THESES ON INFORMATICS AND SYSTEM ENGINEERING C38

Two State Space Reduction Techniques for Explicit State Model Checking

JUHAN-PEEP ERNITS

TALLINN 2007

TALLINN UNIVERSITY OF TECHNOLOGY
Faculty of Information Technology
Department of Computer Science

This dissertation was accepted for the defence of the degree of Doctor of Philosophy in Engineering on October 16, 2007.

Supervisor: Prof. Jüri Vain, Department of Computer Science, Tallinn University of Technology

Opponents: Prof. Kim Guldstrand Larsen, Department of Computer Science, University of Aalborg
Prof. Varmo Vene, Department of Computer Science, University of Tartu

Defence: November 29, 2007

Declaration: Hereby I declare that this doctoral thesis, my original investigation and achievement, submitted for the doctoral degree at Tallinn University of Technology has not previously been submitted for any degree or examination.

/Juhan-Peep Ernits/

ISSN 1406-4731
ISBN 978-9985-59-736-1

INFORMAATIKA JA SÜSTEEMITEHNIKA C38

Kaks olekuruumi kahandamise tehnikat olekute otseesitusega mudelikontrollis

JUHAN-PEEP ERNITS

TALLINN 2007

Two State Space Reduction Techniques for Explicit State Model Checking

Abstract

This thesis is focused on automated analysis of formalised requirements and models of software. We examine more closely two different but partly overlapping techniques for analysis: model checking and model-based testing. In both applications, the automation involves enumerating all possible states of the formalised model. Such state spaces can easily grow too big to be tractable by present day computers. We propose and analyse two different techniques for reducing the state space that needs to be enumerated for proving properties that can be defined in terms of reachability.

Before proceeding to the actual techniques, we have a look at three different formalisms for writing the models. We look at *Promela*, which is the modelling formalism used in *Spin* model checker and various other tools, the visual timed automata formalism used in *Uppaal* and a way of modelling introduced in *NModel* using C# programs that utilise the modelling library of the toolkit.

The notion of symmetry reductions has been around in the model checking and program analysis community for a long time. By introducing the notion of model programs that facilitate modelling using abstract data structures like sets, maps, and multisets, and object instances, we present an algorithm to create state graphs and an algorithm to perform a state graph isomorphism based symmetry reduction. Such reduction allows to explore all reachable structurally distinct states.

Although powerful, the state isomorphism based reduction has its limitations as calculating the state graph and comparing it to previously seen state graphs consumes additional processor time.

To demonstrate how model checking can still yield interesting results for models where full state space search with other reduction methods fails due to memory and processor time limitations, we will introduce an extension of the sequential hashing technique known in the context of bitstate hashing. The method is called iterated search refinement with bitstate pruning. The key idea is that we will iteratively prune the search space by collisions in small bitstate hash tables which can only distinguish a fraction of the reachable states of the model at a time. We show how such method hits interesting results when there exist paths to the state where our condition of interest holds and evaluate the method in several different examples. We also propose a way to extract an abstraction function from the successful runs of the bitstate pruning based search method. Of course, there is a trade-off related to this method as we give up traversing the whole state space of the model. When we find a path to a state where the desired property holds, it is a valid path in the concrete model, but the method is incomplete, as there are parts of the model that may remain unexplored, thus it cannot be used for proving unreachability of states with given properties.

The last part of the thesis provides two case studies of applying the proposed iterated search refinement with bitstate pruning for model-based offline test generation and for a memory arbiter synthesis.

Kaks olekuruumi kahandamise tehnikat olekute otseesitusega mudelikontrollis

Lühikokkuvõte

Käesolev doktoritöö keskendub tarkvara mudelite ja formaliseeritud nõuete automaatsele analüüsile. Me vaatleme lähemalt kahte erinevat, kuid osaliselt kattuvat analüüsitehnikat: mudelikontrolli ja mudelipõhist testimist. Mõlemas rakenduses kätkeb automatiseerimine endas formaliseeritud mudeli kõigi olekute läbivaatust. Selliselt võivad olekuruumid kergesti kasvada tänapäevastele arvutitele täielikuks läbivaatuseks liiga suureks. Me pakume välja ja analüüsime kaht erinevat tehnikat läbitava olekuruumi kahandamiseks saavutatavusülesannete puhul.

Enne konkreetsete tehnikate kirjeldamist vaatleme kolme erinevat modelleerimise formalismi, kus kasutatakse mittedeterminismi. Käsitleme *Promelat*, mis on *Spini* nimelises mudelikontrollijas kasutatav modelleerimiskeel, graafilist ajaga automaatidel põhinevat formalismi, mida kasutatakse *Uppaalis*, ning *NModeli* nimelises tööriistakomplektis kasutatavat C# keeles mudelite kirjutamist kasutades NModeli modelleerimisteegi vahendeid.

Sümmeetriareduktsioonidest on mudelikontrollis ja programmianalüüsis kõneletud juba kaua. Võttes kasutusele mudelprogrammide mõiste, mis võimaldab modelleerimisel kasutada abstraktseid andmetüüpe nagu hulgad ja kujutised ning objekte, mille puhul konkreetne aadress mälus ei ole oluline, saame defineerida uue sümmeetriareduktsiooni, mis tugineb erinevate olekute olekugraafide isomorfismil. Selline reduktsioon võimaldab olekute läbivaatusel uurida kõiki saavutatavaid struktuurilt erinevaid olekuid.

Kuigi kirjeldatud sümmeetriareduktsioon on võimas, kulub olekugraafide tuletamisele ja isomorfismi arvutamisele märkimisväärselt arvutusressurssi.

Näitamaks, kuidas mudelikontroll annab siiski huvitavaid tulemusi mudelite korral, mille puhul terve olekuruumi läbimine ebaõnnestub piisavate operatiivmälu- ja protsessoriressursside puudumise tõttu, kirjeldame ja analüüsime oleku paisktabelis ühe bitina esitamise meetodile tugineva erinevate paiskfunktsioonide järjestikku kasutamise tehnika edasiarendust. Väljapakutavaks tehnikaks on iteratiivne otsingutäpsustus oleku bitina esitamisel põhineva otsinguruumi kärpimisega. Põhiidee seisneb olekuruumi iteratiivses juhuslikus kärpimises väikestes paisktabelites erinevate paiskfunktsioonide korral, kuna mudeli erinevatest olekutest arvutatakse paisktabelis sama aadress ja peetakse olekuid seetõttu sarnasteks. Paiskfunktsiooni muutmine muudab paisktabeli kokkulangevuste mustreid. Me muudame paiskfunktsiooni paisktabeli suuruse muutmisega, alustame väga väikeste paisktabelitega ja näitame, kuidas kirjeldatud meetod annab huvitavaid tulemusi mitmetel erinevatel näidetel, kus mudelis leidub tee meid huvitavasse olekusse. Loomulikult ei kata kõnealune meetod enam garanteeritult mudeli kogu olekuruumi. Kui leiame tee olekusse, kus meid huvitav omadus kehtib, siis on see tee olemas ka meie mudelis, kuid mõned olekuruumi osad võivad jääda läbi vaatamata.

Töö viimases osas vaadeldakse kaht kirjeldatud iteratiivse otsingutäpsustuse rakendust mudelipõhises testimise ja radarisüsteemi mäluarbiitri sünteesi näites.

Acknowledgments

First of all it should be said that writing a PhD thesis is bad for your health. During different phases of writing it causes sleeplessness and permanent fatigue, desire to pull out all of one's hair, lack of exercise and almost no chance to be exposed to fresh air, and basically no social life. It can also wreck the nerves of you and your supervisor. Still there are positive sides to PhD studies too, like getting to know things you would othervise never have the time for and the possibility to travel to conferences and summer schools and meet dozens of exciting people with crazy ideas. The world opened up to me in a wonderful way while I was a PhD student. In addition to numerous shorter trips to many different places I spent several months at Aalborg University at the Distributed Systems and Semantics Group learning about Uppaal and spent three months as an intern at Microsoft Research, Redmond.

Together with my primary and secondary school, International Baccalaureate, BSc, MSc, and PhD studies I have been going to school for a quarter of a century. I guess it is now time to move on with deep gratitude to my family, my teachers, my friends, and the society in general that has facilitated such luxury.

The thesis would not have happened without the support and encouragement from a number of people. First of all I would like to thank my parents for providing the circumstances where obtaining education was viewed as a basic human right and supported in every possible way. My wife, Eneken, has been extremely tolerant and supportive and has been my main source of energy and inspiration. She played a major role in getting me to complete my thesis. I also thank my brother, who I have always looked up to, for providing the example that education matters and borders do not matter when you want to get to know something.

I owe my supervisor, Prof. Jüri Vain, sincere and deep thanks for reqruiting me to the Institute of Cybernetics almost 11 years ago and for providing an abundance of interesting tasks and guidance while still leaving me enough freedom ever since. His work in establishing and running the Department of Computer Science at the Faculty of Information Technology of Tallinn University of Technology that provides courses and specialisation with an emphasis on formal methods and logic has an important impact to the landscape of education in information technology in Estonia.

It is crucial to have a work environment with people with close interests for bouncing ideas and having discussions as understanding complicated things requires occasional verbal expression accompanied by scribbling things on a whiteboard. In addition to interaction with my wonderful colleagues at the Institute of Cybernetics and the Department of Computer Science I have been very lucky to have met many excellent people during guest lectures, winter and summer schools, theory days, and a number of conferences and workshops that have happened in Estonia and abroad. Thus I would like to thank all of my colleagues, in particular Tarmo Uustalu for his energy, kindness, and successful effort in making the work environment international and active, and Jaan Penjam and Jüri Vain for providing me the work environment and for tackling the non-trivial administrative burden.

I thank Prof. Kim G. Larsen for letting me feel as one of the team during my stay in Aalborg and for accepting to be one of my opponents. I thank Margus Veanes for inviting me to do an internship at MSR and for aquainting me to model programs.

I thank all of the co-authors of the papers that form the basis of parts of this thesis: Colin Campbell, Andres Kull, Kullo Raiend, Jüri Vain, and Margus Veanes. I am grateful to Theo Ruys for valuable comments on my thesis.

I would also like to thank the opponents of my thesis whose scrutiny, critique, and comments I sincerely respect and in most aspects agree to.

Last but not least thanks to the organisations that supported me during my PhD work. I was employed part time by the Institute of Cybernetics and part time by the Department of Computer Science of Tallinn University of Technology. Via the latter I was involved in the "Integration Platform for Development Tools of Embedded Systems" project of the ELIKO Competence Centre. In addition, I was supported by the Tiigriülikool and Tiigriülikool+ projects of the Estonian Information Technology Foundation, by the Information and Communication Technology Doctoral School, by a Marie Curie Fellowship from the European Commission, by the Estonian Science Foundation grant number ETF5775, and by the Centre of Excellence for Dependable Computing financed by the Ministry of Education and Science of Estonia.

Contents

CHAPTER 1

INTRODUCTION

This thesis discusses two different state space reduction techniques to be used in explicit state model checking and model-based testing. We will look at symmetry reductions of model programs with abstract data structures and objects, and iterated search refinement with bitstate pruning.

In this introductory chapter we will motivate the our work with the grand challenge of correct software and give a brief overview of some areas which have received extensive attention from the research community. We will give a concise summary of our contribution and a map to the rest of the thesis.

1.1 Correctness of Software

It is essential for an engineer to establish some degree of confidence in that the system he/she builds will actually behave as expected with respect to the requirements. There are a number of ways to build up such confidence and usually a combination of different approaches is used. In the current work we will focus on two of the methods used for the analysis of software – formal verification by model checking and model-based testing.

A typical text about verification, i.e., ways of establishing the correctness relationship, starts with a reference to some examples where a silly software error has caused some major damage, like the $475 million cash setback that the Pentium FDIV bug [Nicely, 1994] caused to Intel. The current thesis is no exception as we already mentioned the Pentium bug, but we refer the reader to more examples listed in texts like, for example, [Edmund M. Clarke, Grumberg and Peled, 1999; Huth and Ryan, 2000; Holzmann, 2003; Zeller, 2005].

Thus, as there are more and more automated systems that people use on daily basis, we need to make sure that the systems, including the software contained in the systems, are built rigorously. Rigorous construction of software means that the key properties of the system are specified in some mathematically precise way and that the validity of the properties in the system is established by some mathematical means. For example, a typical safety requirement for a system involving multiple active concurrent processes would be not to deadlock.

In [Hoare, 2003] Sir Tony Hoare presented a Grand Challenge of creating a verifying compiler. The Verified Software: Theories, Tools, Experiments [VST, 2005] conference in Zürich was attended by a prominent selection of researchers from different branches of computer science. The different but relevant topics are best summarised in a quote from [Hoare, 2006]:

> The relevant topics of research include programming language semantics, programming principles, type theory, compiler construction, program analysis and optimisation, test case generation, mathematical modelling, programming methodology, design patterns, dependability, software evolution, and construction of programmer productivity tools. In addition there are various approaches to mechanical theorem proving, which include proof search, decision procedures, SAT solving, first-order induction, higher order logic, algebraic reduction, resolution, constraint solving, model checking, invariant abstraction, and abstract interpretation. These lists are not intended to be complete; new ideas are very necessary, and will be welcomed from any quarter.

One of the ways to towards the goal is pursued by language theorists work toward developing programming languages that prove a number of properties correct at compile time. An example of a recent important increment of language technology for the masses is the addition of generic types to Java 5 and .Net 2.0. The addition of generics helps to root out a whole class of errors at compile time (even at write time with on-line compilers embedded into integrated development environments like, for example, Eclipse [`http://www.eclipse.org`] and Microsoft Visual Studio [`http://www.microsoft.com/vstudio`]) and makes the programs more readable. Still, the compilers of Java and C# are far from proving properties like deadlock freedom of multithreaded applications and static array bounds checks.

Another way towards the goal of correct implementation is to prove the program correct. This can be done by using interactive proof assistants like PVS, Coq, Isabelle/HOL, to name a few. These proof assistants help to formalise the design requirements and the code implementing them and can cope with simpler proof steps automatically, but generally rely on the user to choose the next proof step. This approach is powerful but laborious and slow. Wolfgang J. Paul said in a talk delivered at the Grand Challenges in Informatics symposium in Budapest [GCI, 2006] that full manual program verification using Hoare logic has a yield of 30-50 lines of code per week per person with an average of 10 lines of proof generated per line of code. He compared the productivity with that of manual testing with the productivity of 50 lines of code per week per person.

An alternative to using proof assistants that facilitate higher order logics with high expressivity but require manual guidance for the proofs is to use a decidable subset of some logic and make the decision procedure fully automatic. Model checking is an example of such an approach.

According to [Myers, Badgett, Thomas and Sandler, 2004], both in 1979 and still in 2004 approximately 50% of the time and more than 50% of the cost of producing software is spent on testing while quite often still several annoying and/or critical errors go undetected in the development phase. While having deep respect towards proving the correctness of systems, this thesis is motivated in the spirit of one of the less known citations of Edsger Dijkstra [Dijkstra, 1965]:

> One can never guarantee that a proof is correct, the best one can say, is: "I have not discovered any mistakes".

As is also stressed in [Myers, 1979] the main goal of testing (or modelling and analysing) a system is to actually discover some concrete issues rather than to state that no issues were found. The main danger when stating that no issues were found is that the requirements had mistakes that went undiscovered.

Simply stating that the system is correct has the danger that we might have also made mistakes in the specification. Thus verification and testing must go hand-in-hand and ideally the results of the proof of correctness should be utilised for designing representative tests for the implemetation.

Another motivation for smarter testing is that an implementation that behaved correctly right after production might have deteriorated over time, thus there is a need to periodically recheck the system. Evidence of the relevance of this claim can be found in processor technology where processors must work under extreme conditions (heat), from corrosion of socket connections, from degradation of electrolytic capacitors, etc.

The following section will give a brief overview of frequent error types that are addressed by whole communities of researchers.

1.2 Common Types of Errors in Software

The goal of this section is to give a non-exhaustive map of the types of errors that are currently or were very recently active targets of research in computer science.

As programmers are humans, they make mistakes. The most common mistakes during writing some program are in the program logic and are related to overlooking some combinations of input data or imprecise assumptions about the data. The correctness of the program can be checked with respect to the requirements according to which the program was written. In the ideal case it would be possible to automatically prove whether the program is correct.

In addition to the problems at the conceptual level, there are whole classes of problems that are specific to the programming task, the system where it is solved in, and to the programming language and compiler which is used for solving the problem.

To give some idea about such types of issues and to pinpoint where we stand in the domain, we briefly outline some of the most common types of errors on which

either separate or ovelapping communities work, providing solutions to dealing with such problems or avoiding them altogether.

The properties of systems are often divided into two categories: *safety* and *liveness*. Safety properties specify which properties should not be violated in the system or which *bad* state should not be reached. Liveness properties specify which properties the system must satisfy or which *desirable* things should happen in the system.

1.2.1 Sequential C code

C code is an example of program code where the user has to manually take care of the resource allocation for the program. Some call it "unmanaged" code. Memory has to be allocated and freed for all data structures except those allocated on the stack. Thus, in addition to potential errors with respect to the initial requirements, the C language introduces a whole variety of errors related to memory management: *memory leaks*, *aliasing*, *buffer overflows*, and *null pointer dereferencing*. Memory leaks are orphaned areas of allocated memory that have not been freed. Aliasing happens when a single memory area is referenced by different pointers and manipulating either variable causes changes to the single instance of the data structure. Buffer overflows occur when insufficient amount of memory is allocated for containing some data and writing past the end of the allocated buffer will corrupt memory areas used for other purposes potentially allowing the execution of arbitrary code. Null pointer dereferencing is taking the address of a pointer that has not been assigned a value (i.e. the pointer is null). This will usually end the execution of the program with the segmentation fault signal. During the recent years also the property of *termination* has received significant attention as an important step for proving liveness of programs.

1.2.2 Sequential Code with Automatic Garbage Collection

With the introduction of programming languages like Java and C#, which have object oriented type systems and built in garbage collection, certain types of memory corruption problems have been eliminated altogether. It is not possible to cause a segmentation fault or corrupt memory by writing past the end of an allocated buffer in a Java or C# program, provided the compiler and the underlying virtual machine do not have errors. Memory leaks in the sense of a C program have also been eliminated as all instances of objects that are not transitively referenced from the root get garbage collected automatically.

Aliasing problem still perisists as it is possible to reference an instance of an object from multiple variables and thus cause *destructive updates* to an instance of an object that is expected to persist in some other part of the program. The buffer overflow problem is reduced to still challenging *array bounds* problem. In the ideal case it should be possible to verify statically that an array index will never exceed array size thus eliminating the need to check array bounds at runtime.

1.2.3 Protocol Related Problems

A great deal of software is organised into libraries that provide application programming interfaces (APIs) to other programs. It is often the case that in addition to knowing the types of the parameters and the return type of an API function, it is neccessary to know the protocol, i.e. the sequential restrictions of how different methods of an API should be accessed. A typical example is the file access library, where a file must be opened prior to being read from. These kinds of issues are closely related to the requirements of the library. There are several relations that can and should be checked: whether the library satisfies the protocol specification and whether the application that uses several APIs with different protocols uses all protocols correctly.

A whole new field of research opens up when one looks at several concurrently active programs that interact.

1.2.4 Concurrency

The problems related to concurrency are often very difficult to debug using methods for sequential code, as it is very difficult to reproduce the exact conditions of all parties interacting in a concurrent environment. Thus there is a whole field of research dedicated to inventing better ways of coping with concurrency related issues.

In addition to making sure that concurrent access to some API does not violate its protocol, a whole new set of problems related to concurrent access to resources emerges. Often such concurrent accesses are guarded by semaphores to denote the *locking* of the resource by some active program. Such protocols using locking are subject to the possibility of deadlocks and livelocks. Such problems are often referred to as *course grain* concurrency.

Sometimes semaphores are omitted for the sake of performance or because they were never introduced as the software was originally written for sequential environments. In such cases the problem of *data races* emerges as there are different possible interleavings of reading and writing a shared variable by different active programs. Some security policies require that there should be no data races present at all, but not all data races necessarily lead to bad behaviour. These problems are sometimes called *fine grain* concurrency.

1.2.5 Time

Some programs interact with the environment via sensors and actuators in addition to the human-computer interface. In cases where programs are used to control processes in the environment the time related properties become increasingly important. Reasoning about the properties of *real-time* systems constitutes another whole domain of research in verification which is not the focus of the current thesis. We will, though, use a real-time model checker Uppaal [Amnell, Behrmann, Bengtsson, D'Argenio, David, Fehnker, Hune, Jeannet, Larsen, Möller, Pettersson, Weise and Yi, 2001] but focusing on other aspects of modelling and verification than time.

1.3 Verification

In previous section we described several different types of problems that occur in software. Computer scientists have worked hard for several decades at providing means to solve such problems. Such methods can be grouped together under the term *verification* by which we mean ways and methods of ensuring that the implementation corresponds to the requirements.

The term "verification" has a bit blurred meaning and may denote different things. Sometimes verification means code reviews and sometimes it means rigorous mathematical proof that the code satisfies its formally stated requirements. Although code reviews can reveal issues in programs, the degree of confidence of correctness is considerably higher when the code is verified using a mathematical method.

One way to establish the relationship between the requirements and the implementation is to prove it by formalising the implementation and the requirements in a suitable logic or logic based specification language and providing a formal proof that the implementation satisfies the requirements. Although powerful, such procedure is also error prone when performed manually.

1.3.1 Automated Verification

The theory of computability [Hopcroft and Ullman, 1979] provides insight to what can be decided by an algorithm. The most well known negative result is of course the result by Alan Turing from 1936, where he proves that a problem now known as the halting problem is *undecidable* [Wikipedia, 2007][1]. Informally speaking, the result says that there is no general algorithmic way to tell whether an arbitrary given program terminates or not. In spite of such results, which exemplify that the problems the theoretical computer science deals with are of complex nature, researchers have come up with a variety of different automatic solutions which establish certain properties in models and/or program code.

1.3.2 Model Checking

Model checking [Edmund M. Clarke et al., 1999] is a technique for verifying finite state concurrent systems. It is a technique that proves or disproves a property of a model either by providing a witness trace to the state where the property is violated or enumerating all reachable states of the model thus proving that the violation of the property is not possible. The property to be proved can either be a reachability property, i.e., whether a certain configuration of valuations of state varialbes is reachable from the initial state, or a property specified in some temporal logic like Linear Temporal Logic (LTL) or Computation Tree Logic (CTL). In the current thesis we will concentrate on reachability properties and refer the reader to, for example, [Huth and Ryan, 2000] for an introduction and pointers about temporal logics.

[1] We are not analysing the halting problem and thus refer the reader to an article in an encyclopedia that has pointers to the original paper and in addition provides some nice examples.

In the previous section we talked about different error types or security policies of programs. Model checking is one possible tool for solving several of them, particularly for proving protocol and concurrency related properties. In some cases the model is extracted from the program code automatically but in other cases it has to be done manually by a verification engineer.

In a typical model checking task there is a model M of the system in some modelling language (timed automata, Promela, BIR, ...) and the design requirements ϕ in some logic (for example computation tree logic, CTL, linear temporal logic, LTL, or μ-calculus). Model checker is used to check whether the model satisfies the requirements or not ($M \models \phi$?). If not, ($M \not\models \phi$), the method gives a *trace* leading to the error. Another benefit of the method is a high level of automation, meaning that once the model and requirements are there, it is possible to "push the button" and wait for the answer. No user intervention is required during the search.

1.3.3 Model-Based Testing

Model-based testing is an automated technique for establishing a *conformance relation* between the requirements and the implementation. Model-based testing is typically used in cases where the implementation under test (IUT) is a black box and the only thing visible to the tester is the interface over which the communication occurs. Although it is not possible to verify the system correct using model-based testing, it is possible to establish a conformance relation, like, for example, alternating refinement [Veanes, Campbell, Schulte and Tillmann, 2005] between the tester model and the IUT.

A very important phase of model-based testing is the formalisation of the requirements. Sometimes the requirements can be documents of several hundred pages long and it is not trivial to build a model and keep the artefacts of the model traceable to a paragraph in the textual specification. It is also very difficult to have any confidence of the correctness of the specification itself unless it is analysed in some way. This is where model-based testing and model checking meet as given the requirements are formalised using a suitable formalism, it is possible to subject the requirements model to automated analysis.

It is important to note that the term *model* has different meanings in model-based testing and model checking. In model-based testing, the model is the formalisation of the requirements, while in model checking the requirements are specified as formulae in some temporal logic and the objective is to establish whether the model implements the specification. On the other hand, model checking the model which is used for model-based testing helps to establish the validity of the model of the requirements which is often obtained by formalising a narrative of several hundreds of pages and is thus an important task.

1.4 References to Previously Published Work

Several parts of this thesis have been previously published as conference papers or journal articles.

Chapter 3 is based on "State isomorphism in model programs with abstract data structures" [Veanes, Ernits and Campbell, 2007], a paper presented at FORTE'07 in June 2007 in Tallinn, Estonia.

Section 5.1 is based on "Generating tests from EFSM models using guided model checking and iterated search refinement" [Ernits, Kull, Raiend and Vain, 2006], a paper presented at FATES/RV'06 in August 2006 in Seattle, USA.

And, Section 5.2 is based on "Memory Arbiter Synthesis and Verification for a Radar Memory Interface Card" [Ernits, 2005], a paper published in the Nordic Journal of Computing.

1.5 Summary of the Contribution

First of all we show on a very simple example how different formalisms can be utilised to model the same aspects of a system. Promela models and Uppaal timed automata are both well established formalisms. We show that model programs with non-determinism on action argument and action selection level and that allow the use of object instances and abstract data types like sets, can be used to model similar systems. Further, we present algorithms to extract object graphs of the states of the model programs at runtime and perform state graph isomorphism based symmetry reduction. In such a way we have established a symmetry reduction that can recognize more symmetries than the scalar set approach with the penalty of more work to be done runtime. Although we were later referred to similar symmetry reduction work in the GROOVE project [Rensink, 2006], the differences of our approach in the context of model programs with explicit state analysis are outlined in Chapter 3.

In Chapter 4 we introduce a yet another variation of the bitstate hashing method with surprising consequences. We apply sequential hashing with the purpose of pruning the search space under each iteration and gain interesting reachability results by using only a few dozen kilobytes of memory for the bitstate hash table. We call the method iterated search refinement with bitstate pruning. Our contribution is the presentation of the approach and the application of the technique to several examples in Chapter 5: offline test generation, memory arbiter synthesis, and solving planning problems. The benefits of the approach are emphasised by easy parallelisability: the time it takes for the iterated search refinement with bitstate pruning to reach a goal is inversely proportional to the number of CPU cores used in the process.

1.6 Organisation of the Thesis

The thesis is organised into 3 logical parts. In Chapter 2 we give an overview of the state space reduction techniques used in explicit state model checking and intro-

duce three alternative modelling formalisms: Promela, Uppaal automata, and model programs of NModel. In Chapter 3 we present a state isomorphism based symmetry reduction technique for model programs containing abstract data structures and objects. In Chapter 4 we present an iterated search refinement with bitstate pruning technique that can be applied for model checking models too large to yield any results using other state space reduction methods. In Chapter 5 there are two case studies of applying the previously introduced iterated search refinement with bitstate pruning method.

CHAPTER 2

PREREQUISITES AND RELATED WORK

This chapter gives a more detailed overview of the techniques used in explicit state model checking with the emphasis on the two methods which will be looked at in more detail in later chapters. In addition this chapter contains examples of modelling the Dining Philosophers example using Promela, Uppaal automata, and model programs of the NModel toolkit.

2.1 State Space Reduction Techniques in Explicit State Model Checking

As already mentioned, explicit state model checking involves the consideration of all possible executions and states of the model. Due to the combinatorial explosion, such state enumerations may need excessive amounts of memory and thus time to be covered. It has therefore been an important direction of research to discover different ways of reducing the requirement for memory but still being able to prove or disprove the properties of interest.

In this section we will first list the basic search algorithms used in model checking and then have a look at the most significant known state space resuction techniques.

2.1.1 Search Algorithms

Typically the implementations of search algorithms used in explicit state model checkers are variations of *depth-first search* (DFS) and *breadth-first search* (BFS) algorithms. The graph the search is performed on is a finite state automaton (FSA) that is constructed as a product of the finite state automata of the active processes. A FSA, for example, [Holzmann, 2003] is a tuple $A = (S, s_0, L, T, F)$ where S is a finite set of states, s_0 is the initial state $s_0 \in S$, L is the finite set of labels, T is the set of transitions $T \subseteq (S \times L \times S)$, and F is a set of final states $F \subseteq S$. The pseudo code for depth-first and breadth-first algorithms is given in Figure 2.1 which summarise the core of the search algorithms used in the Spin model checker.

Depth-First Search

The strategy followed by depth-first search is to search "deeper" in the graph whenever possible [Cormen, Leiserson and Rivest, 1994]. In depth-first search transitions are explored out of the most recently discovered state s that still has unexplored transitions leaving it. When all of the outgoing transitions from s have been explored, the search "backtracks" by popping the stack D to explore transitions leaving the state from which s was discovered. In the case when a state violating the safety property has been found, the path from s_0 to s is directly represented by the stack D.

Breadth-First Search

Breadth-first search systematically explores the transitions of A to "discover" every vertex that is reachable from s_0 [Cormen et al., 1994]. Breadh-first search is so named because it expands the frontier between discovered and undiscovered vertices uniformly across the breadth of the frontier represented by the queue Q in Figure 2.1. That is, the algorithm discovers all vertices at distance k from s_0 before discovering any vertices at distance $k+1$. Due to that, breadth-first search finds the shortest path to a state. If a model checking task running a breadth-first search algorithm finds a witness trace, it is guaranteed to be the shortest possible. In practice, breadth first search will often lead to the exhaution of memory resources sooner than depth-first search. As the `BFSearch` algorithm does not have a stack representing the path from s_0 to s, it is necessary to reconstruct the path in the `FindPath` procedure.

The advantage of the breadth-first search is that it is guaranteed to find the shortest path to a state violating the safety property, but it comes at the cost of larger memory requirements than depth-first search.

Guided Search

Guided search is a variation of breadth first search or depth first search that utilises some guiding function to tell the search algorithm which transition to explore next. In the case of breadth-first the next transition to be taken is selected from among all enabled transitions of the frontier and in case of depth-first search only among the enabled outgoing transitions of the current node. A *cost* function is one of such guiding functions. Guided search in Spin is considered in [Edelkamp, Lafuente and Leue, 2001] and in [Ruys, 2003]. The priced timed automata and guided search principles implemented in Uppaal Cora are described in [Behrmann, Larsen and Rasmussen, 2005].

2.1.2 Partial Order Reduction

The most well known and established state space reduction techniques in explicit state model checking are *partial order reduction* and *symmetry reduction* techniques.

Partial order reduction technique for model checking was introduced in [Godefroid, 1996] Almost a decade later the method received an dynamic extension [Flanagan and

Depth-First Search
Stack $D = \{A.s_0\}$ — Search stack
Set $V = \{A.s_0\}$ — Set of visited states
BOUND — Bound on search depth

```
DFSEARCH()
    s ← StackTop(D)
    if !Safety(s)
            PrintStack(D)
    for each (s,l,s') ∈ A.T
            if !V.Contains(s')
                    V.Add(s')
                    if D.Depth < BOUND
                            D.Push(s')
                            DFSearch(D,V)
    D.Pop
```

Breadth-First Search
Queue $Q = \{A.s_0\}$ — Frontier of search
Set $V = \{A.s_0\}$ — Set of visited states

```
BFSEARCH()
    s ← Q.Head
    Q.RemoveHead
    if !Safety(s)
            FindPath(s)
    for each (s,l,s') ∈ A.T
            if !V.Contains(s')
                    V.Add(s')
                    Q.AddLast(s')
                    BFSearch()
```

Figure 2.1: Depth-first and breadth-first search algorithms for model checking [Holzmann, 2003].

Godefroid, 2005] that makes it possible to omit the static analysis step present in the initial method. The key idea is to not consider all possible interleavings of processes when it is clear that considering a single interleaving is sufficient. This is possible when the statements executed by separate processes are independent, i.e. they do not have effect on shared resources.

We admit that partial order reduction is a very important state space reduction technique, but as it is not in the focus of the current thesis, we refer the reader to, for example, [Peled, 1998].

2.1.3 Symmetry Reductions

Two program states, in the presence of pointers or objects, can be considered equivalent if the structure of the logical links between data objects is equivalent while the concrete physical addresses the pointers point to differ, i.e., when the actual arrangement of objects in memory is different due to the effects of memory allocation and garbage collection. This is known as one form of symmetry reduction and has been used in software model checking. The principles of such symmetry reductions have been outlined by Iosif in [Iosif, 2004]. One of the key ideas in [Iosif, 2004] is to canonize the representation of program heap by ordering the heap graph during a depth first walk. The order of outgoing edges (pointers) from a node (for example an object) is given by a deterministic ordering by edge labels (field name and order number, for example position in the array, in the parent data structure). Lack of such ordering would render state comparison to an instance of the graph isomorphism problem, which requires exponential time in the number of nodes in the general case [Messmer, 1995]. In [Musuvathi and Dill, 2005] Musuvathi and Dill elaborate on

Iosif's algorithm to allow incremental heap canonicalization, i.e., take into account that state changes are often small and modify only a small part of the heap, thus it should not be necessary to traverse the whole heap after each state change.

In addition to dSpin [Demartini, Iosif and Sisto, 1999], where the above mentioned principles were initially implemented, there are several analysis tools specifically targeted for object-oriented software that utilize the approach, for example, *Java Pathfinder* [Visser, Havelund, Brat, Park and Lerda, 2003], *XRT* [Grieskamp, Tillmann and Schulte, 2006], and *Bogor* [Robby, Dwyer and Hatcliff, 2006].

Java Pathfinder, JPF, is an explicit state software model checker for Java bytecode that grew out of a converter of Java to Promela and was originally developed at NASA. It is now in its fifth major release [JPF, 2007].

XRT is a software checker for common intermediate language, CIL. It processes .Net managed assemblies and provides means for analyzing the processed programs.

Bogor is a customizable software model checking engine that supports constructs that are characteristic to object-oriented software. Although there is support for using abstract data types, like sets, the underlying state enumeration and comparison engine performs heap canonicalization based on an ordering of object IDs based on the previously mentioned work by Iosif [Iosif, 2004].

Korat [Boyapati, Khurshid and Marinov, 2002] is a tool for automated test generation based on Java specifications. It also uses the concept of heap isomorphism to generate heaps that are non-isomorphic.

We have layered ASM semantics on top of the underlying programming environment and thus the concrete memory locations have been abstracted by interpreting the program state in the ASM semantics. But in addition to using the concrete data structures, we can declare some types to represent instances of abstract objects and there are some data structures, such as the *Set*, *Map* and *Bag*, that are designed to accommodate such objects, among others.

Symstra [Xie, Marinov, Schulte and Notkin, 2005] uses a technique that linearizes heaps into integer sequences to reduce checking heap isomorphism to just comparing the integer sequence equality. It starts from the root and traverses the heap depth first. It assigns a unique identifier to each object, keeps this mapping in memory and reuses it for objects that appear in cycles. It extends the previously mentioned approaches [Iosif, 2004; Musuvathi and Dill, 2005] in that it also assigns a unique identifier to each symbolic variable, keeps this mapping in memory and reuses it for variables that appear several times in the heap.

In [Darga and Boyapati, 2006] a glass box approach of analyzing data structures is presented. The reductions described therein involve isomorphism-based reductions, but encoding the task requires manual attribution of the data structures to be analyzed. The approach does not present a general way how to handle object-oriented programs containing abstract data types.

Spec Explorer [Veanes, Campbell, Grieskamp, Nachmanson, Schulte and Tillmann, 2005; SpecExplorer, 2006] is a tool for the analysis of model programs written in AsmL and Spec#. It is possible in some cases to specify symmetry reductions in

Spec Explorer using state groupings but the tool does not have a built-in isomorphic state checking mechanisms.

Graph isomorphism is a topic that has received scientific attention for decades. Ullmann's (sub)graph isomorphism algorithm [Ullmann, 1976] is a well known backtracking algorithm which combines a forward looking technique. As the algorithm is relatively straightforward to implement, we used it as an oracle for testing purposes.

The algorithm described in Section 3.4 builds on another well known approach also known as the *Nauty* algorithm, which uses node labelings and partitioning based on such labelings [McKay, 1981].

It is known that there exist certain classes of graphs for which there is a polynomial time algorithm for deciding graph isomorphism. In [Luks, 1982] a method for deciding isomorphism of graphs with bounded valence in polynomial time is presented. The reason why such algorithms are not directly usable in practice is that the polynomial complexity result contains large constants [Fortin, 1996].

There are model checkers, such as for example *Mur*ϕ [Dill, 1996] and *Symmetric Spin* [Bosnacki, Dams and Holenderski, 2000], that allow modeling using scalar sets [Ip and Dill, 1996]. These sets are similar to the sets described in the current thesis but they do not have support for abstract object IDs. A survey of symmetry reductions in temporal logic model checking is given in [Miller, Donaldson and Calder, 2006].

In June 2007 Alastair Donaldson defended his PhD thesis [Donaldson, 2007] where he analyses different methods for utilising symmetries in explicit state model checking. His method is complementary to ours as in addition to utilising scalar sets he exploits the topology of communication channels between the processes.

A good overview of the work on symmetries in Petri nets can be found in [Junttila, 2003].

GROOVE

Quite recently we were referred to the work done in the GROOVE project at the University of Twente [GROOVE, 2007]. In the approach taken there the state spaces of object-oriented programs are generated using graph transformation systems. Such transformations are subjected to CTL model checking [Kastenberg and Rensink, 2006]. In [Rensink, 2006] it is described how the state graph isomorphism based symmetry reduction works in GROOVE. In [Rensink, Schmidt and Varró, 2004] there are some case studies of applying the GROOVE approach among other examples to the dining philosophers problems, as is done in the current thesis. It should be said that from the point of view of model checking object oriented programs, the approach in GROOVE is more mature and the implementation yields better results than our approach. On the other hand, our approach has valuable application in model-based testing and is unique in the sense that it creates the state graphs of the model programs automatically.

2.1.4 State Compression and Hash Compaction

One way to reduce the amount of required memory is to run a generic compression function on the state vector before storing it. Compressing the state this method consumes more cpu resources but enables to store more states in the available amount of RAM.

Hash compaction is a method proposed to be used in verification by Pierre Wolper [Holzmann, 2003]. In hash compaction each state is hashed to some value, for example 64 or 128 bits, and this hash value is stored instead of the full state. The results in [Holzmann, 1998] show that bitstate hashing generally performs better than hash compaction.

2.2 Bitstate Hashing

Model checking in general involves searching possibly very large state spaces for proving or disproving a query — a formula typically in some temporal logic. Bitstate hashing [Holzmann, 1998], also known as supertrace, is a well known method in explicit state model checking for reducing memory requirements for storing the traversed state space by storing only a single bit for each seen state at the address calculated by a hash function. The drawback of the method is the possibility of hash collisions that will result in unexplored parts of the state space, rendering the method to be sound but incomplete. The general significance of reachability checks has been outlined in [Aceto, Bouyer, Burgueño and Larsen, 2003]. Even if it is not possible to prove unreachability, fast reachability checks on formal models that yield a valid trace have applications in, for example, some types of planning and scheduling [Hune, Larsen and Pettersson, 2001; Wijs, van de Pol and Bortnik, 2005; Ruys, 2003], test generation [Hamon, de Moura and Rushby, 2004b; Ernits et al., 2006], software/hardware synthesis [Ernits, 2005], and in debugging [Mercer and Jones, 2005]. In general, the bigger the hash table, the lower the probability of hash collisions. But big bitstate hash tables may still require unavailable amounts of memory.

Bloom filters were introduced by Burton Bloom in [Bloom, 1970] and provide an efficient way of lowering the probability of collisions of the bitstate hashing method [Holzmann, 1998] by storing more than one bit per state in the given hash table. The methods reduces the probability of collisions when the hash table is relatively empty, but collision probabilities become larger than for bitstate hashing when the table becomes more populated. An in-depth probabilistic analysis of bitstate hashing is given in [Kuntz and Lampka, 2004].

2.3 Symbolic Methods

2.3.1 Difference-Bounded Matrices

Difference-bounded matrices provide an efficient region-based symbolic representation that can be used to abstract time. Difference bounded matrice representation is

in fact a weighted directed graph where the vertices correspond to clocks (including zero clock) and the weights on the edges stand for the bounds on the differences between pairs of clocks [Larsen, Larsson, Pettersson and Yi, 2003]. As it gives an explicit bound for the difference between each pair of clocks, its space-usage is in the order of $O(n^2)$. where n is the number of clocks. However, in practice it often turns out that most of these bounds are redundant. Uppaal uses a minimal and canonical representation of DBM-s, which allows efficient inclusion checks. The bitstate pruning based iterated search refinement is orthogonal to this approach and can also be used for models with time.

2.3.2 Binary Decision Diagrams

Binary decision diagrams (BDDs) are the core of a powerful symbolic way of representing Boolean functions and have since their introduction in [Bryant, 1986] been successfully used in a number of applications, particularly in hardware verification by using the graph based symbolic representation of Boolean circuits. The application of BDDs for model checking was described by Kenneth McMillan in his PhD thesis [McMillan, 1992]. We refer the reader to an extensive overview of symbolic model checking in [Edmund M. Clarke et al., 1999].

2.4 Model-Based Testing

Model-based testing is an approach where the task is to establish the conformance relationship between the formalised requirements (the model) and the implementation which is typically a black box with specified interfaces. In model-based testing the model is the specification that is verified against the implementation, thus the model has different role than in model checking. Establishing the conformance relation between the implementation and the model involves exploring the model in full. This is the aspect where model checking and model-based testing meet in the context of the current thesis. The principles of model-based testing are explained in the following recent books: [Utting and Legeard, 2006; Jacky, Veanes, Campbell and Schulte, 2007].

2.5 Modelling Languages

There is a great number of various modelling languages around that are used to specify and model software. We will have a look at three different modelling formalisms which facilitate simulation and model checking of the models.

The model checker Spin and the modelling language Promela used in Spin [Holzmann, 2003] are relevant as Spin is the best-known explicit state software model checker and there is a variety of examples and case studies readily available. Most notably, recently, a database of benchmark models called *BE*nchmarks for *E*xplicit

*M*odel checkers [Pelánek, 2007] was set up with a number of interesting Promela models readily available.

The model checker Uppaal [Amnell et al., 2001] and its guided counterpart Uppaal-Cora [Behrmann, 2005] are used to compare the influence of guiding to the iterated search refinement that will be presented in Chapter 4. Although it has been described how to perform guiding in Promela models [Ruys, 2003], we felt that the built in guiding support of Uppaal Cora provided greater flexibility. In addition, Uppaal offers a nice graphical way of modelling and simulating automata in its graphical user interface.

NModel [NModel, 2007] is a modelling library developed at Microsoft Research, Redmond, that incorporates extensive experience of specification, modelling and model-based testing of reactive software. It is an approach that builds on the experience with SpecExplorer
[SpecExplorer, 2006]. The library accompanies a forthcoming book [Jacky et al., 2007] about model-based testing. One of the key ideas in NModel is that modelling should happen in a known and supported programming language. The motivation is that it is hard to maintain a compiler/interpreter of a custom language and it takes an extra effort of the users to learn to use it. Thus, NModel provides an interesting approach, where learning to model is learning to use a library API. Another difference is that NModel facilitates the use of *object instances* and *abstract data structures*, like sets. This is interesting from several perspectives, but we will have a closer look at a new way of calculating symmetries between states.

The key to the modelling languages in the current context is that they need to be able to provide a reasonable level of abstraction for describing and modelling the desired features and also be analysable.

To illustrate the modelling formalisms at work we use the *dining philosophers* example introduced by Edsger Wybe Dijkstra [Dijkstra, 1971] that has been used for more than 30 years to illustrate concurrency related problems. The description of the problem can be paraphrased in the following way: The life of a philosopher consists of an alternation of thinking and eating. Five philosophers live in a house where the table is laid for them and each philosopher has his own place at the table. The philosophers are served a very difficult kind of spaghetti, so that it has to be eaten with two forks. When a philosopher gets hungry, he sits to the table, picks up the left fork if it is free, then picks up the right fork or waits if it is unavailable, then eats, then puts down the left fork and then, finally ,the right fork, and goes on to thinking again. Such behaviour is modelled with an automaton containing four states: *thinking*, *waiting*, *eating*, and *finishing* and transitions representing the previously described behaviour.

2.5.1 Promela

Promela stands for *Pro*cess *me*ta *la*nguage and is intended to make it easier to find good abstractions of systems designs [Holzmann, 2003]. It provides a vehicle for making abstractions of protocols (or distributed systems in general) that suppress details that are unrelated to process interaction. The intended use of Spin is to verify

fractions of process behaviour, that for one reason or another are considered suspect. The relevant behaviour is modeled in Promela and verified.

Promela programs consist of processes, message channels, and variables. Processes are global objects. Message channels and variables can be declared either globally or locally within a process. Processes specify behavior, channels and global variables define the environment in which the processes run. Unless statements are grouped together by `d_step` or `atomic` keywords, all statements may be interleaved with enabled statements from other processes. The next statement to be executed is chosen nondeterministically from the set of enabled statements. Boolean statements like `fork[0]==0` are blocking statements, they can be traversed when they evaluate to **true**.

```
#define NrOfPhils 3

bit fork[NrOfPhils];

init {
  byte frk;
  atomic {
    frk = 1;
    do
    ::  frk <= NrOfPhils ->
        run philosopher(frk-1, frk%NrOfPhils);
        frk++
    ::  frk > NrOfPhils ->
        break
    od
  }
}

proctype philosopher(byte left, right) {
  think:  if
  ::  d_step {fork[left]==0;fork[left] = 1;}  goto wait;
  fi;
  wait:  if
  ::  d_step {fork[right]==0;fork[right] = 1;}  goto eat;
  fi;
  eat:  if
  ::  fork[left] = 0; goto finish;
  fi;
  finish:  if
  ::  fork[right] = 0; goto think;
  fi;
}
```

Figure 2.2: Promela model of three dining philosophers.

In Figure 2.2 there is an example of three dining philosophers defined in Promela.

The example is a parameterised version of a model taken from the BEEM database [Pelánek, 2007] that provides a selection of benchmark models for explicit state model checkers. The model consists of a bit vector `fork`, where set bits denote which forks are in use, an initialisation proces `init`, and a process template `philosopher`. The given number, `NrOfPhils`, processes are instantiated in an `atomic` block in the `init` process as we are in this case not interested in different interleavings of the creations of the processes.

Please refer to [Holzmann, 2003; Ruys, 2001] for further details about Promela.

2.5.2 Uppaal

Uppaal automata [Behrmann, David and Larsen, 2004] are primarily meant for modelling and verification of real-time systems. Uppaal automata are finite state automata with CCS style synchronisation channels, global and local variables that can be either clocks, booleans, integers, arrays or structs defined in C-like syntax.

Apart from time which is represented internally using difference bounded matrices in Uppaal, it is an explicit state model checker. In addition, Uppaal has a guided counterpart Uppaal-Cora which will be used later in the thesis for analysing the influence of guiding to the proposed iterated search refinement method.

An example how to model the dining philosophers problem in Uppaal is given in Figure 2.3. An instance of the `Phil` automaton is created for each philosopher in the process template instantiation section. When a transition leading from a location of the automaton to the next is enabled the automaton can take a step. If there are multiple transitions enabled from the current location, Uppaal chooses one transition nondeterministically. The transition from one state to another can have two effects: either the transition may be synchronised with some other transitions over synchronisation channels (not present in the current model) or it may trigger a code block that causes the states of the context variables to change. Assignment is a simple instance of such code. `forks[left]=UP` is an example of such assignment.

2.5.3 Model Programs

A third alternative to modelling reactive systems is by using model programs [Veanes, Campbell, Grieskamp, Nachmanson, Schulte and Tillmann, 2005]. The model programs in our case are written in C# and utilise a .NET-based toolkit called NModel [NModel, 2007]. It is a modelling library that accompanies the book: [Jacky et al., 2007]. Models written in an industry standard programming language have the benefit that at least part of the tool, C# language and compiler, will be supported as long as the language and the compiler. The general nature of the language can also be a drawback, as there are various different ways of doing things.

Modelling using model programs is an interesting alternative in modelling compared to other modelling formalisms, as users can learn to model without learning a new syntax. They still have to learn how to use the library and tools of the toolkit.

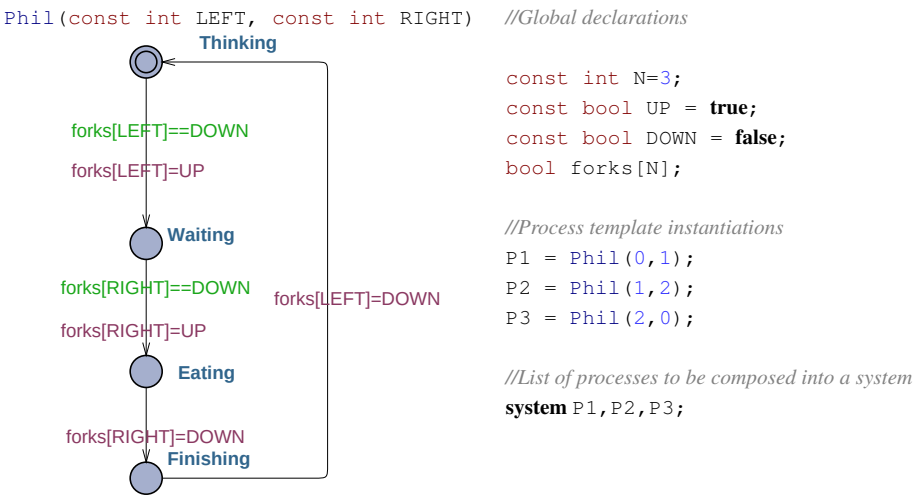


Figure 2.3: Uppaal model of dining philosophers.

Model programs are a useful formalism for software modelling and design analysis and are used as the foundation of industrial tools such as Spec Explorer [Veanes, Campbell, Grieskamp, Nachmanson, Schulte and Tillmann, 2005]. The expressive power of model programs is due largely to two characteristics. First, one can use complex data structures, such as sequences, sets, maps and bags (multisets), which is sometimes referred to as having a *rich background universe*. Second, one can use instances of classes or elements from user-defined abstract types; we use the word *object* to mean either case.

It is possible to can characterize a typical usage scenario of model programs as a three step process [Jacky et al., 2007]: *describe*, *analyze* and *test*.

Describe: A *contract model program* is written to capture the intended behaviour of a system or subsystem under consideration. Complex data structures and abstract elements are utilised to produce a contract, or trace oracle, at the desired level of abstraction.

Analyze: Zero or more *scenario model programs* are written to restrict the contract to relevant or interesting cases. The scenarios are composed with the contract and the resulting model program is *explored* to validate the contract. The possible traces of a composition of model programs is the intersection of possible traces of the constituent model programs.

Test: The model program, that is, the contract possibly composed with additional scenarios, is used to generate test cases or used as a test oracle.

In Figure 2.4 there are the class definitions of objects used in the model program of the dining philosophers. Philosopher has fields referencing the left and the right

```
using System;
using NModel;
using NModel.Terms;
using NModel.Attributes;

namespace DiningPhilosophers
{
    public enum State { Thinking, Waiting, Eating, Finishing }

    [Sort("ControlMode")]
    public enum Mode { Initializing, Running }

    public class Philosopher :  LabeledInstance<Philosopher>
    {
        public State state;
        public Fork left, right;
        public override void Initialize() { state = State.Thinking; }
        public bool leftFree() { return left.isFree(); }
        public bool rightFree() { return right.isFree(); }
    }

    public class Fork :  LabeledInstance<Fork>
    {
        public Philosopher hasMe;
        public override void Initialize() { hasMe = default(Philosopher); }
        public bool isFree() { return hasMe == default(Philosopher); }
        public void take(Philosopher p) { hasMe = p; }
        public void release(Philosopher p) { hasMe = default(Philosopher); }
    }

}
```

Figure 2.4: Model program modelling the dining philosophers problem: definition of the classes.

```
namespace DiningPhilosophers {
  public static class Contract {
    public static Set<Philosopher> phils = Set<Philosopher>.EmptySet;
    public static Mode mode = Mode.Initializing;
    public const int numberOfPhils = 3;
    [Action]
    public static void Init()
    { Philosopher[] tmpP = new Philosopher[numberOfPhils];
      Fork[] tmpF = new Fork[numberOfPhils];
      for (int i = 0; i < numberOfPhils; i++) tmpF[i] = Fork.Create();
      for (int i = 0; i < numberOfPhils; i++)
      { Philosopher p = Philosopher.Create();
        p.left = tmpF[i];
        p.right = tmpF[(i + 1) % numberOfPhils];
        tmpP[i] = p;
      }
      for (int i = 0; i < numberOfPhils; i++) phils = phils.Add(tmpP[i]);
      mode = Mode.Running;
    }
    public static bool InitEnabled() { return (mode == Mode.Initializing); }
    [Action]
    public static void TakeLeft([Domain("phils")] Philosopher p)
    { p.left.take(p);
      p.state = State.Waiting;
    }
    public static bool TakeLeftEnabled(Philosopher p)
    { return mode==Mode.Running && p.leftFree() && p.state==State.Thinking; }
    [Action]
    public static void TakeRight([Domain("phils")] Philosopher p)
    { p.right.take(p);
      p.state = State.Eating;
    }
    public static bool TakeRightEnabled(Philosopher p)
    { return mode==Mode.Running && p.rightFree() && p.state==State.Waiting; }
    [Action]
    public static void ReleaseLeft([Domain("phils")] Philosopher p)
    { p.left.release(p);
      p.state = State.Finishing;
    }
    public static bool ReleaseLeftEnabled(Philosopher p)
    { return mode == Mode.Running && p.state == State.Eating; }
    [Action]
    public static void ReleaseRight([Domain("phils")] Philosopher p)
    { p.right.release(p);
      p.state = State.Thinking;
    }
    public static bool ReleaseRightEnabled(Philosopher p)
    { return mode == Mode.Running && p.state == State.Finishing; }
  }
}
```

Figure 2.5: Model program of the dining philosophers problem: definition of the behaviour.

```
public static ModelProgram Create()
{
    return new LibraryModelProgram(typeof(Contract).Assembly,"DiningPhilosophers");
}
```

Figure 2.6: Model program of the dining philosophers problem: factory method of the class Contract.

fork and Fork has a reference to the philosopher instance that currently possesses it or `null` otherwise.

In Figure 2.5 there is the contract representing the behaviour of the dining philosophers system. The contract describes all possible actions that can take place in the system. Actions are attributed with the `[Action]` attribute. The methods with the name of an action and an ending containing the word "Enabled" are action guards.

Both the `Philosopher` and `Fork` class inherit from the `LabeledInstance` class. The latter is a class of the modelling library that provides required mechanisms for representing objects of the model at run-time. Please refer to [Jacky et al., 2007; Veanes, Campbell and Schulte, 2007a; Veanes, Ernits and Campbell, 2007] for further details about model programs.

Non-determinism is available on the action argument level, i.e. when a model program is analysed, all permutations of action arguments are tried. For example, the domain of the argument of the action `TakeLeft` is the set containing all philosophers. This means that the `TakeLeft` action will be attempted with every instance of the philosophers in the set. Sometimes the action guard might evaluate to false, as in the case where the philosopher is already holding the left fork.

One thing to bear in mind with model programs is that the data structures are immutable, meaning that changes to a data structure have effect only when the variable denoting the data structure is assigned the modified instance. More specifically, all changes to data structures need to be in the form `Set<Element> myset = myset.Add(element);` and not just `Set<Element> myset.Add(element);`.

Another important thing to keep in mind is that it is possible to define models with potentially infinite state space, for example in the case of adding fresh objects to a set. It is possible to enforce the model programs to have a finite state space by introducing state filters that limit the number of elements in certain data structures.

To make it possible to instantiate the model, there has to be a factory method that invoked by the model analysis tools to create an instance of the model program. For the dining philosophers example, the class `Contract` should additionally have a factory method like the one given in Figure 2.6.

CHAPTER 3

SYMMETRY REDUCTIONS

In this chapter[1] we look more closely at model programs that can contain abstract data structures. like sets and maps, and objects. Each state, i.e., the configuration of all variables, can be represented as a graph. The combination of the presence of abstract data structures and objects yields an interesting way to apply state graph isomorphism for symmetry reduction.

3.1 Introduction

Model programs are a useful formalism for software modelling and design analysis and are used as the foundation of industrial tools such as Spec Explorer [Veanes, Campbell, Grieskamp, Nachmanson, Schulte and Tillmann, 2005]. The expressive power of model programs is due largely to two characteristics. First, one can use complex data structures, such as sequences, sets, maps and bags (multisets), which is sometimes referred to as having a *rich background universe*. Second, one can use instances of classes or elements from user-defined abstract types; we use the word *object* to mean either case.

The lack of symmetry checking when program states include both unordered structures and objects is a serious practical concern for users of tools like Spec Explorer. If symmetric states are not pruned, the number of states that must be considered during exploration will often become infeasibly large. Thus one natural way to tackle the *state space explosion* problem is by detecting symmetries. Symmetry reduction is not a universal solution for the state explosion problem but helps to relieve it in many cases. In this chapter we present a symmetry reduction based on state isomorphism for programs that contain both complex data structures and objects.

The expressive power of combining abstract, unordered data types with objects is useful when describing a model but complicates analysis. The core problem is to efficiently identify "relevant" states during exploration. By a state we mean a collection of all state variables and their values at a given point along the exploration path. It is often the case that two states that are isomorphic should be treated as

[1] The work presented in this chapter was done during an internship at Microsoft Research, Redmond, WA, USA.

being equivalent. Isomorphism between states with a rich background universe is well defined. It exists when there is a one-to-one mapping of objects (within each abstract type) that induces a structure-preserving mapping between the states [Blass and Gurevich, 2000].[2] Informally, two states are isomorphic if they differ in choice of object IDs (or elements of the reserve) but are otherwise structurally identical.

Consider for example a state signature containing two state variables V and E. (States as in model programs are introduced in the next section.) The type of V is a set of vertices (distinct values of an abstract type *Vertex*) and the type of E is a set of vertex sets. Let v_1, v_2, v_3, v_4 be vertices and let S_1 be a state where,

$$\begin{aligned} V &= \{v_1, v_2, v_3, v_4\}, \\ E_1 &= \{\{v_1, v_2\}, \{v_2, v_3\}, \{v_3, v_4\}, \{v_4, v_1\}\}. \end{aligned}$$

Intuitively, the state S_1 is an undirected graph that is a rectangle with four vertices. Let S_2 be a state where V has the same value as in S_1 and,

$$E_2 = \{\{v_1, v_3\}, \{v_3, v_2\}, \{v_2, v_4\}, \{v_4, v_1\}\}.$$

States S_1 and S_2 are isomorphic because structure is preserved if the reserve element v_2 is swapped with v_3. This is an isomorphism that maps v_3 to v_2, v_2 to v_3 and every other vertex to itself. Let S_3 be a state where V has the same value as in S_1 and,

$$E_3 = \{\{v_1, v_2\}, \{v_2, v_3\}, \{v_3, v_1\}, \{v_4, v_1\}\}.$$

State S_3 is not isomorphic to S_1, because all vertices in S_1 are connected to two vertices but v_4 is only connected to one vertex in S_3, i.e., there exists no structure-preserving mapping from S_1 to S_3.

The example illustrates the point that state isomorphism is as hard as graph isomorphism, when objects and unordered data structures are combined. A customer survey of Spec Explorer users within Microsoft has shown that this combination occurs often in practice. It occurs in the standard Spec Explorer example included in the distribution [SpecExplorer, 2006] known as the chat model [Veanes, Campbell, Grieskamp, Nachmanson, Schulte and Tillmann, 2005; Utting and Legeard, 2006], where chat clients are objects and the state has a state variable that maps receiving clients to sets of sending clients with pending messages. The state isomorphism problem for reserve elements in unordered structures was not solved in Spec Explorer and to the best of our knowledge has not been addressed in other tools used for model based testing or model checking that support unordered data structures. There are model checkers that support *scalar sets* [Ip and Dill, 1996], which are basically ranges of integers, but we do not know of instances where such sets can contain objects with abstract object IDs.

In practical terms this means that users must either use various pruning techniques that only partially address the problem or extend the model program with custom

[2] In ASM theory, what we call *objects* are called *reserve elements*.

scenario control that tries to work around the problem by restricting the scope of exploration. The results are not always satisfactory.

The pruning techniques that have been partially helpful in this context are state grouping [Grieskamp, Gurevich, Schulte and Veanes, 2002] and multiple state grouping [Campbell and Veanes, 2005], [Veanes, Campbell, Grieskamp, Nachmanson, Schulte and Tillmann, 2005]. But the grouping techniques have an orthogonal usage that is similar to abstraction in model checking, whereas state isomorphism is a generalization of symmetry checking in model checking. In general, it is not possible to write a grouping expression that maps two states into the same value if and only if the states are isomorphic; the "only if" part is the problem.

If objects are not used, then state isomorphism reduces to state equality. State equality can be checked in linear time. This is possible because the internal representation of all (unordered) data structures can then be ordered in a canonical way. The same argument is true if objects are used but no unordered data structures are present. Then state isomorphism reduces to what is called heap canonicalization in the context of model checking and can also be implemented in linear time [Iosif, 2004; Musuvathi and Dill, 2005].

In this chapter we describe a solution for the state isomorphism problem for model programs with states that have both unordered structures and objects. We do so by providing a mapping from model program states to rooted labeled directed graphs and use a graph isomorphism algorithm to solve the state isomorphism problem. The graph construction and the labelling scheme use techniques from graph partitioning algorithms and strong hashing algorithms to reduce the need to check isomorphism for states that are known not to be isomorphic. We also outline a graph isomorphism algorithm that is customized to the particularities of state graphs. Our algorithm extends a linearisation based symmetry-checking algorithm with backtracking and is, arguably, better suited for this application than existing graph isomorphism algorithms.

Before we continue with the detailing how our symmetry reduction works, we illustrate why state isomorphism checking is useful on a small example, shown in Figure 3.1, that we use also in the later sections. The example is small but typical for similar situations that arise for example in the chat model [Utting and Legeard, 2006] or when modeling multithreaded applications where threads are treated as objects [Veanes, Campbell, Schulte and Tillmann, 2005]. The example is written in C# and uses a modeling library and a toolkit called *NModel*. The formal definition of a model program is given in Section 3.2, where it is also explained how the C# code maps to a model program. NModel is available with source code [NModel, 2007] and supports the text book [Jacky et al., 2007] that discusses the use of model programs as a practical modelling technique. All algorithms described in this chapter have been implemented in NModel.

```
namespace Triangle
{
  [Abstract]
  enum Side { S1, S2, S3 }

  [Abstract]
  enum Color { RED, BLUE }

  static class Contract
  {
    static Map<Side, Color> colorAssignments = Map<Side, Color>.EmptyMap;

    static bool AssignColorEnabled(Side s)
    {  return !colorAssignments.ContainsKey(s); }

    [Action]
    static void AssignColor(Side s, Color c)
    {  colorAssignments = colorAssignments.Add(s, c); }
  }
}
```

Figure 3.1: A model program where a color, either `RED` or `BLUE`, is assigned to the sides of a triangle.

3.1.1 Example

Let us look at a simple model program that describes ways to assign colors to the sides of a triangle. The model program is given in Figure 3.1. The triangle in the program has three sides, `S1`, `S2`, and `S3` and each side can be associated with the color `RED` or `BLUE`. The model program has a single action that assigns a color to one side at a time. There are $(|\texttt{Color}|+1)^{|\texttt{Side}|} = 27$ possible combinations of such assignments, including intermediate steps where some sides have not been colored yet. There are three sides; each side has three possible values if you count "no color" as a value.

The state transition graph visualising all possible transitions and all distinct states of the triangle program is given in Figure 3.2. The numbers of the nodes denote exploration sequence. In this case the `[Abstract]` attributes of `Side` and `Color` have not been taken into account and each combination of `Side` $\mapsto$ `Color` is considered distinct.

3.2 Definitions

A formal treatment of model programs builds on the ASM theory [Gurevich, 1995] and can for example be found in [Veanes, Campbell and Schulte, 2007b]. Here we provide some basic terminology and intuition and illustrate the main concepts with examples. A *state* here is a full first-order state, that is intuitively a mapping from a

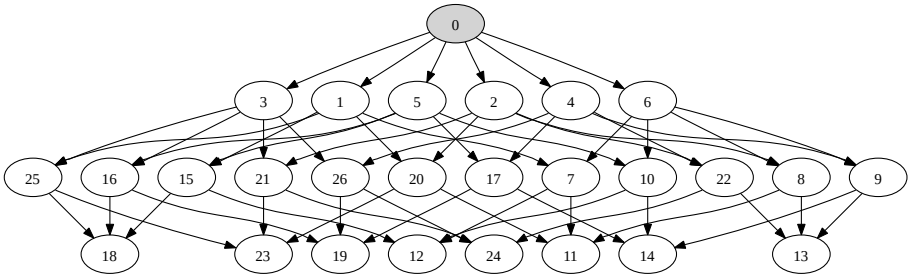


Figure 3.2: The transition system representing the shape of the full state space of the Triangle example in Figure 3.1. Each combination of `Side` $\mapsto$ `Color` is considered distinct and thus the blowup of the state space.

fixed set of *state variables* to a fixed universe of *values*. States also have a rich *background* [Blass and Gurevich, 2000] that contains sequences, sets, maps, sets of sets, maps of sets, etc. We assume here that all state variables are nullary.[3] For example, the model program in Figure 3.1 has one state variable `colorAssignments`.

Since states have a rich background universe they are infinite. However, for representation, we are only interested in the *foreground* part of a state that is the interpretation of the state variables. All values have a *term* representation. Terms that do not include state variables are called *value terms* and are defined inductively over a signature of function symbols. This signature includes constructors for the background elements.[4] We identify a state with a conjunction of equalities of the form $x = t$, where x is a state variable and t a value term.

The interpretation of a value term is the same in all states. Value terms are not unique representations of the corresponding values, i.e., value terms that are syntactically distinct may have the same interpretation. We say value for a value term when it is clear from the context that the particular term representation is irrelevant.

For example, a set of integers, containing the values `1`, `2`, and `3` is represented by the term `Set<int>(1,2,3)`. The term `Set<int>(2,1,3)` has the same interpretation. We use a relaxed notation where the arity of function symbols is omitted but is implicitly part of the symbol. For example `Set<int>(1,2)` represents a set containing 1 and 2, so the constructor `Set<int>` is binary here and ternary in the previous case. Function symbols are typed. For example, a set containing two sets of strings `Set<string>("a")` and `Set<string>("b")` is represented by the term

```
Set<Set<string>>(Set<string>("a"),Set<string>("b"))
```

Model programs typically also have user-defined types that are part of the background. For example the model program in Figure 3.1 has the user-defined type

[3] In ASM theory, state variables are called *dynamic functions* and may have arbitrary arities. Dynamic functions with positive arities can be encoded as state variables whose values are maps.

[4] In ASM theory, these function symbols are called *static*, their interpretation is the same in all states.

`Color`. This type has two elements `Color.RED` and `Color.BLUE`, respectively. The initial state of this model program is (represented by the equality)

```
colorAssignments = Map<Side,Color>.EmptyMap.
```

A user-defined type may be annotated as being *abstract*, e.g., `Color` in Figure 3.1 is abstract. Elements of an abstract type are treated as typed *reserve* elements in the sense of [Gurevich, 1995]. Intuitively this means that they are interchangeable elements so that a particular choice must not affect the behavior of the model program. A valid model program must not explicitly reference any elements of an abstract type. For example, even though the `Color` enumeration type provides an operation to return the string name of a color value, the model program must not use that operation if color is to be considered abstract. Abstract types are similar to objects[5] that are treated the same way.

An *update rule* is a collection of (possibly conditional) assignments to state variables. An update rule p that has formal input parameters $\bar{x}$ is denoted by $p[\bar{x}]$. The instantiation of $p[\bar{x}]$ with concrete input values $\bar{v}$ of appropriate type, is denoted by $p[\bar{v}]$. An update rule p denotes a function $[\![p]\!] : States \times Values^n \rightarrow States$

A *guard* φ is a state dependent formula that may contain free logic variables $\bar{x} = x_1, \ldots, x_n$, denoted by $\varphi[\bar{x}]$; φ is *closed* if it contains no free variables. Given values $\bar{v} = v_1 \ldots, v_n$ we write $\varphi[\bar{v}]$ for the replacement of x_i in φ by v_i for $1 \leq i \leq n$. A closed formula φ has the standard truth interpretation $s \models \varphi$ in a state s. A *guarded update rule* is a pair (φ, p) containing a guard $\varphi[\bar{x}]$ and an update rule $p[\bar{x}]$; intuitively (φ, p) limits the execution of p to those states and arguments $\bar{v}$ where $\varphi[\bar{v}]$ holds.

We use a simplified definition a model program here, by omitting control modes. The state isomorphism problem is independent of the presence of explicit control modes. Thus, this simplification does not affect the main topic of this chapter.

A *model program P* has the following components:

- A finite vocabulary $X_<$ of *state variables*
- A finite vocabulary Σ of *action symbols*
- An *initial state* s_0 given by a conjunction $\bigwedge_{x \in X} x = t_x$ where t_x is a value term.
- A *reset* action symbol $Reset \in \Sigma$.
- A family $(\varphi_f, p_f)_{f \in \Sigma}$ of guarded update rules.
  - The *arity* of f is the number of input parameters of p_f.

[5] By "objects" we mean object IDs. Instance fields associated with objects are considered to be state variables in their own right and not part of any nested structure. In this way, we can consider only global variables without loss of generality.

– The arity of *Reset* is 0 and $[\![p_{Reset}]\!](s) = s_0$ for all $s \models \varphi_{Reset}$.

An *action* has the form $f(v_1, \ldots, v_n)$ where f is an n-ary action symbol and each v_i is a value term that matches the required type of the corresponding input parameter of p_f. We say that an action $f(\vec{v})$ is *enabled* in a state s if $s \models \varphi_f[\vec{v}]$. An action $f(\vec{v})$ that is enabled in a state s can be *executed* or *invoked* in s and yields the state $[\![p_f]\!](s, \vec{v})$.

The model program in Figure 3.1 has a single action symbol `AssignColor`. The guard of `AssignColor` is given by the Boolean function `AssignColorEnabled`. The action $a =$ `AssignColor(Side.S1, Color.RED)` is enabled in the initial state s_0 because `AssignColorEnabled(Side.S1)` returns true in s_0. The execution of a in s_0 yields the state

```
colorAssignments = Map<Side,Color>(Side.S1 ↦ Color.RED).
```

The unwinding of a model program from its initial state gives rise to a labeled transition system (LTS). The LTS has the states generated by the unwinding of the model program as its states and the actions as its labels.

A *rooted directed labeled graph*, G, is a graph that has a fixed root, has directed edges, and contains labels of vertices and edges. Such graph can be formally represented as a triple $G = (v_r, V, E)$ where $v_r \in V$ is the root vertex, V is a set of vertices v that are pairs $v = (id, l_v)$, where id is an identifier uniquely determining a vertex in a graph and l_v is the label of the vertex. E is the set of triples $(v_1.id, l_e, v_2.id)$ where v_1 is the start vertex, v_2 is the end vertex and l_e is the edge label.

3.3 States as Graphs

In this section we present a graph representation of the state of a model program.

The states of a model program can contain object instances and other complex data structures, thus we do not deal only with primitive types, such as integers and Boolean values, but also with instances of objects that can be dynamically instantiated and refer to other instances of objects. The state space of a model program may be infinite, but concrete states are finite first order structures. We look at the configuration of values and object instances that have been assigned to the fields of objects and data structures contained in the program. We do not consider the the program stack to be part of the program state as states are only compared between performing actions.

A state is defined by an assignment of term representations of values to fields, $s = \bigwedge_{x \in X_<} x = t_x$. In the triangle example there is only one state variable and a state of the model program is represented as `colorAssignments = Map<Side,Color>(Side("S1"),Color("BLUE"),Side("S2"),Color("RED"))`. There are two kinds of fields in a model program: global fields, like `colorAssignments` in the

program on Figure 3.1 and fields of dynamically instantiated objects.[6] For the sake of clarity we will here look at states containing global fields and intoduce dynamical instantiations of objects later. Assignments to global fields are simple equations $x = t$. It is important to note that it is possible to establish a binary relation of total ordering, $<$, of field names. This can be achieved by, for example, ordering the field names alphabetically.

Algorithm 1 outlines the procedure of creating a graph from a term representation of a state. In general the procedure is straightforward: the function `CreateGraph` creates the graph by analysing the terms corresponding to each state variable x. The analysis of a term, `TermToGraph`, adds a mapping of the field identifier to a value to the label of the parent node, if t denotes a value and adds a new node to the graph, if t is an object. A specialized procedure is used for creating nodes corresponding to built-in abstract data types[7]. In fact, each ADT is handled in a slightly different way.

A `Set` becomes a node that has the count of its elements in the label of the incoming edge. All outgoing edges of a set are given a label "$\in$", denoting membership in a set. It is possible that the label is extended with more arguments as the set may contain other sets.

The representation of a bag (or multiset) has a sorted list of element multiplicities on the incoming label. The label of the edge pointing to each element of a `Bag` is labelled by the corresponding multiplicity. In fact, a `Bag` is a set of pairs, and using a specialized representation is an optimization that helps to reduce the number of nodes in the state graph. A `Map` is also a `Set` of pairs but can be converted to a reduced fragment of the graph.

The labellings of outgoing edges may be unique, as in the case of different field indices of a structure or with objects in a sequence, or unordered, as in the case of a set.

Thus, it is possible to classify the outgoing edges of a node into *ordered* and *unordered* edges. The graph representations of the abstract data structures `Set`, `Bag`, and `Map` are summarized in Figure 3.3.

In fact, for the purposes of mathematical reasoning about the state graphs of model programs, it suffices to consider only two types of nodes: the *Set* and *Pair* type. Instances of the *Set* type can contain other sets and pairs and all of the edges from *Set* to instances constituting its contents are unordered and labelled with "$\in$". Instances of the *Pair* type, on the other hand, have ordered outgoing edges, labelled $first$ and *second* that can point to sets and pairs or one to a set and the other one to a pair. Any object can be represented in terms of a chain of pairs as the fields of the object are ordered. A *Map* is a *Set* of *key* and *value* pairs. A *Bag* is a set of pairs where the $first$ is the element and *second* is the count. A simplified version of the *TermToGraph* function is given in Figure 3.4:

[6] An instance field can be represented as a global field whose value is a map of objects to their corresponding field values. In other words, objects don't "contain" structure but are distinct identities.

[7] We have omitted the types `Pair` and `Triple` from Algorithm 1 as they are special cases of the type `Sequence`.

Algorithm 1 Pseudocode for generating a rooted labelled directed graph from a state s of a model program.

CREATEGRAPH(s)

$\triangleright$ s – State of the model program in the form $s = \bigwedge_{x \in X_<} x = t_x$

Create an empty rooted labelled directed graph g with root r
for $x \in X$
    $g = TermToGraph(t_x, r, fieldName(x), g)$
return g

TERMTOGRAPH(t, p, fld, g)

$\triangleright$ t – Term
$\triangleright$ p – Current parent vertex in the graph
$\triangleright$ fld – Current field
$\triangleright$ g – Graph of the state

if t is not an object
  then add $fld \mapsto t$ to the label of the parent p
    return g
if $t.functionSymbol == Set$
  then Create a new vertex $setv$ into g and
    add an edge from p to $setv$ with a label
    containing fld and $t.argumentCount$ into g.
    for $targ \in t.arguments$
      $TermToGraph(targ, setv, " \in ", g)$
elseif $t.functionSymbol == Bag$
  then Create a new vertex $bagv$ into g.
    Create a new bag of counts $bagCounts$.
    for `Pair<Term, Term>`$(targ, count) \in t.argumentPairs$
      $TermToGraph(targ, bagv, count, g)$ $bagCounts.Add(count)$
    Add an edge from p to $bagv$ with a label
    containing fld and a sorted list of $bagCounts$ into g.
elseif $t.functionSymbol == Map$
  then for `Pair<Term, Term>`$(key, value) \in t.argumentPairs$
      Create a new vertex $maplet$ into g
      $TermToGraph(key, maplet, "key", g)$
      $TermToGraph(value, maplet, "value", g)$
      Add an edge from p to $maplet$ with a label
      containing fld and $t.argumentPairCount$ into g
elseif $t.functionSymbol == Sequence$
  then Create a new vertex $sequencev$ into g and
    add an edge from p to $sequencev$ with a label
    containing fld and $t.argumentCount$ into g.
    for $targ \in t.arguments$
      $TermToGraph(targ, setv, t.currentArgumentIndex, g)$
  else $\triangleright$ t is not a built-in object type
    if abstract object obj corresp. to $t.functionSymbol$ and $t.firstArgument$, $obj \notin g$
      then add obj into g
    Add an edge fom p to obj with a label containing fld into g.
return g

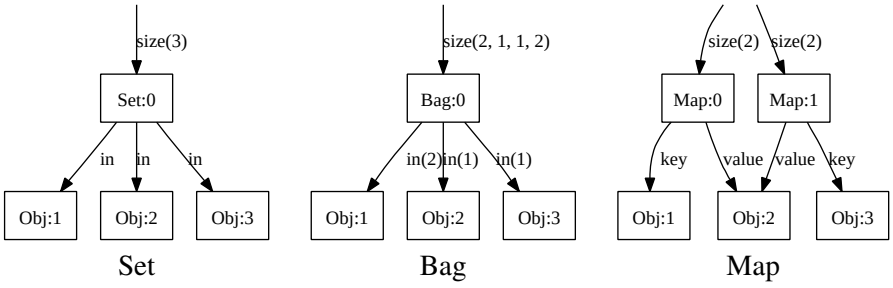


Figure 3.3: Graph representations of abstract data types used by model programs.

TERMTOGRAPH(t, p, fld, g)

$\triangleright$ t – Term
$\triangleright$ p – Current parent vertex in the graph
$\triangleright$ fld – Current field
$\triangleright$ g – Graph of the state

if t is not a `Pair` or `Set`
  then add $fld \mapsto t$ to the label of the parent p
    return g
if $t.functionSymbol == Set$
  then Create a new vertex $setv$ into g and
    add an edge from p to $setv$ with a label
    containing fld and $t.argumentCount$ into g.
    for $targ \in t.arguments$
      $TermToGraph(targ, setv, \text{“} \in \text{”}, g)$
elseif $t.functionSymbol == Pair$
  then Create a new vertex $pairv$ into g and
    add an edge from p to $pairv$ with a label
    containing fld into g.
    $TermToGraph(t.arg1, pairv, \text{“}first\text{”}, g)$
    $TermToGraph(t.arg2, pairv, \text{“}second\text{”}, g)$
return g

Figure 3.4: Pseudocode for *TermToGraph* with only *Set* and *Pair* data types.

There are graphs representing some of the states of the Triangle example in Figure 3.5. The state graphs have been generated using the procedure outlined in Algorithm 1. These state graphs correspond to the states 11, 12, and 14 of the triangle example in Figures 3.2 and 3.6. State 14 is isomorphic to state 12 but neither 12 nor 14 is isomorphic to 11. The abbreviation `cA` stands for `colorAssignments` and `(3)` denotes that there are 3 key-value pairs in the map.

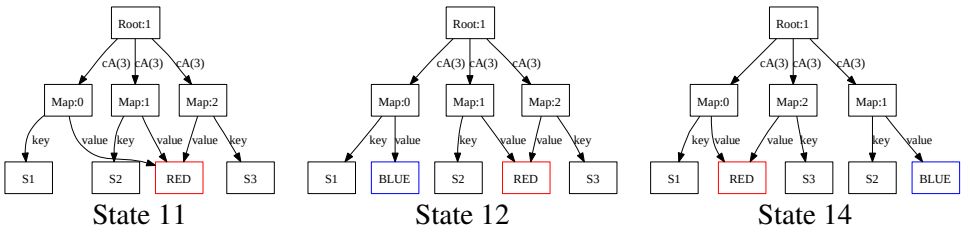


Figure 3.5: State graphs of states 11, 12, and 14 of the triangle example on Fig. 3.2 and Fig. 3.6. State 14 is isomorphic to state 12 but neither 12 nor 14 is isomorphic to 11. The abbreviation `cA` stands for `colorAssignments` and `(3)` denotes that there are 3 key-value pairs in the map.

Figure 3.6 illustrates the effects of isomorphism-based symmetry reduction applied to the triangle example. The state graph on the left shows at which stages of the search isomorphic states were encountered. A dashed arrow points to a previously encountered state that is isomorphic to the state the arrow starts from. The graph on the right is obtained by showing a representative example of a family of isomorphic states.

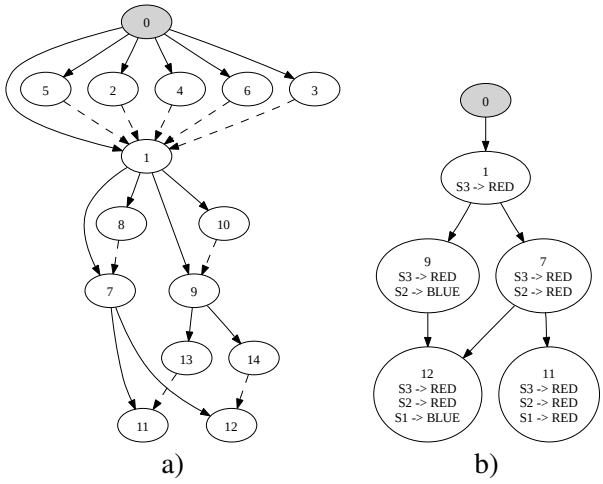


Figure 3.6: State space of the Triangle example from Fig. 3.1, where exploration of isomorphic states has been pruned. The dashed lines on a) exhibit encounters of isomorphic states during exploration, b) exhibits the structure of the state graph when isomorphic states are collapsed.

3.3.1 Field Maps

As mentioned earlier, *objects* are just abstract IDs or reserve elements. So how do we deal with *fields* of objects? Fields of objects are represented by state variables, called *field maps*, whose values are finite maps from objects of the given type to values of the given field type.[8] From the symmetry reduction point of view, field maps are handled in the same way as map-valued state variables. A difference compared to other map-valued state variables is that field maps can not be referenced as values inside a model program, which can be used to simplify the graph representation of a state.

```
namespace Triangle
{
  [Abstract]
  enum Color { RED, BLUE }

  class Side :  LabeledInstance<Side> { public Color color; }

  static class Contract
  {
    static Set<Side> sides = Set<Side>.EmptySet;

    [Action]
    static void AssignColor([New] Side s, Color c)
    {
       s.color = c;
       sides = sides.Add(s);
    }

    static bool AssignColorEnabled(Side s)
    { return sides.Count < 3; }
  }
}
```

Figure 3.7: A version of the triangle model where sides are objects. The `AssignColor` action is enabled if not all sides have been colored. The `New` keyword indicates that the side is a new object (reserve element).

In order to illustrate field maps, consider a version of the triangle example, shown in Figure 3.7, where sides are instances of a class *Side*. The fact that sides are reserve elements is indicated by the base class. This model program has two state variables, `sides` and `color`, where `color` is a field map. In the initial state, both `color` and `sides` are empty. When a color `c` is assigned to a side `s`, the `color` map gets a new entry $\texttt{s} \mapsto \texttt{c}$. For example, the state of the triangle where all sides have been coloured blue has the following representation:

[8] The name of a field map is uniquely determined from the fully qualified name of the class and the name of the field.

```
color = Map(Side(1),Color("BLUE"),Side(2),Color("BLUE"),Side(3),Color("BLUE"))
sides = Set(Side(1),Side(2),Side(3))
```

The presented approach is also extended to states resulting in the composition of model programs, as presented in [Veanes, Campbell and Schulte, 2007b]. The root of a state of a composition of model programs becomes a set of two rooted graphs that may share objects.

Intuitively, field maps represent arrows between objects. Thus, it is possible to rewrite the *CreateGraph* procedure of Algorithm 1 in the following way to produce arrows between objects rather than extra maplets stemming from the root node pointing to instance and the value nodes of the maplets of the field map.

CREATEGRAPH(s)

$\triangleright$ s – State of the model program in the form $s = \bigwedge_{x \in X_<} x = t_x$
$\triangleright$ Create an empty rooted labelled directed graph g with root r
for $x \in X \wedge x$ is not a fieldmap
$g = TermToGraph(t_x, r, fieldName(x), g)$
for $x \in X \wedge x$ is a fieldmap
$g = FieldMapToGraph(t_x, r, fieldName(x), g)$
return g

FIELDMAPTOGRAPH(t, r, n, g)

if t has more than zero arguments
for $(Instance, Value) \in t.ArgumentPairs$
$v = CreateOrGetInstanceNode(Instance, g)$
$g = TermToGraph(Value, v, n, g)$
return g

3.4 Isomorphism Checking

Unlike arbitrary graphs, state graphs are rooted and encode state information in a way that partially reflects the underlying static structure of a program. For example, all objects of a given type have a fixed set of fields that are ordered alphabetically. Several built-in ordered data types, such as sequences and pairs, also have an order of the elements contained in them according to their position. Moreover, user-defined types, other than abstract types, have a fixed alphabetical order of fields. A typical model program uses both ordered and unordered data structures. As explained above, the resulting state graph includes both ordered and unordered edges.

Our intent was to devise an algorithm for graph isomorphism that takes advantage of the ordered edges as much as possible while handling the unordered cases as a last resort through backtracking. The starting point is that all vertices of the graph have been given strong labels through object ID-independent hashing that already reduces the possible pairings of vertices dramatically. In the case when all edges are ordered the algorithm should not do any backtracking at all. The basic idea of the

algorithm is an extension of the linearization algorithm used in Symstra [Xie et al., 2005] with *backtracking*. The algorithm reduces to linearization when the graphs that are being compared are fully ordered, i.e. have no unordered edges. A small difference compared to Symstra is that the linearizations are computed and compared simultaneously for the two graphs as depth first walks, rather than independently and then compared.

3.4.1 Linearization with Backtracking

The following is an abstract description of the algorithm. Given are two state graphs G_1 and G_2. The algorithm either fails to produce an isomorphism or returns an isomorphism from G_1 to G_2. The abstract description of the algorithm is non-deterministic. In the concrete realization of the algorithm the **choose** operation is implemented through backtracking to the previous backtrack point where more choices were possible. The details of the particular backtracking mechanism are omitted here.

We say that an edge with label l is an l-edge. The edge labels that originate from ordered background data structures are called *functional*. It is known that for all functional edge labels l and for all nodes x, there can be at most one outgoing l-edge from x. Other edge labels are called *relational*.

Bucketing: Compute a "bucket map" B_i for all nodes in G_i, for $i = 1,2$. Each node n in G_i with label l is placed in the bucket $B_i(l)$. If either B_1 and B_2 do not have the same labels and the same sizes of corresponding buckets for all labels then **fail**. Otherwise execute **Extend**$(\emptyset, r_1, r_2)$, where r_i is the root of G_i, for $i = 1,2$.

Extend(ρ, x_1, x_2)**:** Given is a partial isomorphism ρ and isomorphism candidates x_1 and x_2. If x_1 and x_2 have distinct labels then **fail**, else if x_1 is already mapped to x_2 in ρ then return ρ, else if either x_1 is in the domain of ρ or x_2 is in the range of ρ then **fail**, else let $\rho_0 = \rho \cup \{x_1 \mapsto x_2\}$ and proceed as follows.

Let $l_1, \ldots, l_k$ be the outgoing edge labels from x_1 ordered according to a fixed label-order.[9] For $j = 1, \ldots, k$,

- For $i = 1,2$, **choose** l_j-edges (x_i, y_i) in G_i for some y_i.
  If **Extend**(ρ_{j-1}, y_1, y_2) fails then **fail**, else let $\rho_j =$ **Extend**(ρ_{j-1}, y_1, y_2).

Return ρ_k.

The corresponding pseudocode is given in Algorithm 2. The pseudo-code is more detailed than the previous description as it distinguishes the functional and relational edges. The non-deterministic choice function that is implemented using backtracking is called **chooseTargetNode** in Algorithm 2 and is used only with relational edges.

[9] At this point we know that x_2 must have the same outgoing edge labels in G_2 as x_1 has in G_1 or else x_2 would have a different label than x_1.

Algorithm 2 Pseudocode for computing the isomorphism of two rooted labelled directed graphs

COMPUTEISOMORHISM(G_1, G_2)

$\triangleright$ ρ – an isomorphism mapping of the graphs G_1 and G_2.
$\triangleright$ B – *Bucketing*, a helper class providing functions used in *Extend*.

if *ComputeBucketing* succeeds
  then if $\rho = Extend(\emptyset, G_1.root, G_2.root)$ succeeds
    then return ρ
return failed

COMPUTEBUCKETING(G_1, G_2)

Create an empty maps of type `Map<Label, Set<Node>>` B_1 and B_2
for $i \in \{1,2\}$
  for node $n \in G_i$
    if $B_i.containsLabel(n.label)$
      then $B_i[n.label] = B_i[n.label].Add(n)$
      else $B_i = B_i.Add(n.label, \texttt{new Set<Node>}(n))$
if label sets of B_1 and B_2 do not match
  then return fail
if node counts in bucket maps B_1 and B_2 do not match
  then return fail
return `new` $Bucketing(B_1, B_2, G_1, G_2)$

EXTEND(ρ, x_1, x_2)

$\triangleright$ ρ – Partial isomorphism
$\triangleright$ x_1, x_2 – Isomorphism candidates

if x_1 and x_2 have distinct labels
  then return fail
if $\{x_1 \mapsto x_2\} \in \rho$
  then return ρ
if $x_1 \in Domain(\rho) \vee x_2 \in Range(\rho)$
  then return fail
$\rho_0 = \rho \cup \{x_1 \mapsto x_2\}$
$k = 0$
for $l_f \in FunctionalOutgoingEdgeLabels(x_1)$
  if $Extend(\rho_k, x_1.getTargetNode(l_f), x_2.getTargetNode(l_f))$ fails
    then return fail
    else $\rho_{k+1} = Extend(\rho_k, x_1.getTargetNode(l_f), x_2.getTargetNode(l_f))$
      $k = k + 1$
for $l_r \in RelationalOutgoingEdgeLabels(x_1)$
  for $x_r \in x_1.getTargetNodes(l_r)$
    if $Extend(\rho_k, x_r, x_2.\textbf{chooseTargetNode}(l_r))$ fails
      then return fail
      else $\rho_{k+1} = Extend(\rho_k, x_r, x_2.\textbf{chooseTargetNode}(l_r))$
        $k = k + 1$
return ρ_k

Notice that the algorithm is deterministic and reduces to linearization when all choices are made from singleton sets. A sufficient (but not necessary) condition for this to be true is when all edge labels are functional. A heuristic we are using in the implementation of this algorithm is that all *functional* edge labels appear before all *relational* edge labels in the label-order that is used in the algorithm.

The implementation of the algorithm has also some optimizations when backtrack points can be skipped, that have been omitted in the above abstract description. One particular optimization is the following. When there are multiple l-edges outgoing from a node x for some fixed relational edge label l, but all of the target nodes of those edges have the same label and degree 1, then an arbitrary *but fixed* order of the edges can be chosen that uses the order of the node labels and choice points can be cut. The algorithm bears certain similarities to the practical graph isomorphism algorithm in [McKay, 1981], by using a partitioning scheme of nodes that eliminates a lot of the backtracking. The algorithm has been implemented in NModel.

3.5 State Isomorphism in the Dining Philosophers Example

Let us now build some intuition of what was presented in previous sections by looking at the Dining Philosophers example specified in Chapter 2. The scalar set based symmetry reduction does not help in the dining philosophers model, because the philosophers are arranged in a restricted topology: not all philosophers have access to all forks. Modelling such topology would require setting up some constraints on the scalar sets but this is not allowed by the definition of the scalar sets. NModel comes with a graphical tool called `mpv.exe`[10] for exploring model programs. We assume that the code given in Figures 2.4, 2.5, and 2.6 is in a single file called Phils.cs. We can compile the example with the following command line using the Mono compiler

```
gmcs /out:Phils.dll /t:library /r:NModel.dll Phils.cs
```

and with the following command line using the C# compiler in the .Net SDK 2.0 by Microsoft:

```
csc /out:Phils.dll /t:library /r:NModel.dll Phils.cs
```

The philosophers model can be opened in the model program viewer, `mpv.exe`, using the following command line:

```
mpv.exe /r:Phils.dll DiningPhilosophers.Contract.Create
```

Model program viewer displays the labelled transition system that results in running the model program until the whole state space has been covered or the maximum number of transitions to explore has been exceeded. By default the transition limit is 100, but it can be changed from the *Advanced properties panel*.

[10] `mpv.exe` is the only tool in the NModel toolkit that runs only in Microsoft Windows as it builds on the GDI interface. The modelling libary and other tools have been tested to work with an open source implementation of .Net and C# compiler called Mono version 1.2.3 and above.

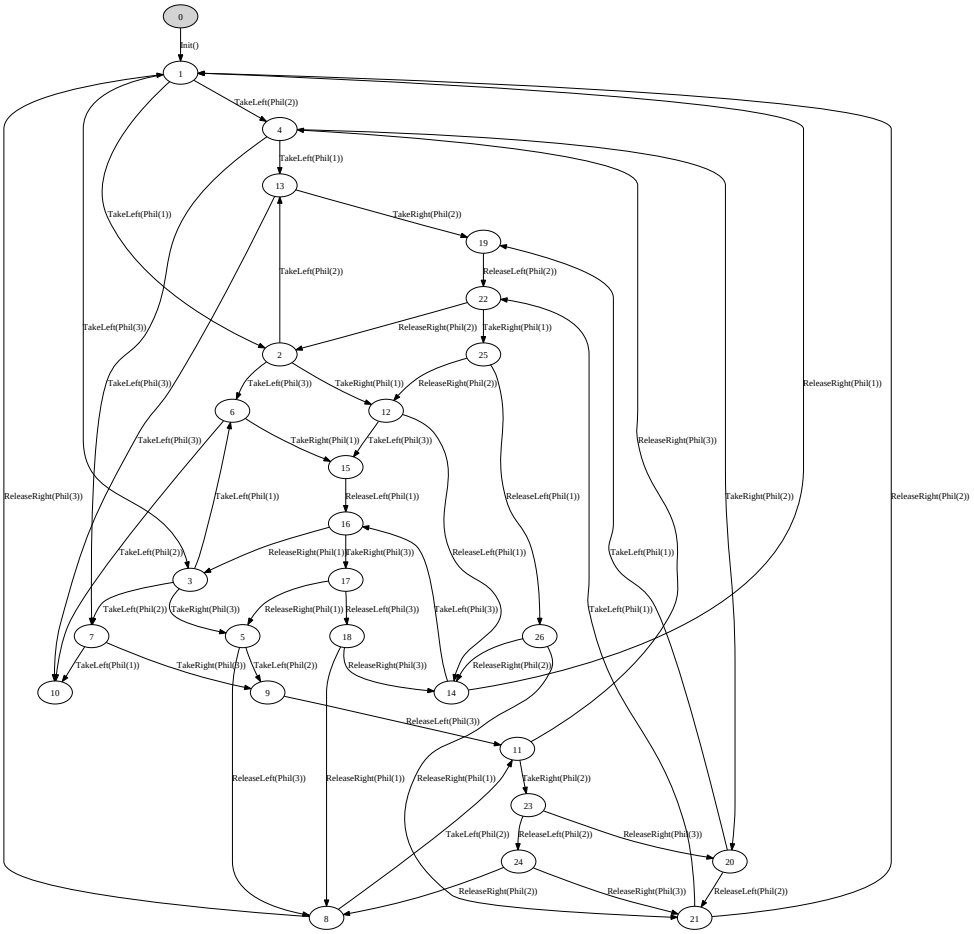


Figure 3.8: A transition system representing the full state space of the dining philosophers model program introduced in Figure 2.5.

When the dining philosophers example with 3 philosophers is opened in `mpv.exe` the user will see a graph representing the state space of the model program. The transition system is given in Figure 4.1.

The graphical user interface of the `mpv.exe` is in Figure 3.9.

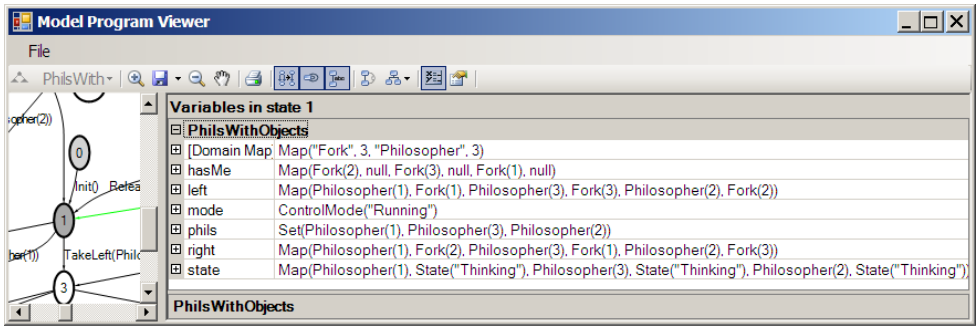


Figure 3.9: The graphical user interface of `mpv.exe` during exploration of the dining philosophers example.

One can look at the values of the global fields and field maps of the object fields at a model program location by selecting the corresponding node and opening the *state viewer* toolbox. The variables `mode` and `phils` are global fields, the fields `left`, `right`, and `state` are the fields of the instances of `Philosopher`, and the field `hasMe` is the field of the instances of `Fork`. The `[Domain Map]` is a built in field that is used for providing fresh abstract object IDs for new instances of objects.

The state isomorphism visualisation can be invoked by selecting *ExcludeIsomorphicStates* from the advanced properties panel, which appears when the appropriate button from the toolbar is pressed. In Figure 3.10 one can find the transition system representing the state space when state isomorphism checking has been switched on.

Figure 3.11 depicts the family of isomorphic states in the dining philosophers example and is obtained by collapsing all isomorphic states into one.

In Figure 3.12 a) there is the state graph of the state where all philosophers are thinking. The state graph in Figure 3.12 b) is the deadlock state where all philosophers have acquired one fork and cannot proceed. Obviously these states have distinct structure and are not isomorphic.

In Figure 3.13 there are the state graphs of states where one philosopher has aquired one fork. In the both cases the concrete philosopher having a fork is different, but the structure of the states is isomorphic.

In Figure 3.14 there are some experimental results of the effects of state isomorphism based symmetry reductions. We can see that the number of structurally distinct states grows much slower than the number of states in the system. Memory consumption is also lower in the symmetric case than in the explicit case, but processor time utilisation grows very fast with the number of philosophers. The latter indicates that the current implementation of linearisation is not good enough for larger models as

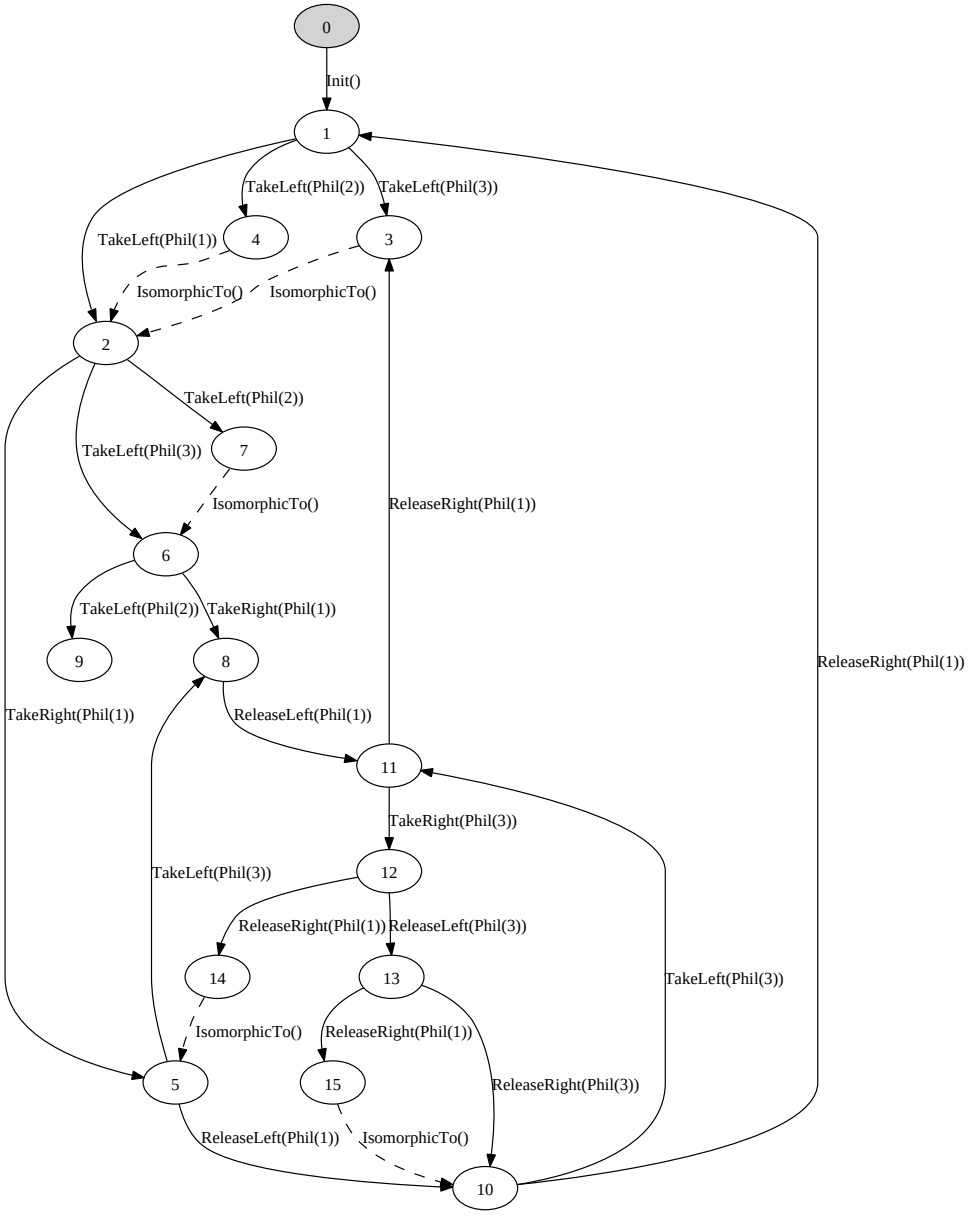


Figure 3.10: A transition system representing the state space of the dining philosophers model program with isomorphic state detection switched on.

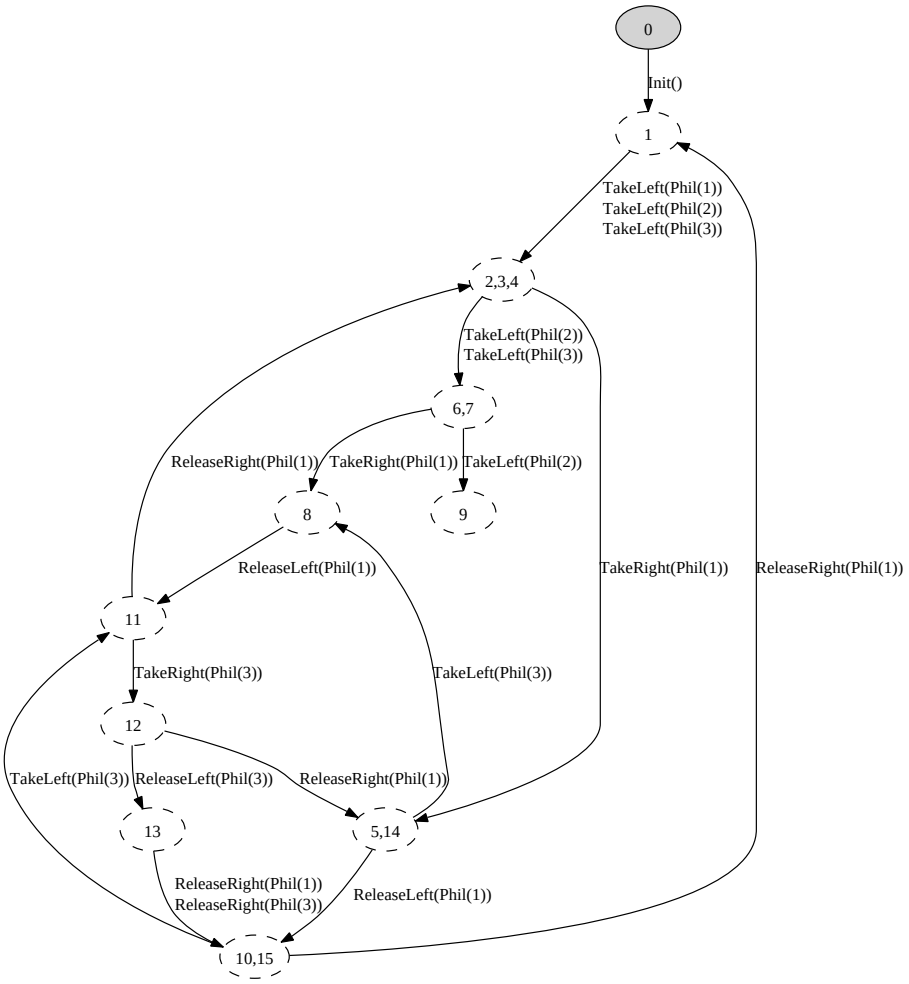


Figure 3.11: A transition system representing the family of isomorphic states in the dining philosophers example.

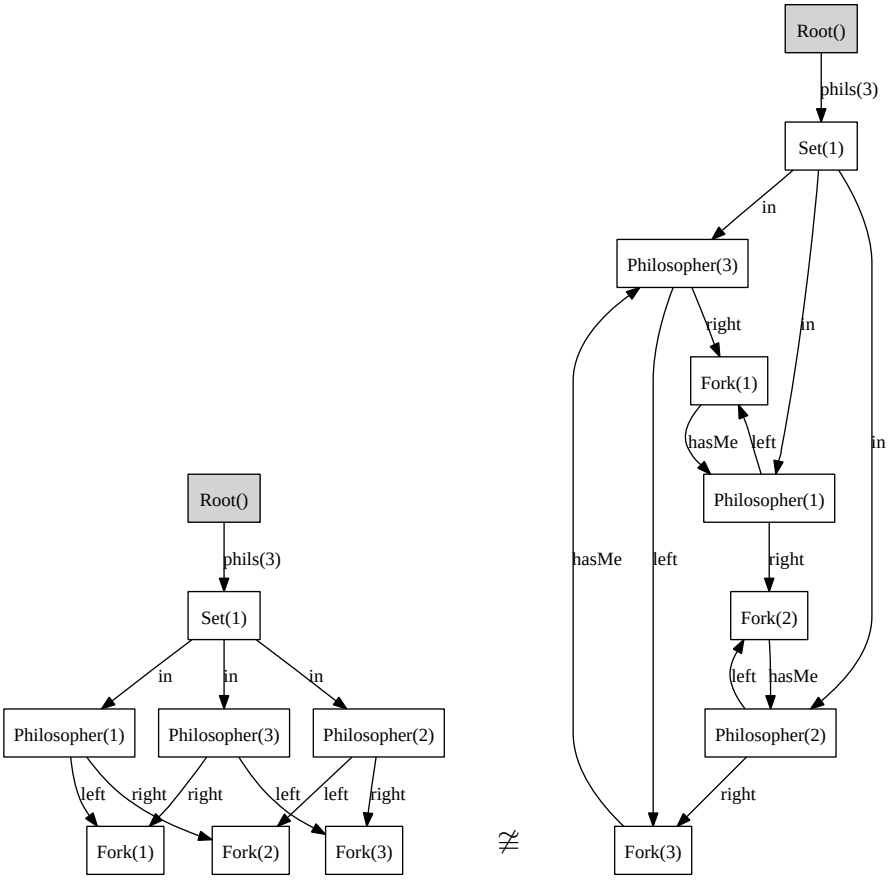
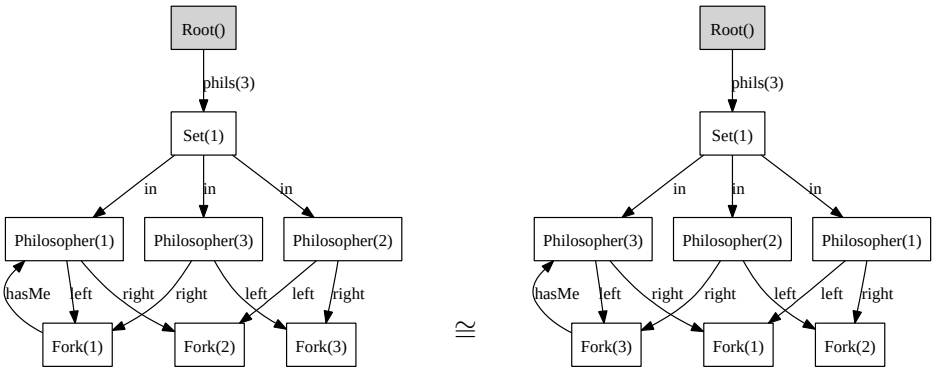


Figure 3.12: The state graphs of the dining philosophers example: a) The state where all forks are on the table. b) The deadlock state where each philosopher has one fork.



a) b)

Figure 3.13: Isomorphic states with sequence numbers 2 (a) and 3 (b) from the dining philosophers example.

isomorphism checks need to carry out a substantial amount of backtracks. Improving the implementation is planned as a future pursuit.

It should be mentioned that the symmetries exhibited by the dining philosophers example are not total symmetries, i.e., the symmetry is dependent on the topology of the arrangement of philosophers. Thus, it is not possbile to exploit such symmetries in the symmetry reductions built on scalar sets, as is the corresponding implementation of Uppaal [Hendriks, Behrmann, Larsen, Niebert and Vaandrager, 2003].

3.6 Conclusion

In this chapter we showed how state isomorphism for states with both unordered structures and objects may be understood in the context of model programs. We reviewed how the concept of background structures and reserve elements can formalise the meaning of isomorphism for program states. We then described how to represent state as a rooted directed labeled graph so that existing isomorphism algorithms could be applied. We showed an isomorphism-checking algorithm that takes advantage of the information contained in states with elements drawn from a rich background universe. Finally we showed how the state isomorphism reduction influences the dining philosophers example.

The techniques in this chapter can be applied in a variety of industrially relevant modeling and testing contexts and are motivated by practical concerns that arose from the industrial use of the Spec Explorer tool in Microsoft.

While this current chapter gives a solid notion how program states of object-oriented programs can be viewed as graphs, it also leads to a number of interesting open problems. For example, how can one speed up isomorphism checking for the particular graphs of program states? Would it be useful to describe graph isomor-

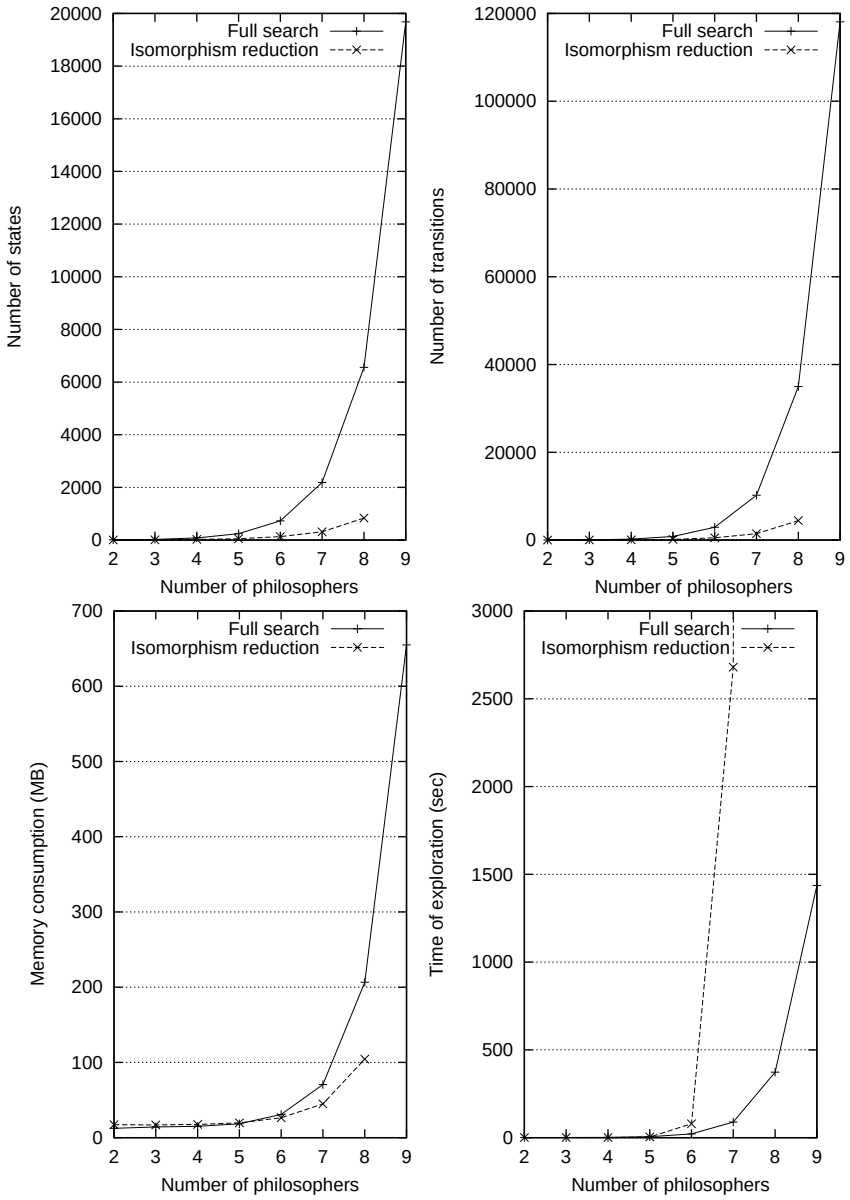


Figure 3.14: The effects of symmetry reductions for dining philosopher problem with different number of philosophers.

phism as a SAT problem? How could this be accomplished?

As future work, we plan on showing how hashing techniques can be used to improve the performance of isomorphism checks for larger numbers of states.

There is one drawback in the state isomorphism approach presented here. The problem is that state isomorphism is calculated on every transition, i.e., even if we have perfect linearisation, the processing time increases when we have larger models. Thus the next chapter looks at a different approach for state space reduction that sacrifices completeness but is applicable on very large examples.

CHAPTER 4

ITERATED SEARCH REFINEMENT WITH BITSTATE PRUNING

In this chapter we present a simple method of iterated search refinement where bitstate hash tables are used for pruning the search space during individual iterations of running explicit state model checking tasks. The method enables to apply model checking for larger models than full state space search or previously known bitstate hashing and Bloom filters based methods would allow due to very large memory requirements.

4.1 Introduction

It is desirable to model software and systems in a way that renders itself to automated analysis. Logic model checking is one of such automated analyses that has been proven to be useful and applicable on practical systems. In the case of explicit state model checking the method enumerates potentially all states of the system that are reachable from the initial state and checks whether properties specified by the user hold in the model. If a state violating the desired properties is found, model checker produces a trace to the error state thus easing the debugging of why it was possible to reach such state. Still, due to state space explosion, it is easy to construct models which have state spaces intractable with the state enumeration approach.

In recent years model checking has also been applied in different contexts, for example planning and scheduling where the target of search is not finding an error but finding a valid trace to the desired state. In either case, when we find an error or when we look for a plan satisfying the constraints set by our model, it is typically desirable to find an as short as possible witness trace.

A number of methods has been devised for reducing the number of states that needs to be enumerated for proving a property with model checking. The most notable of such methods in explicit state model checking are partial order and symmetry reduction methods that prune the search space by reducing the number of state permutations that needs to be considered for proving certain properties. In addition, there are a number of methods that sacrifice completeness for resources, i.e. the need for

memory and cpu time is reduced by approximations that may overlook some states that are reachable in the model. Examples of such methods in explicit state model checking are bitstate hashing and hash compaction.

In bitstate hashing only a single bit per state is stored in the table of already visited states at an address calculated by a hash function from the full state vector. Such method drastically reduces the need for memory for storing the state already seen but introduces the possibility that calculating the hash value of two distinct states yields a coinciding address in the hash table. In such case the state encountered later in the search is falsely considered as seen and thus parts of the state space may remain uncovered by the search. This typically motivates increasing the size of the bitstate hash table thus reducing the probability of omissions. Methods of multihash and Bloom filters further refine reducing the probabilities of omissions in cases where the number of bits in the hash table is of larger order of magnitude than the number of reachable states of the model.

We take a different perspective on the bitstate hashing method. The hash table is both in previous and our approaches used for distinguishing already seen states from those not yet encountered. If we set the hash table to be *much smaller* than the expected state space of the system, the bitstate hash table together with the search algorithm *prunes* the full search tree since collisions are bound to happen. We will see later in the chapter, that by changing the hash function the search tree gets pruned in a random fashion, thus covering different parts of the search tree in different runs. The same phenomenon is utilised in a different way in the Bloom filters approach where the probability of collisions is lowered by storing more than one bit per state at addresses calculated by independent hash functions. The fundamental difference with our case is that we populate hash tables independently and potentially by tasks running in parallel on a different cpu core or cluster node.

Another effect of the intentionally small bitstate hash table is that we bound the depth of the search by the size of the hash table. Intuitively the maximum depth of the search is the number of states we are able to distinguish — the number of bits in the bitstate hash table. In practice the maximum depth reached is always smaller than the size of the hash table. Combination of such effect and depth first search yiels significantly shorter witness traces than with regular depth first search. Thus the method can also be used for shortening witness traces found with depth first search of unpruned state space.

We show that such method can find traces to error or goal states in several large models from the Benchmarks for Explicit Model Checkers (BEEM) database [Pelánek, 2007] by presenting results obtained by running a prototype implementation that runs such iterations in parallel on the available number of processor cores. We present the results obtained by a prototype implementation of the iterated search refinement with bitstate pruning using an appropriately patched instance of the Spin model checker. We also give a reference to an experiment of the effects of guiding in the context of iterated search refinement with bitstate pruning by using Uppaal-Cora.

Example

We will reuse the *dining philosophers* example introduced in Chapter 2. For experiments with Spin we will use models of the dining philosophers that are crafted in the same manner as in Figure 2.2.

Figure 4.1 shows the finite state automaton representing the full state space and all possible transitions of the three dining philosophers model. We reduced the number of philosophers from five to three for the purpose of visualising the effects of iterated search refinement with bitstate pruning. There are 26 states and 52 transitions in the model with 3 philosophers and there are 242 states and 806 transitions in the model with 5 philosophers.

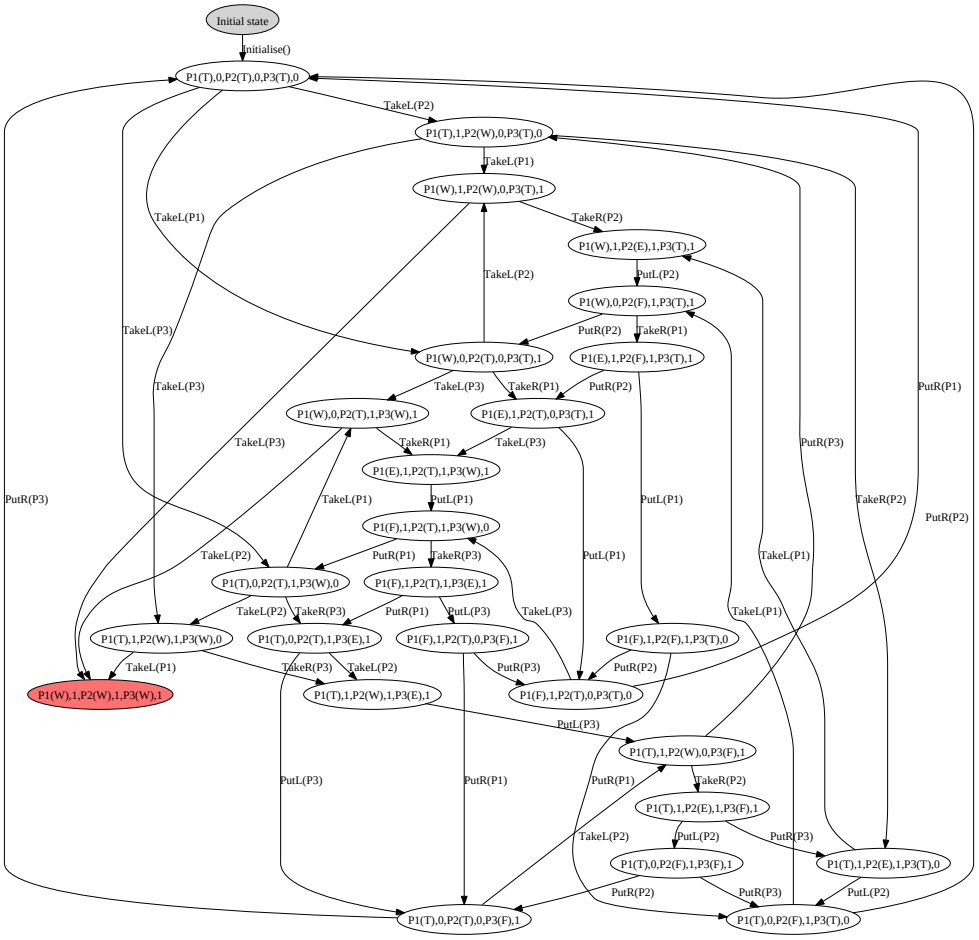


Figure 4.1: Transition system representing the full state space of the dining philosophers example with 3 philosophers. Philosophers are denoted by *P1*, *P2*, and *P3*. Philosopher states are *thinking* (T), *waiting* (W), *eating* (E), and *finishing* (F).

4.2 Related Work

There is a version of iterated search refinement built into Spin [Holzmann, 2003]. It is possible to pass a `-i` or `-I` command line key to the checker to iteratively reduce the depth of the witness trace. The method leads to the shortest possible path if there are sufficient resources for performing the search. In most larger cases considered in the current chapter, such approach did not succeed because of the exhaustion of memory resources.

Iterated search refinement is briefly mentioned in [Holzmann and Smith, 2000]. We build on the idea and extend it with the surprising result that in many cases it suffices to distinguish only a fraction of different states in the reachable state space of a model to find a trace to state exhibiting the property we are interested in.

Gerard J. Holzmann describes a sequential multihash method in [Holzmann, 1998], where the hash table is populated by independent hash functions under each iteration. The idea is similar to the iterated search refinement with bitstate pruning with the distinction that we overpopulate the hash table on purpose and exploit hash collisions for searching only a part of the total search space under each iteration. For us the bitstate hash table acts as a random pruning function. Gerard J. Holzmann also refers to Bloom filters named after Burton H. Bloom [Bloom, 1970], which permit to lower the probability of hash collisions in relatively empty hash tables by storing more than one bit per state. Gerard J. Holzmann chose to use double hashing for Spin [Holzmann, 2003].

In [Dillinger and Manolios, 2004b], Peter C. Dillinger and Panagiotis Manolios improve the double hashing method of Spin that is based on Jenkins' [Jenkins, 1997] hash function. In [Dillinger and Manolios, 2005] the method is further developed and implemented in 3Spin and 3Murphi [Dillinger and Manolios, 2004a]. They do not consider overpopulating the hash table on purpose but improve the efficiency of using Bloom filters in Spin. They also show that although using hash table sizes of powers of 2 has a certain performance advantage as the `MOD` function becomes bit masking during compilation, the impact of using arbitrary sizes does not have that great impact.

Bloom filters act in a direction unfavourable to iterated search refinement with bitstate pruning.

The current approach has similarities with the re-initialisation enhancement of the random walk method described in [Pelánek, Hanžl, Černá and Brim, 2005], but our method has native support for trace generation and uses either depth first search or random best depth first search [Behrmann, Larsen and Rasmussen, 2004] in the guided case. The randomness is due to pruning the search space with Bloom filters and in the guided case additionally due to the random factor in the random best depth first search. In [Barnat, Brim and Chaloupka, 2003] parallel breadth first search for LTL model checking is discussed. Our approach, when run in parallel, runs depth first search on different partially overlapping parts of the state space bounded by the size of the hash table. Additionally, when the hash table becomes large enough, it

will facilitate search of the full state space with some probability. Ways to calculate this probability are discussed in [Holzmann, 1998].

4.3 Bitstate Hashing

Model checking in general involves searching possibly very large state spaces for proving or disproving a query — a formula typically in some temporal logic. Bitstate hashing [Holzmann, 1998], also known as supertrace, is a well known method in explicit state model checking for reducing memory requirements of storing the traversed state space by storing only a single bit for each seen state at the address calculated by a hash function. The method has the feature of the possibility of hash collisions that will result in unexplored parts of the state space, rendering the method to be incomplete. Still, if a state exhibiting a property of interest is reached, the trace produced is a valid trace in the model. Thus bitstate hashing is sound, i.e. it does not yield false positives. In general, the bigger the hash table, the lower the probability of hash collisions. But big bitstate hash tables may still require unavailable amounts of memory.

The general significance of reachability checks has been outlined in [Aceto et al., 2003]. Even if it is not possible to prove unreachability, fast reachability checks on formal models that yield a valid trace have applications in, for example, some types of planning and scheduling [Hune et al., 2001; Wijs et al., 2005; Ruys, 2003], test generation [Hamon et al., 2004b; Ernits et al., 2006], software/hardware synthesis [Ernits, 2005], and in debugging [Mercer and Jones, 2005].

Let us have a look at the example with three dining philosophers in Figure 4.1. Let us assume that the states of a single philosopher are encoded using two bits in the following way $thinking = 00$, $waiting = 01$, $eating = 10$, and $finishing = 11$. A fork is encoded using a single bit. If a fork is picked up it is 1 and if it is on the table it is 0. The state vectors of the dining philosophers example with 3 philosophers are given in Table 4.1. In the right part of the table there are the addresses of respective states in a bitstate hash table obtained by calculating the division remainder (*mod*) of the state. of the size indicated in the column header. The red (grey) numbers indicate which states are reachable with depth-first (DF) and breadth-first (BF) search. If we consider the states encountered during all separate DF searches, we get that by using 5 to 12 bits for distinguishing states in the model we encountered 17 of the total 26 states. This is the key to iterated search refinement with bitstate pruning. The deadlock state (10) was first encountered when the bitstate hash table was just 7 bits.

In this small example the difference between breadth-first and depth-first search does not show very clearly. The only difference is that breadth-first search encountered just 14 states of the total 26 over the searches. Experiments show that depth-first search performs far better for bitstate pruning than breadth-first search. The intuition is that depth first search will end up in deeper random corners of the search space due to bitstate pruning while breadth-first search fills up the bitstate hash table by reaching much shallower depths.

Table 4.1: State vectors of the dining philosophers example and reachable states using depth-first (DF) and breadth-first (BF) search with different bitstate hash table sizes in bits.

State	State number	Bit 1 of P1 Bit 0 of P1 Fork 1 Bit 1 of P2 Bit 0 of P2 Fork 2 Bit 1 of P3 Bit 0 of P3 Fork 3	State as a decimal	Address in bitstate hash table and search coverage															
				5 DF	5 BF	6 DF	6 BF	7 DF	7 BF	8 DF	8 BF	9 DF	9 BF	10 DF	10 BF	11 DF	11 BF	12 DF	12 BF
P1(T),0,P2(T),0,P3(T),0	1	0 0 0 0 0 0 0 0 0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
P1(W),0,P2(T),0,P3(T),1	2	0 1 0 0 0 0 0 0 1	129	4	4	3	3	3	3	1	1	3	3	9	9	8	8	9	9
P1(T),0,P2(T),1,P3(W),0	3	0 0 0 0 0 1 0 1 0	10	0	0	4	4	3	3	2	2	1	1	0	0	10	10	10	10
P1(T),1,P2(W),0,P3(T),0	4	0 0 1 0 1 0 0 0 0	80	0	0	2	2	3	3	0	0	8	8	0	0	3	3	8	8
P1(T),0,P2(T),1,P3(E),1	5	0 0 0 0 0 1 1 0 1	13	3	3	1	1	6	6	5	5	4	4	3	3	2	2	1	1
P1(W),0,P2(T),1,P3(W),1	6	0 1 0 0 0 1 0 1 1	139	4	4	1	1	6	6	3	3	4	4	9	9	7	7	7	7
P1(T),1,P2(W),1,P3(W),0	7	0 0 1 0 1 1 0 1 0	90	0	0	0	0	6	6	2	2	0	0	0	0	2	2	6	6
P1(T),0,P2(T),0,P3(F),1	8	0 0 0 0 0 0 1 1 1	7	2	2	1	1	0	0	7	7	7	7	7	7	7	7	7	7
P1(T),1,P2(W),1,P3(E),1	9	0 0 1 0 1 1 1 0 1	93	3	3	3	3	2	2	5	5	3	3	3	3	5	5	9	9
P1(W),1,P2(W),1,P3(W),1	10	0 1 1 0 1 1 0 1 1	219	4	4	3	3	2	2	3	3	3	3	9	9	10	10	3	3
P1(T),1,P2(W),0,P3(F),1	11	0 0 1 0 1 0 1 1 1	87	2	2	3	3	3	3	7	7	6	6	7	7	10	10	3	3
P1(E),1,P2(T),0,P3(T),1	12	1 0 1 0 0 0 0 0 1	321	1	1	3	3	6	6	1	1	6	6	1	1	2	2	9	9
P1(W),1,P2(W),0,P3(T),1	13	0 1 1 0 1 0 0 0 1	209	4	4	5	5	6	6	1	1	2	2	9	9	0	0	5	5
P1(F),1,P2(T),0,P3(T),0	14	1 1 1 0 0 0 0 0 0	448	3	3	4	4	0	0	0	0	7	7	8	8	8	8	4	4
P1(E),1,P2(T),1,P3(W),1	15	1 0 1 0 0 1 0 1 1	331	1	1	1	1	2	2	3	3	7	7	1	1	1	1	7	7
P1(F),1,P2(T),1,P3(W),0	16	1 1 1 0 0 1 0 1 0	458	3	3	2	2	3	3	2	2	8	8	8	8	7	7	2	2
P1(F),1,P2(T),1,P3(E),1	17	1 1 1 0 0 1 1 0 1	461	1	1	5	5	6	6	5	5	2	2	1	1	10	10	5	5
P1(F),1,P2(T),0,P3(F),1	18	1 1 1 0 0 0 1 1 1	455	0	0	5	5	0	0	7	7	5	5	5	5	4	4	11	11
P1(W),1,P2(E),1,P3(T),1	19	0 1 1 1 0 1 0 0 1	233	3	3	5	5	2	2	1	1	8	8	3	3	2	2	5	5
P1(T),1,P2(E),1,P3(T),0	20	0 0 1 1 0 1 0 0 0	104	4	4	2	2	6	6	0	0	5	5	4	4	5	5	8	8
P1(T),0,P2(F),1,P3(T),0	21	0 0 0 1 1 1 0 0 0	56	1	1	2	2	0	0	0	0	2	2	6	6	1	1	8	8
P1(W),0,P2(F),1,P3(T),1	22	0 1 0 1 1 1 0 0 1	185	0	0	5	5	3	3	1	1	5	5	5	5	9	9	5	5
P1(T),1,P2(E),1,P3(F),1	23	0 0 1 1 0 1 1 1 1	111	1	1	3	3	6	6	7	7	3	3	1	1	1	1	3	3
P1(T),0,P2(F),1,P3(F),1	24	0 0 0 1 1 1 1 1 1	63	3	3	3	3	0	0	7	7	0	0	3	3	8	8	3	3
P1(E),1,P2(F),1,P3(T),1	25	1 0 1 1 1 1 0 0 1	377	2	2	5	5	6	6	1	1	8	8	7	7	3	3	5	5
P1(F),1,P2(F),1,P3(T),0	26	1 1 1 1 1 1 0 0 0	504	4	4	0	0	0	0	0	0	0	0	4	4	9	9	0	0

In the case of bitstate hashing a hash is calculated from the state vector to be stored in the bitstate hash table. The hash function can either be the division remainder function or some faster hash function, but in both cases the hash function determines the address of the bit to be stored. The division remainder function can be used to wrap the hash value to bitstate hash table of any given size. The faster hash function used in Spin to hash the states is the hash function by Jenkins [Jenkins, 1997] accompanied with a division remainder function. Such combination also helps to lower the influence of the ordering of the variables in the state vector.

4.3.1 Collision probabilities

Let us consider the probability distribution of collisions in a bitstate hash table. We assume that we have a perfect hash function, which will cause a uniform distribution of hash values along the bitstate hash table with size m.

The probability of collision is trivially 0 for the first bit entered into the hash table. The probability of collision when entering the second bit is $1/m$, the third bit $2/m$ etc. Thus we get a linear distribution function $F_{k=1}(x) = x/m$ where x is the number of states stored and m is the hash table size in bits. A collision probability distribution

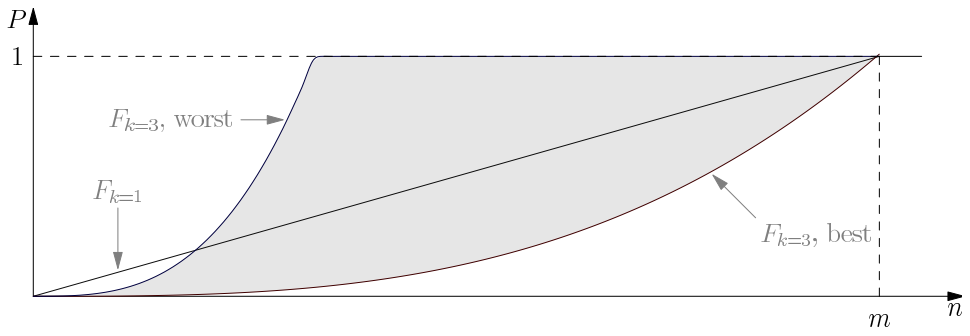


Figure 4.2: Collision probability distribution for a bitstate hash table of size m bits. $F_{k=1}$ represents the collision distribution when a single bit per state is stored. $F_{k=3}$ represents the case when 3 bits per state are stored in the hash table. n is the axis representing the number of states stored and P represents the collision probability.

function in such case is represented in Figure 4.2.

The analysis of the probability distribution of a Bloom filter requires us to consider the best and the worst case. The collision probability of entering the first k bits is trivially 0. In the best case, entering the first state occupies k bits and entering every subsequent state occupies just one extra bit. The collision probability distribution function is $F_{best}(x,k) = x/(m-k)$. In the worst case, entering every subsequent state into the hash table occupies k bits. The distribution function in this case would be $F_{worst}(x,k) = kx/(m-k)$. Obviously, the actual collision probability is somewhere in between the two extremes denoted by the grey area in Figure 4.2. Thus, Bloom filters actually counteract the effect of bitstate pruning as they reduce the probability of collisions towards the beginning of the search thus skewing the search tree towards the "left" if we assume that depth first search explores left successors first. In fact, applying bitstate hashing with a single bit per state and modifying the hash function (by changing the hash table size by 1 byte) has the same effect as the best case of Bloom filters: it pushes the probability distribution downwards as is described in Figure 4.3.

4.4 Iterated Search Refinement

The algorithm of iterated search refinement with bitstate pruning is given in Algorithm 3. $ModelCheck_{bitstate}(M,q,k,b)$ is a function of calling a model checker with the model M, reachability query q (may also be empty if we are looking for assertion violations), bitstate hash table size k, and the depth bound b. If the depth bound is 0 when calling *IterationTask*, the depth bound is set to $8k$ as the number of bits in the bitstate hash table is the theoretical maximum of the witness trace length.

The algorithm works in the following way. Initially an *IterationTask*, which is a batch of small steps, i.e., n_{ss} increments of one byte in the hash table size per

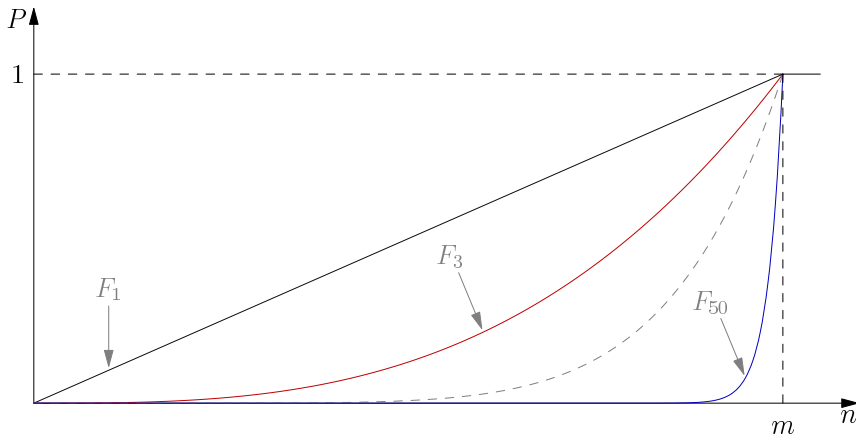


Figure 4.3: Collision probability distribution for repeated bitstate hashing with distinct hash functions at a hash table size of m bits. F_1 represents the collision distribution when a single bit per state is stored. F_3 represents the probability distribution for 3 runs and F_{50} the probability distribution of 50 runs.

iteration step, is asynchronously assigned to each available processor starting each at a different base value. If any of the *IterationTasks* finishes, a new batch is submitted and the bitstate hash table size is increased. If an *IterationTask* returns a trace to an error or goal state, all other tasks are terminated and a set of fresh batches with the new depth bound b is initiated. b is the length of the newly discovered trace. The values of k and c, the current big step multiplier, are reset. The idea of starting to increase the size of the bitstate hash table by the factor of c_{bs} only after the threshold l_{bs} is because the iterations with very small bitstate hash tables take relatively little time but often yield interesting results. The threshold l_{bs} and the number of small steps per batch n_{ss} may be tuned at runtime according to the specifics of the model.

The algorithm does not terminate after finding the first trace to a state of interest. Instead, it continues to look for a shorter trace.

4.5 Prototype implementation

Implementation of the prototype consists of two parts: the implementation of submitting search refinement tasks to the available number of processor cores and a patched version of Spin that generates instances of checkers where it is possible to specify the size of the bitstate hash table in *bytes* not just *megabytes* and *gigabytes* as by default. The corresponding trivial but for our purpose necessary patch introducing the command line key "`-B`" enabling to specify the bistate hash table size in bytes to Spin 4.3.0 is rather straight forward.

The models, details of the experimental results, an implementation of the prototype, and the -B patch to Spin 4.3.0 are available from `http://www.cc.ioc.`

Algorithm 3 Iterated search refinement with bitstate pruning

M	–	Model of the system
q	–	Reachability query
P	–	Pool of processors to which we can asynchronously submit tasks
c_{bs}	–	Big step multiplier
l_{bs}	–	Big step threshold
n_{ss}	–	Number of small steps
b	–	Bound for the search depth

```
ITERATEDSEARCH()
    ▷ k – Size of the bitstate hash table in bytes
    k ← 1
    b ← 0
    c ← 1
    for each processor ∈ P
            P.Submit(IterationTask(k,b))
            k ← (k + n_ss) * c
            if k > l_bs
                    c ← c_bs
    while (result,trace) = P.GetFinishedTask()
            if result == FoundTrace
                    currentBestTrace ← trace
                    b ← trace.Length − 1
                    P.KillAll
                    k ← 1
                    c ← 1
                    for each processor ∈ P
                            P.Submit(IterationTask(k,b))
                            k ← (k + n_ss) * c
                            if k > l_bs
                                    c ← c_bs
            if k < MaxHashTableSize
                    P.Submit(IterationTask(k,b))
                    k ← round((k + n_ss) * c)

ITERATIONTASK(k,b)
    if b == 0
            b_1 ← 8k
    for i in {1,...,n_ss}
            (result,trace) ← ModelCheck_bitstate(M,q,k + i,b_1)
            if result == FoundTrace
                    return (result,trace)
```

```
#define MAX 999
int xx, yy ;

active proctype Inc() { do :: xx = (xx + 1) % MAX od }
active proctype Dec() { do :: yy = (yy - 1) % MAX od }
// Monitor process
#define P !((xx==MAX-1)&(yy==1-MAX))
active proctype monitor() { do :: assert(P) od }
```

Figure 4.4: A toy model in Promela representing an incrementer and a decrementer process [Ruys, 2001].

`ee/~juhan/bitprune`.

We performed all of the following experiments on a node with 8 dual core Opteron processors (16 cores) running at 2.4 GHz and containing 1 MB of L2 cache per core. The node has 32 GB of 667 MHz DDR2 memory and runs 64-bit GNU/Linux.

4.6 Evaluation

Let us consider the model of an incrementer and decrementer process `inc-dec.pr` [Ruys, 2001] written in Promela given in Figure 4.4.

Theo Ruys demonstrated in [Ruys, 2001] that this kind of model is particularly awkward for bitstate hashing. Using iterated search refinement with bitstate pruning on the described experiment node yielded a trace to an error state in 143 seconds with the length of 6016 steps using a single hash function and the bitstate hash table size of 2855 bytes. The latter result is summarised in Figure 4.5.

Dining philosophers example

On Figure 4.6 there are the results of applying the iterated search refinement with bitstate pruning on the dining philosophers example with varying number of philosophers. The iteration parameters were the following: from 1 to 10000 bytes the hash table was increased in one byte increments. From 10000 onwards the big step multiplier of 1.2 was applied. The iteration was terminated when the bitstate hash table exceeded 200000 bytes.

In the upper left diagram in Figure 4.6 we see that the length of the first trace found may vary to a quite large degree. The lengths of the traces to deadlock found by the time the iteration terminates differ by a factor of approximately 10 from the minimal result.

In the upper right diagram in Figure 4.6 there are the times in seconds it took to reach the deadlock states with the iteration method. The spike in time it took to find the shortest trace in the case of 20 philosophers is due to the phenomenon that sometimes the iteration finds just one step shorter trace during each iteration and the

iteration is restarted from the hash table size 1 each time a shorter depth bound is found.

The lower diagram in Figure 4.6 represents the hash table sizes in bytes at which the traces to deadlock were found. Although the hash table sizes show steady increase, the sizes are still surprisingly low, for example just 514 bytes for finding the deadlock in the instance of 255 dining philosophers. The results of running Spin with the dining philosophers example with 255 philosophers is given in Figure 4.7,

We could not go beyond 255 processes because of the process count limit of Spin. We did not list the memory requirements of the instances of model checking tasks because they are minor, just 10-50 MB depending on the amount of memory allocated for the stack. It has to be kept in mind that in the case of the 255 philosopher example, the C compiler `gcc` used to compile the checker generated by Spin required 3 Gb of memory. The memory requirements of the compiler can be reduced by reducing the level of optimisation thus sacrificing some speed.

Blocks example

In the BEEM database [Pelánek, 2007] there is a number of models where the goal is to find a trace to the goal state. One of those examples is a typical blocks world planning example. The experiment with running our implementation against the hardest

```
> this filename  : inc-dec.pr.out
> promela file   : inc-dec.pr,
> options file   : produced by BitPrune iterator
> date           : 16-Sep-2007 12:35:29
> spin version   : Spin Version 4.3.0 -- 22 June 2007, with '-B' patch
> gcc version    : gcc (GCC) 4.1.2 20061115 (prerelease) (Debian 4.1.1-21)
> spin options   : -a
> gcc options    : -DBITSTATE -DSAFETY -DVECTORSZ=65535
> pan options    : -k1 -B2855 -m22840 # 8*2855=22840

./pan -k1 -M2855 -m22840
pan: assertion violated  !(((xx==(999-1))&(yy==(1-999)))) (at depth 5992)
pan: wrote inc-dec.pr.trail
(Spin Version 4.3.0 -- 22 June 2007, with '-B' patch)
Warning: Search not completed
        + Partial Order Reduction

Bit statespace search for:
        never claim             - (none specified)
        assertion violations    +
        cycle checks            - (disabled by -DSAFETY)
        invalid end states      +

State-vector 44 byte, depth reached 6016, errors: 1
    6815 states, stored
    8634 states, matched
   15449 transitions (= stored+matched)
       0 atomic steps

hash factor: 1230.9 (best if > 100.)

bits set per state: 1 (-k1)

6.553   memory usage (Mbyte)
0.01 user, 0.02 system, 0.11 elapsed
```

Figure 4.5: Results found by iterated bitstate pruning for running Spin on the `inc-dec.pr` example.

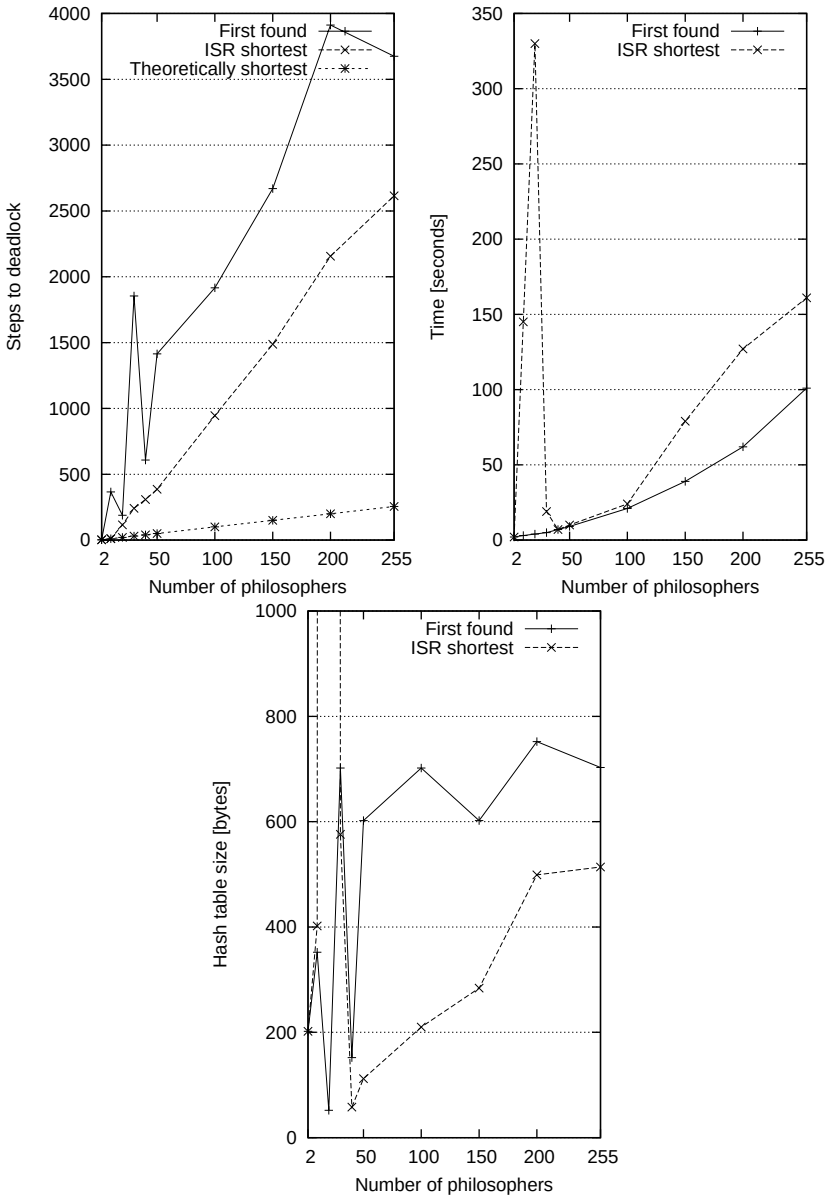


Figure 4.6: Trace lengths to deadlock, times, and bitstate hash table sizes for finding the deadlock states in dining philosopher examples with 2 . . . 255 philosophers.

of the blocks examples in the BEEM database, `blocks.4.pm`, yielded a trace in 144 seconds containing 1525 steps to the goal state. The bitstate hashtable size was 2915 bytes. The Promela code of the model is given in Appendix B for reference.

The witness trace producing result of running the `blocks.4.pm` example is in Figure 4.8

The effects of guiding

An experiment on the effects of guiding on iterated search refinement with bitstate pruning can be found in the next chapter. The results indicate that cost helps to shorten the resultant traces but the benefit depends highly on the how well we succeed in defining the cost function.

Bitstate pruning and processor cache utilisation

One of the reasons why the presented approach works quite well in practice is that the calls to model checker that use small bitstate hash tables allow the processor to be very cache efficient. Experiments with Valgrind's Cachegrind [Nethercote, Walsh

```
> this filename  : phils.255.pr.out
> promela file   : phils.255.pr,
> options file   : produced by BitPrune iterator
> date           : 16-Sep-2007 18:08:11
> spin version   : Spin Version 4.3.0 -- 22 June 2007, with '-B' patch
> gcc version    : gcc (GCC) 4.1.2 20061115 (prerelease) (Debian 4.1.1-21)
> spin options   : -a
> gcc options    : -DBITSTATE -DSAFETY -DVECTORSZ=65535
> pan options    : -k1 -B514 -m3276

pan: invalid end state (at depth 2614)
pan: wrote phils.255.pr.trail
(Spin Version 4.3.0 -- 22 June 2007, with '-B' patch)
Warning: Search not completed
        + Partial Order Reduction

Bit statespace search for:
        never claim             - (none specified)
        assertion violations    +
        cycle checks            - (disabled by -DSAFETY)
        invalid end states      +

State-vector 2300 byte, depth reached 2615, errors: 1
    2623 states, stored
    2063 states, matched
    4686 transitions (= stored+matched)
       0 atomic steps

hash factor: 3198.1 (best if > 100.)

bits set per state: 1 (-k1)

Stats on memory usage (in Megabytes):
6.064   equivalent memory usage for states (stored*(State-vector + overhead))
1.049   memory used for hash array (-B514)
0.184   memory used for DFS stack (-m3276)
5.321   other (proc and chan stacks)
6.553   total actual memory usage
0.02 user, 0.01 system, 0.11 elapsed
```

Figure 4.7: Results found by the iterated bitstate pruning for running Spin on the dining philosophers problem with 255 philosophers.

```
> this filename  : blocks.4.pm.out
> promela file   : blocks.4.pm,
> options file   : produced by BitPrune iterator
> date           : 16-Sep-2007 17:08:11
> spin version   : Spin Version 4.3.0 -- 22 June 2007, with '-B' patch
> gcc version    : gcc (GCC) 4.1.2 20061115 (prerelease) (Debian 4.1.1-21)
> spin options   : -a
> gcc options    : -DBITSTATE -DSAFETY -DVECTORSZ=65535
> pan options    : -k1 -B2915

pan: invalid end state (at depth 1524)
pan: wrote blocks.4.pm.trail
(Spin Version 4.3.0 -- 22 June 2007, with '-B' patch)
Warning: Search not completed
        + Partial Order Reduction

Bit statespace search for:
        never claim             - (none specified)
        assertion violations    +
        cycle checks            - (disabled by -DSAFETY)
        invalid end states      +

State-vector 44 byte, depth reached 1525, errors: 1
    1894 states, stored
    1238 states, matched
    3132 transitions (= stored+matched)
       0 atomic steps

hash factor: 4429.04 (best if > 100.)

bits set per state: 1 (-k1)

6.553   memory usage (Mbyte)
0.01 user, 0.00 system, 0.11 elapsed
```

Figure 4.8: Verification result producing a plan for the `blocks.4.pm` example. The desired end state is encoded as an assertion failure, thus Spin reports it as an "invalid end state".

and Fitzhardinge, 2006] indicate close to 100% utilisation of level 1 cache which means that the processor can work with negligible need to wait for memory operations to complete.

4.7 Discussion and further work

The results where it is possible to reach some interesting state by distinguishing only a fraction of the states of the reachable state space leads to the question of whether it is possible to use the bitstate pruning approach for detecting useful abstractions of the model. If a witness trace to a state with an interesting property is found, we can use the parameters of bitstate pruning to define a distinguishing function $\delta : S \rightarrow E$ on the automaton A where E defines a set of equivalence classes on S, i.e. several distinct $s \in S$ map to one $e \in E$.

Bitstate pruning acts as a distinguishing function with the following parameters: the hash function $hash()$ used for calculating a hash of the state vector, bitstate hash table size k, the bound on the depth of the search b, and the search algorithm $search()$. Thus the distinguishing function of bitstate pruning is a function depending on four arguments: $\delta(hash(), k, b, search())$. We may wish to construct a function that maps states to equivalence classes in the same manner as δ but that would not depend on the search algorithm or hash function.

A primitive way to construct such function would be to construct sets of equivalence classes while traversing the concrete search tree defined by $\delta(hash(), k, b, search())$. All states that have colliding hashes in the hash table would go into the same equivalence class. There would need to be one extra equivalence class for the states not encountered during the concrete search run. This is obviously not the best possible way of constructing a new δ. But we consider the analysis of ways how to learn state distinguishing functions from successful runs of bitstate pruning as part of the future work.

4.8 Conclusion

In this chapter we presented a practical method of iterated search refinement with bitstate pruning that can be applied in explicit state model checking to check models that are otherwise intractable due to memory requirements. We showed that such method can find traces to error or goal states in several large models — hundreds of dining philosophers and a planning example of the blocks world. The results were obtained using a prototype implementation that runs the iterations in parallel on any available number of processor cores. The method works for finding traces to errors or desired states and can be used for shortening traces found in depth first search. We argued that Bloom filters act in an unfavourable direction to be used in this kind of application as they reduce the probability of collisions in the beginning of the search thus skewing the search density towards the “left” of the search tree. We carried out the experiments with a prototype implementation that supports a modified version of

Spin 4.3.0. In the next chapter we will have a look at two different case studies where the method presented in the current chapter yields interesting results.

CHAPTER 5

APPLICATIONS OF ITERATED SEARCH REFINEMENT WITH BITSTATE PRUNING

In this chapter we will have a look at two case studies where iterated search refinement with bitstate pruning enables to use model checking for finding a solution. The first case study is about model-based generation of preset tests and contains additionally an analysis of different search strategies and effects of cost guiding in conjunction with the ISR method. The second case study is about applying model checking for the synthesis of a memory arbiter for data streaming in a radar application.

5.1 Generating Preset Tests

In this chapter we will focus on two examples where the previously described iterated search refinement with bitstate pruning makes it possible to solve the problem with a model checker. The first example is about generating preset tests from models of specification for off-line testing.

5.1.1 Introduction

In this section we target test generation for software systems from specifications in the form of extended finite state machines (EFSMs). We propose a method of test generation that combines techniques of model construction with iterated search refinement with bitstate pruning in model checking.

One possible motivation for working with EFSMs is that specifications provided in terms of, for example, suitably restricted UML statecharts can be converted into EFSMs. Converting UML statecharts to EFSMs is not the topic of the current section and thus we use EFSMs as the starting point for the reason that they provide a semantically well-defined model representation that can be applied for test generation. The problem of generating test sequences is formulated as a bounded reachability problem and solved by model checking.

The procedure of searching for a suitable test sequence is simple if the software is modeled as a finite state machine that has neither variables nor guard conditions. Introducing variables and guard conditions, as in EFSMs, makes the search much more complex.

The complexity arises from the large number of combinations of values that the variables can be assigned and from the need to satisfy guard conditions for taking transitions. One well known option for generating tests for EFSMs is to use the search machinery provided out-of-the-box by model checkers.

If a model checker solves a reachability task, it generates a witness trace that corresponds to an abstract test sequence.

The most critical factor of space exploration based methods is scalability, i.e., the ability to handle the exponential growth of the search space.

One example of problems where scalability quickly becomes acute, is targeting some structural test coverage criteria that result in long traces. For example *all transitions* of the Implementation Under Test (IUT) model or *all possible subsequences of transitions of some length* $k > 1$ of the IUT. Our goal is to generate preset tests for models of deterministic IUT models.

We compare different search strategies and iterated search refinement on the well-known benchmark examples of stopwatch and the INRES protocol [Hogrefe, 1991].

We show how guiding the search with a cost variable influences the lengths and required amounts of memory of test generation. In fact, we merge guiding together with iterated search refinement to reduce the lengths of generated test sequences and to improve the scalability of applying explicit state model checking for test generation.

We use the model checker Uppaal and its guided counterpart Uppaal Cora [Behrmann, Larsen and Rasmussen, 2004] because it enables us to demonstrate both, the influence of guiding, and iterated search refinement, in the presented context of test generation.

5.1.2 Related Work

The most common coverage criteria in the context of model-based testing are structural coverage criteria, such as state coverage and transition coverage [Farchi, Hartman and Pinter, 2002]. Test generation according to structural coverage criteria is often treated as a reachability problem and solved either by symbolic or explicit state model checking [Edmund M. Clarke et al., 1999].

An automated test generation tool SAL-ATG based on SAL2 symbolic model checker is proposed in [Hamon, de Moura and Rushby, 2004a]. An alternative approach to test case generation by explicit state model checking is studied extensively on the basis of the Uppaal family of tools[1]. Special testing environments Uppaal Tron [Larsen, Mikucionis, Nielsen and Skou, 2005] and Uppaal CoVer [Blom, Hessel, Jon-

[1]The representation of time in Uppaal is symbolic. The representation of locations and integer and boolean variables is explicit state [Bengtsson, 2002].

sson and Petterson, 2005], [Hessel, Larsen, Nielsen, Petterson and Skou, 2004] have been built upon the main search engine of Uppaal.

Cost automata based Uppaal Cora [Behrmann, Larsen and Rasmussen, 2004] is designed for solving cost guided reachability problems and can be used also for introducing context information to guide test case generation. One important problem in using model checking for test case generation is encoding test coverage criteria. In [Hong, Lee, Sokolsky and Ural, 2002] the structural coverage criteria are represented by a set of CTL formulae. Similarly, temporal logics LTL and CTL are used respectively in [Gunter and Peled, 2005] and in [Blom et al., 2005] for specifying path conditions that are transformed to property automata. In SAL-ATG the test purpose is stated directly as an observer state machine. Finding a minimal-cost or time optimal witness for a formula is combinatorially hard. Existing model checkers search minimal-cost witnesses typically by breadth-first search (enhanced with some heuristic) of state space that is known to be a NP-hard problem [Hong et al., 2002].

The search options of model checking tools have a significant influence on the performance of reachability search when the whole state space need not be traversed. For instance, traversal options such as depth first, breath first, random first etc are supported by the majority of model checkers. Optimization techniques used in model checking include also preprocessing of the model, for example, cone of influence reduction [Hamon et al., 2004a]. Instrumenting the model with trap variables is a standard technique used in prioritized traversal [Blom et al., 2005]. One step further is combining model checking with other methods using scriptable model checkers as reported in [Hamon et al., 2004a]. It is shown that combining different methods by scripting allows even a bounded model checker to reach deep states at low resource footprint.

The work presented in the current section takes a different approach by combining guiding of Uppaal Cora with iterated search refinement.

5.1.3 Case Studies

We use the following two case studies in the section: stopwatch [Hamon et al., 2004a] and a modified INRES protocol [Hogrefe, 1991].

Stopwatch.

In [Hamon et al., 2004a] it was claimed that explicit state model checkers are not suitable for finding test cases from models that have deep counter-dependent loops. Such a counter (in the range 0..6000) is present in the stopwatch example. Referring to our experiments with Uppaal Cora we show how guiding and iterated arch refinement improve test generation using explicit state model checking.

The stopwatch in [Hamon et al., 2004a] is modeled using Stateflow notation. In Figure 5.1 there is an equivalent UML state machine. For our experiments we used a flattened representation in Uppaal.

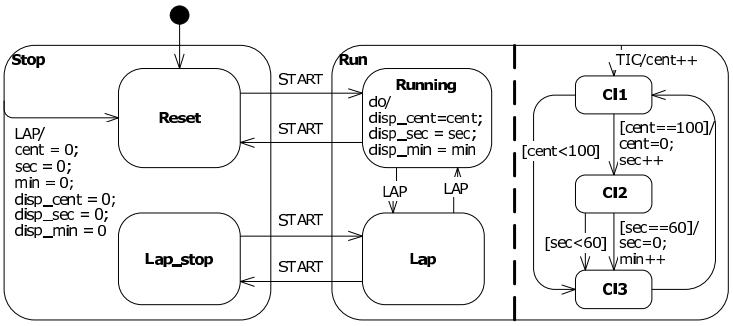


Figure 5.1: Stopwatch as UML state machine

Modified INRES Protocol.

INRES protocol is a well-known example in the model verification and test generation community. The protocol is simple but not trivial and provides a good reference for studying performance and scalability issues of competing methods. We use it to demonstrate the scalability of our test generation method. The case study shows that the generation of test sequences for ”all transition triples” test coverage results in very long test sequences. The protocol was introduced in [Hogrefe, 1991] and was modified in [Bourhfir, Dssouli, Aboulhamid and Rico, 1997] and is depicted in Figure 5.2 as an EFSM. We chose this particular model because it has several loops, for example, a self loop (at the `Sending` state) and a (minimally) two-step loop (`Sending`, `Blocked`, `Sending`), the depths of which depend on the input parameters `datarequest.n` and `datarequest.b` respectively.

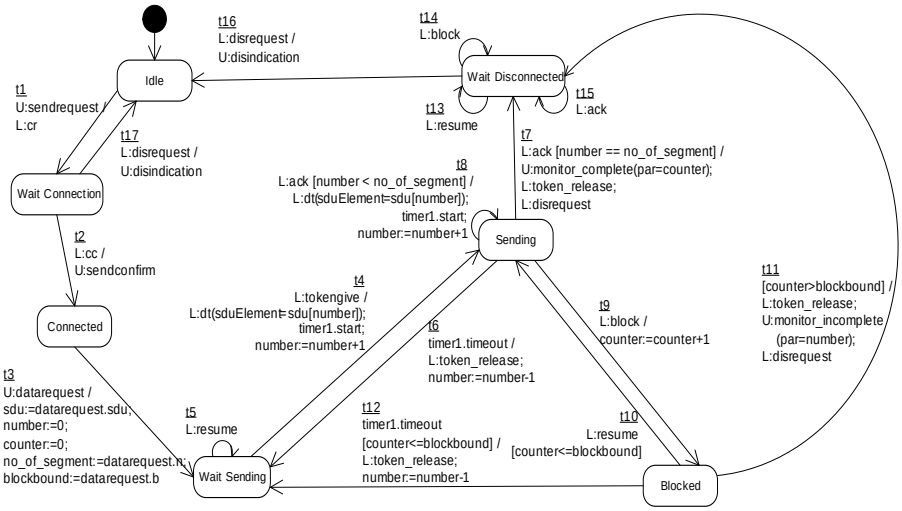


Figure 5.2: Modified version of the INRES protocol [Bourhfir et al., 1997].

5.1.4 Model Construction for Test Generation

An Extended Finite State Machine (EFSM) is a FSM extended with variables with finite domains, for example, Booleans and bounded integers. In addition to input and output, every transition may have a guard condition and assignments to variables. We assume that the EFSM of the initial IUT is deterministic and strongly connected.

The source EFSM that is given as a UML state machine is transformed into a Uppaal automaton in three steps. In the first step, the UML state machine is flattened and parallel states are sequentialized. The result is transformed to a Uppaal automaton in the second step. We are interested in finding a sequence of transitions that satisfies the selected structural coverage criterion in the model, thus the inputs and outputs of the model are abstracted away, so that only the information influencing the control flow of the Uppaal model is kept. Thus we reduce the search space in a way that makes trace generation by model checking feasible. In the last step the model is annotated with auxiliary variables to mark passing certain states or transitions. Such trap variable declarations, trap variable assignments and, additionally, cost functions are added to each transition according to the selected coverage criterion. After the generation of test sequence the inputs and outputs associated with each transition in the test sequence are reintroduced in the tester code generation step, which is beyond the scope of the current section.

As in [Hamon et al., 2004a], [Hong et al., 2002], [Mücke and Huhn, 2004], and [Hessel, Larsen, Nielsen, Pettersson and Skou, 2003], we encode the coverage criterion as a reachability problem using trap variables. For example, in the case of all transitions criterion, an initially false boolean trap variable t_i is added to the model for each transition and an assignment $t_i = true$ is added to each transition. A witness trace that passes all transitions at least once is generated by the model checker by checking reachability of the property $E\Diamond(t_1 \wedge t_2 \wedge ... \wedge t_n)$, where n is the number of transitions in the model. We extend this approach for k-switch [Chow, 1978] coverage criterion.

1-switch criterion requires that all pairs of consecutive transitions are covered by a test sequence at least once. For the construction of a reachability property corresponding to the 1-switch criterion we add trap variables $t_i t_j$ for each feasible transition pair (t_i, t_j). Trap variables $t_i t_j$ are initially set to false. To remember the previously visited transition an auxilary variable *prev* is declared. On each transition t_j a case statement is added for assigning 1-switch trap variables to true depending on the previously passed transition, in Figure 5.3 (left), where $t_{i1}, ..., t_{il}$ are incoming transitions to the source state of transition t_j. The property to be checked involves a conjunction of all feasible 1-switch trap variables $t_i t_j$: $E\Diamond \bigwedge_{i,j}(t_i t_j)$.

2-switch is a triple of consecutive transitions and a test satisfying *all 2-switches coverage* criterion passes all feasible transition triples. For transforming all 2-switches criterion to a reachability problem we add a trap variable $t_i t_j t_k$ for each feasible triple and auxilary variables `prev` and `befprev` to remember the previous and before-the-previous traversed transition, respectively. In Figure 5.3 (right) there is an example of a nested case statement that is added to each transition t_k for assigning 2-switch

```
select (prev) {
  case (prev==ti1) tiltj=true;
                   break;
  ...
  case (prev==til) tiltj=true;
}
```

```
select (prev) {
  case (prev==i1)
    select (befprev) {
      case (befprev==tj1)
                   tj1tiltk=true;
                   break;
      ...
      case (befprev==tjm)
                   tjmtiltk=true;
    }
  ...
  case (prev==til)
    select (befprev) {
      case (befprev==tj1)
                   tj1tiltk=true;
                   break;
      ...
      case (befprev==tjm)
                   tjmtiltk=true;
    }
}
```

Figure 5.3: Trap variable assignments for 1-switch for t_j (left) and 2-switch for t_k (right)

trap variables where $t_{j1},...,t_{jm}$ are incoming transitions to the source state of transition t_j. The property to be checked contains a conjunction of all feasible 2-switch trap variables $t_i t_j t_k$: $E\Diamond \bigwedge_{i,j,k}(t_i t_j t_k)$.

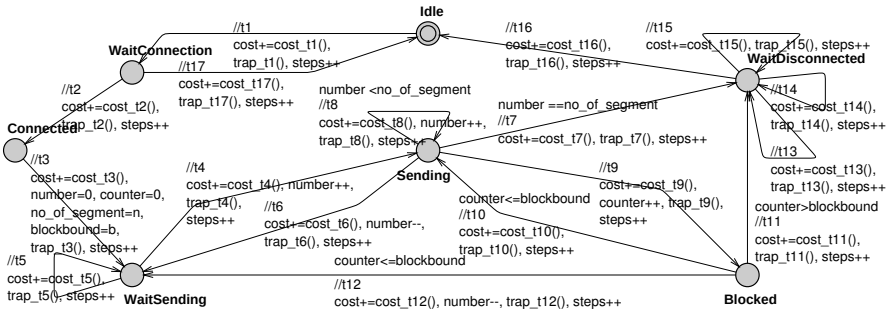


Figure 5.4: Uppaal model of the modified INRES protocol with traps and cost functions

In Figure 5.4 there is a Uppaal representation of the INRES model in Figure 5.2 for generating all 2-switch test sequence. In Figure 5.5 (left) there is an example of the relevant trap variable assignment function, where case statements are implemented in terms of if-then-else.

Uppaal Cora has support for guiding the reachability search with a built-in cost variable which can be used to minimise the lengths of generated test sequences. We define the cost variable assignment on each transition so that the cost increment is zero while the switch has not been passed (the trap variable of the switch is false) and increases the cost by a fixed penalty after it is set to true. In Figure 5.5 (right) there is an example of a cost function used in the experiments.

```
// 2-switch trap variable assignments
// procedure on the transition t1

void trap_t1() {
  if (prev==16) {
     if (befprev==7) t7t16t1=true;
     else if (befprev==11) t11t16t1=true;
     else if (befprev==13) t13t16t1=true;
     else if (befprev==14) t14t16t1=true;
     else if (befprev==15) t15t16t1=true;
  }
  else if (prev==17)
        if (befprev==1)
           t1t17t1=true;
  befprev=prev; prev=1;
}
```

```
// 2-switch cost function on
// the transition t1

int cost_t1() {
  if (prev==16 and (
     (befprev==7 and t7t16t1) or
     (befprev==11 and t11t16t1) or
     (befprev==13 and t13t16t1) or
     (befprev==14 and t14t16t1) or
     (befprev==15 and t15t16t1)
     )) return PENALTY;
  if (prev==17 and (
     (befprev==1 and t1t17t1)
     )) return PENALTY;
return 0;
}
```

Figure 5.5: Implementations of the trap assignment function (left) and cost assignment function (right)

5.1.5 Iterated Search Refinement for Test Generation

Model checking in general involves searching possibly very large state spaces to prove or disprove a query — a formula typically in some temporal logic. We make use of the feature of model checking to generate witness traces. We specify one test coverage criterion at a time as a reachability query.

We chose to use Uppaal Cora version 060206 because it enabled us to demonstrate the behaviour of regular search options and in addition the influence of guiding and iterated search refinement in the presented context of test generation using a single model format and thus avoiding influences to results that may be introduced by converting a model to several modelling formalisms.

Standard Search and Trace Generation Options of Model Checking

Standard search strategies typically used to traverse the state space are *depth first* and *breadth first*. The standard version of Uppaal implements both [Behrmann, David and Larsen, 2004] and additionally also a *random depth first* search strategy. Breadth first search looks for all reachable states at current search depth before proceeding deeper while depth first search takes one path and goes along it deeper until the property is satisfied or it needs to backtrack to look at alternative paths. Reachability queries considered in the current context do not in practice require full traversal of the state space if the property is satisfiable.

Trace generation options that Uppaal provides [Behrmann, David and Larsen, 2004] are for generating *some*, *shortest*, and *fastest* trace. Since we have currently omitted the use of clocks in our models, we do not use the latter option in the experiments.

Additional search strategies of guided model checking provided by Uppaal Cora are *best first*, *random best depth first*, and *smallest heur first* [Behrmann, Larsen and Rasmussen, 2004]. As we use only the *cost* variable for guiding, the latter search option is not used in the experiments.

An additional trace generation option of guided model checking provided by Uppaal Cora is *best* trace. This means that the trace generated has the lowest aggregate value of cost in the context of the search strategy used.

Iterated Search Refinement Using Bitstate Hashing

Bitstate hashing, also known as supertrace, is a well known method applied for model checking and thoroughly analysed in [Holzmann, 1998] for reducing memory consumption of the whole state space search by storing only a single bit for each seen state at the address calculated by a hash function. The drawback of the method is the possibility of hash collisions that will result in unexplored parts of the search space, rendering the method sound but incomplete. Still, fast reachability checks that yield a valid trace can be quite useful for applying model checking, for example, for test sequence generation from an EFSM model.

In general, the bigger the hash table, the lower the probability of hash collisions. But big hash tables may still require unavailable amounts of memory. Iterated search refinement is briefly mentioned in [Holzmann and Smith, 2000] and is based on the idea of iteratively increasing the size of the hash table and thus search thoroughness. We make use of the property of a division remainder based hash function to distribute hash collisions pseudorandomly as the divisor (the hash table size) is changed. Thus, the states considered similar by collisions change too. Since Uppaal uses a modulus based hash function [Bengtsson, 2002] for bitstate hashing, we use unmodified Uppaal Cora to compare the influences of different search options.

Basic Iterated Search Refinement, ISR, works as follows. There is a model M and a reachability query q. The bitstate hash table is initially set very small (for example 1 bit). The reachability of the query q is checked on model M. If a trace to the reachable state is not found then the bitstate hash table size is increased by 1. The hash table size is increased by small steps for some configurable number of times and then it is increased by some factor, for example 2. The small steps are necessary to try several different paths at each thoroughness level and big steps are to speed up finding the appropriate hash table size for the particular task. The minimal size of the bitstate hash table yielding a trace may differ by many orders of magnitude for different tasks. The bigger the hash table, the longer each iteration step takes.

Improvement of the first result gained in the basic approach is possible for some specific types of models. Let us assume that we look for a trace that is as short as possible and exhaustive search is not possible due to memory and/or processor time limits. Then we can iteratively constrain the reachability query by the trace length bound found in the previous step. In such an approach there is no clear criterion when to stop, as we cannot be sure if the result gained at some iteration step is actually the shortest possible. The most important criterion is the amount of time we have to wait for an improved result.

Combining ISR with guiding is a very important aspect in the current approach. Namely, the shape of the reachable search space of a model given a bitstate hash table size is dependent on search strategy, as the state hashing to some address in the

bitstate hash table is traversed only during the first visit and the next states hashing to the same value are already considered seen.

5.1.6 Comparison of Search Strategies for Test Generation

In this section we present a comparison of different search strategies and trace generation options that can be used in model checking for test sequence derivation. The experiments are run on an EFSM represented as a Uppaal model of the stopwatch example described in Section 5.1.3. All experiments described in this section were run on a 2.4 GHz Xeon processor with 512 kB of cache, 533 MHz FSB and 6 GB of 266 MHz DDR memory.

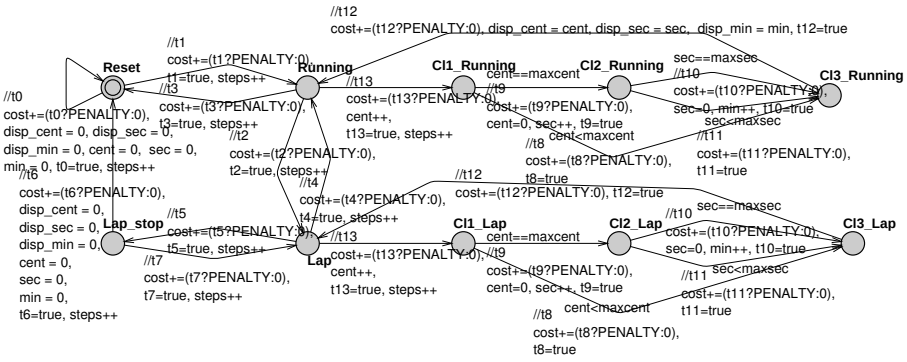


Figure 5.6: Uppaal model of the stopwatch with trap variables and cost assignments.

The Uppaal model of the stopwatch is presented in Figure 5.6. The model is decorated with trap variables that are used for finding a trace that passes through all transitions. The transitions are labelled by the names of trap variables, for example `//t0`. To make the comparison of all available search options possible, the model is optimised by declaring variables `min`, `disp_cent`, `disp_sec`, `disp_min`, and `steps` as *hidden* (*meta*), meaning that the states where only the values of such *hidden* variables are different are considered equivalent by the model checker. The `steps` variable is used for capturing the length of the trace.

The model is also decorated with assignments to a special purpose built-in `cost` variable which is used for guiding the model checker.

In Table 5.1 there are experimental results of applying Breadth First (BF), Depth First (DF) and Random Depth First (RDF) search strategies on the model in Figure 5.6 with the goal of covering all transitions (equivalent to all trap variables `t0`...`t13` becoming true). The trace generation option is set to *some* because setting it to *best* caused the model checker to run out of memory (3GB per process due to 32 bit architecture). Breadth first search did not yield an answer without declaring some of the integer variables to be *hidden*. We can see that depth first search yielded an answer quickly but the trace is 5 times longer than the minimal, which is 6011 steps

Table 5.1: Test sequence lengths found using different search options for the model without guiding

Search order	Trace	No. of steps	Time [sec]	Memory [MB]
BF	some	6012	21	146
DF	some	30009	52	45
RDF	some	8988	12	12

Table 5.2: Test sequence lengths found using ISR and model without guiding (first trace found)

Search order	Trace	No. of steps	Time [sec]	Mem. [MB]	Hash table [Mbit]
BF	some	6141	276	48	44
DF	some	6137	106	11	1
RDF	some	9000	80	11	3

in length. Random depth first search yielded a better answer than regular depth first.

In Table 5.2 there are results for applying the iterated search refinement with the same search strategies. The figures show that using breadth first search consumes considerably more memory and requires considerably more time to find an answer than depth first and random depth first search. By comparing the results in Table 5.1 and in Table 5.2, we can see that the result obtained by depth first search using ISR is considerably shorter.

But can we improve these results? Intuitively, if we could guide the search, we should find a shorter trace sooner.

First, we add cost assignments to all transitions that are equipped with trap variables. The cost assignments `cost+=(trap?PENALTY:0)` are C style assignments, meaning that `PENALTY` is added to cost only when the corresponding trap variable has already become true before evaluating the assignment.

Table 5.3 summarizes the results of applying Uppaal Cora with Best First (BeF) and Random Best Depth First (RBDF) search. One can see that best first strategy yields the optimal answer but requires a considerable amount of memory for this rather small example. In fact, the result is very close to breadth first search in the model without guiding. Random best depth first did not yield an answer at all due to running out of memory.

Next we combine guiding and ISR. The results of running ISR with cost assign-

Table 5.3: Test sequence lengths found using guiding with cost variable definition

Search order	Trace	No. of steps	Time [sec]	Memory [MB]
BeF	best	6011	22	147
RBDF	best	N/A	2230	out of memory

Table 5.4: Test sequence lengths found using ISR and a model with cost assignments on all transitions (first trace found)

Search order	Trace	No. of steps	Time [sec]	Mem. [MB]	Hash table [Mbit]
BeF	some	6302	6063	622	1408
BeF	best	6155	5837	628	1408
RBDF	some	8508	138	17	3
RBDF	best	8505	138	21	3

Table 5.5: Test sequence lengths found by Uppaal Cora using ISR and model with guiding and loop entry optimisations

Search order	Trace	No. of steps	Time [sec]	Mem. [MB]	Hash table [Mbit]
First trace found					
BeF	some	6226	5745	625	1408
BeF	best	6254	5485	599	1408
RBDF	some	7279	259	12	5
RBDF	best	6714	286	27	5
Shortest trace found before system memory or hash table overflow					
BeF	some	6151	7431	628	1408
BeF	best	6133	8059	599	1408
RBDF	some	6011	3810	265	176
RBDF	best	6011	3515	310	176

ments on every transition are presented in Table 5.4. The results show that using the best first search strategy combined with ISR produces considerably worse results than breadth first search. Random best depth first search gives interesting results that are comparable to random depth first search in the uniterated case (Table 5.1) and to depth first and random depth first in the iterated case without guiding (Table 5.2).

The results are not significantly improved. Can we tune guiding for better results?

We tune the model by removing the cost of taking entry transitions to loops where counters are incremented, for example transition `t13` in Figure 5.6. In this way we relieve multiple entries to loops from penalties and thus make the model checker choose such transitions more often. This requires an extra analysis of the model which is currently not automated.

The results of running the ISR on the tuned guided model are presented in Table 5.5. The first results obtained by the ISR algorithm by random best depth first search with either some or best trace generation option are significantly better than in the previous case. Additionally, if the iteration is continued, the actual optimum is also reachable by ISR (the lower half of Table 5.5.). The drawback of ISR is that there is no indication how far the current result is from the optimal value.

We presented a comparison of different search strategies on a relatively small and optimised example. In the next section we look at how depth first search without iteration, depth first search with iteration and random best depth first with best trace

Table 5.6: Combinations of search options used for the INRES case study

Abbreviation	ISR	Search order	Trace	Guiding
DF	-	depth first	some	-
IterDF	first result	depth first	some	-
IterRBDF	first result	random best depth first	best	uniform
IterRBDF tuned	first result	random best depth first	best	tuned

generation option behave on a larger example. These options are chosen because these have low memory footprint and yield relatively good results and thus have the potential to be scalable.

5.1.7 Scalability of ISR and Guiding for Test Generation

The modified INRES protocol in Figure 5.2 contains a self-loop where a variable is incremented (transition `t8`) and several cycles of two or more transitions (for example, a variable is incremented in the cycle containing `t9` and `t10`). The test sequence length depends on the parameters n and b defining the upper limits of loop counters. A manually obtained estimation of the shortest length of all 2-switch test sequence can be given as $352 + 15n + 26b$, when $n \geq 5$ and $b \geq 3$.

Next we present the results of searching for all 2-switch test sequences in the model in Figure 5.2 using options listed in Table 5.6. The results that are obtained using random best depth first search and ISR are average values of 3 runs. While the first value found can vary considerably in different runs, the value obtained by refining the initial result for some proportional amount of time converges fast. *Uniform* guiding means that all trap variables are associated with similar cost and *tuned* guiding means that the cost functions have been modified not to penalize for entering the loops where counters are incremented, i.e. consequent incrementations of the counters is favoured.

The trace lengths of all 2-switch test sequences generated with different search options are given in Figure 5.7. *Estim.* stands for the estimated value. The line representing DF search ends at $n = 300$ on the rightmost diagram because the model checker ran out of memory. We see that the iterated approach scales with all selected combinations of options for larger models than the depth first search. Tuned guiding yields traces that are quite close to the estimated shortest.

The maximum amount of memory that was required to generate the traces is given in Figure 5.8. We can see that DF search takes little memory in the case where counters are shallow (the diagram on the left) but the amount of required memory increases rapidly when the counters become deeper (the diagram on the right). The iterated approach requires much less memory than plain DF search.

The time it took to generate the traces is given in Figure 5.9. We can see that the gain in memory and shorter trace lengths is paid for with processor time. The iterative approach takes generally much longer than depth first search. This problem can be relieved by running the iterations on multiple processors in parallel as each iteration

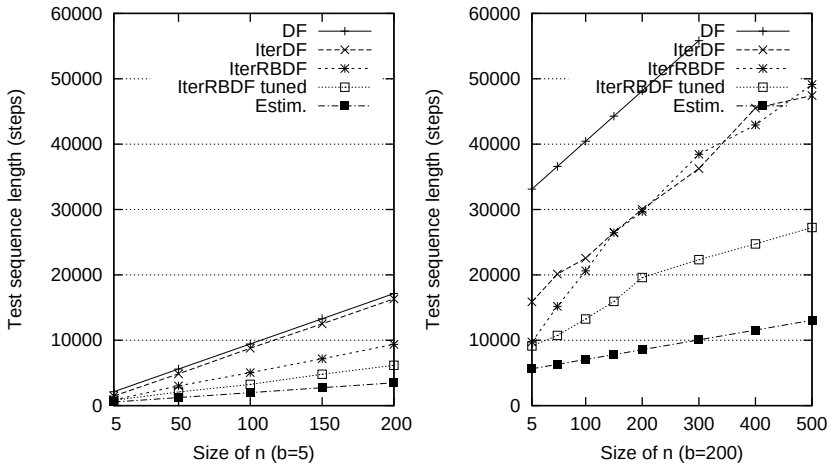


Figure 5.7: Lengths of sequences in the INRES model for the 2-switch coverage criterion

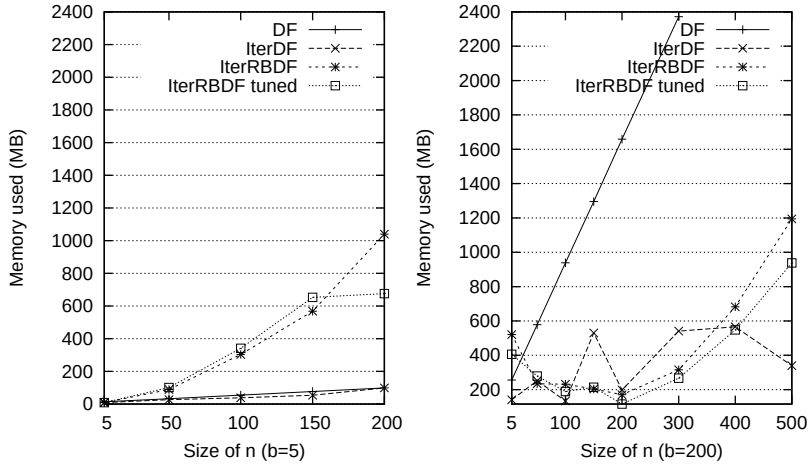


Figure 5.8: Memory required to find sequences for the 2-switch coverage criterion

is independent. In addition, in most cases, it is acceptable to wait for more than just a few seconds for a test sequence satisfying some stronger structural coverage criterion.

5.1.8 Conclusion and Discussion

We presented a way to build Uppaal models from EFSM models to generate test sequences covering some structural criteria, for example all transitions, all transition pairs and all transition triples. We conducted a comparison of different search strategies on a stopwatch model. The comparison confirmed what has previously been stated in the literature, that explicit state model checking does not scale well for test sequence generation purpose: breadth first search, which would yield a short

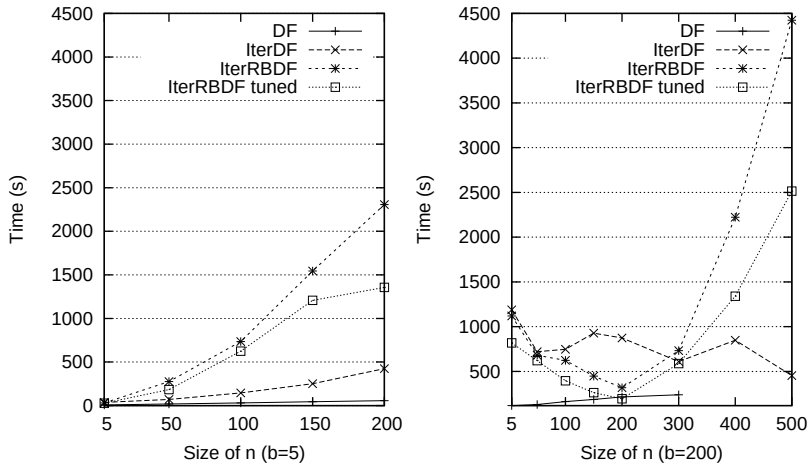


Figure 5.9: Time spent for finding sequences for the 2-switch coverage criterion

sequence, runs out of memory with quite simple models and depth first search produces very long sequences while consuming large amounts of memory as the model becomes more complex. A bitstate hashing based iterated search refinement method for checking reachability proved to be more scalable on unmodified models for test generation than the traditional search strategies used in model checking. Additionally, extending the EFSM model with guiding cost expressions yielded better results in terms of sequence length. Some tuning of the cost expressions further improved the results. Thus, we have shown how the lengths of test sequences generated using explicit state model checking can be improved by combining guiding and iterated search refinement.

5.2 Memory Arbiter Synthesis for a Radar Memory Interface Card

5.2.1 Introduction

In this section we analyse a radar system memory interface card described in a case study from the IST Advanced Methods for Timed Systems project, [AMETIST, 2002–2005]. It was contributed to the project by Terma A/S [Behrmann, Bernicot, Hune, Larsen, Lecamp and Skou, 2002]. The memory interface card performs signal processing calculations on two streams of input data and their delayed counterparts. The stream data is temporarily stored in synchronous dynamic RAM (SDRAM) that is shared by all streams. Dynamic memory is generally considerably less costly in larger amounts than static memory which is used for intermediate buffers.

We present a way to *synthesise* a memory arbiter for the system in a way that minimises the amount of static RAM used for buffering the streams. In addition, we *verify* that the resulting arbiter indeed does not deadlock and never starves nor overflows any of the intermediate buffers. Both the synthesis and the verification problem are solved by *model checking*.

The synthesis task is generally computationally harder than verification, thus we need to apply a number of abstractions to the system to make the task solvable by model checking. Still, even with the manually abstracted model, full state space search requires unavailable amounts of memory. The synthesis part succeeds while consuming modest memory resources when we use the iterated search refinement with bitstate pruning.

Model checking

In this section we specify our questions to the system as reachability queries. We apply the approach for establishing the existence of a schedule for the system and for establishing the correctness of the resulting schedule. We apply model checking for two distinct purposes—synthesis and verification. In the case of synthesis we set up the problem in such a way that a positive answer to the reachability question gives us a trace from which we extract the schedule for the memory arbiter. Due to memory limitations we bound the depth of the search.

Thus, after we have found a recurring cycle in the system by using a depth bounded search, we build a system based on the schedule gained in the previous stem and verify it without bounds to see whether it behaves expectedly.

Choice of the Model Checking Tool

There is a variety of different implementations of model checkers available. We chose Uppaal [Amnell et al., 2001] because it enabled us to

- Model the current case study conveniently using the Uppaal extended timed automata formalism;

- Leverage an implementation of bit-state hashing symbolic state space representation feature built into Uppaal;
- Leverage the uniformly priced timed automata extension of Uppaal with only minor modifications to the model.

The latter extension is available in a recently released version of extended Uppaal — Uppaal CORA [Behrmann, Larsen and Rasmussen, 2004].

Outline

The rest of the section is organised as follows. Section 5.2.2 gives an overview of related work. Section 5.2.3 introduces the radar memory interface board. In Section 5.2.4 we describe a set of abstractions to tailor the model to arbiter synthesis for the memory interface board. In Section 5.2.5 we give an overview of the steps of synthesising the arbiter for the shared memory bus and of verification of the resultant system containing the synthesised arbiter.

5.2.2 Related Work

This case study has previously been analysed by [Weiss, 2002]. The solution described in the current section differs from the former solution described in the following aspects:

- The one presented by [Weiss, 2002] involves using SMV [McMillan, 1999]. The current solution uses Uppaal [Pettersson and Larsen., 2000] as the model checker;
- In [Weiss, 2002] the schedule for the example is reached using a parameterised model as model checking the full system was infeasible due to state space explosion. In the current approach, the Uppaal model of the component system contains abstractions that enable the schedule to be synthesised for the relevant aspect of the whole system.

Uppaal has been previously applied for batch plant scheduling by [Hune et al., 2001]. The approach presented therein is similar to the work presented here in the sense that they use model checking for finding a valid schedule. In addition, they leverage bit-state hashing to reduce memory consumption of the model checking task. Our work differs from the latter in that we model the system as a synchronous system and the state of the target system is represented in terms of integers. We look for a suitable ordering of such states. The application domain is also different — job shop scheduling versus hardware analysis. We model the state of the target system in terms of integer variables which are updated on one transition by a nontrivial update, thus disregarding uninteresting interleaving.

[Goel and Lee, 2000] present a case study based on IBM CoreConnect^{TM} processor local bus arbiter core. The case study presented therein is similar to current

problem. The authors of the paper call for potential solutions for analysing the arbiter core.

[Amnell, Fersman, Pettersson, Yi and Sun, 2002] present a way to synthesise code for LEGO RCX bricks. The approach has more emphasis on the timed aspect of target systems.

5.2.3 Radar Memory Interface Card

As was mentioned above, we use a case study from the IST AMETIST project by [Behrmann et al., 2002]. The case study was provided to the project by Terma who is producing radar sensors mainly used for traffic control in ports and airports and for coastal surveillance. Some configurations of their radar sensor systems employ a technique known as *frequency diversity*. In this mode, two subsequent pulses that differ slightly in frequency are emitted right after each other from the antenna. As usually, the echo (in this case of two signals) is received, but due to the characteristics of the antenna, the signals got propagated in slightly different directions, and therefore the two simultaneously received signals do not correspond to exactly the same direction. To align the signals, one of the signals has to be delayed. This approach has two immediate benefits: it increases the output power of the radar for the purpose of better range and more reliable signal as two pulses are emitted and provides *time diversity*, i.e., makes it possible to compare the echoes of two signals from one direction at two slightly different subsequent moments for distinguishing, for example, a big wave from a small boat.

To remove noise, another technique, known as *sweep integration*, is used. The idea is to integrate the signal at the same direction from multiple sweeps. In the case of frequency diversity, sweep integration is performed on both return signals before the signals are combined.

In total, the signal processing board serves four purposes: sweep integration, frequency diversity combination, noise cancellation and a kind of differentiation (high pass filtering). The board uses dynamic synchronous RAM (SDRAM) for intermediate storage of the two input streams and their processed counterparts.

The signal processing board consists of signal processing units, SDRAM connected by a shared memory bus interfaced by 9 FIFO (First In First Out) buffers and governed by an arbiter which is responsible for setting up communication between memory and one FIFO at a time so that none of the buffers is neither starved nor overflowed. A block diagram of the system is presented in Figure 5.10 ([Behrmann et al., 2002]).

Figure 5.11 represents the internal structure of the 9 FIFO buffers in the context of their surroundings. The size of the buffers range from 512 bytes to 2048 bytes (2KBytes) and they are implemented as ring buffers. The buffers that mediate data streams *to* SDRAM are called *input buffers* and, that mediate data streams *from* SDRAM are called *output buffers*.

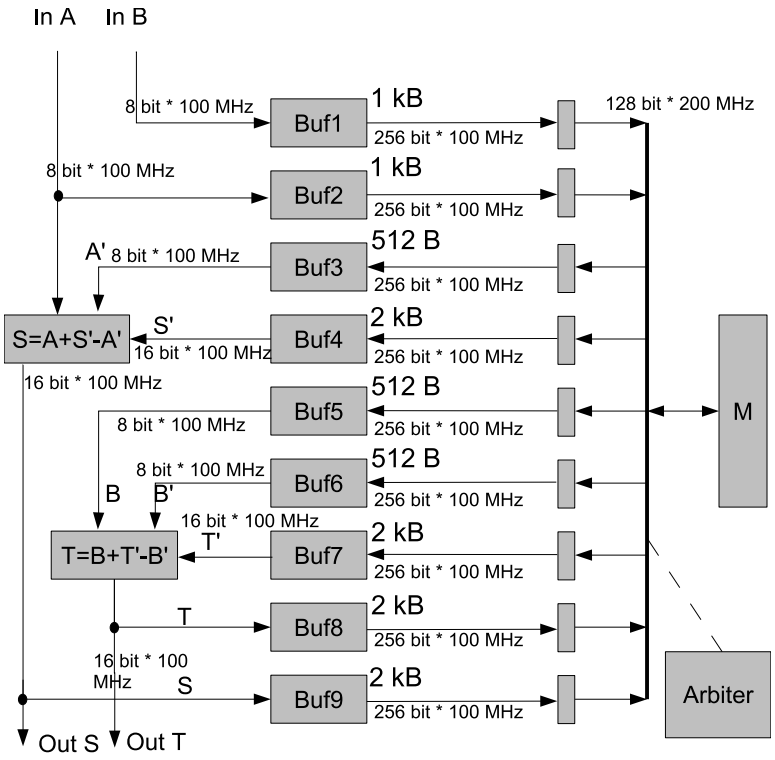


Figure 5.10: Radar memory interface card.

Observations of the system

1. The smallest quantity of data that can be moved in the system is 1 byte. For example, inputs A and B in Figure 5.10.

2. Data is written at fixed rate in 1 byte or 2 byte quantities (Fig. 5.11, ***a***). It is assumed that uninterrupted data flow of 1 byte at the frequency of 100 MHz is fed into the inputs A and B. Equivalently, it is assumed that outputs S and T can always be written to at the rate of 2 bytes at 100 MHz.

3. Data is read from the ring buffer into the register in 4 byte quantities (Fig. 5.11, ***b***) whenever the two registers are not full and there are at least 4 bytes in the buffer at the beginning of a system clock cycle. The duration of this transfer is one clock cycle. If there is less data in the buffer or the registers are full, no data is transferred. (This works vice versa in the case of output buffers. Data is written from the register to the buffer whenever the register is not empty and there is at least 4 bytes worth of space in the ring buffer).

4. On the memory bus (Fig. 5.11, ***c***) data is always transferred when the register is full, i.e. in quantities of 512 bits = 64 bytes. (In the case of output buffers, data is transferred whenever the register is empty).

5. There is a multiplexer on the bus from the register to SDRAM as the memory bus runs in double data rate (DDR) mode and, is 128 bits wide but the register outputs 256 bits at 100 MHz. The DDR mode is achieved by transferring data

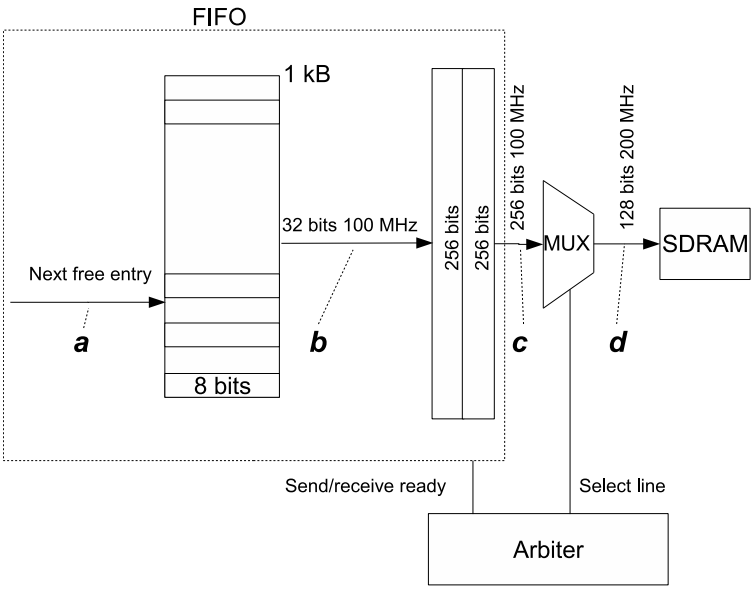


Figure 5.11: The structure of a FIFO buffer in the context of its surroundings.

between memory and the multiplexer on both the rising and the descending edge of a clock cycle.

6. Whenever an output register is full, the *send* signal is set. Whenever an input register is empty, the *receive* signal is set.

Arbiter

The arbiter is responsible for setting up connections between SDRAM and registers that are ready for communication. The arbiter, that is presented in the case study, checks the *send/receive* signals of the buffers in a round-robin way. When a buffer is ready, a *select line* signal is issued by the arbiter.

1. Whenever an output register is full, the *send* signal is set (Fig. 5.11). Whenever an input register is empty, the *receive* signal is set.

2. The calculation of the memory addresses for memory communication is not considered in the current approach.

3. Transferring 64 bytes between the registers and SDRAM takes two additional system clock cycles, making it a total of 4 cycles.

4. SDRAM needs to be refreshed every 15625 ns. The refresh takes 100 ns (10 system clock cycles).

The aim of the original case study was to verify that the behaviour of the round-robin scheduling algorithm is correct. A further step would be to synthesise a scheduler for a set of buffers.

We modify the original goal slightly and seek solution to the following problems:

1. Check that no input buffers are full at the beginning of a clock cycle.

2. Check that no output buffers are empty at the beginning of a clock cycle.

3. Check that the system does not deadlock.

4. Synthesise an arbiter for the memory bus that guarantees that the above properties are always satisfied.

5. Find the smallest possible buffer values for the arbiter synthesised.

6. Check that the schedule guarantees that no data transfers are interrupted by a memory refresh.

Table 5.7: Data rates of the data streams. (A, ..., T' denote corresponding streams in Figure 5.10. ***b*** denotes the corresponding stream in Figure 5.11.)

Data stream	A	A'	B	B'	S	S'	T	T'	***b***
Bus width (bytes)	1	1	1	1	2	2	2	2	4
Frequency (MHz)	100	100	100	100	100	100	100	100	100
Abstract bus width (bytes)	4	4	4	4	8	8	8	8	32
Abstrtract frequency (MHz)	25	25	25	25	25	25	25	25	25
Rate (MByte/s)	100	100	100	100	200	200	200	200	400

5.2.4 Construction of the Abstract Model

In this section we summarise the decisions taken during the process of modelling the memory interface board. We are interested in one specific aspect of the behaviour of the board, namely the *control behaviour of the memory arbiter*, and thus we disregard all detail that we manage to classify as not directly relevant. We model the system in terms of Uppaal modelling language [Bengtsson and Yi, 2004], which corresponds to finite state automata extended with integer variables and clocks.

The most significant observation in approaching the memory card is that *the system is fully synchronous meaning that all events are aligned to the system clock ticks*. Memory refresh, as specified by Observation 4 in Section 5.2.3, is the only exception to this rule. Keeping the synchronous nature of the system in mind enables us to reduce the number of intermediate states that should be distinguished in automatic analysis.

The first step in the current modelling approach is to choose the aspect of the system to focus on. We assume that the system is modelled in terms of some other (more or less formal) language, for example some block diagram language as in Figure 5.10. If the system is sufficiently specified it is possible for the engineer to point to some part of the system and ask for assistance there. We assume that the engineer pointed to the shared memory bus and asked to remove nondeterminism from the model or, in other words, synthesise an arbiter.

Memory bus

As said, we pay special attention to the memory bus as the arbiter of the memory bus is what we are specifically interested in. We observe closely how the other components of the system interact with the memory bus. The properties of the memory bus determine the parameters of our model, such as, for example, time and data granularity.

Observation 3 in Section 5.2.3 refers to that data is transferred in bursts along the memory bus and that it takes 4 system clock cycles per burst. Thus we align our abstract system clock to the bursts on the memory bus.

Let us consider the main characteristics of the data streams, when dividing the system clock frequency by 4. The properties of the data streams are summarised in Table 5.7. As the data rates should stay constant, we abstract the busses by widening

them proportionally to the factor by which we reduced the clock frequency. The effect of this modification will be discussed in detail below.

Now let us look at how the implementation details of the memory bus affect this approach. The data transfer from a register to memory and vice versa is performed on a 16 byte wide 200 MHz bus in the quantities of 64 bytes (Observation 4). As mentioned above, it takes 4 clock cycles to complete the transfer, so the memory-buffer data rate can be considered to be $64/4 = 16$ bytes/cycle $= 1600$ Mbytes/sec.

It is easy to estimate the solvability of the task in this case by the following simple sanity check:

$$throughput_{mb} - refresh \geq \sum_{streams} throughput_{stream}, \tag{5.1}$$

where $throughput_{mb}$ is the (abstract) data rate of the memory bus (1600 Mbytes/sec), *refresh* is bandwidth lost by staying in refresh state (100 ns / 15625 ns $\times throughput_{mb}$), and the right hand side is the aggregate throughput of the data streams.

We assume that there is enough stream data stored in the SDRAM, so that we can always initiate a burst from some memory location to an output register when the latter is empty and vice versa for the input register. This assumption allows us to concentrate on the data levels of the buffers and registers and not to model the double data rate behaviour of the dynamic memory (Figure 5.11, bus ***d***).

We will model the behaviour of the system in terms of data levels in the buffers and the registers. Under the above assumption we do not need to model data levels in memory. We model the data levels as values of integer variables in Uppaal automata.

Communication between Inputs, Outputs, and Buffers

We now turn to how the buffers and signal processing units behave from the point of view of the memory bus.

We observe that all of the input-adder, input-buffer, adder-output, adder-buffer, buffer-adder (Figure 5.10) communication occurs in constant streams. We can model a constant flow by adding or subtracting a constant value to the integer variable representing a particular buffer at every (coarse) clock tick. Thus we can omit modelling the adders altogether.

Let us now have a look at how the interconnection of buffers and registers behaves. For example, take data transfer from buffer to register (Figure 5.11, ***b***). The width of the buffer to register bus is 4 bytes. In one abstract clock cycle (four concrete cycles) this amounts to 16 bytes. If the data level in the register is less than the allowed maximum of 64 bytes, data is written from the buffer to the register. Such analysis is repeated for all buffer-register pairs.

Figure 5.12 gives an overview of the effect of the widening of the data paths that happens due to the coarsened clock. In the case of an input buffer, the buffer is considered empty until it is possible to read a whole coarse unit of data from it. In the case of output buffers, the buffer must have space for a whole coarse unit of data before data can be written to it.

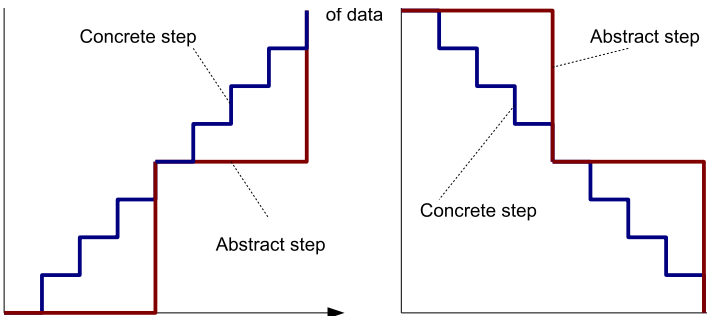


Figure 5.12: Abstractions of the data flow.

Due to synchronicity we are not interested in the interleaving between the filling and emptying of independent buffers and registers. Therefore all such interleaving is discarded by performing the updates of all registers and buffers as an update of a single transition of the automata model.

Assembly of the Model

The resultant model consists of three automata:

- The automaton which is responsible for updating the states of buffers and registers is called *Buffers*.
- The automaton that simulates the behaviour of memory (its need for refreshes) is called *MemoryRefresh*.
- The automaton for generating clock ticks is called *Clock*.

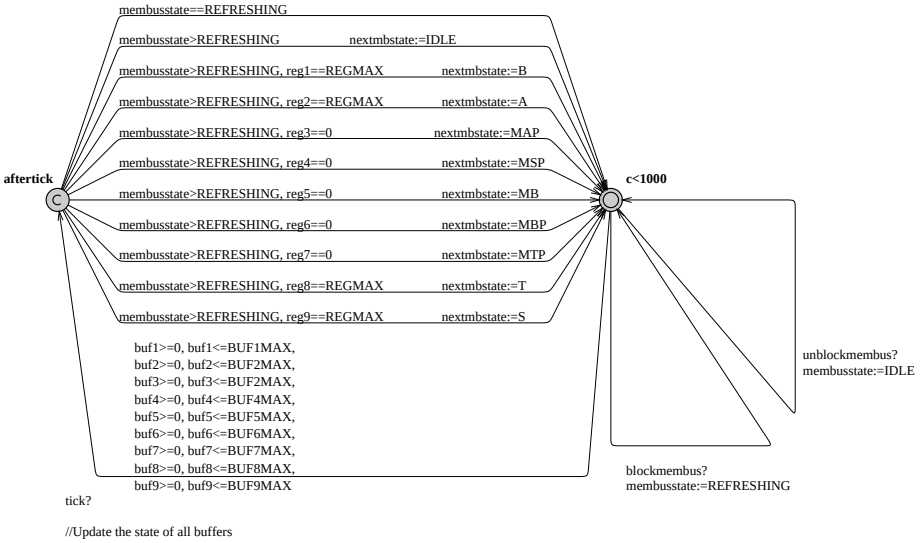
In the next section we describe how the synthesis and verification problems are approached and what additional modifications are needed to the model described thus far.

5.2.5 Arbiter Synthesis and Verification

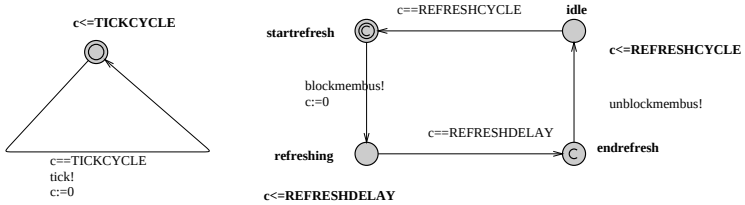
For the purpose of arbiter synthesis we create a conservative model of the system. By conservative we mean that we allow only valid behaviours of the system. We explicitly restrict the starvation and overflow of any of the buffers by introducing relevant guards.

The model in Figure 5.13 has the following characteristics:

- It consists of three concurrent automata : *Buffers*, *Clock*, and *MemoryRefresh*. *Buffers* models the states of the buffers and registers, *Clock* provides system clock ticks and *MemoryRefresh* models the need of SDRAM to be refreshed periodically.



Buffers



Clock *MemoryRefresh*

Figure 5.13: The Uppaal model for schedule synthesis consists of three automata: Buffers, Clock and MemoryRefresh.

- The state of the buffers and registers is represented by integers in the *Buffers* automaton.
- To model synchronicity, all variables representing buffers and registers are updated on one state transition, thus discarding a great deal of interleaving that is irrelevant in the current context. There is a separate variable representing the behaviour of the arbiter that controls which register can access memory at a time.
- The granularity of clock ticks is aligned with the duration of a transfer on the shared bus. The buffers and registers are updated by relevant multiples of bytes at the beginning of each such abstract tick.

For each buffer-to-memory and memory-to-buffer communication there is a transition in the *Buffers* automaton that sets up relevant communication in the next clock cycle. In addition, there is a transition for memory update and idling.

The update of the states of the buffers and registers is performed by an update presented in Figure 5.14. The semantics is that the update is taken at the end of the abstract clock cycle, to make sure that no memory refresh would invalidate a data burst.

Memory refreshes are triggered by the *MemoryRefresh* automaton.

Schedule synthesis

The current solution builds on the Uppaal model checker and its bit-state hashing implementation of symbolic representation of state space described by [Bengtsson, 2002]. Bit-state hashing is a memory consumption reduction technique that is applied to finding schedules. A clock variable that is never reset and is checked in an invariant ($c < 1000$ ns) in a state of the *Buffers* automaton (Figure 5.13) is used for bounding the depth of the search.

Specification of the Synthesis Property

The schedule is found using a reachability query given in Figure 5.15.

The query poses the following question to the system: Does there exist a state, apart from the initial state, along some path of computation, where the sum of data in each corresponding register and buffer equals to the initial sum? It is important to note that memory refresh is triggered at first clock cycle, meaning, that the memory bus is blocked at start. The amounts of data specified in the query are set to a quarter of the capacity of each buffer (we have reduced the buffer sizes by a factor of two). Notice that the specified reachability property does not hold along the path to the desired state. To make sure that only valid paths are explored we incorporated the path property (no buffers can be starved nor overflowed) into the guards of the synthesis model. This reduces the reachable state space.

```
//Update the state of all buffers
//first we need to set up help variables:
// (for input buffers)
buf1help=((reg1<=(REGMAX-BUF1OUT))&&(buf1>=BUF1OUT)?1:0),
// (for output buffers)
buf3help=((reg3>=BUF3IN)?1:0),
// (for memory)
// next memory transaction:
// input buffers
reg1idle=((nextmbstate==1)&&(reg1==REGMAX)&&(membusstate>REFRESHING)?0:1),
// output buffers
reg3idle=((nextmbstate==3)&&(reg3==0)&&(membusstate>REFRESHING)?0:1),
//
//
//input buffers
//
buf1=(buf1help?buf1+BUF1IN-(reg1idle*BUF1OUT):buf1+BUF1IN),
reg1=(buf1help?reg1+(reg1idle*BUF1OUT):reg1),
//
//
// output buffers
//
buf3=(buf3help?buf3-BUF3OUT+(reg3idle*BUF3IN):buf3-BUF3OUT),
reg3=(buf3help?reg3-(reg3idle*BUF3IN):reg3),
//
// Memory
//
membusstate=(membusstate>REFRESHING?0:membusstate),
//
membusstate=(reg1idle==0&&membusstate>REFRESHING?1:membusstate),
membusstate=(reg3idle==0&&membusstate>REFRESHING?3:membusstate),
memB=(reg1idle==0&&membusstate==1?memB+REGMAX:memB),
memA=(reg3idle==0&&membusstate==3?memA-REGMAX:memA),
//trasfer to and from registers
reg1=(reg1idle==0&&membusstate==1?0:reg1),
reg3=(reg3idle==0&&membusstate==3?REGMAX:reg3),
//
// clear help variables
reg1idle=1,
reg3idle=1,
//clean up:
buf1help=0,
buf3help=0,
initcomplete=(membusstate>REFRESHING?1:initcomplete)
```

Figure 5.14: The update on the lower transition of the *Buffers* automaton that updates the states of registers and buffers. (The actual updates are only shown for one input buffer and register (buf1, reg1), and one output buffer and register (buf3, reg3).)

```
E<> Buffers.buf3+Buffers.reg3==128 and
    Buffers.buf5+Buffers.reg5==128 and
    Buffers.buf6+Buffers.reg6==128 and
    Buffers.buf1+Buffers.reg1==64 and
    Buffers.buf2+Buffers.reg2==64 and
    Buffers.buf4+Buffers.reg4==256 and
    Buffers.buf7+Buffers.reg7==256 and
    Buffers.buf8+Buffers.reg8==256 and
    Buffers.buf9+Buffers.reg9==256 and
    Buffers.initcomplete==1
```

Figure 5.15: The query used for finding the schedule for the memory arbiter.

Table 5.8: Two synthesised schedules resulting from depth first search and cost optimally guided search.

Time (ns)	40	80	120	160	200	240	280	320
Depth first schedule	-1	-1	0	7	9	8	4	2
Cost optimal schedule	-1	-1	0	4	7	2	1	8
Time (ns)	360	400	440	480	520	560	600	640
Depth first schedule	7	9	8	6	5	4	1	3
Cost optimal schedule	9	4	7	6	3	5	8	9

The schedule is produced as a sequence of memory bus states indicated by the MemBusState variable. Examples of schedules can be found in Table 5.8. The numbers indicate corresponding numbers of registers that communicate with memory in each 40 ns time slot (an abstract clock cycle). Number 0 stands for either an idle state of the memory bus or a state where the memory bus becomes idle by the end of a clock cycle. Number -1 stands for memory refreshing state. The data should be interpreted in the following way: memory bus is blocked due to refresh (-1) at the end of the first 40 ns. It is also blocked during the next 40 ns ending with the 80th nanosecond. The bus becomes idle by the end of the third abstract clock cycle (120th ns). During the period 120 ns to 160 ns 64 bytes are transferred from the memory to register number 7 (in the case of the depth first schedule). Etc.

Buffer Memory Minimisation

Uppaal CORA [Behrmann, 2005] is a tool that contains two different extensions to the Uppaal timed automata formalism. One extension is called Linearly Priced Timed Automata (LPTA) and the other Uniformly Priced Timed Automata (UPTA) [Behrmann and Fehnker, 2001]. In this example we make use of the UPTA extension by specifying a variable `cost` that is increased monotonously according to the increase in the range of buffer memory used. The cost value is the sum of differences of the maxima and minima of the amount of data in buffers. The appropriate modifications to the model that are shown in Figure 5.16.

A schedule derived using the cost guided model is presented in Table 5.8. The difference between the behaviours of resultant schedules in terms of buffer sizes is visualised in Figure 5.17. Attention should be paid to the difference between the maxima and minima of corresponding data levels in buffers. This suggests that the

```
// Input buffers
//
buf1=(buf1help?buf1+BUF1IN-(reg1idle*BUF1OUT):buf1+BUF1IN),
cost+=(buf1>buf1max?1:0),
buf1max=(buf1>buf1max?buf1:buf1max),
cost+=(buf1<buf1min?1:0),
buf1min=(buf1<buf1min?buf1:buf1min),
reg1=(buf1help?reg1+(reg1idle*BUF1OUT):reg1),

// Output buffers
//
buf3=(buf3help?buf3-BUF3OUT+(reg3idle*BUF3IN):buf3-BUF3OUT),
cost+=(buf3>buf3max?1:0),
buf3max=(buf3>buf3max?buf3:buf3max),
cost+=(buf3<buf3min?1:0),
buf3min=(buf3<buf3min?buf3:buf3min),
reg3=(buf3help?reg3-(reg3idle*BUF3IN):reg3),
```

Figure 5.16: Specification of the model in terms of Uniformly Priced Timed Automata (modifications to the original model).

buffer sizes can be reduced drastically even when looking at the problem from a conservatively abstract point of view. The model lends itself better for further analysis with reduced buffer sizes.

It is visible from Figure 5.17 that the schedule obtained by using the guided model utilises less memory than the previous one. The top-most graph is gained with bounded depth first search and the sum of utilised buffer ranges is 536 bytes. The lower one is obtained by using uniformly priced timed automata extensions to the model. The sum of the utilised buffer ranges is 444 bytes. It should be stated that search was equally bounded by 1000 ns in both cases.

Processor Time and Memory Requirements of the Synthesis Tasks

The synthesis task ran out of memory on a 32 bit x86 machine (the limit is 3 GB of memory per process) when using a model checker (Uppaal 3.4.x) in the explicit state space representation mode. Thus the need for some relevant space optimisation. Turning on bit-state hashing may yield a desired trace or an uninformative ”may be” answer in which case the search should be pursued further by modifying the model or, preferably, the hash table size. It appears that a synthesis task described above can be solved using just a few kilobytes of memory for the bit-state hashing table of passed states. It is shown in Figure 5.18 that, for example, approximately 25% of hash table sizes around 200 kB yield a schedule for the synthesis task. But about 75% of hash table sizes return a ”may be” answer. This suggests that iterated search refinement using bit-state hashing may need very small amount of memory to provide an aswer for a reachability task.

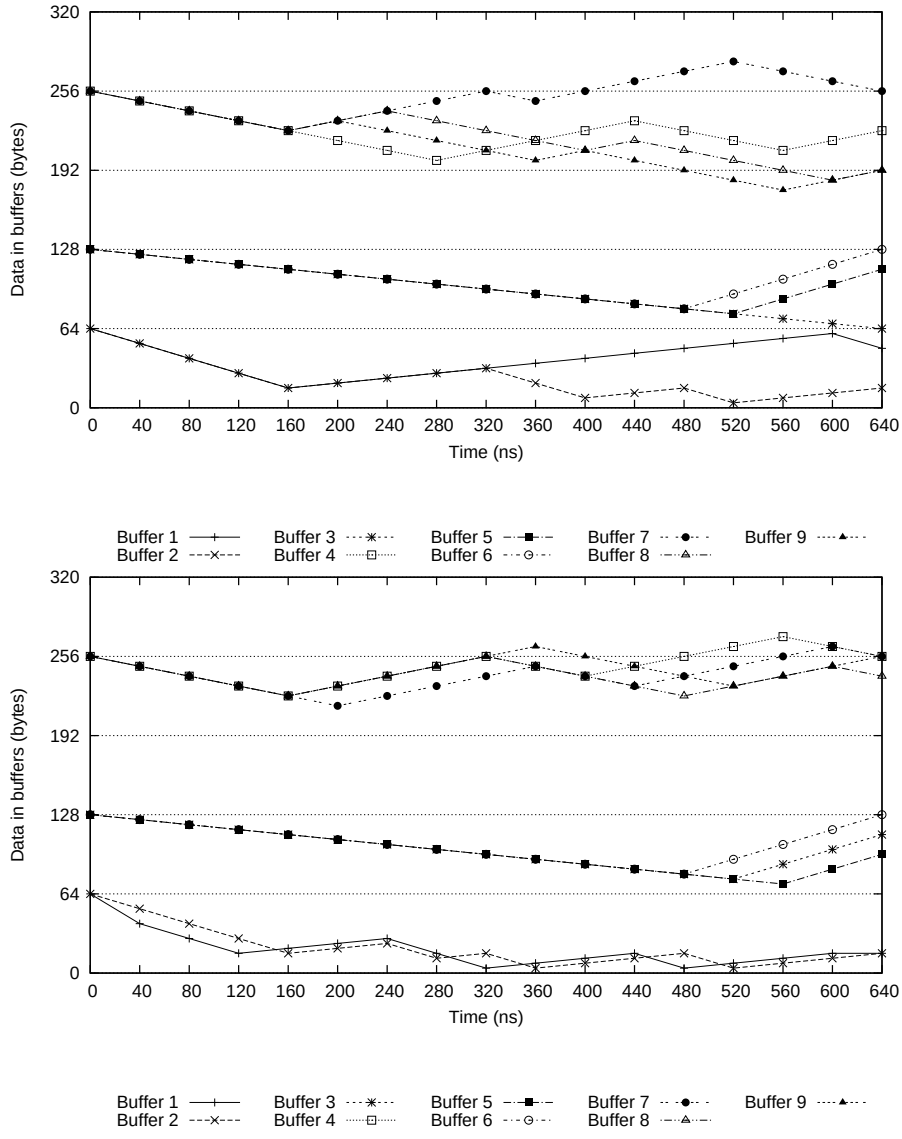


Figure 5.17: Data fluctuations in buffers during a synthesised cycle. The top-most graph is gained with bounded depth first search and the lower one is by using uniformly priced timed automata extensions to the model.

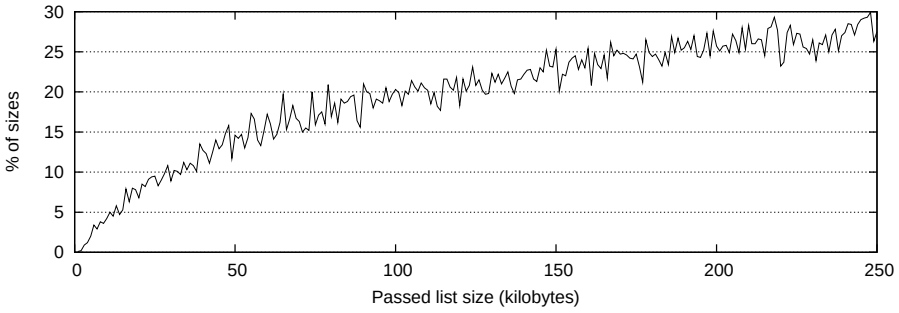


Figure 5.18: The percentage of passed list (hashtable) sizes that yield a concrete schedule. The rest yield a ”may be” answer.

Modelling and Verification of the Memory Arbiter

In the previous section we created a schedule for the memory arbiter that should satisfy the desired properties presented in the case study. As we did not present any formal correctness proofs for modifications for simplifying the synthesis task, for example depth bounding the search, we should show in some way that the synthesised schedule and the resultant arbiter is indeed such that does not deadlock nor causes any over or underflow of buffers. In addition, as was indicated in Figure 5.17, the initial sizes of the buffers were unnecessarily big, so we can reduce the buffer sizes accordingly. The resultant verification model is represented in Figure 5.19.

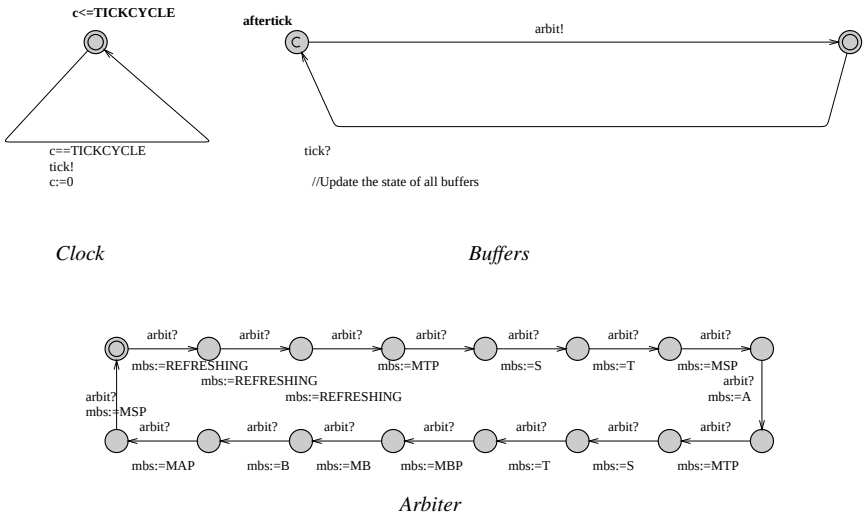


Figure 5.19: The Uppaal model for verification of the system. The Arbiter automaton is based on the cost optimal synthesised schedule presented in Table 5.8.

Modelling the Arbiter

The *Arbiter* automaton (Figure 5.19) consists of a linear sequence of locations and transitions that are connected into a loop. On each transition there is an assignment of a variable that dictates the memory bus state of the *Buffers* automaton. The assignments to the *mbs* variable are taken from the cost optimally generated schedule presented in Table 5.8.

Verification of the Arbiter

After building the *Arbiter* automaton we assemble the model of the system by adding the *Clock* and the *Buffers* automata. The resultant model is represented in Figure 5.19. The update on the lower transition of the *Buffers* automaton is the same as in Figure 5.14 but the transitions enabling switch to different memory transfers are removed. The guards that restricted the buffers from being starved and overflowed are also removed.

It appears that the result is a deterministic automaton that should satisfy the properties of never deadlocking and never starving nor overflowing any of the buffers under the assumption that memory refresh can be scheduled. The *MemoryRefresh* automaton that was present in Figure 5.13 is omitted from the verification model as memory refreshes are scheduled by the arbiter. It is necessary to construct a more complicated arbiter if memory refreshes are for some reason required to be allowed to occur periodically out of sync of the arbiter. In the current case such deterministic automaton is desirable because if the memory burst set-up needs to precede the actual burst by some short interval, it can be directly integrated into the scheduler.

```
a) A[] Buffers.buf1>=0 and Buffers.buf1<=Buffers.BUF1MAX and
    Buffers.buf2>=0 and Buffers.buf2<=Buffers.BUF2MAX and
    Buffers.buf3>=0 and Buffers.buf3<=Buffers.BUF3MAX and
    Buffers.buf4>=0 and Buffers.buf4<=Buffers.BUF4MAX and
    Buffers.buf5>=0 and Buffers.buf5<=Buffers.BUF5MAX and
    Buffers.buf6>=0 and Buffers.buf6<=Buffers.BUF6MAX and
    Buffers.buf7>=0 and Buffers.buf7<=Buffers.BUF7MAX and
    Buffers.buf8>=0 and Buffers.buf8<=Buffers.BUF8MAX and
    Buffers.buf9>=0 and Buffers.buf9<=Buffers.BUF9MAX

b) A[] not deadlock

c) E<> Buffers.c>1240
```

Figure 5.20: The queries used for the analysis of the verification model of the memory arbiter.

The verification model proved itself useful in the following ways: It was possible to adjust the buffer sizes and initial values to satisfy the required properties a) and b) in Figure 5.20. The diagnostic traces generated by the model checker brought out a typing error in the query that was used for generating the schedule. Additionally it was possible to generate a trace that represents the behaviour of the arbiter. The trace was generated by using query c) in Figure 5.20 and the results are presented in Figure 5.21. The graphs indicate a regular pattern with the exception of the three first cycles

(0-120 ns). The model can be further enhanced with specifying initial buffer and register levels to reduce the required buffer sizes. Note that all data fluctuations are of the order of magnitude of the registers, so with further analysis it could be shown that the buffers could be omitted altogether and the registers modified accordingly.

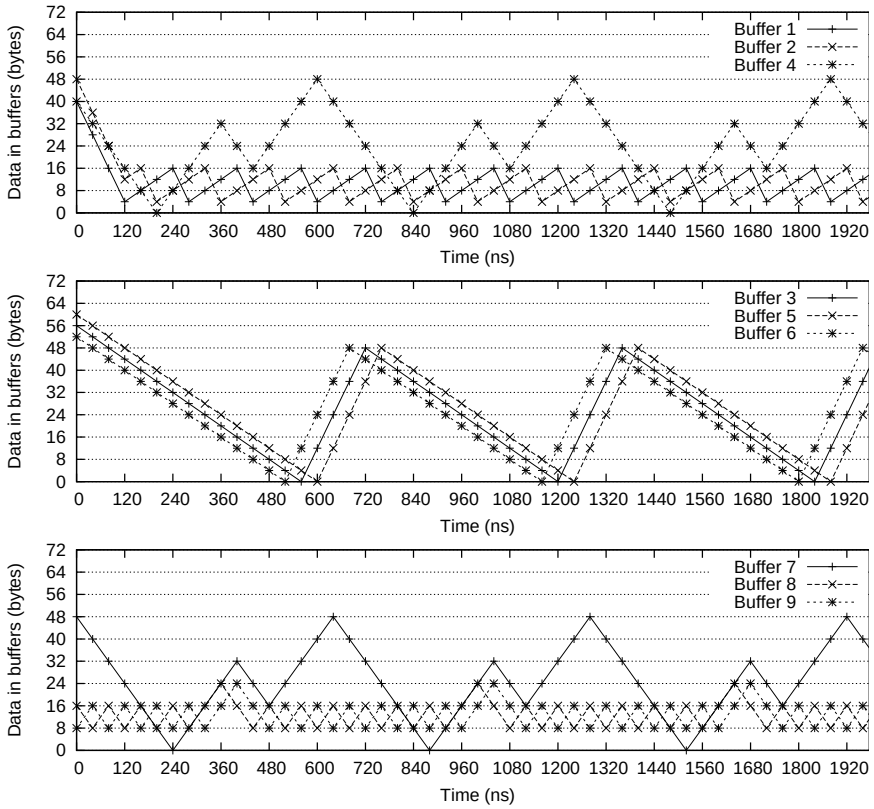


Figure 5.21: Data fluctuations in buffers in the system containing a synthesised arbiter. Additionally, buffer sizes were chosen based on the cost optimal synthesis task. (Data fluctuations have been divided into three different diagrams for the sake of readability).

5.2.6 Conclusion

In this section we analysed the radar memory interface card presented by Terma AS to the IST AMETIST project. We outlined a way to synthesise a memory arbiter for the system, to minimise the amount of static RAM used for buffering the streams and to verify that the the synthesised arbiter does not deadlock and never starves nor overflows any of the intermediate buffers. We presented the synthesis and verification tasks as two different problems of logic model checking. We designed the model for the synthesis task very carefully to reduce the potential state space that needs to be

searched in the process. To achieve that, unwanted behaviour was restricted on the guard level of the model by disabling transitions which would starve or overflow the intermediate buffers. We used the assumption of synchronicity in modelling. This enabled us to make a number of simplifications in the model. As we used bounded (in terms of search depth) search for arbiter synthesis, it was necessary to verify whether it behaves expectedly under all circumstances. For this purpose we constructed another model which differs from the first one by containing an arbiter but not the restrictions mentioned in the synthesis model case. The model was verified against the buffer starvation/overflow invariant and was checked that it is deadlock free.

Synthesis tasks are generally computationally more challenging than verification tasks. The success of the synthesis task is largely concealed in the rather simple abstract model of the system. The other key factor is the application of a sound but incomplete method for space saving—bit-state hashing, where each visited state is represented by one bit in the hash table in a location determined by the hash of the state. We showed that it is possible to synthesise a reasonable arbiter using this approach. Additionally, we augmented the model with cost variables (taking into account the range of buffers used) and applied the guided version of Uppaal—Uppaal-Cora to the synthesis task. This resulted in an optimal schedule in terms of buffer memory on the abstraction level of the model. Experiments with a resultant model of the arbiter showed that the fluctuations of data levels in buffers were reduced to amounts that are equivalent to the sizes of registers.

CHAPTER 6

CONCLUSION

This thesis focused on automated analysis of formalised requirements and models of software. We had a look at two different but partly overlapping techniques for analysis: model checking and model-based testing and a problem occurring in both — the need for enumerating states for performing reachability analysis. Such analysis can be used for proving that the model can not reach an unsafe state and for finding desired behaviours as in planning and scheduling tasks. The state spaces of such models can easily grow too big to be tractable by present day computers. We proposed and analysed two different techniques for reducing the state space that needs to be enumerated for proving properties that can be defined in terms of reachability.

Before proceeding to the actual techniques, we had a look at three different formalisms for writing the specifications and models of software. We looked at *Promela*, which is the modelling formalism used in *Spin* model checker and various other tools, the visual timed automata based formalism used in *Uppaal* and a way introduced in *NModel* for modelling using the C# programming language and predefined constructs and data types from the modelling library. All of the modelling approaches were used later in the thesis in different examples.

By introducing the notion of model programs that allow modelling using abstract data structures like sets and object instances, we introduced a new symmetry reduction which is based on state graph isomorphism of distinct states. Such reduction allows to explore all structurally distinct states that are reachable from the inital state thus reducing the total number of states that need to be considered.

Although powerful, the state isomorphism based reduction techique has its limitations as calculating the state graph and comparing it to previously seen state graphs consumes both extra memory and extra processor time.

We demonstrated how model checking can still yield interesting results for specification models where full model checking fails due to memory and processor time limitations, we observed that deliberately overpopulated bitstate hash tables act as a sort of pruning mechanism of the search tree. Based on the result we defined an iterated search refinement algorithm that utilises bitstate pruning. The algorithm runs on parallel architectures, thus the method can natively be speeded up by increaseing the number of processors available to the iteration algorithm. Although the method

implies that the parts of the state space of the model get traversed over and over again during different iterations, the fact that each individual iteration process consumes very little memory compensates by making each iteration much more processor cache efficient than search with hundreds of megabytes or gigabytes of memory. Of course, it should be stressed, that the iterated search refinement with bitstate pruning cannot be used for disproving a property as it is not guaranteed to cover the full state space.

Iterated Search Refinement with bitstate pruning is a powerful alternative for discovering shorter traces to error / desired state. Usually reachability of such states can be established using depth first search but the resultant trace is unfeasibly long. Breadth first search theoretically yields the shortest trace but often the amount of memory resources required exceeds the amount available yielding no answer. Iterated search refinement with bitstate pruning makes it possible to discover shorter traces using low memory resources. The tradeoff lies in that the method requires large amount of processing power, but the tradeoff can be remedied by applying multiple processors in the search.

The research presented in the current thesis uncovered several topics that will require attention during future research. Experiments with the current implementation of the isomorphism checking algorithm in *NModel* show that the implementation needs to be improved by, for example, extending the labelling of the nodes in the state graph.

Experimenting with iterated search refinement with bitstate pruning opened up the question whether the distinguishing functionality of the bitstate hash table, the search function and the search depth bound could be used for learning abstractions from the model.

Bibliography

Aceto, L., Bouyer, P., Burgueño, A. and Larsen, K. G.: 2003, The power of reachability testing for timed automata, *Theor. Comput. Sci.* **300**(1-3), 411–475.

AMETIST: 2002–2005, The eu information society technologies project ist-2001-35304: "advanced methods for timed systems", http://ametist.cs.utwente.nl.

Amnell, T., Behrmann, G., Bengtsson, J., D'Argenio, P. R., David, A., Fehnker, A., Hune, T., Jeannet, B., Larsen, K. G., Möller, M. O., Pettersson, P., Weise, C. and Yi, W.: 2001, UPPAAL - Now, Next, and Future, *in* F. Cassez, C. Jard, B. Rozoy and M. Ryan (eds), *Modelling and Verification of Parallel Processes*, number 2067 in *Lecture Notes in Computer Science Tutorial*, Springer–Verlag, pp. 100–125.

Amnell, T., Fersman, E., Pettersson, P., Yi, W. and Sun, H.: 2002, Code synthesis for timed automata, *Nordic J. of Computing* **9**(4), 269–300.

Barnat, J., Brim, L. and Chaloupka, J.: 2003, Parallel breadth-first search LTL model-checking, *Proceedings 18th IEEE International Conference on Automated Software Engineering* **00**, 106.

Behrmann, G.: 2005, Uppaal CORA, http://www.cs.aau.dk/~behrmann/cora/.

Behrmann, G., Bernicot, S., Hune, T., Larsen, K. G., Lecamp, S. and Skou, A.: 2002, Case study 2: Memory interface for radar system, Deliverable to the IST AMETIST project No. IST-2001-35304.

Behrmann, G., David, A. and Larsen, K. G.: 2004, A tutorial on Uppaal., *in* M. Bernardo and F. Corradini (eds), *SFM*, Vol. 3185 of *Lecture Notes in Computer Science*, Springer, pp. 200–236. (updated version available from `http://www.uppaal.com`).

Behrmann, G. and Fehnker, A.: 2001, Efficient guiding towards cost-optimality in uppaal, *TACAS 2001: Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Springer-Verlag, pp. 174–188.

Behrmann, G., Larsen, K. G. and Rasmussen, J. I.: 2004, Priced timed automata: Algorithms and applications., *in* F. S. de Boer, M. M. Bonsangue, S. Graf and W. P. de Roever (eds), *FMCO*, Vol. 3657 of *Lect. Notes in Comp. Sci.*, Springer, pp. 162–182.

Behrmann, G., Larsen, K. G. and Rasmussen, J. I.: 2005, Optimal scheduling using priced timed automata, *SIGMETRICS Perform. Eval. Rev.* **32**(4), 34–40.

Bengtsson, J.: 2002, *Clocks, DBMs and states in timed systems*, PhD thesis.

Bengtsson, J. and Yi, W.: 2004, Timed automata: Semantics, algorithms and tools, *in* W. Reisig and G. Rozenberg (eds), *In Lecture Notes on Concurrency and Petri Nets*, Lecture Notes in Computer Science vol 3098, Springer–Verlag.

Blass, A. and Gurevich, Y.: 2000, Background, reserve, and gandy machines, *Proceedings of the 14th Annual Conference of the EACSL on Computer Science Logic*, Springer-Verlag, London, UK, pp. 1–17.

Blom, J., Hessel, A., Jonsson, B. and Petterson, P.: 2005, Specifying and Generating Test Cases Using Observer Automata, *in* J. Gabowski and B. Nielsen (eds), *Proc. of the 4th International Workshop on Formal Approaches to Testing of Software (FATES 2004)*, number 3395 in *Lect. Notes in Comp. Sci.*, Springer, pp. 125–139.

Bloom, B. H.: 1970, Space/time trade-offs in hash coding with allowable errors, *Communications of the ACM* **13**(7), 422–426.

Bosnacki, D., Dams, D. and Holenderski, L.: 2000, Symmetric spin., *in* K. Havelund, J. Penix and W. Visser (eds), *SPIN*, Vol. 1885 of *Lecture Notes in Computer Science*, Springer, pp. 1–19.

Bourhfir, C., Dssouli, R., Aboulhamid, E. and Rico, N.: 1997, Automatic executable test case generation for extended finite state machine protocols, *Proceedings of the 10th International IFIP Workshop on Testing of Communicating Systems (IWTCS'97), Cheju Islands, Korea*, Chapman & Hall, pp. 75–90.

Boyapati, C., Khurshid, S. and Marinov, D.: 2002, Korat: automated testing based on java predicates, *SIGSOFT Softw. Eng. Notes* **27**(4), 123–133.

Bryant, R. E.: 1986, Graph-based algorithms for boolean function manipulation, *IEEE Trans. Comput.* **35**(8), 677–691.

Campbell, C. and Veanes, M.: 2005, State exploration with multiple state groupings, *in* D. Beauquier, E. Börger and A. Slissenko (eds), *12th International Workshop on Abstract State Machines, ASM'05*, Laboratory of Algorithms, Complexity and Logic, Créteil, France, pp. 119–130.

Chow, T. S.: 1978, Testing software design modeled by finite-state machines., *IEEE Trans. Software Eng.* **4**(3), 178–187.

Cormen, T. H., Leiserson, C. E. and Rivest, R. L.: 1994, *Introduction to Algorithms*, McGraw-Hill, Inc., New York, NY, USA.

Darga, P. T. and Boyapati, C.: 2006, Efficient software model checking of data structure properties, *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, ACM Press, New York, NY, USA, pp. 363–382.

Demartini, C., Iosif, R. and Sisto, R.: 1999, dSPIN: A dynamic extension of SPIN, *Proceedings of the 5th and 6th International SPIN Workshops on Theoretical and Practical Aspects of SPIN Model Checking*, Springer-Verlag, London, UK, pp. 261–276.

Dijkstra, E. W.: 1965, Programming considered as a human activity, *Proc. IFIP Congress*, Vol. 65, pp. 213–217.

Dijkstra, E. W.: 1971, Hierarchical ordering of sequential processes, *Acta Inf.* **1**, 115–138.

Dill, D. L.: 1996, The murphi verification system, *CAV '96: Proceedings of the 8th International Conference on Computer Aided Verification*, Springer-Verlag, London, UK, pp. 390–393.

Dillinger, P. C. and Manolios, P.: 2004a, Bloom filters in probabilistic verification., *in* A. J. Hu and A. K. Martin (eds), *FMCAD*, Vol. 3312 of *Lecture Notes in Computer Science*, Springer, pp. 367–381.

Dillinger, P. C. and Manolios, P.: 2004b, Fast *and* accurate bitstate verification for SPIN, *11th SPIN Workshop on Model Checking Software*, Vol. 2989 of *LNCS*, Springer-Verlag.

Dillinger, P. C. and Manolios, P.: 2005, Enhanced probabilistic verification with 3spin and 3murphi., *in* P. Godefroid (ed.), *SPIN*, Vol. 3639 of *Lecture Notes in Computer Science*, Springer, pp. 272–276.

Donaldson, A. F.: 2007, *Automatic Techniques for Detecting and Exploiting Symmetry in Model Checking*, PhD thesis, University of Glasgow.

Edelkamp, S., Lafuente, A. L. and Leue, S.: 2001, Directed explicit model checking with hsf-spin, *SPIN '01: Proceedings of the 8th international SPIN workshop on Model checking of software*, Springer-Verlag New York, Inc., New York, NY, USA, pp. 57–79.

Edmund M. Clarke, J., Grumberg, O. and Peled, D. A.: 1999, *Model checking*, MIT Press, Cambridge, MA, USA.

Ernits, J.: 2005, Memory arbiter synthesis and verification for a radar memory interface card, *Nordic Journal of Computing* **12**(2), 68–88.

Ernits, J. P., Kull, A., Raiend, K. and Vain, J.: 2006, Generating tests from efsm models using guided model checking and iterated search refinement, *in* K. Havelund, M. Núñez, G. Rosu and B. Wolff (eds), *FATES/RV*, Vol. 4262 of *Lecture Notes in Computer Science*, Springer, pp. 85–99.

Farchi, E., Hartman, A. and Pinter, S. S.: 2002, Using a model-based test generator to test for standard conformance., *IBM Systems Journal* **41**(1), 89–110.

Flanagan, C. and Godefroid, P.: 2005, Dynamic partial-order reduction for model checking software, *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, ACM Press, New York, NY, USA, pp. 110–121.

Fortin, S.: 1996, The graph isomorphism problem.

GCI: 2006, Academia europaea informatics symposium: Grand challenges of informatics budapest. Hungary, 19–20 September, 2006. See `http://www.jaist.ac.jp/~bjorner/ae-is-budapest/`.

Godefroid, P.: 1996, *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*, Springer-Verlag New York, Inc., Secaucus, NJ, USA. Foreword By-Pierre Wolper.

Goel, A. and Lee, W. R.: 2000, Formal verification of an ibm coreconnect processor local bus arbiter core, *DAC '00: Proceedings of the 37th conference on Design automation*, ACM Press, pp. 196–200.

Grieskamp, W., Gurevich, Y., Schulte, W. and Veanes, M.: 2002, Generating finite state machines from abstract state machines, *ISSTA'02*, Vol. 27 of *Software Engineering Notes*, ACM, pp. 112–122.

Grieskamp, W., Tillmann, N. and Schulte, W.: 2006, XRT — exploring runtime for .Net architecture and applications., *Electr. Notes Theor. Comput. Sci.* **144**(3), 3–26.

GROOVE: 2007, Groove web page. `http://groove.cs.utwente.nl/`.

Gunter, E. L. and Peled, D.: 2005, Model checking, testing and verification working together., *Formal Asp. Comput.* **17**(2), 201–221.

Gurevich, Y.: 1995, *Specification and Validation Methods*, Oxford University Press, chapter Evolving Algebras 1993: Lipari Guide, pp. 9–36.

Hamon, G., de Moura, L. and Rushby, J.: 2004a, Generating efficient test sets with a model checker, *2nd International Conference on Software Engineering and Formal Methods*, IEEE Computer Society, Beijing, China, pp. 261–270.

Hamon, H., de Moura, L. and Rushby, J.: 2004b, Generating efficient test sets with a model checker, *SEFM '04: Proceedings of the Software Engineering and Formal Methods, Second International Conference on (SEFM'04)*, IEEE Computer Society, Washington, DC, USA, pp. 261–270.

Hendriks, M., Behrmann, G., Larsen, K. G., Niebert, P. and Vaandrager, F. W.: 2003, Adding symmetry reduction to Uppaal, *in* K. G. Larsen and P. Niebert (eds), *FORMATS*, Vol. 2791 of *Lecture Notes in Computer Science*, Springer, pp. 46–59.

Hessel, A., Larsen, K. G., Nielsen, B., Petterson, P. and Skou, A.: 2004, Time-optimal Real-Time Test Case Generation using UPPAAL, *in* A. Petrenko and A. Ulrich (eds), *Proc. of the 3rd International Workshop on Formal Approaches to Testing of Software (FATES 2003)*, number 2931 in *Lect. Notes in Comp. Sci.*, Springer, pp. 136–151.

Hessel, A., Larsen, K., Nielsen, B., Pettersson, P. and Skou, A.: 2003, Time-optimal realtime test case generation using UPPAAL, *FATES'03*, Montreal.

Hoare, T.: 2003, The verifying compiler: A grand challenge for computing research, *J. ACM* **50**(1), 63–69.

Hoare, T.: 2006, The ideal of verified software, *in* T. Ball and R. B. Jones (eds), *CAV*, Vol. 4144 of *Lecture Notes in Computer Science*, Springer, pp. 5–16.

Hogrefe, D.: 1991, OSI-formal specification case study: The INRES protocol and service. Technical Report 91-012, University of Bern.

Holzmann, G. J.: 1998, An analysis of bitstate hashing, *Form. Methods Syst. Des.* **13**(3), 289–307.

Holzmann, G. J.: 2003, *The Spin Model Checker, Primer and Reference Manual*, Addison-Wesley, Reading, Massachusetts.

Holzmann, G. J. and Smith, M. H.: 2000, Automating software feature verification, *Bell Labs Technical Journal* **5**(2), 72–87.

Hong, H. S., Lee, I., Sokolsky, O. and Ural, H.: 2002, A temporal logic based theory of test coverage and generation, *TACAS '02: Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Springer-Verlag, London, UK, pp. 327–341.

Hopcroft, J. E. and Ullman, J. D.: 1979, *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, Reading, Massachusetts.

Hune, T., Larsen, K. G. and Pettersson, P.: 2001, Guided Synthesis of Control Programs using UPPAAL, *Nordic Journal of Computing* **8**(1), 43–64.

Huth, M. R. A. and Ryan, M. D.: 2000, *Logic in Computer Science: Modelling and Reasoning about Systems*, Cambridge University Press, Cambridge, England.

Iosif, R.: 2004, Symmetry reductions for model checking of concurrent dynamic software., *STTT* **6**(4), 302–319.

Ip, C. N. and Dill, D. L.: 1996, Better verification through symmetry, *Form. Methods Syst. Des.* **9**(1-2), 41–75.

Jacky, J., Veanes, M., Campbell, C. and Schulte, W.: 2007, *Model-based Software Testing and Analysis with C#*, Cambridge University Press. Forthcoming.

Jenkins, B.: 1997, Algorithm alley: Hash functions, *Dr. Dobbs* **22**(9).

JPF: 2007, Java Pathfinder web page. `http://javapathfinder.sourceforge.net/`.

Junttila, T.: 2003, On the symmetry reduction method for Petri nets and similar formalisms, *Research Report A80*, Helsinki University of Technology, Laboratory for Theoretical Computer Science, Espoo, Finland. Doctoral dissertation.

Kastenberg, H. and Rensink, A.: 2006, Model checking dynamic states in groove, *in* A. Valmari (ed.), *SPIN*, Vol. 3925 of *Lecture Notes in Computer Science*, Springer, pp. 299–305.

Kuntz, M. and Lampka, K.: 2004, Probabilistic methods in state space analysis, *in* C. Baier, B. R. Haverkort, H. Hermanns, J.-P. Katoen and M. Siegle (eds), *Validation of Stochastic Systems*, Vol. 2925 of *Lecture Notes in Computer Science*, Springer, pp. 339–383.

Larsen, K. G., Larsson, F., Pettersson, P. and Yi, W.: 2003, Compact data structures and state-space reduction for model-checking real-time systems, *Real-Time Syst.* **25**(2-3), 255–275.

Larsen, K. G., Mikucionis, M., Nielsen, B. and Skou, A.: 2005, Testing real-time embedded software using UPPAAL-TRON: an industrial case study, *EMSOFT '05: Proc. of the 5th ACM International Conference on Embedded Software*, ACM Press, New York, NY, USA, pp. 299–306.

Luks, E. M.: 1982, Isomorphism of graphs of bounded valence can be tested in polynomial time., *J. Comput. Syst. Sci.* **25**(1), 42–65.

McKay, B. D.: 1981, Practical graph isomorphism, *Congressus Numerantium* **30**, 45–87.

McMillan, K. L.: 1992, *Symbolic Model Checking: An approach to the state explosion problem*, PhD thesis, Carnegie Mellon University. CMU-CS-92-131.

McMillan, K. L.: 1999, The SMV language. Cadence Berkeley Labs.

Mercer, E. and Jones, M.: 2005, Model checking machine code with the GNU debugger., *in* P. Godefroid (ed.), *SPIN*, Vol. 3639 of *Lecture Notes in Computer Science*, Springer, pp. 251–265.

Messmer, B. T.: 1995, Efficient graph matching algorithms.

Miller, A., Donaldson, A. and Calder, M.: 2006, Symmetry in temporal logic model checking, *ACM Comput. Surv.* **38**(3), 8.

Mücke, T. and Huhn, M.: 2004, Generation of optimized testsuites for UML statecharts with time., *in* R. Groz and R. M. Hierons (eds), *TestCom*, Vol. 2978 of *Lecture Notes in Computer Science*, Springer, pp. 128–143.

Musuvathi, M. and Dill, D. L.: 2005, An incremental heap canonicalization algorithm., *in* P. Godefroid (ed.), *SPIN*, Vol. 3639 of *Lecture Notes in Computer Science*, Springer, pp. 28–42.

Myers, G. J.: 1979, *Art of Software Testing*, John Wiley & Sons, Inc., New York, NY, USA.

Myers, G. J., Badgett, T., Thomas, T. M. and Sandler, C.: 2004, *Art of Software Testing*, John Wiley & Sons, Inc., New York, NY, USA. 2nd edition.

Nethercote, N., Walsh, R. and Fitzhardinge, J.: 2006, Building workload characterization tools with Valgrind, *Invited tutorial, IEEE International Symposium on Workload Characterization (IISWC 2006)*, San José, California, USA.

Nicely, T. R.: 1994, Pentium FDIV flaw. `http://www.trnicely.net/pentbug/pentbug.html`.

NModel: 2007, NModel web site. `http://www.codeplex.com/NModel`.

Pelánek, R.: 2007, BEEM: Benchmarks for explicit model checkers, *in* D. Bosnacki and S. Edelkamp (eds), *SPIN*, Vol. 4595 of *Lecture Notes in Computer Science*, Springer, pp. 263–267.

Pelánek, R., Hanžl, T., Černá, I. and Brim, L.: 2005, Enhancing random walk state space exploration, *FMICS '05: Proceedings of the 10th international workshop on Formal methods for industrial critical systems*, ACM Press, New York, NY, USA, pp. 98–105.

Peled, D.: 1998, Ten years of partial order reduction, *CAV '98: Proceedings of the 10th International Conference on Computer Aided Verification*, Springer-Verlag, London, UK, pp. 17–28.

Pettersson, P. and Larsen., K. G.: 2000, UPPAAL2k, *Bulletin of the European Association for Theoretical Computer Science* **70**, 40–44.

Rensink, A.: 2006, Isomorphism checking in GROOVE, *Electronic Communications of the EASST* **1**, 1–11.

Rensink, A., Schmidt, Á. and Varró, D.: 2004, Model checking graph transformations: A comparison of two approaches, *in* H. Ehrig, G. Engels, F. Parisi-Presicce and G. Rozenberg (eds), *ICGT*, Vol. 3256 of *Lecture Notes in Computer Science*, Springer, pp. 226–241.

Robby, Dwyer, M. B. and Hatcliff, J.: 2006, Domain-specific model checking using the bogor framework, *ASE '06: Proceedings of the 21st IEEE International Conference on Automated Software Engineering (ASE'06)*, IEEE Computer Society, Washington, DC, USA, pp. 369–370.

Ruys, T. C.: 2001, *Towards Effective Model Checking*, PhD thesis, University of Twente.

Ruys, T. C.: 2003, Optimal scheduling using branch and bound with SPIN 4.0., *in* T. Ball and S. K. Rajamani (eds), *SPIN*, Vol. 2648 of *Lecture Notes in Computer Science*, Springer, pp. 1–17.

SpecExplorer: 2006, `http://research.microsoft.com/SpecExplorer`.

Ullmann, J. R.: 1976, An algorithm for subgraph isomorphism, *J. ACM* **23**(1), 31–42.

Utting, M. and Legeard, B.: 2006, *Practical Model-Based Testing - A tools approach*, Elsevier Science.

Veanes, M., Campbell, C., Grieskamp, W., Nachmanson, L., Schulte, W. and Tillmann, N.: 2005, Model-based testing of object-oriented reactive systems with Spec Explorer. Tech. Rep. MSR-TR-2005-59, Microsoft Research. Preliminary version of a book chapter in the forthcoming text book *Formal Methods and Testing*.

Veanes, M., Campbell, C. and Schulte, W.: 2007a, Composition of model programs, *in* J. Derrick and J. Vain (eds), *FORTE*, Vol. 4574 of *Lecture Notes in Computer Science*, Springer, pp. 128–142.

Veanes, M., Campbell, C. and Schulte, W.: 2007b, Parallel and serial composition of model programs, *Technical Report MSR-TR-2007-22*, Microsoft Research.

Veanes, M., Campbell, C., Schulte, W. and Tillmann, N.: 2005, Online testing with model programs, *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, ACM Press, New York, NY, USA, pp. 273–282.

Veanes, M., Ernits, J. P. and Campbell, C.: 2007, State isomorphism in model programs with abstract data structures, *in* J. Derrick and J. Vain (eds), *FORTE*, Vol. 4574 of *Lecture Notes in Computer Science*, Springer, pp. 112–127.

Visser, W., Havelund, K., Brat, G., Park, S. and Lerda, F.: 2003, Model Checking Programs, *Automated Software Engineering* **10**(2), 203–232.

VST: 2005, Conference on verified software: theories, tools, experiments. Eidgenössische Technische Hochschule Zürich, Zürich 10–13, October 2005. See `http://vstte.ethz.ch`.

Weiss, G.: 2002, Optimal Scheduler for a Memory Card, *Research report*, Weizmann.

Wijs, A., van de Pol, J. and Bortnik, E.: 2005, Solving scheduling problems by untimed model checking: the clinical chemical analyser case study, *FMICS '05: Proceedings of the 10th international workshop on Formal methods for industrial critical systems*, ACM Press, New York, NY, USA, pp. 54–61.

Wikipedia: 2007, Halting problem — Wikipedia, the Free Encyclopedia. `http://en.wikipedia.org/w/index.php?title=Halting_problem`.

Xie, T., Marinov, D., Schulte, W. and Notkin, D.: 2005, Symstra: A framework for generating object-oriented unit tests using symbolic execution, *in* N. Halbwachs and L. D. Zuck (eds), *TACAS*, Vol. 3440 of *Lecture Notes in Computer Science*, Springer, pp. 365–381.

Zeller, A.: 2005, *Why Programs Fail: A Guide to Systematic Debugging*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

List of Publications

- Jüri Vain, Kullo Raiend, Andres Kull, Juhan-Peep Ernits: 2007, Synthesis of test purpose directed reactive planning tester for nondeterministic systems. *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE'07). November 5-9, 2007 in Atlanta, Georgia, USA,* ACM Press, 363 – 372.
- Margus Veanes, Juhan-Peep Ernits, Colin Campbell: 2007, State isomorphism in model programs with abstract data structures. *In* J. Derrick and J. Vain (eds) *Formal Techniques for Networked and Distributed Systems - FORTE 2007 : 27th IFIP WG 6.1 International Conference, Tallinn, Estonia, June 27-29, 2007, Proceedings.* Berlin: Springer, Vol 4574 of Lecture Notes in Computer Science, 112 – 127.
- Juhan-Peep Ernits, Andres Kull, Kullo Raiend, Jüri Vain: 2006, Generating tests from EFSM models using guided model checking and iterated search refinement. *In* K. Havelund, M. Núñez, G. Rosu and B. Wolff (eds) *Formal Approaches to Software Testing and Runtime Verification : First Combined International Workshops FATES 2006 and RV 2006, Seattle, WA, USA, August 15-16, 2006, Revised Selected Papers.* Berlin: Springer, Vol 4262 of Lecture Notes in Computer Science, 85 – 99.
- Juhan-Peep Ernits, Andres Kull, Kullo Raiend, Jüri Vain: 2006, Generating TTCN-3 test cases from EFSM models of reactive software using model checking. *In* C. Hochberger and R. Liskowsky *Informatik 2006 - Informatik für Menschen : Proceedings: Beiträge der 36.Jahrestagung der Gesellschaft für Informatik e.V.(GI), 2.bis 6.Oktober 2006 in Dresden*. Bonn: Köllen, Vol. 94 of Lecture Notes in Informatics; Bd 2, 241 – 248.
- Juhan-Peep Ernits: 2005, Memory arbiter synthesis and verification for a radar memory interface card. *Nordic Journal of Computing*, **12**(2), 68 – 88.
- Jüri Vain, Juhan-Peep Ernits: 2003, Model checking in pattern based control systems design. *Proceedings of the 15th IFAC World Congress : International Federation of Automatic Control, 21-26 July 2002, Barcelona, Spain. Vol. L. Computers for Control.* Amsterdam: Pergamon, 237 – 242.
- Jüri Vain, Ingmar Randvee, Tiit Riismaa, Juhan-Peep Ernits: 2002, Solving line balancing problems with model checking. *Proceedings of the Estonian Academy of Sciences.Engineering*, **8**(4), 211 – 222.
- Juhan-Peep Ernits: 2002. Model checking hybrid dynamical systems. *MSc Thesis* Tallinn: Tallinn Technical University, 62 pp.
- Jüri Vain, Juhan-Peep Ernits, Mati Littover, Ingmar Randvee, Tiit Riismaa: 1999, A tool for flexible planning of resource routes. *Control in Natural Disasters (CND '98): A Proceedings volume from the IFAC Workshop, Tokyo, Japan, 21-22 September 1998.* Oxford: Pergamon, 1999, 85 – 90.

Curriculum Vitae

Personal Data

Name	Juhan-Peep Ernits
Birth date and place	16.12.1974, Tartu
Citizenship	Estonian
Marital status	married

Contact Data

Address	Institute of Cybernetics at TUT, Akadeemia tee 21, 12618 Tallinn
Phone	6204194
E-mail	juhan@cc.ioc.ee

Education

School	Year	Degree
St. Olav Videregående Skole	1994	International Baccalaureate
Tallinn Technical University	1999	BSc, comp. and systems technology, Cum Laude
Tallinn Technical University	2002	MSc, computer science

Positions Held

2002 - ...	Tallinn University of Technology, Institute of Cybernetics; researcher
2002 - ...	TUT, Department of Computer science, researcher
2007 - 2007	Internship at Microsoft Research, Redmond, WA USA (01/2007-03/2007)
2003 - 2004	Marie Curie Fellowship to visit BRICS, Denmark (09/2003-01/2004)
1997 - 2002	Tallinn University of Technology, Institute of Cybernetics, assistant

Research Interests

Formal modelling of software and systems with the emphasis on automated analysis; methods of automated analysis of formal models – model checking; model-based testing.

Professional Activities

2007	Chair of local organisation, 19th IFIP International Conference on Testing of Communicating Systems and 7th International Workshop on Formal Approaches to Testing of Software, and 27th IFIP WG 6.1 International Conference on Formal Methods for Networked and Distributed Systems, TestCom/FATES'07 and FORTE'07
2007	Member of the organising and programme committee, 6th Estonian Summer School in Computer and Systems Science, ESSCaSS'07
2006	One of the organisers, ROBOTEX 2006
2006	One of the local organisers, 8th International Conference of Mathematics of Program Construction and 11th International Conference on Algrebaic Methodology and Software Technology, MPC/AMAST 2007
2006	Member of the organising and programme committee, ESSCaSS'06
2005	One of the organisers, ROBOTEX 2005
2005	Main organiser and member of the programme committee, ESSCaSS'05
2005	One of the local organisers, 6th International Symposium on Trends in Functional Programming, 10th ACM SIGPLAN International Conference on Functional Programming, and 4th International Conference on Generative Programming and Component Engineering, TFP/ICFP/GPCE
2004	Main organiser and member of the programme committee, ESSCaSS'04
2004	One of the local organisers, 2nd APPSEM II Workshop
2003	Main organiser and member of the programme committee, ESSCaSS'03
2002	One of the local organisers, 14th Nordic Workshop on Programming Theory, NWPT'02

Elulookirjeldus

Isikuandmed

Ees- ja perekonnanimi	Juhan-Peep Ernits
Sünniaeg ja koht	16.12.1974, Tartu
Kodakondsus	Eesti
Perekonnaseis	abielus

Kontaktandmed

Aadress	TTÜ Küberneetika Instituut, Akadeemia tee 21, 12618 Tallinn
Telefon	6204194
E-posti aadress	juhan@cc.ioc.ee

Hariduskäik

Õppeasutus	Aasta	Haridus
St. Olav Videregående Skole	1994	International Baccalaureate
Tallinna Tehnikaülikool	1999	tehnikateaduste bakalaureus, arvuti- ja süst.-tehnika
Tallinna Tehnikaülikool	2002	tehnikateaduste magister, arvutiteadus

Teenistuskäik

2002 - ...	Tallinna Tehnikaülikool, Küberneetika Instituut; teadur
2002 - ...	TTÜ, Infotehnoloogia teaduskond, Arvutiteaduse instituut, teadur
2007 - 2007	Intern Microsoft Researchi Redmondi uurimislaboris (01.2007-03.2007)
2003 - 2004	Doktorikool BRICS Taanis, Marie Curie Fellowship (09.2003-01.2004)
1997 - 2002	TTÜ Küberneetika Instituut, insener

Teadustöö põhisuunad

Tarkvara formaalne modelleerimine rõhuga automaatsel analüüsil; formaalsete mudelite automaatse analüüsi meetodid – mudelikontroll; mudelipõhine testimine.

Teadusorganisatsiooniline ja - admistratiivne tegevus

2007	Kohaliku korraldustoimkonna juht, 19. IFIP International Conference on Testing of Communicating Systems and 7th International Workshop on Formal Approaches to Testing of Software, and 27. IFIP WG 6.1 International Conference on Formal Methods for Networked and Distributed Systems, TestCom/FATES'07 and FORTE'07
2007	Korraldus- ja programmitoimkonna liige, 6. Eesti Arvuti- ja Süsteemiteaduse Suvekool, ESSCaSS'07
2006	Üks korraldajatest, ROBOTEX 2006
2006	Kohaliku korraldustoimkonna liige, 8. International Conference of Mathematics of Program Construction and 11. International Conference on Algrebaic Methodology and Software Technology, MPC/AMAST 2007
2006	Korraldus- ja programmitoimkonna liige, ESSCaSS'06
2005	Üks korraldajatest, ROBOTEX 2005
2005	Põhikorraldaja ja programmitoimkonna liige, ESSCaSS'05
2005	Kohaliku korraldustoimkonna liige, 6. International Symposium on Trends in Functional Programming, 10. ACM SIGPLAN International Conference on Functional Programming, and 4. International Conference on Generative Programming and Component Engineering, TFP/ICFP/GPCE
2004	Põhikorraldaja ja programmitoimkonna liige, ESSCaSS'04
2004	Kohaliku korraldustoimkonna liige, 2. APPSEM II Workshop
2003	Põhikorraldaja ja programmitoimkonna liige, ESSCaSS'03
2002	Kohaliku korraldustoimkonna liige, 14. Nordic Workshop on Programming Theory, NWPT'02