tretmans and E. Brinksma, editors, *Proceedings of FATES'01 - Formal Approaches to Testing of Software*, 2001.

[LBR98] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: a java modeling language. In *Formal Underpinnings of Java Workshop (at OOPSLA '98)*, 1998.

[LBR99] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: A notation for detailed design. In Haim Kilov, Bernhard Rumpe, and Ian Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer Academic Publishers, 1999.

[LCL88] F.J. Lin, P.M. Chu, and M. Liu. Protocol verification using reachability analysis: the state explosion problem and relief strategies. *ACM*, pages 126–135, 1988.

[Let03] M. Lettrari. Using abstractions for heuristic state space exploration of reactive object-oriented systems. In *FME 2003: Formal Methods, International Symposium of Formal Methods Europe, Pisa, Italy, September 8-14, 2003, Proceedings*, volume 2805 of *Lecture Notes in Computer Science*, pages 462–481. Springer, 2003.

[LG81] G. Levin and D. Gries. A proof technique for communicating sequential processes. *Acta Informatica*, 15:281–302–385, 1981.

[LK01] Marc Lettrari and Jochen Klose. Scenario-based monitoring and testing of real-time uml models. In *UML'01: Proceedings of the 4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools*, pages 317–328, London, UK, 2001. Springer-Verlag.

[LME04] P. Leven, T. Mehler, and S. Edelkamp. Directed error detection in C++ with the assembly-level model checker StEAM. In *Model Checking of Software (SPIN)*, 2004.

[LV01] F. Lerda and W. Visser. Addressing Dynamic Issues of Program Model Checking. In *Proc. 8th SPIN Workshop*, volume 2057 of *Lecture Notes in Computer Science*, pages 80–102, 2001.

[McM93] K.L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[ME04] M. Musuvathi and D.R. Engler. Model checking large network protocol implementations. In *Proceedings of theFirst Symposium on Networked Systems Design and Im-plementation*, 2004.

[Mey92] B. Meyer. *Eiffel: The Language*. Prentice Hall, 1992.

[Mey97] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1997.

[MMZM01] M. Moskewics, C. Madigan, Y. Zhao, and L. Zhang S. Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 39. Design Automation Conference (DAC)*, Las Vegas, 2001.

[MP91] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems - Specification*. Springer, 1991.

[MPC+02] M. Musuvathi, D.Y. Park, A. Chou, D.R. Engler, and D.L. Dill. CMC: A pragmatic approach to model check-ing real code. In *Proceedings of the Fifth Symposium onOperating Systems Design and Implementation*, 2002.

[MS95] S. Miller and M. Srivas. Formal verification of the AAMP5 Microprocessor: A Case Study in the Industrial Use of Formal Methods. In *IEEE Computer Society, WIFT'95: Workshop on Industrial-Strength Formal Specification Techniques*, 1995.

[NNH99] F. Nielson, H.R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.

[OG76a] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6:319–340, 1976.

[OG76b] S. Owicki and D. Gries. Verifying properties of parallel programs: an axiomatic approach. *Communications of the ACM*, 19:279–285, 1976.

[ORSS95] S. Owre, J. Rushby, N. Shankar, and M. Srivas. A tutorial on using PVS for hardware verification. In *Conference on Theorem Provers in Circuit Design (TPCD)*, volume 901 of *LNCS*, pages 258–279, Bad Herrenalb, Germany, 1995. Springer-Verlag.

[Pau94] L. Paulson. Isabelle: A Generic Theorem Prover. *LNCS*, 828, 1994.

[Pea85] J. Pearl. *Heuristics*. Addison-Wesley, 1985.

[Pnu77] A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th IEEE Symposium on Foundations of Computer Science*, , pages 46–57. , 1977.

[Poh70] I. Pohl. Heuristic search viewed as path finding in a graph. *Artificial Intelligence*, 1:193–204, 1970.

[QN04] K. Qian and A. Nymeyer. Guided invariant model checking based on abstraction and symbolic pattern databases. In *Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004*, volume 2988 of *Lecture Notes in Computer Science*, pages 497–511. Springer, 2004.

[QS82] J.P. Quielle and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proceedings of the 5th International Symposium on Programming*, volume 137 of *Lecture Notes in Computer Science*, pages 337–371. Springer-Verlag, 1982.

[QS83] J.P. Quielle and J. Sifakis. Fairness and Related Properties in Transition Systems - A Temporal Logic to Deal with Fairness. *Acta Informatica*, 19:195–220, 1983.

[SD95a] U. Stern and D.L. Dill. Automatic verification of the SCI cache coherence protocol. In *IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, pages 21–34, 1995.

[SD95b] U. Stern and D.L. Dill. Improved probabilistic verification by hash compaction. In *IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, pages 206–224, 1995.

[Sht89] O. Shtrichman. Tuning SAT checkers for bounded model checking. In *Proc. 12th Intl. Conference on Computer Aided Verification (CAV'00)*, Lecture Notes in Computer Science. Springer-Verlag, 1989.

[Sil95] J.P.M. Silva. *Search Algorithms for Satisfiability Problems in Combinational Switching Circuits*. Ph.D. Dissertation, University of Michigan, 1995.

[Str00] B. Stroustrup. *The C++ Programming Language*. Addison Wesley, Reading MA, 2000.

[TPR01] T.Ball, A. Podelski, and S.K. Rajamani. Boolean and Cartesian Abstraction for Model Checking C Programs. In *Proceedings TACAS'01*, volume 2031 of *Lecture Notes in Computer Science*, pages 268–283. Springer-Verlag, 2001.

[WL93] P. Wolper and D. Leroy. Reliable hashing without collision detection. In *Proc. of the 5th International Conference on Computer Aided Verification (CAV-93)*, volume 7, pages 59–70, Berlin, 1993. Springer-Verlag.

[WR95] M. Wallace and C. Runciman. Extending a Functional Programming System for Embedded Applications. *Software - Practice and Experience*, 25(1):73–96, 1995.

[YD98] C.H. Yang and D.L. Dill. Validation with guided search of the state space. In *DAC*, pages 599–604, 1998.

[Yof97] S. Yofine. Kronos: a verification tool for real-time systems. *Software Tools for Technology Transfer*, 1(1+2):123–133, 1997.

[Zha97] H. Zhang. An Efficient Propositional Prover. In *International Conference on Automated Deduction (CADE'97)*, volume 1249 of *LNAI*, pages 272–275. Springer Verlag, 1997.

A SymC++ Restrictions

The following C++ constructs are either not supported in SymC++ or are treated with a slightly different semantics than defined in the ISO-standard [ISO03].

- Function pointers are not supported.
- Class temporaries created under a conditional operator are not destroyed after completely executing the conditional operator as it is defined in the standard [ISO03].
- Integral types `unsigned char`, `unsigned short`, `unsigned int`, `unsigned long`, `unsigned long long`, `char`, `short`, `long` and `long long` are treated as `int`.
- Floating point types `float`, `double` and `long double` are not supported.
- Casts of integral integer types are ignored.
- Virtual base classes are treated as nonvirtual base classes.
- Exceptions are not supported.
- Union-types are treated as ordinary `struct`-types.
- Reference-variables are treated as pointer variables.
- Unnamed class-/struct-/union-/enum-types are not supported.
- Pointer to members are not supported.
- Type declarations within functions are not supported.
- Inline assembler code provided by the `asm` statement is not supported.
- The `dynamic_cast` operator is not supported.
- Bit-fields, i.e., explicit declaration of the number of bits used to represent member variables in structures are ignored.
- Ellipsis, i.e., functions with varying number of arguments as e.g. the function `printf` are not supported.
- C++ runtime type information function `typeid()` is not supported.

- Temporaries that have been bound to a static or global reference are destructed after completely executing the corresponding expression that created the temporaries and not at the end of the block corresponding to the expression as it is defined in the standard [ISO03].
- Member-operator functions `|`, `|=` and `||` are not supported.
- Type-casts between pointer types that have no inheritance relation are not supported.

B PBX Sample Program

The PBX program describes the behavior of a telephone system consisting of 4 telephones. As all our test programs, the PBX program has been generated by the automatic code generator of the commercial UML design tool Rhapsody. Within Rhapsody, parts of the structure and behavior of programs can be described by means of UML diagrams like *object model diagrams* or *statecharts*, and other parts like e.g. the behavior of operations are described by C++ code directly. By using the built-in code generator of Rhapsody one can automatically generate a complete C++ program that contains both C++ representations of the graphical elements of a Rhapsody model and the explicitly provided C++ code of the operations. Figure B.1 shows an object model diagram of the PBX model. An object model diagram describes classes, objects and relationships between the classes and objects of a particular model. In the PBX model, there is a so-called *composite class* PBX that contains different so-called *parts*. Each part is represented as one or more objects of other classes. More precisely, the PBX class contains one object of the class Registry, one object of the class CallRouter, 4 objects of the class Connection, 4 objects of the class Line and 4 objects of the class Telephone. The number of objects for each class can be seen in the top left corner of the corresponding class symbol. Additionally, the object model diagram also shows some relationships between the

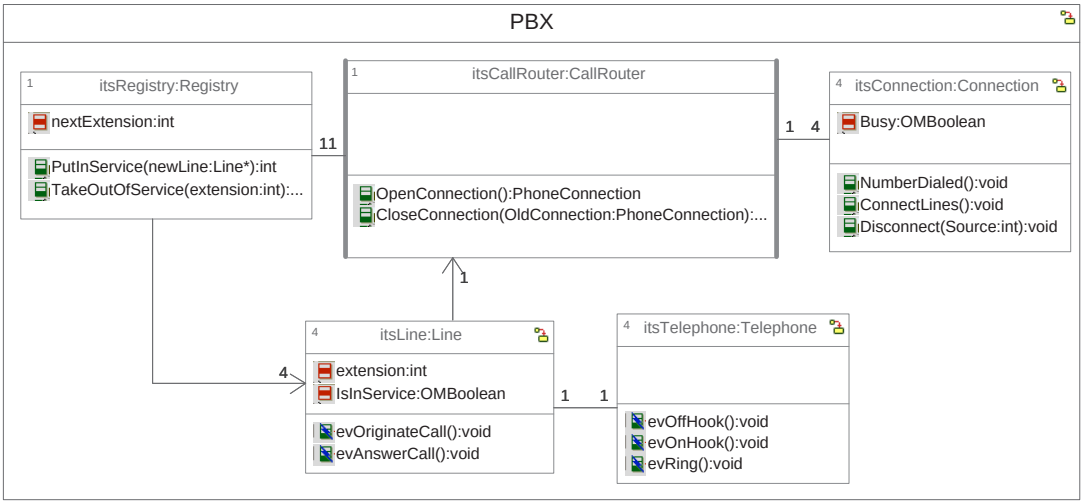


Figure B.1: **Object model diagram of class PBX**

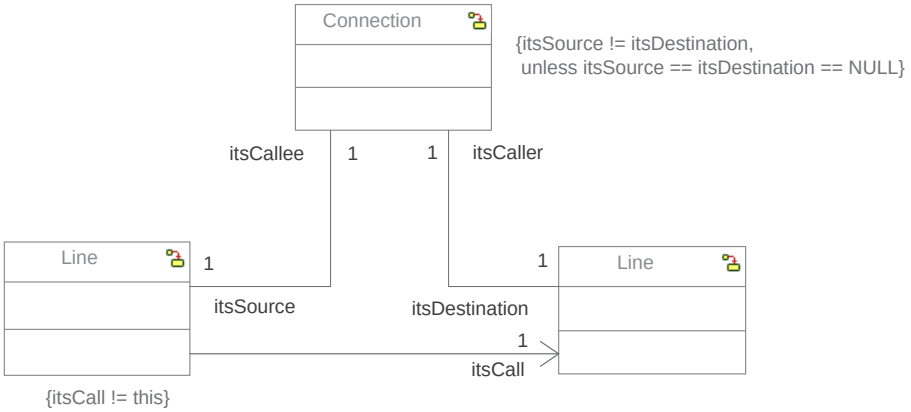


Figure B.2: **Object model diagram of class Connection**

different objects inside the PBX class by means of so-called *links*, which are represented graphically as directed or undirected lines between objects. A link is an instantiation of a so-called *association*, which represents a relationship between classes. Whenever there is an association between a class A and a class B, an object (or instance) of class A can have a link to an object of class B. In fig. B.1 one can see that the Registry object has a link to the CallRouter object and one link to each of the 4 Line objects. Since the line between the Registry object and the CallRouter object is undirected, also the CallRouter object has a link to the Registry object. Contrary to this, since the line between the Registry object and the Line objects is directed, only the Registry has links to the Line objects, but not vice versa. Furthermore, fig. B.1 also shows that each instance of Line is linked to one instance of Telephone and vice versa, the CallRouter instance has links to each of the 4 instances of Connection and each instance of Connection has a link to the instance of CallRouter.

Another object model diagram showing the relationship between class Line and class Connection is depicted in fig. B.2. There exist several named associations between an instance of Connection and an instance of Line. Each connection has two associations named `itsSource` and `itsDestination`, whereby `itsSource` represents an association to the line corresponding to the calling telephone and `itsDestination` represents an association to the line corresponding to the called telephone. Furthermore, each line has three associations named `itsCallee`, `itsCaller` and `itsCall`, whereby `itsCallee` represents an association to a connection that is used if the line belongs to the telephone that initiates a call, `itsCaller` represents an association to a connection that is used if the line belongs to the telephone that receives a call, and `itsCall` represents a direct association to the connected line once a call has been established.

Besides object model diagrams, Rhapsody offers a so-called *browser* that contains a hierarchical representation of the entire UML model. Figure B.3 shows the browser of

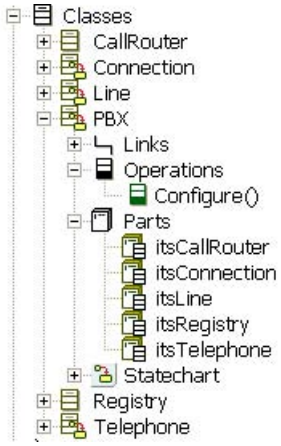


Figure B.3: **Browser view of the entire PBX model**

the PBX model. The browser shows the classes CallRouter, Connection, Line, PBX, Registry and Telephone. As can be seen in fig. B.3, the class PBX has an operation `Configure()`, and it contains the parts we have already seen in fig. B.1. To get an impression about the way the C++ code generated by the automatic code generator reflects classes, objects and their relationships, in the following we list the C++ code that contains the declarations for the class PBX[1]:

```
#ifndef PBX_H
#define PBX_H
#include <oxf/oxf.h>
#include "PbxPkg.h"
#include <oxf/omthread.h>
#include <oxf/omreactive.h>
#include <oxf/state.h>
#include<oxf/event.h>
#include <oxf/omlist.h>
//## package PbxPkg
//-----------------------------------------------------------------
// PBX.h
//-----------------------------------------------------------------
class CallRouter; class Connection; class Line;
class Registry; class Telephone;
// The PBX is a system level object which is responsible
// for instantiating and configuring all of the objects in the
// system.  This object is instantiated from the main() function.
```

[1]The listed C++ code is slightly formatted for better readability.

```
//## class PBX
class PBX : public OMReactive {
////    Friends    ////
public :
////    Constructors and destructors    ////
public :
   PBX(OMThread* p_thread = OMDefaultThread);
   ~PBX();
////    Operations    ////
public :
   //## operation Configure()
   void Configure();
////    Additional operations    ////
public :
   CallRouter* getItsCallRouter() const;
   CallRouter* newItsCallRouter();
   Connection* newItsConnection();
   Line* newItsLine();
   Registry* getItsRegistry() const;
   Registry* newItsRegistry();
   Telephone* newItsTelephone();


////    Framework operations    ////
public :
   OMIterator<Connection*> getItsConnection() const;
   OMIterator<Line*> getItsLine() const;
   OMIterator<Telephone*> getItsTelephone() const;
   //rootState:
   inline int rootState_IN() const;
   virtual void rootState_entDef();
   virtual int rootState_dispatchEvent(short id);
   //Ready:
   inline int Ready_IN() const;
   virtual OMBoolean startBehavior();
protected :
   void initRelations();
   void initStatechart();
////    Relations and components    ////
protected :
   CallRouter* itsCallRouter;
   OMList<Connection*> itsConnection;
   OMList<Line*> itsLine;
```

```
    Registry* itsRegistry;
    OMList<Telephone*> itsTelephone;
////    Framework    ////
protected :
    //states enumeration:
    enum PBX_Enum{ OMNonState=0, Ready=1 };
    int rootState_subState;
    int rootState_active;
};
#endif
```

In the generated C++ code, the class PBX inherits from the framework class OMReactive. Furthermore, the generated C++ code contains several declarations of variables and operations. For instance, member variables `itsConnection`, `itsLine` and `itsTelephone` are declared. As can be seen in the definition of the class PBX, these member variables are so-called *containers* that can hold an arbitrary number of objects of a specific type. These containers are realized as C++ template classes, and they are also part of the Rhapsody UML framework. For instance, the member variable `itsTelephone` has the type `OMList<Telephone*>`. The template class `OMList` implements a double linked list and provides several operations to manage the list. For instance, one can use so-called *iterators* to access individual elements in the list. An iterator is an object that can access the elements stored in a container object. Normally, iterators provide operations to get the element the iterator is currently pointing at, and also operations to move from the current element to the first, last and next element of the associated container. With the declaration

```
OMList<Telephone*> itsTelephone;
```

a member variable `itsTelephone` is defined as a container that is organized as a list of pointer of Telephone objects. Additional declarations of member variables are generated for the other parts of the class PBX, i.e. for the lines, connections, the CallRouter and the Registry. The constructor operation `PBX(OMThread*p_thread=OMDefaultThread)` is called whenever a new object of the class PBX is created, and it is implemented as follows:

```
PBX::PBX(OMThread* p_thread) :
 itsConnection(), itsLine(), itsTelephone() {
    setThread(p_thread, FALSE);
    initRelations();
    initStatechart();
}
```

The argument of the constructor of the type `OMThread*` which is used as a parameter in the call of the operation `setThread` specifies to which thread the newly created reactive object should belong. Additionally, the member variables `itsConnection`, `itsLine` and

itsTelephone are initialized, and the operations initRelations and initStatechart are called. The operation initRelations realizes the creation of the parts of a PBX object and is implemented as follows:

```
void PBX::initRelations() {
   itsCallRouter = newItsCallRouter();
   for (int i = 0; i < 4; i++) newItsConnection();
   for (int i = 0; i < 4; i++) newItsLine();
   itsRegistry = newItsRegistry();
   for (int i = 0; i < 4; i++) newItsTelephone();
   OMIterator<Connection*> frIter(itsConnection);
   while (*frIter){
     (*frIter)->setItsCallRouter(itsCallRouter);
     frIter++;
   }
   OMIterator<Telephone*> frIter(itsTelephone);
   OMIterator<Line*> toIter(itsLine);
   while (*frIter){
     (*frIter)->setItsLine(*toIter);
     toIter++;
     frIter++;
   }
   itsCallRouter->setItsRegistry(itsRegistry);
   OMIterator<Line*> frIter(itsLine);
   while (*frIter){
     (*frIter)->setItsCallRouter(itsCallRouter);
     frIter++;
   }
}
```

Firstly, the CallRouter object is created, and then 4 connections, 4 lines, a registry and 4 telephones. This is realized by calling the operations newItsCallRouter, newItsConnection, newItsLine, newItsRegistry and newItsTelephone. Within these operations, pointers to the generated objects are stored in the corresponding member variables of the PBX object. After all parts have been created, the links between the objects are instantiated by using iterator objects on the member variables of the PBX object. For instance, the links of the connections to the CallRouter are set within the following loop:

```
OMIterator<Connection*> frIter(itsConnection);
while (*frIter){
  (*frIter)->setItsCallRouter(itsCallRouter);
  frIter++;
}
```

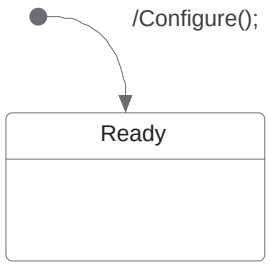


Figure B.4: **Statechart of class PBX**

After the call of `initRelations`, in the constructor of the PBX class the operation `initStatechart` is called. This operation realizes the starting of the startechart that is associated with the PBX class. Figure B.4 shows the statechart for the PBX class. It consists of only one state and a default transition into this state. When an object of the class PBX is created, the statechart associated with PBX is started, and the default transition is taken. In the action part of the default transition the operation `Configure` is called, which is implemented as follows:

```
void PBX::Configure() {
   //#[ operation Configure()
   OMIterator<Line*> iLine(itsLine);
   for (iLine.reset(); *iLine; iLine++) {
     (*iLine)->setItsCallRouter(itsCallRouter);
     itsRegistry->PutInService((*iLine));
   }
   //#]
}
```

The operation `Configure` sets the links from the lines to the CallRouter. Additionally, all lines are put into service by calling the operation `PutInService` of the registry object. After the call of `Configure` returns, also `initStatechart` and therefore the constructor of the PBX class terminate, i.e., after creating one object of the PBX class the whole PBX system is completely initialized.

The reactive behavior of the running PBX system can be understood by looking at the statechart of the class Telephone that is depicted in fig. B.5. A telephone can react to different events which are special objects that can be send to reactive objects. The initial state of a telephone is `Idle`. Independent of current state of a telephone, whenever a telephone receives an event `evRelease` or an event `evOnHook` it switches to the `Idle` state. When the telephone is idle and it receives an event `evOffHook` it changes its state to `Calling`. When taking this transition, it sends an event `evOriginateCall` to its associated line object whose statechart can be seen in fig. B.6 on page 195. When

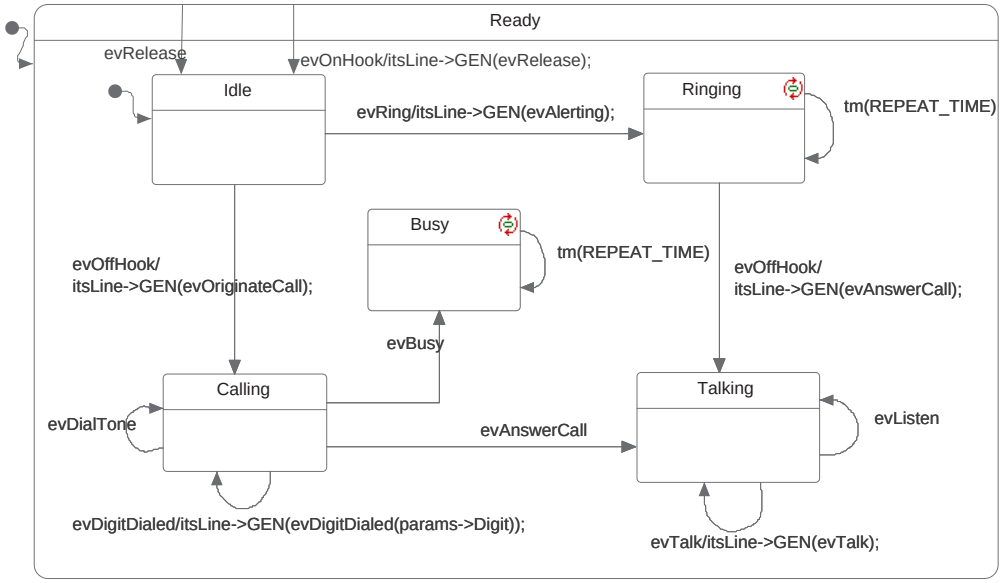


Figure B.5: **Statechart of class Telephone**

receiving this event, the line object takes the transition from state `Idle` to `Dialing`. In the action part of this transition, the line first calls the operation `OpenConnection` and then emits an event `evDialTone` back to the telephone. The operation `OpenConnection` is implemented as follows:

```
void Line::OpenConnection() {
    //#[ operation OpenConnection()
    Connection * NewConnection = NULL;
    setItsCall(NULL);
    setItsCaller(NULL);
    NewConnection = itsCallRouter->OpenConnection();
    setItsCallee( NewConnection );
    NewConnection->GEN(evOriginateCall);
    Source = 1;
    //#]
}
```

This operation computes a pointer to a free connection `NewConnection` by calling the operation `OpenConnection` of the CallRouter. This operation checks if one of the available connections are currently unused, and returns a pointer to an appropriate connection. After having computed a free connection, within the operation `OpenConnection` an event `evOriginateCall` is sent to the chosen connection, whose statechart is depicted in fig.

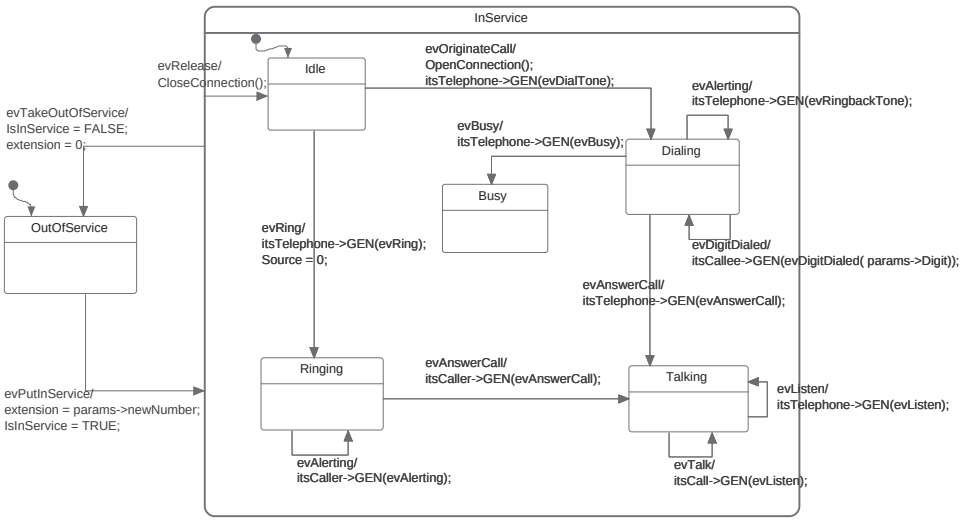


Figure B.6: **Statechart of class Line**

B.7 on page 196. When receiving this event, the connection takes the transition from state `Disconnected` to state `CollectDigits`.

Among other events, within state `Calling` a telephone can receive an event `evDigitDialed` that represents the dialing of a certain digit. When receiving such an event, the telephone propagates this event to its line and the line propagates it to its current associated connection. The connection collects the dialed digits until the operation `DialingDone` used as a guard in the transition with the trigger `evDigitDialed` returns true, which is the case if two or more digits have been entered. When `DialingDone` is true, by calling the operations `NumberDialed` and `ConnectLines` a pointer to the destination line associated to the telephone belonging to the dialed number is stored in the member variable `itsDestination` of the connection:

```
void Connection::ConnectLines() {
   Line * DestinationLine =
     itsCallRouter->getItsRegistry()->getItsLine( Extension );
   if (!IS_IN(DestinationLine->Idle)) itsSource->GEN(evBusy);
   else {
     setItsDestination( DestinationLine );
     itsDestination->setItsCallee(NULL);
     itsSource->setItsCall( itsDestination );
     itsDestination->setItsCall( itsSource );
     DestinationLine->GEN(evRing); } }
```

In the operation `ConnectLines`, first a pointer named `DestinationLine` is computed that points at the line belonging to the entered number. If the destination line be-

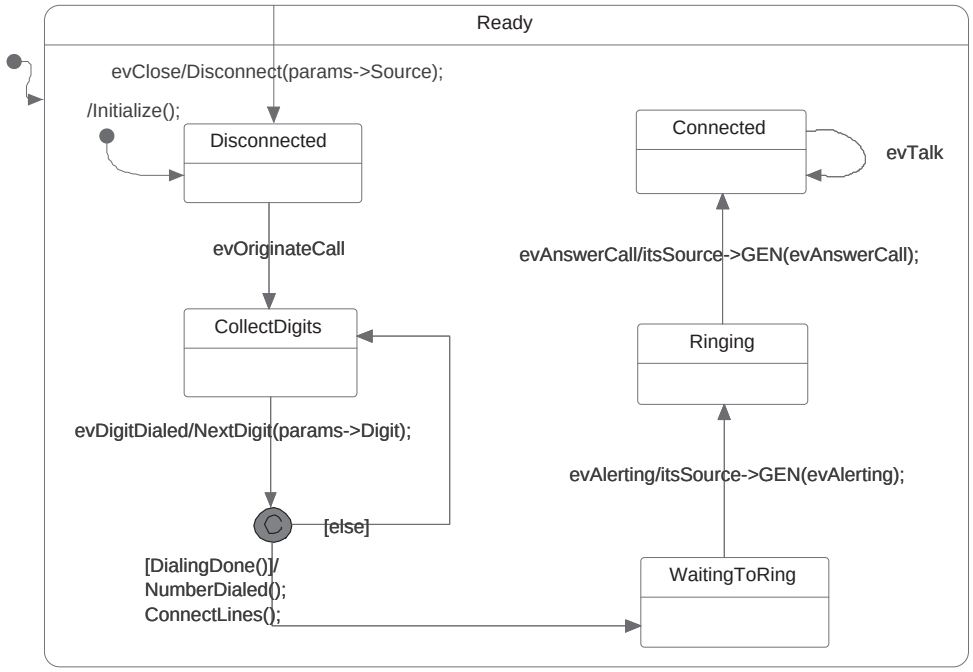


Figure B.7: **Statechart of class Connection**

longs to a telephone that is not in its idle state, an event `evBusy` is sent to the source line and propagated from the source line to its telephone. Otherwise, if the destination line belongs to a telephone that is indeed in its idle state, the member variable `itsDestination` of the connection is set and also the member variables `itsCallee`, `itsCaller` and `itsCall` of the source line and the destination line, i.e., a connection between the source line and the destination line has been established. As a result, an event `evRing` is sent to the destination line and propagated to the destination telephone, i.e., the telephone belonging to the dialed number is ringing since it changes its state from `Idle` to `Ringing`.

If an event `evOffHook` is now sent to the ringing telephone, it emits an event `evAnswerCall` to its line and changes its state to `Talking`. The event `evAnswerCall` is consumed by the associated line, which also emits an event `evAnswerCall` to the line of the calling telephone. This line, which is currently in state `Dialing`, consumes this event, sends also an event `evAnswerCall` to its associated telephone and changes its state to `Talking`. After that, the source telephone receives this event and also changes its state to `Talking`, i.e., both the source and the destination telephone are in state `Talking`. As long as both telephones are in this state, each telephone can receive events `evTalk` and `evListen`. If a telephone receives an event `evTalk`, it sends an event `evListen` via its line to the other telephone. If one of the telephones receives an event `evOnHook` it

changes its state again to `Idle` and it emits an event `evRelease` to the other telephone. As a result, also the other telephone changes its state to `Idle` again, thereby terminating the call. For this program, in the experiments from sect. 4.1.4 and sect. 4.2.1 we checked the reachability of a property that is fulfilled if a call was established correctly, i.e., if a certain telephone is in state `Talking`. Additionally, in the experiments from sect. 5.3 we checked the reachability of a property that is fulfilled if a certain telephone is in state `Busy` which is the case if this telephone is calling a telephone which is already part of an established call.

Curriculum Vitae

19. Februar 1975	geboren in Osnabrück
1981 - 1985	Grundschule Ueffeln
1985 - 1987	Orientierungsstufe Bramsche
1987 - 1994	Greselius-Gymnasium Bramsche, abgeschlossen mit dem Abitur
1994 - 1995	Grundwehrdienst in Rotenburg und Fassberg
Oktober 1995 - April 2000	Studium der Informatik an der Universität Oldenburg mit Nebenfach Physik. Die Diplomarbeit mit dem Thema ,,Eine Testautomatensemantik für Constraint Diagrams und ihre Anwendung“ wurde von Prof. Dr. Ernst-Rüdiger Olderog und Dr. Henning Dierks begutachtet.
Mai 2000 - August 2005	Wissenschaftlicher Mitarbeiter am Institut OFFIS im Bereich ,,Sicherheitskritische Systeme“
Seit September 2005	Mitarbeiter der OSC Embedded Systems AG
25. November 2005	Disputation