Masaryk University
Faculty of Informatics

Reduction and Abstraction Techniques for Model Checking

Radek Pelánek

Ph.D. Thesis

2006

Abstract

Model checking is an increasingly popular method for verification of safety-critical systems. The main obstacle of this verification method is a state space explosion problem and consequently high computational requirements of model checking algorithms. In order to make the model checking method practically feasible, it is necessary to develop powerful techniques for fighting state space explosion.

This thesis is focuses on fighting state space explosion in the context of embedded system verification. Verification of embedded systems is particularly difficult due to intricate interferences of software and real-time aspects of these systems. In this setting, the most useful techniques are abstraction and reduction. These represent the main topics of this thesis.

The thesis contributes in several ways to the development of abstraction and reduction techniques, which are both practical and theoretically grounded. Our first contribution is the systematic presentation of reduction and abstraction techniques in a single formal setting. This facilitates understanding and application of these techniques. Our main innovative contribution lies in the novel under-approximation refinement algorithm for software model checking. Similarly to other automatic refinement algorithms, our algorithm is based on predicate abstraction. However, it uses under-approximation refinement instead of the classical over-approximation refinement. The thesis also contains several important technical results about abstraction and reduction techniques. Particularly, we provide two interesting results for timed automata: a decidability result for a non-emptiness problem of timed automata with sampled semantics and a new extrapolation technique for zone based abstractions of timed automata.

Acknowledgements

"You ought to return thanks in a neat speech," the Red Queen said. [47]

I am afraid that my speech will not be neat. Anyway, I definitely ought to return thanks to many people.

First of all, I thank to Ivana Černá, my supervisor. She was the right supervisor for me: giving me the freedom to pursue my ideas, but at the same time gently guiding my way. She always managed to find time to read my numerous drafts, discuss proposals, and give valuable feedback. I also appreciated having a supervisor with very similar values and understanding for my non-PhD activities.

Pavel Krčál was, despite the large physical distance which separated us during our PhD studies, one of my closest collaborators. To be specific, I thank him for the cooperation on the FSTTCS paper and for accommodating me during my visit in Uppsala. But I am even more grateful for a lot of very useful feedback on my ideas, drafts, and papers during different stages of my PhD studies. His sincere criticism was a great complement to the polite feedback of my supervisor.

During my PhD, I spent 10 weeks as a NASA intern. I was lucky to be assigned to work with Willem Visser and Corina Păsăreanu. Doing research with Willem and Corina was really interesting. Particularly, I enjoyed the research-by-competition, as we sometimes carried it with Corina (although it was unfair, since Corina has a magic computer), as well as trying to answer some of Willem's "million dollar questions".

I was also very glad to cooperate with many other bright researchers: Gerd Behrmann, Kim G. Larsen, Patricia Bouyer, Jan Strejček, Tomáš Hanžl. In all cases the cooperation was very enjoyable and fruitful.

I thank to all members of the Parallel and distributed systems laboratory (ParaDiSe) for making it such a nice environment full of interesting people (not to mention watering of flowers during summer). I am especially thankful to members of the DiVinE group, particularly Jiří Barnat and Pavel Šimeček. Although, at the end, I have not used DiVinE very much in my thesis, collaboration on this project was very useful to me. Luboš Brim, chief of the ParaDiSe laboratory, engaged me, together with Ivana, in the laboratory in early days of my master studies and supported me in different ways till the end of my studies.

Life is not just computer science. And even writing computer science PhD thesis is not just about computer science. I am convinced that I gained very much from experiences and skill which I acquired as an organizer of summer camps, outward bound activities, and city games. So I am very indebted to all my friends who

cooperated with me on these activities and who helped me to acquire these skills. It was also fun and it helped me to stay sane.

Ivana Černá, Pavel Krčál, and Pavel Šimeček provided useful feedback about the final structure of the thesis, Jan Holeček gave me valuable typographical advice, and Nikola Betlachová pointed out some of my English language mistakes.

Last but not least, I thank to my whole family, for giving me support and firm background.

Radek Pelánek

Contents

Chapter 1

Introduction

"Found it," the Mouse replied rather crossly: "of course you know what 'it' means."

"I know what 'it' means well enough, when I find a thing," said the Duck: "it's generally a frog or a worm. The question is, what did the archbishop find?" [47]

In this chapter we introduce the context of the thesis and discuss what we found[1]. The chapter begins with a brief description of the motivation and the context of the work. Then it goes on to the description of the model checking method and discusses current research trends. After this general introduction we discuss the main unifying themes of the thesis, we give an overview of the thesis and discuss its contributions.

1.1 Motivation

This section describes a a high-level motivation for the research presented in the thesis. We argue that there is a great need for reliable verification methods and that formal verification is an important one.

1.1.1 Computers and Bugs

Computers and computer-controlled systems have a direct impact on more and more aspects of our everyday life. Often they make life easier. Nevertheless, we are all familiar with all these irritating faults. In most situation, computer bugs lead to relatively small problems. Unfortunately, many examples of very significant computer bugs can be found. Some of the best known are the following:

Ariane 5 The Ariane 5 rocket exploded on its first flight. It inherited some parts of a code from its predecessor, Ariane 4, without proper verification [88].

Therac-25 Therac-25 was a radiation therapy machine. Due to a software error, six people are believed to die because of overdoses [134].

Pentium FDIV bug A design error in a floating point division unit (more specifically, an FDIV instruction) of a Pentium processor led to wrong results. Intel was forced to offer replacement of all flawed processors [1].

Some of these bugs are "only" very expensive (e.g., space flight, processor design, stock exchange, telephone control), but some are *safety-critical*, i.e., they lead to

[1] We leave the question "What did the archbishop find?" open.

threat to health or even life (e.g., medical systems, embedded controllers in vehicles, nuclear reactor controllers). It is crucial to verify functionality of these systems really properly.

Unfortunately, systems that are safety-critical usually have typical characteristics of systems that are very hard to design correctly:

embedded: a system is not a "stand-alone" computer, but the processor is rather embedded in a larger physical systems and it also controls mechanical parts of the system,

reactive: a system does not have an input-output behaviour of a "classical" computer program, but rather runs indefinitely and reacts to events from the external environment,

concurrent: a system is not a single sequential program, but rather a set of concurrently running threads which interleave in unexpected ways,

real-time: a system reacts with its environment in real-time and exact timing is often critical for correct functionality of the system.

All these characteristics make these systems much harder to design and verify than "classical" computer systems. Taking into account the safety-critical nature of these systems and the difficulty of their verification, it is clear that the powerful and reliable verification methods are very important.

1.1.2 Formal Verification

In verification we face the following problem: for a system and a set of requirements, there is a need to either to verify that the system satisfies these requirements or to find an example which demonstrates an incorrect behavior of the system. Several different verification methods can be employed, e.g., inspections, testing, simulation, and formal verification.

None of these methods is superior to others, each of them has its advantages, disadvantages, and domains of application. Here we focus on formal verification. In comparison to the other verification methods, formal verification can give us much higher assurance of system correctness. However, it is a very difficult and time-consuming method. Therefore it is not very convenient for verification of usual software applications, but rather for systems which are safety-critical. At the moment, formal verification is used mainly in the following domains: embedded systems [77], computer network communication protocols [108], supervision of traffic (airlines, railways [26]), systems for space flights [136], hardware [85, 19].

There exist two basic approaches to formal verification:

Deductive methods These methods aim at producing mathematical proof of the satisfaction of the requirements by a system. Although this process can be partially automatized (simple proofs can be constructed algorithmically), deductive methods have to be performed by experts and are very time-consuming.

$$
\begin{array}{lcl}
pc_1 = N & \longmapsto & pc_1 := W \\
pc_1 = W \wedge free = 1 & \longmapsto & pc_1 := C, free = 0 \\
pc_1 = C & \longmapsto & pc_1 := N, free = 1 \\
\\
pc_2 = N & \longmapsto & pc_2 := W \\
pc_2 = W \wedge free = 1 & \longmapsto & pc_2 := C, free = 0 \\
pc_2 = C & \longmapsto & pc_2 := N, free = 1
\end{array}
$$

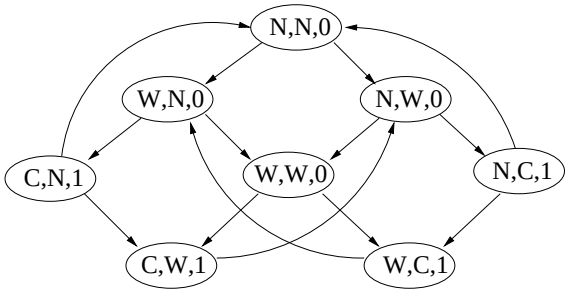


Figure 1.1: Example of a model and its state space.

Automatic methods These methods use brute-force and try to test *all* possible behaviors of the system and verify that all of them satisfy the requirements. This approach is fully automatic and in a case that the system does not satisfy the requirements an automatic method can produce a demonstration of a wrong behavior (counterexample).

1.2 Model Checking

Automatic formal verification methods include model checking. The verification by model checking proceeds in three major phases:

1. modeling,
2. formalization of properties,
3. verification.

Modeling

As the first step, a model of a given system in a suitable mathematical formalism has to be created. In order to apply automatic methods, we need to close the system, i.e., we must model also the environment of the system.

The modeling is done in some formal specification language. In the thesis we consider two specification languages: a guarded command language and timed automata. These are rather low-level specification languages suitable for theoretical treatment. For practical purposes it is necessary to use more high-level languages.

Semantics of specification languages is defined via labeled transition systems. For a model M we denote the semantics $[\![M]\!]$. The semantics describes a state space of the model, i.e., all possible behaviours of the model.

Consider an example in Figure 1.1. The example formalizes a very simple mutual exclusion protocol for two processes. The protocol is formalized in the guarded command language. The figure also shows the state space of the model — the states are tuples $(pc_1, pc_2, \mathit{free})$, transition leads between two states if it is possible to move from one state to another according to the rules given by the model (semantics of the language is formalized in Chapter 2).

Formalization of Properties

To formalize properties of systems, we use formal logics. The thesis is concerned only with verification of properties expressed in predicate logic. Predicate logic can be used to express (some) safety properties — properties of the type "nothing bad happens". This is practically the most useful type of properties. In order to express more complex properties of systems, one has to use temporal logics, e.g., linear temporal logic (LTL) or branching time temporal logics (CTL, CTL^*).

Consider the example in Figure 1.1 again:

1. The mutual exclusion property can be expressed in predicate logic as:

$$\neg(pc_1 = C \wedge pc_2 = C)$$

2. Property "if the process reaches a wait section, then it will eventually enter critical section" can be formalized using LTL as:

$$\mathbf{G}\,(pc_1 = W \Rightarrow \mathbf{F}\,pc_1 = C)$$

The semantics of these logics is defined with respect to transition systems, i.e., we define relation $\models$ between transition systems and formulas such that $T \models \varphi$ iff a transition system T satisfies φ.

Verification

Given a model and a property, we want to check *automatically* whether the model satisfies the property. The model checking problem can be formalized as follows:

Input:	model M, property φ
Question:	does $[\![M]\!] \models \varphi$?

Moreover, in the case that the answer is "no", we require the model checking algorithm to output an counterexample, i.e., a behaviour violating the property.

Model checking algorithms are based on state space exploration — a state space $[\![M]\!]$ of the model, as for example illustrated in Figure 1.1, is explicitly constructed and the property of interest is verified on this state space by suitable graph algorithms.

This is, in fact, a "brute force" approach — in order to verify the property we test all possible behaviours of the model. This could be contrasted with, for example, the theorem proving method which tries to (dis)prove a property by reasoning about the model without an explicit construction of a state space. By taking the "brute force" approach, model checking gains the advantage of being fully automatic. There is, of course, a disadvantage that has to be paid — a state space explosion and consequently high computational requirements of the method.

In the next chapter, we demonstrate the first two steps on a more involved example. The focus of the thesis is, however, on the third step of the process — automatic verification, particularly on fighting the state space explosion problem.

1.2.1 Fighting State Space Explosion

Main disadvantages of model checking are big memory and time requirements. These are caused by so called *state space explosion problem* — the number of states in the state space grows very fast with respect to the size of the model. If we consider the example in Figure 1.1 parametrized by a number n of processes, we find that the size of the state space is $2^n + n \cdot 2^{n-1}$, i.e., the size of the state space grows exponentially for this example.

Fighting state space explosion is a key research direction in the model checking research. There are several basic approaches how to fight this problem:

- Reduce the number of states that need to be explored (abstraction, reductions based on equivalences, compositional methods).
- Reduce the memory requirements needed for storing explored states (storage size reduction, symbolic representation).
- Increase the amount of available memory (distributed environment, magnetic disk).
- Give up the requirement on completeness and explore only part of the state space (heuristics, randomization).

We briefly review main techniques for fighting state space explosion.

Abstraction

Any method which uses models is fundamentally based on abstraction. In model checking, however, the use of abstraction does not end by modeling. Abstraction is very useful also in obtaining smaller state spaces by abstracting away some information from the model. Since abstraction is one of the main themes of this thesis it is discussed in more detail in other places.

Reductions Based on Equivalences

Due to state space explosion, even small models have large number of possible behaviours. Some of these behaviours are, however, "equivalent" (we discuss later in

more detail what equivalent means). It is, therefore, not necessary to explore all behaviours of the model — it is sufficient to explore at least one from each equivalence class. This goal is pursued by different reductions, e.g., symmetry, partial order, cone of influence. Reductions are another important theme of the thesis, so we postpone their discussion as well.

Compositional Methods

It is also possible to exploit the structure of the system. Systems are often specified as a composition of several components. This structure can be exploited in two ways:

– Compositional generation of state spaces [120]. State space is not generated directly, but rather gradually in the following steps:
  1. Generate a state space for each component.
  2. Reduce each of these state spaces according to a suitable behavioral equivalence (usually the bisimulation equivalence).
  3. Compute the final state space from reduced state spaces for components.
– Assume-guarantee approach [158, 99, 81]. For each component we express in temporal logic what it assumes about the rest of the system and what it guarantees. The properties have to be expressed in such a way that the conjunction of all assumptions and guarantees gives the required property of the system. We use model checking to verify that individual components behave properly according to assumptions and guarantees.

Storage Size Reductions

The main source of memory requirements of the exploration algorithm is a data structure which stores already visited states. The memory consumed by this structure can be reduced in several ways:

– State compression [111, 169, 84].
– Do not store all states: caching [91, 83, 157], selective storing [23], sweep line method [54].
– Implicit representation of the data structure [113].

Symbolic Representation

The above discussed techniques for storage size reduction modify the representation of the data structure *States*, but the exploration algorithm remains the same. With symbolic representation, we have to modify the algorithm as well.

The most common type of symbolic representation is based on binary decision diagrams (BDDs). This data structure can be used to concisely represent sets and to efficiently perform operations over sets. Algorithms that use this data structure do not manipulate individual states but rather sets of states [45].

Another approach based on symbolic representation is bounded model checking with SAT [28]. All behaviours of the system up to some fixed depth are characterized by formula in a propositional logic. The correctness of the system (up to a given depth) is expressed as a satisfiability query of this formula. The satisfiability problem is then solved by SAT solver.

More Brute Force

Another approach how to manage a large number of states is to use more brute force, particularly more memory. Classical algorithm usually uses only the RAM memory of a single computer. This can be extended in two ways.

1. Use a magnetic disk. Simple use of magnetic disk leads to an extensive swapping which slows down the computation extremely. So the magnetic disk have to be used in a sophisticated way in order to minimize disk operations [165].
2. Use a distributed environment, e.g., networks of workstations. The exploration algorithm can be easily extended to work in a distributed environment with message passing [164, 132, 82].

Randomized Techniques and Heuristics

If the state space is too large even after the application of the above given techniques, we can use randomized techniques and heuristics. These techniques explore only part of the state space, i.e., under-approximation of the whole state space. Therefore they can help only in the detection of error states; they cannot assist us in proving absence of errors. The advantage is that they scale much better.

The main techniques from this category are the following: heuristic search (e.g., A* search) [97, 123, 161], partial searches and random walks through the state space [107, 135, 100, 119, 137], bitstate hashing [109], genetic algorithm [93], bounded search.

1.2.2 Other Current Trends

Except for fighting state space explosion, which we describe above, there are several other directions currently pursued by model checking researchers:

1. Extensions of techniques to work with infinite state models (e.g., models with infinite data domain variables and unbounded recursion, real-time models, hybrid models, probabilistic models).
2. Combination of model checking with other verification methodologies, e.g., testing, theorem proving, or static analysis.
3. Applications of model checking to new domains (e.g., scheduling, biology).
4. Improvements of front-ends, such as translations from more high-level modeling languages (including programming languages and UML), automatic abstractions, providing user with more informations, making model checkers look more 'debugger-like'.

5. Efficient implementations of model checkers for particular application domains.

Although the thesis is concerned mainly with fighting state space explosion, it also touches some of these other trends, particularly extensions to infinite state models.

1.3 Main Themes

The thesis describes several different techniques for fighting the state space explosion problem in model checking. Although individual chapters consider different aspects of the problem, there are several unifying themes common to all covered topics: behavioral equivalences, abstraction, approximation, and refinement. Since some of these notions are bit overloaded — they are used with several different meanings, even in the computer science community — we discuss them in detail here.

We start the discussion of individual themes by definitions from dictionaries and then we clarify how we use these notions in the thesis. Only intuitive ideas of technical notions are provided here; formal treatment is given in following chapters.

We illustrate notions on a simple labeled transition system (LTS). The "full" system is on Figure 1.2 (a).

1.3.1 Behavioral Equivalences

Equivalent — of the same size, value, importance, or meaning as something else.

(MacMillan English Dictionary)

If we look more closely at state spaces of practical problems, we quickly notice that there is some redundancy — many states/paths in state spaces are in some sense 'equivalent'. Thus it seems plausible to identify these equivalent states/paths and to explore only one from each 'equivalence class'. In order to employ this basic idea, we have to address several basic questions.

What does it mean that two systems are equivalent? This question is more difficult to answer than it seems. There exist many different reasonable notions of equivalence. The basic ones that we are interested in include:

trace equivalence: systems are trace equivalent if they produce the same sets of traces,

(bi)simulation equivalence: systems are (bi)simulation equivalent if they can simulate each others behaviour in a step-wise manner (distinction between simulation and bisimulation is discussed in Chapter 2),

weak equivalences: there are also weak variants of the above mentioned equivalences which do not take into account invisible (internal) actions of the system.

For our example from Figure 1.2 we get: (a) and (b) are equivalent with respect to trace equivalence but not with respect to bisimulation, (a) and (c) are weak bisimulation equivalent.

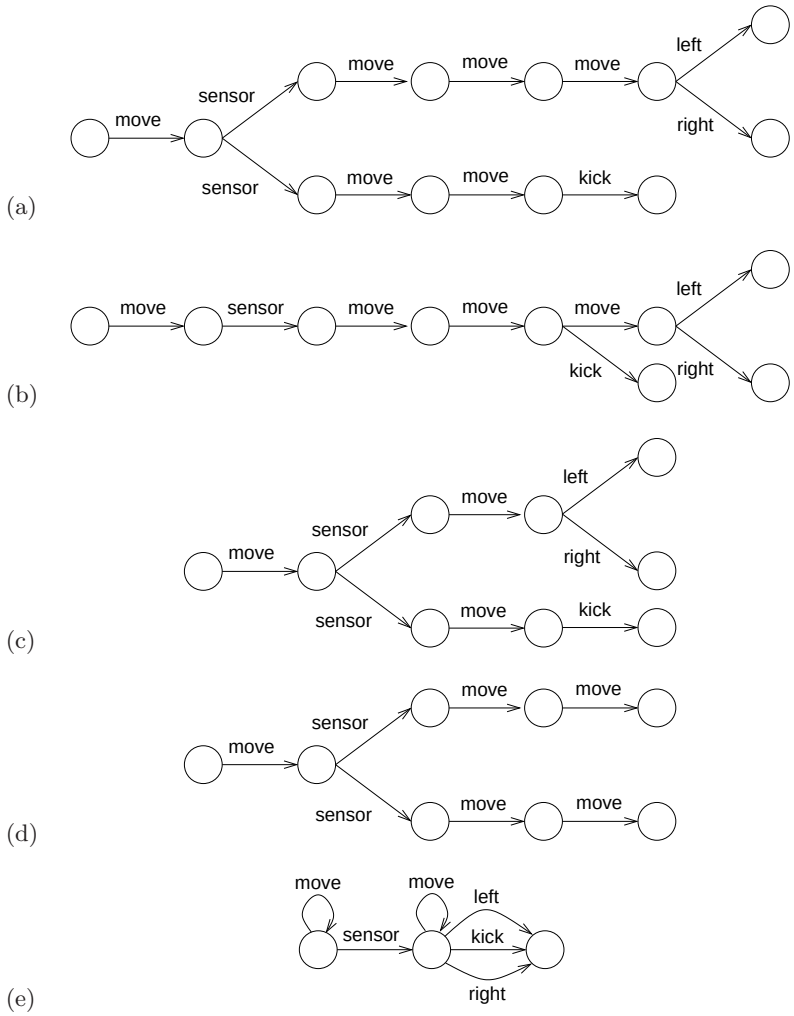


Figure 1.2: Several labeled transition systems.

From point of view of model checking, it is important to study which logics are preserved by which equivalences. A logic is preserved by an equivalence if two equivalent structures satisfy the same formulas in the logic. This problem, as well as formal definition of equivalences, is discussed in Chapter 2.

How do we recognize whether two structures are equivalent? Once we fix an equivalence, a natural questions arises: how do we to decide whether two given structures are equivalent with respect to this equivalence. This is the basic stone of an approach called 'equivalence checking' — specification is given by the user as a small structure and the tool checks whether implementation is equivalent to it. This question is not important to us, since in model checking the specification is given as temporal logic formula. In model checking, we are rather interested in the following question.

How do we construct a smaller equivalent structure? There exist two possible approaches to this question:

– We generate the full state space and *then* reduce it with respect to a given equivalence. Since this approach does not reduce peak memory requirements, it is useful only for model checking expressive logics for which model checking algorithms have high time complexity (like μ-calcul) or for compositional generation of state spaces.
– We produce a smaller equivalent structure *during* the exploration of the full state space (on-the-fly). With this approach we trade some reduction (in the on-the-fly manner we cannot identify all equivalent states/paths), but we reduce the peak memory requirements.

In the thesis we are concerned only with the second approach — on-the-fly reductions.

Which equivalences are preserved by a given reduction? This question is related to on-the-fly reductions. Given some reduction we want to know which equivalences are preserved by this reduction. We encounter this question several times in the thesis.

It is useful to consider not only equivalences, but also preorders (it makes sense to consider preorders with respect to trace and simulation, but not with respect to bisimulation). Preorders are useful for formalizing *approximations*, which are one of the other main themes of the thesis.

1.3.2 Approximations and Refinement

Approximation — a result that is not necessarily exact, but is within the limits of accuracy required for a given purpose

(Webster's Encyclopedic Unabridged Dictionary)

Refinements are small alterations or additions that you make to something in order to improve it.

(Cobuild Learner's Dictionary)

We use approximation in the following meaning: instead of the full state space (e.g., structure (a) in Figure 1.2) we use some approximate structure which is smaller, but in some sense similar (e.g., structures (b), (c), (d), (e) in Figure 1.2). This approximate structure is then used as usual, i.e., model checking over this structure is performed by classical algorithms.

Since approximation has many meanings in computer science, it may be useful to specifically say, what meanings are *not* used here: approximate answer with some probability; approximate solution (not optimal but still good).

The notion of approximation (i.e., what it means similar structure) is in our setting formalized by some preorder. Depending on whether the approximate structure is larger or smaller with respect to the preorder, we talk either about over-approximation or under-approximation. The used preorder is usually either trace

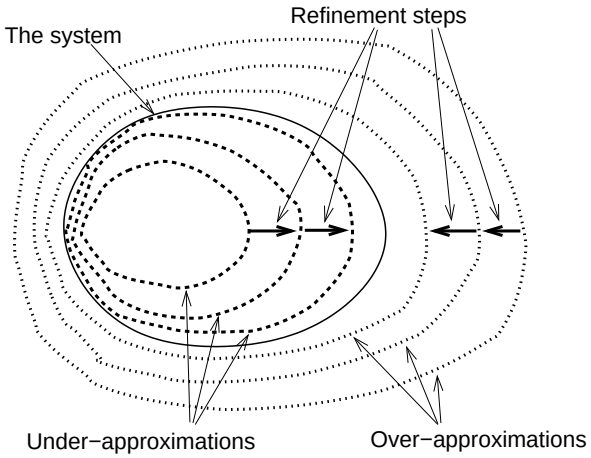


Figure 1.3: Illustration of refinements.

preorder or reachability preorder, in the thesis we implicitly consider approximations with respect to reachability preorder.

Over-approximations contain more behaviours than the full system. Thus if there is no error in an over-approximation then there is no error in the full system. Therefore, over-approximations are suitable for verification. On the other hand, errors found in an over-approximation can be spurious — they need not correspond to any real error in the full system. Thus over-approximations are not very good for detection of errors. Typical way to obtain an over-approximation is *abstraction* (Figure 1.2 (e)).

Under-approximations contain less behaviours than the full system. Thus if there is an error in an under-approximation then this error is a real error in the full system. This makes under-approximations suitable for falsification (error detection). On the other hand, absence of errors in an under-approximation does not imply absence of errors in the full systems. Therefore, under-approximations are not very good for verification. Typical way to obtain an under-approximation is a bounded search (Figure 1.2 (d)).

Approximations are often used together with some kind of *refinement* that is used to produce a sequence of approximations of increasing exactness. Figure 1.3 illustrates the classical setting. We start with either a coarse under-approximation or a coarse over-approximation and by successive refinements we get closer and closer to the full system until we get an approximation which is good enough to answer the question at hand (or until we run out of resources).

Figure 1.4 gives a basic pseudocode for this approach. The meaning of '(in)conclusive answer' depends on what type of approximation we use: if we use under-approximations than error detection is a conclusive answer and no-error result is inconclusive; whereas if we use over-approximations then no-error result is conclusive and error detection is inconclusive.

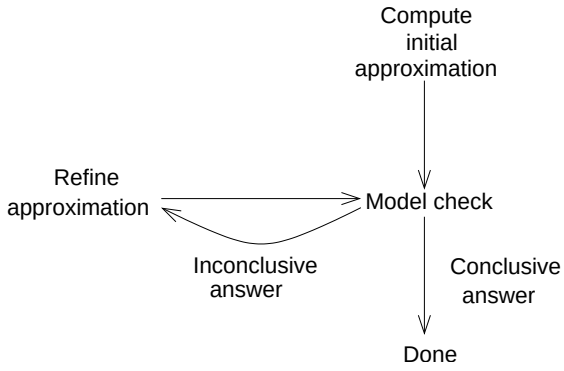


Figure 1.4: Basic refinement loop.

The refinement step often makes use of the result of model-checking step, the most well known instance of this approach is a 'counterexample guided abstraction refinement' (CEGAR). With this approach, we use spurious counterexamples to refine approximations which are obtained by predicate abstraction.

The approximation refinement approach usually leads to semi-algorithms: nothing prevents us from falling into the refinement loop forever. In this thesis, we are bit sloppy about this issue and we often call 'algorithm' what should be more correctly called 'semi-algorithm'.

1.3.3 Abstractions

Abstraction — the act of considering something as a general quality or characteristic, apart from concrete realities, specific objects, or actual instances.

(Webster's Encyclopedic Unabridged Dictionary)

Abstraction is the thought process wherein ideas are distanced from objects.

(Wikipedia)

Abstraction is a key ingredient of any approach that uses models. The model building process is inherently based on abstraction and the success or failure of the model checking activity often depends on the modeler's ability to choose the right level of abstraction. Although this aspect of abstraction is very important, in this thesis we are not concerned with this type of abstraction.

In model checking, the role of abstraction does not end at the modeling process. Different types of abstractions can be used to cope with the state space explosion problem. Here we use abstraction mainly in the following way: we take a model produced by the user (called concrete model) and produce an abstract model which is smaller, i.e., it has smaller state space. The abstract model is used for model checking instead of the concrete one. Optimally, the abstract model is *equivalent* to the concrete one, but more often it is only *approximation* of it. In such a case it may be necessary to perform *refinement* of the abstraction.

Since the notions of abstraction and approximation are often used in confusing ways, we clarify the relation of these two in our setting:

– abstraction often leads to approximation (usually over-approximation), but it can also be exact, i.e., abstract structure can be equivalent to full system with respect to some equivalence,
– not all approximations are obtained by abstraction.

Some authors use the terminology of sound and complete abstractions. Here we prefer to use the terminology of over-approximations and under-approximations, which have the same meaning.

Let us briefly discuss an abstraction on the example from Figure 1.2. Let's imagine that individual states are labeled by position of a brick on the belt and that we use abstraction which distinguishes only two predicates: $position < 2, position > 4$. Then we obtain a system on Figure 1.2 E.

1.4 Contribution and Organization

Having discussed the context of the work, we can state the contribution of the thesis and outline the organization of the rest of the work.

The author has worked on several aspects of model checking, most of them related to fighting state space explosion. He is an author or a co-author of 1 journal paper, 10 conference papers, 4 workshop papers, and 7 technical reports. The thesis is concerned only with abstraction and reduction techniques. We discuss separately contributions contained in the thesis and other author's contributions.

1.4.1 Thesis Contribution

Contributions of the thesis can be divided into three types: technical, innovative, and methodological.

Technical Contribution

The thesis presents several important technical contributions which improve the state-of-the-art of reduction and abstraction techniques. The main technical contributions are the following:

– a decidability result for a non-emptiness problem of timed automata in sampled semantics with an unknown period (Chapter 6); this problem was previously wrongly labeled as undecidable in [5],
– a new extrapolation technique for zone based abstractions of timed automata and a proof of its correctness (Chapter 7); this techniques improves performance of state of the art model checkers,
– a description of an under-approximation refinement algorithm and proofs of its correctness and termination properties (Chapter 5),

– experimental evaluation of on-the-fly reductions which provides a realistic assessment of their usefulness (Chapter 3),
– several novel reduction techniques, particularly a dynamic reduction technique for acyclic systems (Chapter 3),
– demonstration of an error in [142] (Chapter 4).

Innovative Contribution

In Chapter 5 we present a novel under-approximation refinement algorithm for software model checking. This approach provides a novel approach to application of predicate abstraction. Moreover, it shows that there is a viable alternative to the prevailing use of refinement for over-approximations. Therefore, the importance of this contribution is not only in its technical aspects (correctness of the algorithm) but also as an innovative way to the software model checking which can, hopefully, inspire other research in this direction.

Methodological Contribution

Finally, the thesis present systematic overviews of several areas:

– on-the-fly reductions (Chapter 3),
– predicate abstractions techniques (Chapter 4),
– behavioral equivalences between dense and sampled semantics of timed systems (Chapter 6),
– non-emptiness problems for timed systems (Chapter 6).

These overviews present mainly known results, although we also fill some missing pieces. Results, often collected from different notations by different authors, are presented here in a single framework. The presentation provides novel insight into these areas.

Insight obtained from the overview and evaluation of on-the-fly reductions is summarized into an advice for developers of model checking tools.

Related Publications

Some of the results presented in the thesis were published in the following papers:

1. *Lower and Upper Bounds in Zone Based Abstractions of Timed Automata.* Gerd Behrmann, Patricia Bouyer, Kim G. Larsen, and Radek Pelánek. International Journal on Software Tools for Technology Transfer (STTT), Springer, 2005.
2. *Concrete Search with Abstract Matching and Refinement.* C. Păsăreanu, R. Pelánek, and W. Visser. Computer Aided Verification (CAV 2005), LNCS 3576, pages 52-66, Springer, 2005.
3. *On Sampled Semantics of Timed Systems.* P. Krčál and R. Pelánek. Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2005), LNCS 3821, pages 310-321, Springer, 2005.

4. *Evaluation of On-the-fly State Space Reductions.* R. Pelánek. Mathematical and Engineering Methods in Computer Science (MEMICS'05), pages 121-127, 2005.

1.4.2 Other Contributions

Now we briefly review contributions to other areas on which the author worked and which are not part of the thesis.

Structural Properties of State Spaces

State spaces explored by model checking algorithms are directed graphs with some additional information (atomic propositions, labels). Since huge state spaces are generated from rather small descriptions, these graphs are not arbitrary. Using six model checking tools, the author gathered a large set of state spaces, studied their typical properties, and discussed possible applications of these properties in model checking (a SPIN 2004 paper [150]).

Selective Storing and Random Walk

With K. G. Larsen and G. Behrmann we proposed a reachability algorithm which stores only some states during the exploration. We discussed several strategies for selection of states that are stored during the exploration. We also proposed a new strategy based on static analysis of the model (a CAV 2003 paper [23]).

With L. Brim, I. Černá, and T. Hanžl we studied the behavior of random walks on state space. We also studied different enhancements of random walks — techniques which use the available memory to store some information (an FMICS 2005 paper [152]).

Distributed Cycle Detection

This research, conducted with L. Brim, I. Černá, and P. Krčál, focused on detection of cycles in distributed environment with an application to LTL model checking. Results were published on FST-TCS 2001 [40], SOFSEM 2001 [41], and SPIN 2003 [50].

Study of LTL Fragments

This research focused on study of fragments of LTL, hierarchies of these fragments, relations between fragments of LTL and automata on ω-words, and connections to model checking. Results were published on MFCS 2003 [51] (a paper with I. Černá) and CIAA 2005 [153] (a paper with J. Strejček).

Test Case Generation

With C. Păsăreanu and W. Visser we studied application of state space exploration techniques to test case generation for Java units, particularly for container-like classes (an ASE 2005 [148] paper).

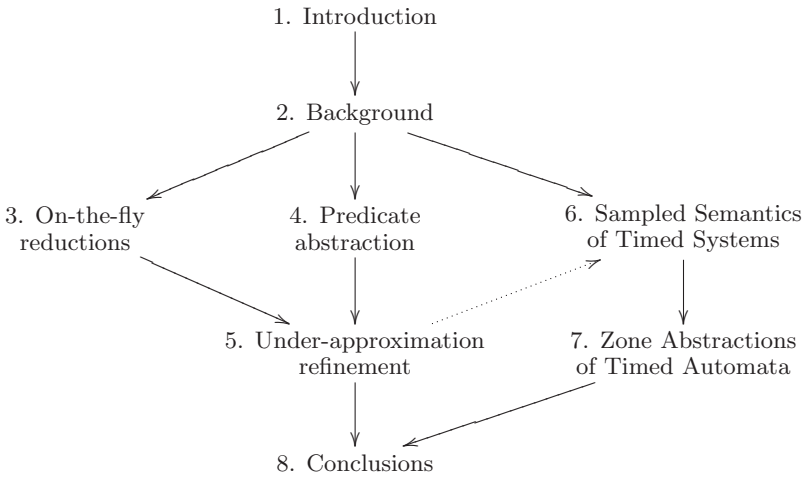


Figure 1.5: Chapter dependencies.

1.4.3 Thesis Organization

The thesis is self-contained — all used notions are formally introduced. We try to illustrate all new notions and techniques on examples. However, several nontrivial (but classical) notions are introduced only formally and not illustrated by example (e.g., bisimulation or region graph). Therefore, a reader who is not familiar with model checking research may have some difficulty reading the text; it could be useful to consult some introductory text, e.g., [56].

Main chapters of the thesis (Chapters 3-7) have the following structure. At the beginning, we give a specific introduction to the content of the chapter and we discuss relation to our main themes. Then we discuss the main content of the chapter and in most cases we also provide some evaluation. Each chapter ends with a separate discussion of related work.

Figure 1.5 gives an overview of relations among chapters. Chapters 3, 4, and 5 are guided mainly towards applications in software model checking, whereas Chapters 6 and 7 are concerned with real-time aspects of model checking.

Chapter 2 gives background. It introduces the mathematical preliminaries necessary for reading the thesis and then it formally defines notions discussed in introduction. It also describes a Sorter example which is used thorough the thesis as a running example.

Chapter 3 deals with on-the-fly reductions of state spaces. This chapter contains mainly an overview and evaluation of different techniques; we also discuss some novel techniques. We concentrate on which *equivalences* are preserved by which reduction techniques and we reformulate some known techniques in terms of *abstraction* (this is useful for next chapters). We also briefly mention *approximation* and *refinement*,

although these are not usually used in this context.

Chapter 4 overviews techniques for predicate *abstraction* and its *refinement*. Predicate abstraction is used to construct *approximations* of the system. Using refinement loop we obtain semi-algorithms for deciding verification problems. Since we cannot guarantee termination, we study different termination properties.

Chapter 5 presents a novel *under-approximation refinement* approach. It combines the idea of concrete search with abstract matching (introduced and discussed in Chapter 3) and predicate *abstraction* with refinement (introduced in Chapter 4).

Chapter 6 is concerned with sampled semantics of timed systems. Sampled semantics can be viewed as *approximation* of the classical dense time semantics. We discuss theoretical results concerning behavioral *equivalences* between dense and sampled semantics. Motivated by *under-approximation refinement*, we also study certain non-emptiness problem of timed automata with sampled semantics and show decidability of this problem.

Chapter 7 deals with zone based *abstractions* of timed automata. We introduce a novel abstraction technique which takes into account more information from the model. This technique preserves less *equivalences* than previously known technique, but leads to coarser abstractions and thus improves significantly the performance of a model checker.

Chapter 8 provides summary of the work, critique, and directions for future work.

Chapter 2

Background

"Well," said Owl, "the customary procedure in such cases is as follows."

"What does Crustimoney Proseedcake mean?" said Pooh. "For I am a Bear of Very Little Brain, and long words Bother me." [139]

In this chapter we formally define some long words, so that they do not bother us (so much). We also illustrate the model checking technique on the Lego© Sorter example, which is used thorough the thesis as a running example.

2.1 Mathematical Preliminaries

Here we discuss some basic mathematical notions. The purpose of this section is to fix the notation — we suppose that the reader is acquainted with these notions.

A *relation* R on sets S_1, S_2 is defined as a set of pairs ($R \subseteq S_1 \times S_2$). We use either set notation ($(s_1, s_2) \in R$) or infix notation ($s_1 R s_2$). Infix notation is used particularly in the case of a transition relation ($s_1 \longrightarrow s_2$).

Let S be a set and R a relation over $S \times S$. The relation R is a *preorder* if it is reflexive and transitive. It is a *partial order* if it is reflexive, transitive, and anti-symmetric. For partial order, the pair (S, R) is called a *partially ordered set* (shortly a *poset*). The relation R is an an *equivalence* relation if it is reflexive, transitive, and symmetric. An *equivalence class* of s is $[s]_R = \{s' \mid (s, s') \in R\}$.

A *core* of a function $f : A \to B$ is a relation $\equiv_f \subseteq A \times A$ defined as follows: $a \equiv_f b \Leftrightarrow f(a) = f(b)$. We also say that f induces a relation $\equiv_f$. Note that a core is an equivalence. Function f between two posets $(P, \leq_P)$ and $(Q, \leq_Q)$ is *monotone* iff $\forall p, q \in S, p \leq_P q \Rightarrow f(p) \leq_Q f(q)$.

A *Galois connection* between two posets P, Q is a tuple of function (α, γ), where $\alpha : P \to Q$ (an abstraction function), $\gamma : Q \to P$ (a concretization function) satisfy: $x \leq_P \gamma(y) \Leftrightarrow \alpha(x) \leq_Q y$ (or equivalently α, γ are monotone and satisfy $x \leq_P \gamma(\alpha(x)), \alpha(\gamma(x)) \leq_Q x$). Given the function α, the function γ is uniquely determined (and vice versa). Therefore, we sometimes specify a Galois connection only by α (resp. γ).

2.2 Labeled Transition Systems

As a semantics of a model, we use labeled transition systems (LTS). Another way to define semantics of models is to use Kripke structures (KS). A Kripke structure does not have action names on transition, but has labels on states. For results presented in this thesis this distinction is purely technical and the choice (LTS versus KS) is rather arbitrary. It would be possible to formulate all the results in terms of Kripke structures[1].

An LTS is a tuple $T = (S, Act, \longrightarrow, s_0)$ where S is a (possibly infinite) set of states, Act is a finite set of actions containing a special invisible action τ, $\longrightarrow \subseteq S \times Act \times S$ is a transition relation, and $s_0 \in S$ is an initial state.

At first, we introduce some notation:

- $s \longrightarrow s'$ iff $\exists a \in Act : s \xrightarrow{a} s'$,
- $s \longrightarrow^* s'$ iff there exists sequence of states $s_1, s_2, \ldots, s_n$ such that $s = s_1, s_n = s'$ and for each $1 \leq i < n : s_i \longrightarrow s_{i+1}$,
- $s \stackrel{a}{\Longrightarrow} s'$ iff $\exists s_1, s_1'$ such that $s \xrightarrow{\tau}{}^* s_1 \xrightarrow{a} s_1' \xrightarrow{\tau}{}^* s'$.

State s is deadlocked if there is no outgoing transition from s. A set of *reachable actions* $RA(T)$ is a set $\{a_i \in Act \mid s_0 \longrightarrow^* s_n \xrightarrow{a_i} s_{n+1}\}$. A set of reachable states is a set $\{s \mid s_0 \longrightarrow^* s\}$. We say that LTS is *finite* if the set of reachable states is finite[2].

A *run* of T over a trace $w \in Act^* \cup Act^\omega$ is a sequence of states $s_0, s_1, \ldots$ starting in the initial state such that $s_i \xrightarrow{w(i)} s_{i+1}$. A *weak run* of T over a weak trace $w \in \{Act \setminus \tau\}^* \cup \{Act \setminus \tau\}^\omega$ is a sequence of states $s_0, s_1, \ldots$ starting in the initial state such that $s_i \stackrel{w(i)}{\Longrightarrow} s_{i+1}$.

A *language* (resp. an *ω-language*) of the transition system T is a set of finite (resp. infinite) traces:

$$\begin{array}{lcl} L(T) & = & \{w \in Act^* \mid \text{ there exists a run of } T \text{ over } w\} \\ L_\omega(T) & = & \{w \in Act^\omega \mid \text{ there exists a run of } T \text{ over } w\} \end{array}$$

A *weak language* (resp. an *weak ω-language*) of the transition system T is a set of finite (resp. infinite) weak traces:

$$\begin{array}{lcl} WL(T) & = & \{w \in \{Act \setminus \tau\}^* \mid \text{ there exists a weak run of } T \text{ over } w\} \\ WL_\omega(T) & = & \{w \in \{Act \setminus \tau\}^\omega \mid \text{ there exists a weak run of } T \text{ over } w\} \end{array}$$

Let $X \subseteq S$ be a set of states. The *weakest precondition* of the set X with respect to an action a_i is a set $pre(a_i, X) = \{s \mid \forall s \xrightarrow{a_i} s' : s' \in X\}$. The *strongest*

[1] Note, for example, that the paper [151], which is the base of Chapter 3, uses Kripke structures to formalize results. Also note that in the case of Kripke structures it is customary to talk about stutter equivalences rather than weak equivalences.

[2] Note that this terminology is a bit nonstandard, but it is useful in our setting, where we use the same specification language (guarded command language) to specify both finite and infinite state systems.

postcondition of the set X with respect to an action a_i is a set $post(a_i, X) = \{s \mid \exists s' \in X : s' \xrightarrow{a_i} s\}$.

One focus of this thesis is an abstraction. In general, we consider abstraction to be an operator on LTSs. An abstraction operator (usually denoted $\mathcal{A}$) takes a LTS T and produces another LTS T'. In some cases T' is over the same set of states (i.e., abstraction only changes the transition relation). We usually apply abstraction to an LTS obtained as a semantics of some model ($[\![M]\!]$). We call abstraction *effectively computable* if it can be computed directly from the model M without going through $[\![M]\!]$ (this can be impossible, since $[\![M]\!]$ is sometimes infinite).

2.3 Specification Languages

To make the presentation readable, we use very simple specification languages. To model software (respectively software aspects of systems) we use a guarded commands language over integer variables. To model real-time systems we use basic timed and stopwatch automata.

Techniques discussed in the thesis carry to more sophisticated specification languages (e.g., with data types, synchronization, communication, invariants, pointers) which are necessary for practical applications. But the simple specification languages that we use are more suitable to explain the essence of presented techniques.

2.3.1 Guarded Command Language

Syntax

Let V be a finite set of integer variables. Expressions over V are defined using standard boolean $(=, <, >)$ and binary $(+, -, \cdot, ...)$ operations. Act is a set of action names. A *model* is a tuple $M = (V, E)$; $E = \{t_1, \ldots, t_n\}$ is a finite set of transitions, where $t_i = (a_i, g_i, u_i)$ with $a_i \in Act$, predicate g_i (a boolean expression over V), and update $u_i(\vec{x})$ (a sequence of assignments over V).

Semantics

The semantics of a model is an LTS. States are variable valuation $V \to \mathbb{Z}$. The semantics of a predicate ϕ (a relation $s \models \phi$) and the the semantics of an update u_i (a function $[\![u_i]\!] : \vec{x} \mapsto u_i(\vec{x})$) are defined as usual. The semantics of model M is an LTS $[\![M]\!] = (S, Act, \to, s_0)$ where

– $S = 2^{V \to \mathbb{Z}}$,
– $s \xrightarrow{a_i} s'$ iff there exists $(a_i, g_i, u_i) \in T$ such that $s \in [\![g_i]\!], s' = u_i(s)$,
– s_0 is the zero valuation $(\forall v \in V : s_0(v) = 0)$.

We also use the following notation:

– $[\![\phi]\!]$ is a set $\{s \in 2^{V \to \mathbb{Z}} \mid s \models \phi\}$

– $s(x)$ is a value of variable x in state s,
– $s[x := a]$ is a state s' such that $s'(x) = a$ and $\forall y \in V, y \neq x : s'(y) = s(y)$.

2.3.2 Timed and Stopwatch Automata

Syntax

Let $\mathcal{C}$ be a set of non-negative real-valued variables called *clocks*. The set of guards $G(\mathcal{C})$ is defined by the grammar $g ::= x \bowtie c \mid x - y \bowtie c \mid g \wedge g$ where $x, y \in \mathcal{C}, c \in \mathbb{N}_0$ and $\bowtie \in \{<, \leq, \geq, >\}$. A *stopwatch automaton* is a tuple $A = (L, Act, \mathcal{C}, q_0, E, stop)$, where:

– L is a finite set of locations,
– $\mathcal{C}$ is a finite set of clocks,
– $l_0 \in L$ is an initial location,
– $E \subseteq L \times Act \times G(\mathcal{C}) \times 2^{\mathcal{C}} \times L$ is a set of edges labeled by an action name, a guard, and a set of clocks to be reseted,
– $stop : L \to 2^{\mathcal{C}}$ assigns to each location a set of clocks that are stopped at this location.

A clock $x \in \mathcal{C}$ is called a *stopwatch* clock if $\exists q \in L : x \in stop(q)$. We use the following special types of stopwatch automata:

– a *timed automaton* is a stopwatch automaton such that there are no stopwatch clocks (i.e., $\forall l \in L : stop(l) = \emptyset$),
– a *closed* automaton uses only guards with $\{\leq, \geq\}$,
– a *diagonal-free* automaton uses only guards defined by $g := x \bowtie c \mid g \wedge g$.

We also consider combinations of these types, e.g., closed timed automaton.

Semantics

Semantics is defined with respect to a given time domain D. We suppose that time domain is a subset of real numbers which contains 0 and is closed under addition. A *clock valuation* is a function $\nu : \mathcal{C} \to D$. If $\delta \in D$ then a valuation $\nu + \delta$ is such that for each clock $x \in \mathcal{C}$, $(\nu + \delta)(x) = \nu(x) + \delta$. If $Y \subseteq \mathcal{C}$ then a valuation $\nu[Y := 0]$ is such that for each clock $x \in \mathcal{C} \setminus Y$, $\nu[Y := 0](x) = \nu(x)$ and for each clock $x \in Y$, $\nu[Y := 0](x) = 0$. The satisfaction relation $\nu \models g$ for $g \in G(\mathcal{C})$ is defined in the natural way.

The semantics of a stopwatch automaton $A = (L, Act, \mathcal{C}, l_0, E, stop)$ with respect to the time domain D is an LTS $[\![A]\!]_D = (S, Act, \to, s_0)$ where $S = L \times D^{\mathcal{C}}$ is the set of states, $s_0 = (l_0, \nu_0)$ is the initial state, $\nu_0(x) = 0$ for all $x \in \mathcal{C}$. Transitions are defined with the use of two types of basic steps:

– time step: $(l, \nu) \xrightarrow{delay(\delta)} (l, \nu')$ if $\delta \in D, \forall x \in stop(l) : \nu'(x) = \nu(x), \forall x \in \mathcal{C} \setminus stop(l) : \nu'(x) = \nu(x) + \delta$,

– action step: $(l,\nu) \xrightarrow{action(a)} (l',\nu')$ if there exists $(l,a,g,Y,l') \in E$ such that $\nu \models g, \nu' = \nu[Y := 0]$.

The transition relation of $[\![A]\!]_D$ is defined by concatenating these two types of steps: $(l,\nu) \xrightarrow{a} (l',\nu')$ iff there exists (l'',ν'') such that $(l,\nu) \xrightarrow{delay(\delta)} (l'',\nu'') \xrightarrow{action(a)} (l',\nu')$.

We consider the following time domains: $\mathbb{R}_0^+, \mathbb{Q}_0^+, \{k \cdot \epsilon \mid k \in \mathbb{Z}_0^+\}$. The semantics with respect to the last domain is denoted $[\![A]\!]_\epsilon$ (also called *sampled semantics*). We use the following shortcut notation: $L(A) = L([\![A]\!]_{\mathbb{R}_0^+}), L_\omega(A) = L_\omega([\![A]\!]_{\mathbb{R}_0^+}), L^\epsilon(A) = L([\![A]\!]_\epsilon), L^\epsilon_\omega(A) = L_\omega([\![A]\!]_\epsilon)$.

2.4 Basic Algorithm

Model checking algorithms are based on exhaustive search, i.e., on construction of the full state space of the model. Figure 2.1 gives the basic algorithm for construction of the state space. The algorithm takes as an input a model M and constructs reachable part[3] of $[\![M]\!]$.

The algorithm is a classical graph traversal algorithm. It starts in the initial state and then generates all reachable states using transitions according to the given model. We give the algorithm explicitly, because later in the thesis we present several modifications and improvements of this algorithm. The algorithm uses the following data structures (we stick to this notation for the rest of the thesis):

– *States* is a set of states that were visited during the exploration. Since we need to perform a test of membership in this data structure, the set is usually represented by hash table.
– *Transitions* is a set of transition that were traversed during the exploration.
– *Wait* is a set of states that need to be explored. The implementation of this data structure (queue/stack) determines the search order of the algorithm.

Once we have generated $[\![M]\!]$, we can directly use it to verify simple safety properties. For this purpose it is sufficient to use a simple reachability analysis in $[\![M]\!]$. In this thesis we consider mainly this type of properties — they are, anyway, the practically most useful ones. Therefore, we are concerned mainly with generation and exploration of $[\![M]\!]$.

Model checking problems for more complex properties are usually reduced to some non-emptiness problem and this problem is in turn solved by some graph algorithm. For example, model checking of LTL logic can be reduced to Buchi automata non-emptiness which can be solved by cycle detection. We do not discuss these algorithms in more detail; an interested reader is referred to [56].

[3] In this thesis, we are bit sloppy about the distinction between $[\![M]\!]$ and reachable part of $[\![M]\!]$. It is clear that with respect to all verification problem, only reachable part of the state space is important.

```
proc GenerateLTS(M)
  add s_0 to Wait
  add s_0 to States
  while Wait ≠ ∅ do
        remove s from Wait
        foreach s -a_i-> s' do
          add (s, a_i, s') to Transitions
          if s' ∉ States then
                              add s' to Wait
                              add s' to States
          fi od
  od
  return (States, Transitions, s_0)
end
```

Figure 2.1: The basic algorithm for generating $[\![M]\!]$.

2.5 Behavioral Equivalences

Definitions

Let $T_1 = (S_1, Act, \rightarrow_1, s_0^1)$, $T_2 = (S_2, Act, \rightarrow_2, s_0^2)$ be two labeled transitions systems. A relation $R \subseteq S_1 \times S_2$ is a *simulation relation* iff for all $(s_1, s_2) \in R$ and $s_1 \xrightarrow{a}_1 s_1'$ there is s_2 such that $s_2 \xrightarrow{a}_2 s_2'$ and $(s_1', s_2') \in R$. System T_1 is simulated by T_2 if there exists a simulation R such that $(s_0^1, s_0^2) \in R$. A relation R is a *bisimulation relation* iff R is a symmetric simulation relation. A *bisimulation* $\sim$ is the largest bisimulation relation.

– *reachability equivalent* iff $RA(T_1) = RA(T_2)$,
– *deadlock equivalent* iff T_1 contains reachable deadlocked state only if T_2 contains reachable deadlocked state,
– *trace equivalent* iff $L(T_1) = L(T_2)$,
– *infinite trace equivalent* iff $L_\omega(T_1) = L_\omega(T_2)$,
– *simulation equivalent* iff T_1 simulates T_2 and vice versa,
– *bisimulation equivalent* (bisimilar) iff $s_0^1 \sim s_0^2$.

By using weak language WL instead of language L we define, weak (infinite) trace equivalence. Using $\Longrightarrow$ instead of $\longrightarrow$ we define weak (bi)simulation equivalence.

For some results we need a technical notion of a k-bisimulation. Any relation $R \subseteq S_1 \times S_2$ is a 0-bisimulation relation. A relation $R \subseteq S_1 \times S_2$ is a $(k+1)$-bisimulation relation ($k \geq 0$) if there exists a k-bisimulation relation R' such that for all $(s_1, s_2) \in R$:

– if $s_1 \xrightarrow{a_i}_1 s_1'$ then there is s_2' such that $s_2 \xrightarrow{a_i}_2 s_2'$ and $(s_1', s_2') \in R'$
– if $s_2 \xrightarrow{a_i}_2 s_2'$ then there is s_1' such that $s_1 \xrightarrow{a_i}_1 s_1'$ and $(s_1', s_2') \in R'$

A k-bisimulation is the largest k-bisimulation relation, denoted $\sim_k$. Note that bisimulation is a k-bisimulation relation for each k.

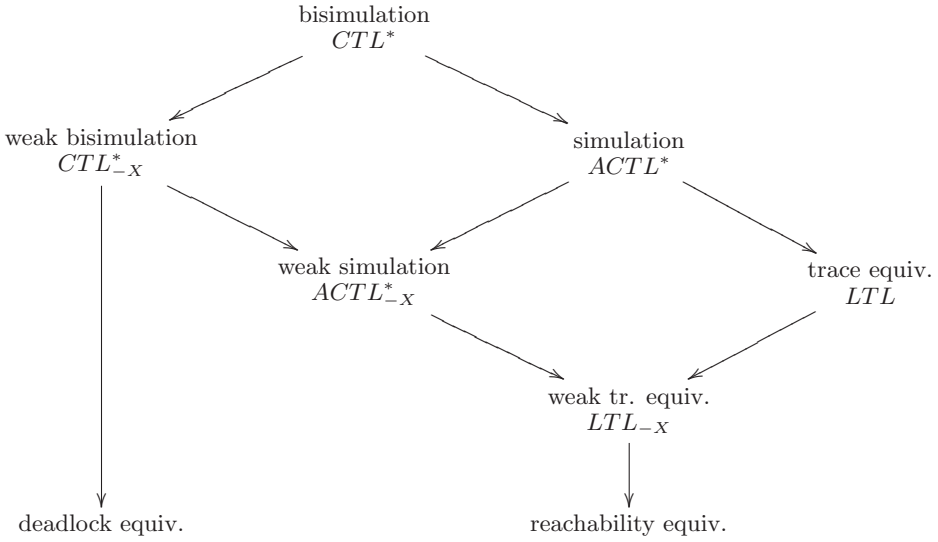


Figure 2.2: Relations among equivalences and temporal logics. For each equivalence we give temporal logic which is preserved by the equivalence. Proofs of can be found in [43, 99, 156].

Relations

Figure 2.2 shows relations among different equivalences. It also gives for each behavioral equivalence a temporal logic which is preserved by the equivalence (temporal logic is preserved by an equivalence if two equivalent structures satisfy the same set of formulas).

Bisimulation Quotient

A *bisimulation quotient* $T_\sim$ is an LTS $(S_\sim, Act, \rightarrow_\sim, [s_0]_\sim)$, where $S_\sim$ is the set of equivalence classes of $\sim$, $[s_0]_\sim$ is the equivalence class containing s_0, and $\rightarrow_\sim$ is defined as $C_1 \xrightarrow{a_i} C_2 \Leftrightarrow \exists s_1 \in C_1, s_2 \in C_2, s_1 \xrightarrow{a_i} s_2$. Bisimulation quotient $T_\sim$ is, trivially, bisimilar to T.

2.6 Sorter Example

Finally, we describe a Lego© Sorter example. This example is used throughout the thesis to illustrate different notions and techniques. This example is not a classical case study — it was consciously constructed in order to be *illustrative*, so it is in some sense artificial. Nevertheless it still illustrates nicely many aspects of a typical application of the model checking method.

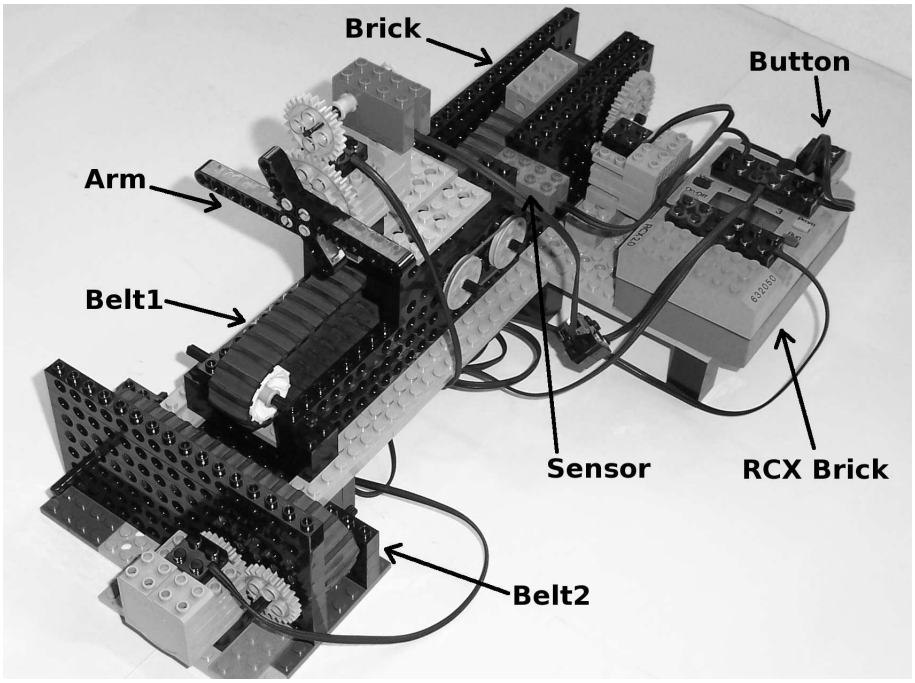


Figure 2.3: The Lego© Sorter system.

2.6.1 Description of the System

The example is a sorter of bricks built using a Lego© Mindstorms systems. The system is depicted on Figure 2.3. Lego© Mindstorms is an expansion of a popular Lego© system: it contains sensors, motors, and a RCX brick with a small processor. The RCX brick has six communication ports: three sensor inputs and three actuator outputs. Control programs for the RCX brick can be written in several languages, one of them an NQC (Not Quite C) language which is similar to the language C. Example of an NQC code is given in Figure 2.4. Lego© Mindstorms systems have been used in several verification case studies [117, 130, 46], our example is significantly influenced by Iversen et al. [117] (our sorter is slightly more complex than theirs).

The Sorter consists of the following parts:

– 2 belts which are used to transport bricks,
– a light sensor which can detect passing bricks,
– an arm which can kick bricks from the belt[4],
– a button which is used to "order" bricks for processing.

The intended behaviour of the system is the following. Bricks are placed by the

[4]The system also contains rotation sensor which is used to stop rotation of the arm. For the sake of simplicity, we ignore this sensor in our discussion.

```
task light_sensor_control() {
  int x=0;
  while (true) {
    if (LIGHT > LIGHT_TRESHOLD) {
      PlaySound(SOUND_CLICK);
      Wait(30);
      x = x + 1;
    } else {
      if (x>2) {
        PlaySound(SOUND_UP);
        ClearTimer(0);
        brick = LONG;
      } else if (x>0) {
        PlaySound(SOUND_DOUBLE_BEEP);
        ClearTimer(0);
        brick = SHORT;
      }
      x = 0;
    }
  }
}
```

Figure 2.4: An example of an NQC code.

user on the first belt. Bricks which are too long (length is detected with the use of light sensor) are kicked out from the belt by the arm. Short bricks are transported to the second belt. The second belt transports them either to a "processing" side or to a "not-processing" side depending on whether a brick has been ordered by pressing the button.

Although the system is rather simple and artificial, it has several features typical for embedded systems:

– the system runs on a dedicated, small processor,
– the system interacts with non-trivial environment,
– the system runs parallel software (RCX brick support up to 10 parallel tasks, our implementation contains 5 tasks),
– the system contains many real-time aspects and its behaviour is very sensitive to timing.

All these features make it very difficult to debug the system using standard testing approaches[5] and thus a reasonable candidate for the use of formal verification methods.

2.6.2 Modeling the System

In order to use the model checking method, we have to model the system in a formal modeling formalism. The modeling process is not a routine work — it requires a certain degree of craftsmanship. It is necessary to use a right degree of abstraction,

[5] In this case, author really talks from his own experience.

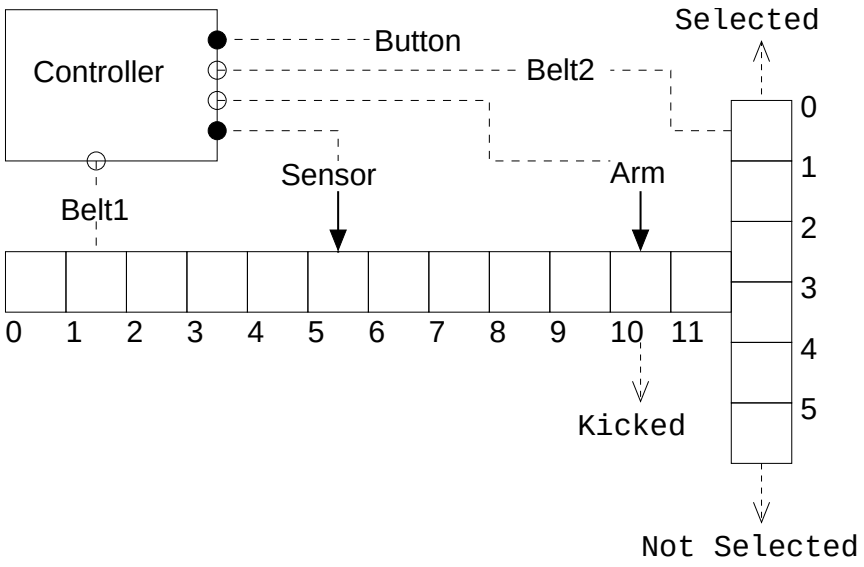


Figure 2.5: A schematic diagram of the Sorter system. Black circles denote inputs from sensors, white circles denote outputs to actuators.

particularly when modeling the environment. We have to include in a model all the relevant information and at the same time keep the model as simple as possible.

The basic structure of a possible model[6] is shown in Figure 2.5. The diagram shows schematically main parts of the system and connections among them. The most significant abstraction in our model is the discretization of space: we have divided the belt into fixed, finite number of positions and we consider bricks only in these positions.

Now we need to choose a suitable modeling language and express the environment and the control programs in this formalisms. We use the guarded command language and timed automata. These specification languages are defined in Chapter 2, full models are given in Appendix B. To give the reader a basic impression of the formalisms used, we present here just fragments of models corresponding to the light controller, see Figure 2.6.

2.6.3 Model Checking

As a next step we need to express and formalize properties that the system should satisfy. For our system, properties of interest include:

1. On/off commands to motors alternate.
2. Long bricks are always kicked out of the belt.
3. Short bricks are never kicked out of the belt.

[6]Since the main goal of this example is to be *illustrative*, the provided model is rather simple. For more realistic verification, we would need to use a bit more detailed model, particularly of brick's positions on the belt.

4. The arm never kicks without hitting a brick.
5. Every brick inserted on the belt eventually leaves the sorter.
6. If the button is pressed then the next short brick is delivered to the "processing" site.
7. The number of bricks delivered to the "processing" site is never larger then the number of times the button was pressed.

The first four properties are typical safety properties (nothing bad happens). These properties can be checked by simple reachability analysis. In this thesis we are concerned mainly with this type of properties.

Once we have a formal model of the system and a formal formulation of the problem, we can use a model checker to automatically check whether the system satisfies the property. In our case, the systems works correctly only if bricks are placed on the belt with an sufficient interval. If two bricks are placed on the belt in a short succession then the system invalidates several of the properties — these behaviours can be found and explored with the use of the model checker.

This example also illustrates the weakness of model checking: state space explosion. Table 2.1 shows the increase in number of states with respect to number (and type) of bricks in the model.

Guarded Command Language

$$
\begin{array}{lcl}
LC = 0 \wedge token = 3 \wedge LightSensorLevel = 0 & \longmapsto & token := 4 \\
LC = 0 \wedge token = 3 \wedge LightSensorLevel = 1 & \longmapsto & LC := 1, x := 1, token := 4 \\
LC = 1 \wedge token = 3 \wedge LightSensorLevel = 1 & \longmapsto & x := x + 1, token := 4 \\
LC = 1 \wedge token = 3 \wedge LightSensorLevel = 0 & \longmapsto & LC := 2, timer := 0 \\
LC = 2 \wedge x \leq 2 & \longmapsto & LC := 0, brick := 3, token := 4 \\
LC = 2 \wedge x > 2 & \longmapsto & LC := 0, brick := 4, token := 4
\end{array}
$$

Timed Automata

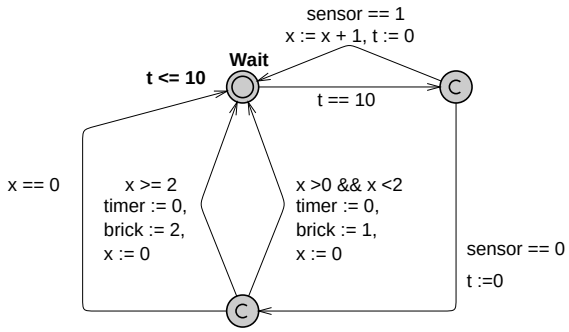


Figure 2.6: Snapshots of formal models of the light controller task from Figure 2.4.

1 long brick	7,592
1 short brick	20,544
2 long bricks	296,148
2 short bricks	1,288,478
1 short and 1 long brick	> 4,000,000

Table 2.1: Size of the state space of the Sorter example.

Chapter 3

On-the-fly Reductions

"In that direction," the Cat said, waving its right paw round, "lives a Hatter: and in that direction," waving the other paw, "lives a March Hare. Visit either you like: they're both mad."

In many cases, there is some redundancy in the state space. For example, we can sometimes visit either branch we like: they're both equivalent. This chapter is concerned with techniques based on such observations — reduction techniques which reduce the size of the state space during the exploration. The focus is on techniques which perform the reduction by matching on abstract states. We pay special attention to acyclic systems. We also briefly mention reductions which preserve weak equivalences (partial order reduction and similar techniques). The chapter contains extensive evaluation of several reduction techniques on realistic model checking case studies and challenges the common beliefs in their usefulness.

3.1 Introduction

As we discuss in Chapter 1, state spaces contain many redundancies — there are many equivalent states/paths (with respect to some suitable behavioral equivalence). For example, there is some kind of symmetry in our sorter example: if we have two bricks of the same type on the belt, it is not necessary to distinguish whether the first brick is on position 1 and the second brick on position 8 or vice versa.

As another example, let us consider an example in Figure 3.1. The full state space of the model has 10 states, whereas the shown reduced structure has only 2 states. The reduced structure is weakly bisimilar to the full state space and hence it is sufficient for most verification purposes.

To generalize this observation, we can say that instead of checking the specification over the (very big) state space we can check it over some (smaller) equivalent structure. One possibility is to generate the whole state space, reduce it by suitable equivalence, and finally check the specification over the reduced structure. This approach, however, does not reduce peak memory requirements which are the main practical limitation of model checking.

Another possibility is to employ static analysis and use specific information about the model to compute a reduced structure on-the-fly. There are many techniques for

$$
\begin{array}{lcl}
(\ a, x = 0 & \longmapsto & x := 1\ ) \\
(\ b, x = 1 \wedge y \geq 1 & \longmapsto & x := 0, z := 1\ ) \\
(\ \tau, y < 2 & \longmapsto & y := y + 1\ )
\end{array}
$$

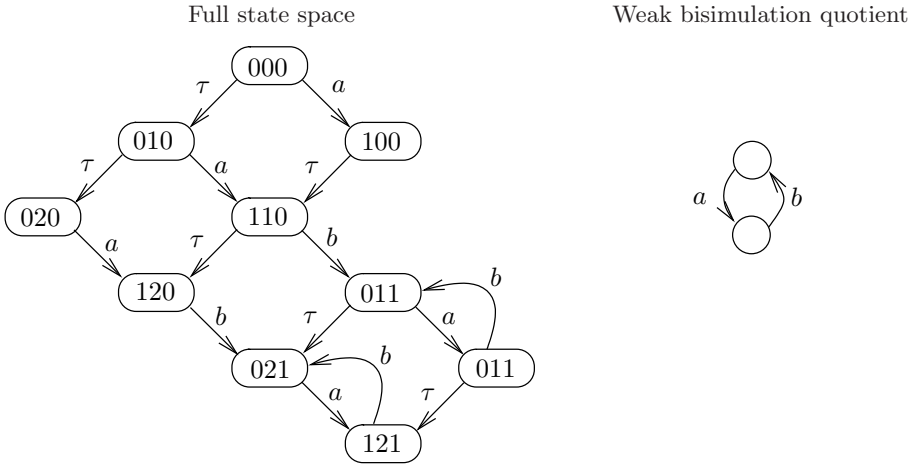


Figure 3.1: A simple program, the corresponding full state space (states are given as vectors (x, y, z)), and a reduced structure.

such on-the-fly reduction. In this chapter we provide an overview of these techniques, propose some novel ones, and give an experimental evaluation.

The stress is on the approach which we call *concrete search with abstract matching* (αSEARCH algorithm). We formulate some well-know techniques in this setting (symmetry reduction, dead variable reduction) and propose some novel ones (based on identification of equivalent values, equivalent states, and on linear transformations of variables). This part also serves as a background for Chapter 5 in which we extend the approach to an under-approximation refinement scheme.

The concrete search with abstract matching approach uses statical abstraction function which is determined *prior* to the search. The abstraction function is chosen in such a way, that it preserves bisimulation classes. Another option is to approximate bisimulation classes *during* the search. This approach can lead to more significant reduction, but it introduces additional overhead during the generation which can outweigh benefits of the reduction. We study this approach for a special case of acyclic state spaces. Acyclicity can be exploited to make the technique more efficient.

We also discuss reductions preserving weak equivalences, particularly the partial order reduction, and we present some novel techniques preserving reachability and deadlock equivalence.

As can be seen from the above given discussion, there are many different reduction techniques. However, it is not clear what are the practical merits of these reductions.

Researchers usually demonstrate the effectiveness of proposed reduction on just one or two (well selected) examples. Often we can find claims about exponential or at least "drastic" reduction. Are these claims appropriate?

We give a realistic evaluation of merits of discussed reductions. For the evaluation we use three model checking tools and many case studies previously studied in the literature. The results show that the effect of reductions can be significant but it is not so drastic as often claimed in the literature.

Relationship to Main Themes

– Equivalences. For each reduction technique, it is important to determine which equivalences are preserved by this reduction. The chapter is organized according to equivalences preserved by reductions.
– Abstractions. Some reduction techniques can be formulated in terms of abstraction function and matching on abstract states. We pay special attention to this class of reductions.
– Approximations and refinement. Most of the reductions that we study in this chapter are exact (with respect to some equivalence). But we also briefly mention some approximate techniques and possibilities for their refinement.

3.2 Reductions Preserving Bisimulation

At first, we study reductions that try to reduce the number of explored equivalent states — the equivalence of choice here is bisimulation. The basic idea is simple: when we reach a new state, we check whether we have already seen some bisimilar state during the exploration. It is, however, not so easy to implement this idea.

We introduce a general algorithm, called αSEARCH, which is based on exploring concrete state space while matching on abstract states. This general algorithm is used to formulate several classical reduction techniques. In Chapter 5 we extend this algorithm into a novel, more involved technique.

3.2.1 αSearch Algorithm

Figure 3.2 shows the reachability procedure that performs model checking with abstract matching. It is basically a concrete state space exploration with matching on abstract states; the main modification with respect to classical state space exploration is that we store $\alpha(s)$ instead of s. At the moment, let abstraction function α be an arbitrary function.

The following lemma states that αSEARCH explores under-approximation of the state space.

Lemma 3.1 $RA(\alpha\text{SEARCH}(M, \alpha)) \subseteq RA([\![M]\!])$.

Proof: Since the algorithm explores only concrete transitions, all states/transitions in the generated state space are reachable in the concrete one. It is easy to verify

```
proc αSearch(M, α)
  add s_0 to Wait
  add α(s_0) to States
  while Wait ≠ ∅ do
        remove s from Wait
        foreach s -a_i-> s' do
          add (α(s), a_i, α(s')) to Transitions
          if α(s') ∉ States then
                                add s' to Wait
                                add α(s') to States
          fi od
  od
  return (States, Transitions, s_0)
end
```

Figure 3.2: The αSEARCH algorithm.

that the following is an invariant of the algorithm: *Wait* is a subset of reachable states. □

The structure computed by αSEARCH depends on the search order. This is illustrated by the following example.

Example 3.1 *Let us consider the example from Figure 3.1 and an abstraction function* $\alpha([x, y, z]) = [x = 0, y = 0, x + y > 1]$ *(this is an example of a predicate abstraction function; we use this type of abstraction functions intensively in the following chapters).*

Algorithm αSEARCH *with the breadth-first search order explores the following states:* 000 *(stores* 110*),* 010 *(stores* 100*),* 100 *(stores* 010*),* 020 *(stores* 101*),* 110 *(stores* 001*),* 120 *(matched on* 001*),* 011 *(matched on* 100*).*

With depth-first search order the following states are explored: 000 *(stores* 110*),* 010 *(stores* 100*),* 020 *(stores* 101*),* 120 *(stores* 001*),* 021 *(matched on* 101*),* 100 *(stores* 010*),* 110 *(matched on* 001*).*

We say that abstraction function α is *exact* if the core $\equiv_\alpha$ of the function α is a bisimulation on $[\![M]\!]$. The following lemma justifies this terminology.

Lemma 3.2 *If an abstraction function* α *is exact then* αSEARCH(M, α) *is bisimilar to* $[\![M]\!]$.

Proof: Let us consider the following relation R: $(s, s') \in R$ iff $s \sim s'$ and s' is the first state from $[s]_\sim$ which was visited during αSEARCH(M, α). It is easy to verify that R is a bisimulation relation between $[\![M]\!]$ and αSEARCH(M, α). □

Example 3.2 *Figure 3.3 gives an example of reduction according to an exact abstraction function* $\alpha([x, y, z]) = [x, y]$ *(this is in fact dead variable reduction that we discuss below).*

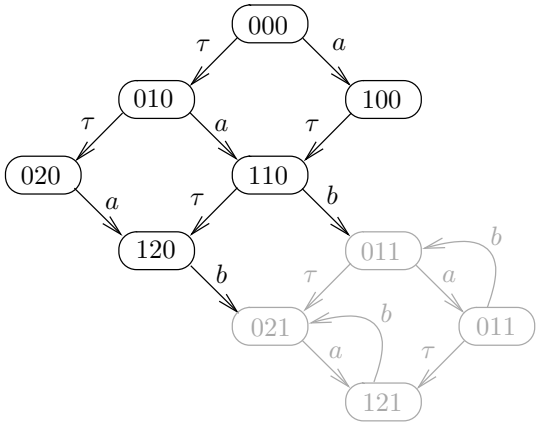


Figure 3.3: Example: exact abstraction function (dead variable reduction).

In this setting, we often use formulation via 'canonization' rather than 'abstraction'. The reason is rather technical: if we work with abstract states, we need to implement representation of two types of entities (concrete states and abstract states). It is more convenient to work just with concrete states and to represent abstract states $\alpha(s)$ via some representant of respective equivalence class $[s]_{\equiv_\alpha}$. A function, which for a given state produces some corresponding representant, is called a canonization function (denoted *canonize*).

3.2.2 Exact Abstraction Functions

We start with the study of abstraction functions which are exact.

Dead Variables

A variable $x \in V$ is *dead* in a state $s \in S$ iff the value of the variable is not used in any computation starting in the state s, i.e., the state s is bisimilar to all states obtained by changing value of x (for all $a \in \mathbb{Z} : s \sim s[x := a]$). Dead variables can be computed (respectively safely approximated) by standard static analysis algorithms. We can get better results by incorporating a dependency analysis: variables which cannot influence neither the control flow nor the value of variables which occur in atomic propositions are dead — this idea is also called faith variable analysis or cone of influence reduction. The definition of dead variables lends itself to the following canonization function: $\mathit{canonize}(s) = s'$ where $s'(x) = 0$ if x is a dead variable and $s'(x) = s(x)$ otherwise.

For more sophisticated specification languages the idea of dead variables can be extended:

– For models with arrays it is useful to consider that each individual position in array can be dead. The static analysis is more complicated — it is useful to perform

"constant propagation" (in order to find constant array indexes) and "live range analysis".
– For models with message queues it is possible to generalize the notion of deadness to the content of a queue [79].
– For timed automata it is customary to talk about "active clock reduction" [68].

Equivalent Values

Values $a, b \in \mathbb{Z}$ are *equivalent for a state* $s \in S$ *and a variable* $x \in V$ iff $s[x := a] \sim s[x := b]$. This is an extension of the idea of dead variables. Note that the region constructions for timed automata (discussed in Chapter 6) is in fact based on equivalent values. In the case of timed automata, a special symbolic representation is used for the whole set of equivalent values.

Static detection of equivalent values is more complicated than dead variables detection. It is possible, for example, in the following cases:

– Variable x is (locally) used only in expression $x = k$ and for several values the same action is performed, e.g., process is waiting for an reset signal, all other signals are discarded.
– Monotonically increasing variable x is used only in guards $x \leq k$, e.g., in (discrete) time models and scheduling problems. In this case all values larger than the maximal constant to which x is compared are equivalent.

We can use the following canonization function: $\mathit{canonize}(s) = s'$ where $s'(x) = \min\{a \mid \text{values } a \text{ and } s(x) \text{ are equivalent values for } s \text{ and } x\}$.

Symmetry

Another way to identify bisimilar states is to exploit symmetries in the model. Symmetries are formalized by the notion of an automorphism. An *automorphism* is a bijection $h : S \to S$ such that $s \xrightarrow{a} s' \Leftrightarrow h(s) \xrightarrow{a} h(s')$. If h is an automorphism then $\forall s \in S : s \sim h(s)$. Automorphisms can be detected by static analysis if the model exhibit some kind of symmetry. A typical example is the use of permutation for specification languages based on networks of machines. If the model contains several identical machines then any permutation of these identical machines is an automorphism. In this case, we can use as a canonization function a permutation which sorts first n locations in a state vector. This basic idea can be extended to a more general class of models with the use of special data type scalarset [116] (only 'symmetrical operations' are allowed over scalarset).

We propose another approach for detecting automorphisms. It is based on linear transformations. For the moment, let us suppose that the data domain can be also $\mathbb{Z}_n$. We say that a set of variables $V' \subseteq V$ is *linearly transformable* iff all uses of these variables are either in guards $x \bowtie y + k$ or in effects $x := y + k$ $(x, y \in V', k \in \mathbb{N})$. A typical example of model with linearly transformable variables is an "alternating

bit protocol" (where the data domain is $\mathbb{Z}_2$). Many other protocols use modular arithmetic as well.

Lemma 3.3 *Let $V' \subseteq V$ be a set of linearly transformable variables. Then the function $h_k(s) = s[V' := V' + k]$ is an automorphism for each $k \in \mathbb{Z}$.*

We can use following canonization function: we select one fixed variable $v \in V'$ and then use $canonize(s) = h_{-s(v)}(s)$, i.e., the canonization function always sets value of v to zero.

3.2.3 Non-Exact Abstraction Functions

Now we consider functions which are not necessary exact. In this case, the αSEARCH leads to an under-approximation of the state space. We briefly discuss three techniques. The first two of them are usually not considered as abstractions, but in our setting it is possible to consider them in this way. For each technique we briefly mention possibilities of refinement.

Bitstate Hashing

This technique is usually treated as a probabilistic storage reduction technique and not as an abstraction function. So let us start with a usual, 'implementation view' description of bitstate hashing [109].

The usual way to implement the data structure *States* is a hash table. For each state we compute a hash value, which gives us address of a row in the hash table. Since the hash function can give the same value for two different states, we need to resolve collisions: this is usually done by keeping a linked list of states for each row in the hash table. Bitstate hashing does not resolve collisions. For each row in the hash table, we store just 1 or 0 depending on whether we have already seen *some* state with this hash value during the exploration. Since we do not resolve collisions, it may happen that we treat new state as a visited state and thus we obtain only under-approximation.

If we consider this technique in our setting, the hash function is just a special case of an abstraction function α where the image of α are natural numbers (rows in the hash table). This abstraction function, of course, in most cases does not satisfy the exactness requirement of inducing bisimulation.

This technique is very good for error detection, but from theoretical point it is not very plausible. There is no way to recognize whether the result is exact or whether it is only under-approximation. Moreover, there is no way to guide the refinement of this approximation: the only way to do the refinement is to increase the image of an abstraction function (i.e., the number of rows in the hash table).

Lossy Compression

Similarly to bitstate hashing, this technique is also usually treated as probabilistic storage reduction technique. We start again by description of the 'implementation

view'.

In model checker implementations, states are represented as vectors. These vectors are rather long (in order of 100 bytes). In many cases, these long vectors contain lot of redundancy and can be substantially compressed. The usual compression is reversible: for a compressed representation we can construct the original state. But we may also consider more aggressive, irreversible, lossy compression. This leads to more significant memory savings, but we lose some information during the compression and it may happen that two different states map to the same compressed representation. Since the compression usually does not satisfy exactness requirement, we obtain under-approximation.

Again, we may view lossy compression as an abstraction function. In this case, this formulation is rather natural — lossy compression actually does abstract some part of the state. Otherwise, however, this technique is similar to bitstate hashing — we are not able to recognize whether the result is exact and there is no reasonable way to guide the refinement.

Predicate Abstraction

Another way to obtain inexact abstraction is to use predicate abstraction: we fix a set of predicates and the abstraction function is given by evaluating these predicates. In this case, it is possible to do the refinement in guided way and to recognize, whether the abstraction is exact. This technique is more involved and we devote to it whole Chapter 5.

3.3 Reductions Preserving Weak Equivalences

Secondly, we consider reductions that try to reduce the number of explored equivalent paths — equivalence of choice here is some weak equivalence. The basic idea is the following: paths that differ only by the order of invisible action (i.e., τ action) are equivalent.

This type of reduction techniques is based on changing order and execution of invisible actions. In order to maintain correctness, we usually further need the notion of independence. Two actions $a, b \in Act$ are *independent* iff they satisfy the following commutativity requirement:

$$s \xrightarrow{a} s_1, s \xrightarrow{b} s_2 \Longrightarrow \exists s' : s_1 \xrightarrow{b} s', s_2 \xrightarrow{a} s'$$

In this section, we give some details about partial order reduction. It is the most often used reduction and we also use it to nicely illustrate the relation between conditions on the reduction and the preserved equivalence. Then we briefly mention several other techniques based on similar ideas. Finally, we discuss how to use these techniques to compute approximations.

3.3.1 Partial Order Reduction

The basic idea of partial order reduction is the following: visit only some paths through the state space, in such a way that from each equivalence class at least one path is visited. This idea is implemented in a simple way: during the state space exploration we traverse only subset $ample(s)$ of all actions enabled in the given state. We use the basic exploration algorithm (Figure 2.1) with one modification, the line

foreach $s \xrightarrow{a_i} s'$ **do**

is changed into:

foreach $s \xrightarrow{a_i} s'$ such that $a_i \in ample(s)$ **do**

The relation between the generated reduced transition system and the full state space one depends on conditions satisfied by the function *ample*. Let us consider the following conditions:

PC (Persistence condition) On all paths in $[\![M]\!]$ (the full state space) starting at s the following condition holds: an action that is dependent on action in $ample(s)$ cannot occur before the first action from $ample(s)$. Moreover $ample(s)$ is empty only if $enabled(s)$ is empty.

CC (Cycle condition) On every cycle in the reduced transition system there is at least one state s such that $ample(s) = enabled(s)$.

IC (Invisibility condition) If $ample(s) \neq enabled(s)$ then all actions in $ample(s)$ are invisible.

SC (Singleton condition) If $ample(s) \neq enabled(s)$ then $|ample(s)| = 1$.

How do we ensure these conditions during the on-the-fly exploration of a state space? Conditions **IC** and **SC** are easy to check locally. Conditions **PC** and **CC**, however, state properties of the full (resp. reduced) state space and cannot be checked locally (by considering the current state and its immediate neighborhood). Therefore, these conditions are usually ensured by some weaker, heuristically checkable conditions (see, e.g., [56]).

The full state space $[\![M]\!]$ and the structure T computed by the modified algorithm using function *ample* are related as follows:

- If *ample* satisfies **PC** then $[\![M]\!]$ and T are deadlock equivalent [89].
- If *ample* satisfies **PC** and **CC** then $[\![M]\!]$ and T are reachability equivalent for local properties [18, 89] (for definition of local properties see [18]).
- If *ample* satisfies **PC**, **CC**, and **IC** then models $[\![M]\!]$ and T are weak trace equivalent [154, 56].
- If *ample* satisfies **PC**, **CC**, **IC**, and **SC** then models $[\![M]\!]$ and T are weak bisimulation equivalent [86].
- It is possible to formulate conditions under which $[\![M]\!]$ and T are weak simulation equivalent [155]. These conditions are, however, bit more technically complicated and we do not present them here.

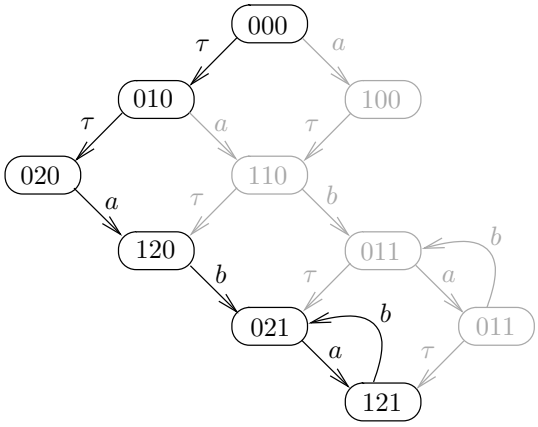


Figure 3.4: Example: partial order reduction.

Example 3.3 *Figure 3.4 illustrates the effect of partial order reduction on the example from Figure 3.1.*

There are many variations of this basic idea. These variations differ mainly in the way they ensure persistence condition. Moreover, the basic idea can be extended by using sleep sets [89], which improve the reduction by taking into account the history of the search.

3.3.2 Other Techniques

There are several other techniques based on similar idea as partial order reduction. We just briefly review them.

Slicing

Slicing identifies parts of the model that are relevant to the verified property [101]. Actions which cannot influence visible actions are removed by a static transformation of the model prior to the state space exploration. The resulting model is weak trace equivalent to the original model.

Transition Merging

If there are two consecutive local invisible actions in the model then we can statically merge them into one atomic action. This basic idea have been formalized in different ways and under different names. Each formalization preserves some weak equivalence: transition merging, transition compression (weak trace equivalence) [125, 74, 110], "next" heuristics (weak simulation) [6], path reduction (weak bisimulation) [173].

A special case of transition merging is loop acceleration. Loop acceleration aims at reducing the 'fragmentation' of state space caused by repeated execution of some

simple cycle. The cycle is statically substituted by meta-transition, which captures repeated executions of the cycle. This techniques is usually used with some kind of symbolic representation [31, 140, 102, 30].

Confluence

Similar technique as partial order reduction is based on the identification of τ-confluent actions [29, 145] — invisible actions satisfying some additional confluence requirements which are similar to the independence requirements of partial order reduction. A on-the-fly reduction algorithm works either in the same way as the modified algorithm for partial order reduction, i.e., by choosing only subset of enabled actions (τ-prioritization [145]) or by using τ-confluent actions to compute canonization function [29]. In both cases the reduced structure is weakly bisimilar to the full state space.

Simultaneous Reachability Analysis

When faced with several possible interleavings of independent and invisible actions, partial order reduction tries to traverse only one of these interleavings. Simultaneous reachability analysis rather tries to perform *all* these actions at once [144, 168], i.e., instead of executing individual actions it executes combined actions. To preserve correctness, it is necessary to ensure that there is no dependence among combined actions and that there is at most one visible action among combined actions.

3.3.3 Approximate Techniques

All of the above discussed techniques have the following form: if a certain condition is satisfied then omit some part of the state space during the exploration. So far we considered only exact techniques of this type — the condition was always safe, i.e., we were able to guarantee that the reduced state space is equivalent (up to some weak equivalence).

What if we consider more aggressive approach? If we consider conditions which are not necessary safe (i.e., we cannot guarantee that the reduced structure is equivalent), we obtain an under-approximation and we can use the approximation refinement scheme discussed in Chapter 1. If we do not find an error in the under-approximation, we refine. The refinement is done by using more precise condition for the reduction. We refine until an error is found or until the condition is safe. Let us discuss simple example of this approach.

Let us suppose that we have a model of system with several parallel processes and that for each process we have a program counter pc_i, which identifies the current location in a modeled program (this is a usual feature of most models).

We divide program locations (program counter values) into two sets: interleaving and non-interleaving. Generation of state space works as follows: if the current state s has at least one process i in a non-interleaving location, then we considered

as successors of s only successors via actions of the process i. If all processes are in interleaving locations then we compute all successors as usual.

This approach leads to weakly equivalent structure only if the division into interleaving and non-interleaving location satisfy certain correctness conditions. These correctness conditions are similar to classical partial order reduction conditions and we do not state them explicitly. They are, however, rather restrictive, so the 'safe' approach would not lead to any significant reduction.

The under-approximation refinement works as usual:

1. Start with all location marked as non-interleaving.
2. Generate the state space.
3. If an error is found or if the correctness condition is satisfied then terminate.
4. Mark some non-interleaving location as interleaving. Go to step 2.

This is just a rough sketch which need to be further elaborated, particularly it is necessary to address the issue of guiding the refinement (step 4). We leave this as a future work (see Chapter 8 for reasons and discussion).

3.4 Reductions Preserving Reachability and Deadlock

At third, we discuss reductions which preserve reachability and deadlock equivalence. These reduction try to foresee the future: they stop the exploration in same state when they can guarantee the results of exploring successors of the given state. We discuss several ways to detect appropriate states.

In general, these techniques employ the following trade-off: prior to the exploration, they precompute some information about the model (this takes some time), during the exploration this information is used to omit searching some parts of the state space (this saves some time).

Doomed States

We say that a state s is *a-doomed* (*deadlock-doomed*) if we can guarantee that an action a (deadlock) is reachable from s.

De Alfaro et al. [70] detect doomed states using the notion of uncontrollability. Their specification language is given as a network of machines. A location of a machine is uncontrollable if no environment can prevent the machine from reaching an action a. With our specification language, we can detect a-doomed states simply by computing several iterations of weakest precondition of the action a.

For an analysis of deadlock-doomed states we can employ an analysis of local cycles. A *covering set* [124, 23] is a set of actions such that each cycle in the structure $[\![M]\!]$ contains at least one of these actions. A covering set can be computed by static analysis of local cycles in the model and it is often very small [23]. Detection of deadlock-doomed states can be based on the following observation: if no action from a covering set is reachable from a state s then this state is deadlock-doomed.

Boring States

We say that a state s is *a-boring* (*deadlock-boring*) if we can guarantee that action a (deadlock) is not reachable from s.

For the detection of boring states we can employ the notion of progress function which was proposed for the sweep line method of state space exploration [54]. Function f is a progress function if $s \rightarrow s' \Rightarrow f(s) \leq f(s')$. If we can show that $f(s_1) \leq k$ for all states s_1 for which there is a-transition from s_1 and $f(s) > k$, then s is a-boring. The progress function usually needs to be provided by the user.

Boring states can also be detected by an analysis of abstract models. Let $\mathcal{A}$ be an over-approximating abstraction function, i.e., $M \preceq \mathcal{A}(M)$. If no state satisfying a is reachable from $\alpha(s)$ in an abstract structure $\mathcal{A}(M)$ then the state s is boring in the structure $[\![M]\!]$.

Dominating Values

If $s \succeq s'$ then it is sufficient to visit successors of the state s in the reachability analysis. Thus we can modify the basic algorithm by changing the line

if $s' \notin States$ **then**

into a line

if not$(\exists s'' : s'' \in States \wedge s' \preceq s'')$ **then**

In this case, the number of visited states during the search depends on the order in which states are visited. We demonstrate in Section 3.6 that in practice there are quite significant differences between breadth-first and depth-first search order. Nevertheless, the correctness is ensured for each order of visits.

In order to apply this technique, we need to be able to safely approximate the simulation relation $s \succeq s'$. This can be done, for example, by an analysis of dominating values. Let $s \in S$ and $x \in V$. We say that value $a \in \mathbb{Z}$ *dominates* $b \in \mathbb{Z}$ *for* s *and* x if $s[x := a] \succeq s[x := b]$. This is an extension of the notion of equivalent values. The fact that one value dominates another can be detected by analyzing monotone variables. If x is a monotone increasing variable which is used only in guards $x \leq k$ then smaller values of x dominate larger values. This situation occurs in models with discrete time (for dense time we use this idea in Chapter 7), in scheduling problems, in models with restricted number of occurrences of certain event (e.g., bounded retransmission protocol), or in cryptographic protocols (the intruder knowledge represented by boolean variables is monotone). Similar reasoning works for other combinations of increasing/decreasing variables and lower/upper bounds.

3.5 Reductions for Acyclic State Spaces

In this section we return to the approach based on concrete search and abstract matching and consider a dynamic approach to reduction from Section 3.2. Now

we restrict our attention to acyclic systems — instead of choosing an abstraction function statically prior to the exploration, we compute it dynamically during the exploration.

For acyclic systems we can compute bisimulation classes inductively during the depth-first search traversal of the state space — if we know bisimulation classes of all successors of a state s, then we can easily compute the bisimulation class of the state s.

In the usual model checking application domain (i.e., reactive systems), acyclicity is a rather restrictive condition. Nevertheless, many practically important systems satisfy this condition, e.g., leader election protocols and scheduling problems. Moreover, bounded model checking leads to exploration of acyclic state spaces. This approach is useful particularly in software verification where the verification of the full program is often infeasible (the approach is also called systematic testing). Utilization of acyclicity was recently advocated by Flanagan and Godefroid [80]; they exploit acyclicity to perform dynamic partial order reduction.

We propose a novel algorithm for on-the-fly reduction of acyclic state spaces. The algorithm is based on dynamically computing sets of bisimilar states. The complexity of our algorithm is better than the complexity of a dynamic reduction algorithm for general state spaces which was proposed by Lee and Yannakakis [131]. We also briefly discuss implementation issues: how to to represent sets during the algorithm and how to approximate operations and perform them more efficiently. This section presents the basic idea of the approach. The approach needs to be further elaborated.

In this section we consider only acyclic systems.

3.5.1 Characterization of Bisimulation Classes

Our algorithm computes bisimulation classes inductively with the use of the following characterization.

Lemma 3.4 *Let s be a state in $[\![M]\!]$ and $X_s = \bigcap_{i=1}^{n} Z_i$, where*

$$Z_i = \begin{cases} [\![g_i]\!] \cap pre(u_i, [s_i]_\sim) & \text{if } s \xrightarrow{a_i} s_i \\ [\![\neg g_i]\!] & \text{otherwise} \end{cases}$$

Then $X_s = [s]_\sim$.

Proof: "$X_s \subseteq [s]_\sim$": Let $s' \in X_s$. We show that $s \sim s'$:

1. If $s \xrightarrow{a_i} s_i$ then $Z_i = [\![g_i]\!] \cap pre(u_i, [s_i]_\sim)$ and therefore $s' \models g_i$ and $s' \in pre(u_i, [s_i]_\sim)$. Hence there exists s'_i such that $s' \xrightarrow{a_i} s'_i$ and $s'_i \sim s_i$.
2. If $s' \xrightarrow{a_i} s'_i$ then $s' \models g_i$. From the construction of X follows that $s' \in [\![g_i]\!] \cap pre(u_i, [s_i]_\sim)$ and $s \xrightarrow{a_i} s_i$ where $s_i \sim s'_i$.

"$[s]_\sim \subseteq X_s$": Suppose $s' \sim s$. It is easy to verify that $\forall 1 \leq i \leq n : s' \in Z_i$ and therefore $s' \in X_s$. □

```
proc ReducedSearch(s)
  explore s
  X := S
  foreach (g_i, u_i) ∈ E do
    if s ⊨ g_i then
                s' := u_i(s)
                Y_i := FindSet(Visited, s')
                if Y_i = ∅ then Y_i = ReducedSearch(s') fi
                X := X ∩ [[g_i]] ∩ pre(u_i, Y_i)
            else
                X := X ∩ [[¬g_i]] fi
  od
  AddSet(Visited, X)
  return X
end
```

Figure 3.5: Algorithm for on-the-fly reduction of acyclic state spaces.

3.5.2 On-the-fly Reduction Algorithm

Now we present an algorithm which performs a dynamical reduction according to bisimulation. The algorithm inductively computes bisimulation classes with the use of Lemma 3.4.

Figure 3.5 gives the algorithm which explores the reduced transition system and computes bisimulation classes. It performs depth-first search traversal of the acyclic transition system and during backtracking it updates the information about visited states. This information is kept in a global data structure *Visited*, which is a set of disjoint sets and supports the following operations:

– FindSet($Visited, s$) returns a set containing s; if such a set does not exists it returns $\emptyset$,
– AddSet($Visited, X$) adds a set X to *Visited*.

Lemma 3.5 *The algorithm* ReducedSearch *satisfies the following:*

1. ReducedSearch$(s) = [s]_\sim$,
2. *at any time during the algorithm* FindSet($Visited, s$) *returns either* $\emptyset$ *or* $[s]_\sim$.

Proof: A *rank* of a state s in acyclic transition systems is defined inductively as follows:

– if s has no successor then $rank(s) = 0$,
– otherwise $rank(s) = \max\{rank(s') \mid \exists a_i : s \xrightarrow{a_i} s'\} + 1$.

We prove the lemma by induction with respect to the rank of s:

– if $rank(s) = 0$ then s has no successors and ReducedSearch$(s) = \bigcap_{i=1}^{n} [\![\neg g_i]\!] = [s]_\sim$,
– if $rank(s) = k$ then the result follows from the induction premise, the invariant about FindSet, and Lemma 3.4.

In both cases it is easy to verify that the invariant (second property) stays valid. □

Lemma 3.6 *The complexity of* ReducedSearch(s_0) *is* $O(n_\sim \cdot k_a + m_\sim \cdot k_f)$*, where* $n_\sim$ *(*$m_\sim$*) is the number of states (transitions) of bisimulation quotient of* $[\![M]\!]$*,* k_a *is the complexity of the* AddSet *operation, and* k_f *is the complexity of* FindSet *operation.*

Proof: For each bisimulation class C reachable in the bisimulation quotient there exists exactly one state $s \in C$ which is explored during ReducedSearch(s_0). This follows from the Lemma 3.5 and from the fact that ReducedSearch(s') is called only if FindSet($\mathit{Visited}, s'$) returns $\emptyset$. The lemma follows. □

The algorithm can be directly used for verification of reachability properties. The algorithm can be also easily modified to generate the bisimulation quotient of $[\![M]\!]$.

The algorithm can be used for verification of programs with input values from an infinite domain (i.e., with an infinite number of initial states), if a finite bisimulation quotient exists. We just need to be able to repeatedly pick initial state which does not belong to any set in *Visited*.

3.5.3 Data Structures

How do we represent the data structure *Visited* and necessary operations over this structure? If we represent sets explicitly (by enumeration) then the complexity of the algorithm is worse than the full traversal. So the whole approach makes sense only if we can represent sets in some symbolic way.

In practice it is always useful to separate a control and a data part of models and to keep the control part concrete. In the following we suppose that the control part of a model is kept concrete and stored in some standard way (e.g., by hashing which gives us pointer to representation of data part). We discuss possible ways for representing the data part of states.

For the representation of sets we need the following operations: representation of sets $[\![g]\!]$, intersection, and weakest precondition. We also need to represent the structure *Visited*: a set of disjoint sets which supports operations FindSet and AddSet. The basic options are the following:

– Symbolic expressions (formulas over predicates, boolean or higher order). This representation can trivially represent sets $[\![g]\!]$ and intersection. Computation of weakest precondition can be performed easily by syntactic manipulation. This representation is, however, not very succinct and the set *Visited* can be represented easily only as a list which requires linear traversal.
– Decision diagrams or minimized deterministic automata. Representation of guards and intersection can be performed using standard operations. The computation of weakest precondition involves intersection with decision diagram representation of transition relation and quantification over variables — in symbolic model checking this is a standard step and is usually called 'preimage computation'. This step is,

however, rather expensive (from practical point of view). The structure *Visited* can be represented by multi-terminal binary decision diagrams and the FindSet operation can be performed efficiently.

For specialized types of models we can use other representation like difference bound matrices (for timed automata), regular expressions (for FIFO automata), covering sharing trees, octagons, or intervals. In general, there is a trade-off between efficiency of set operations (particularly weakest precondition), succinctness of the representation, efficiency of set manipulation operations (particularly FindSet), and expressivity of a specification language.

3.5.4 Approximate Operations

As we have mentioned above, each representation have its disadvantage. However, we do not need to compute sets of bisimilar states exactly but we can only under-approximate them[1]. In such a case, the algorithm does not visit *exactly* one state from each bisimulation class, but it visits *at least* one state from each bisimulation class, i.e., it may visit some states unnecessary, but it remains correct.

We can view the algorithm as making trade-off between effort invested into computation of sets of bisimilar states and effort invested into exploring states. It makes sense to compute sets of bisimilar states only if the time that we save by not exploring these states and their successors is longer then the time needed to compute sets of bisimilar states. So in practice, it may be more useful to compute quickly some coarse under-approximation of the set of bisimilar states.

Dead Variables

One possibility is to distinguish only live/dead variables (static application of dead variables is discussed on page 35). Dead variables can be partially detected by standard static analysis techniques, but the dynamic approach can be more exact.

In this case, sets of states are represented by vectors over $\mathbb{Z} \cup \{*\}$ where $*$ means any value (i.e., given variable is dead). These sets can be represented, for example, by decision diagrams. The computation of weakest precondition can be significantly simplified: we do not need to perform conjunction with a transition relation and quantification, we just insert $*$ values according to which variables are used/set in guards and updates.

Predicate Abstraction

Let $\Phi = \{\phi_1, \ldots, \phi_n\}$ be a fixed set of predicates over program variables. We can represent sets of states by monomials over Φ, these monomials can be easily represented by decision diagrams. Since this representation is not closed on all necessary operations (particularly weakest precondition), we need to approximate them. The

[1] Note that although we use approximate operation during the algorithm, the result of the algorithm is always correct.

computation of weakest precondition can be done with the use of predicate-cartesian abstraction [15] — we study the effect of the update on each predicate in isolation; in this way the weakest precondition operator can be precomputed and the operation can be performed efficiently (for more detailed discussion of this issue see Chapter 4).

It may happen that we obtain an empty set as an under-approximation and thus we even revisit the same state twice. Nevertheless, this approach can still be useful in cases where concrete states are two complex to be stored and the stateless search is a default option (e.g., C++ verification in VeriSoft [90]).

3.6 Evaluation

Now we provide an experimental evaluation of techniques discussed in this chapter. For most of the discussed techniques it is easy to come up with an (artificial) example on which the reduction gives an exponential, or at least very significant, improvement. Unfortunately, many authors evaluate their techniques on such examples. Moreover, reduction techniques are often evaluated on toy models with high values of parameters — this makes state spaces regular and reductions seem significant. However, in real usage of model checkers, models are complex and parameter values are low. It happens very rarely that a model is correct for low values of parameters and erroneous for high values. In our previous experimental work [150], we argue that experiments on such toy models can lead to misleading conclusions. In this section we provide an evaluation of merits of individual reductions on realistic models. It turns out that the reduction is more temperate than often claimed.

Most of the models that we use for the evaluation are well-known model checking case studies: alternating bit protocol, Peterson's mutual exclusion protocol, bounded retransmission protocol, I-protocol, firewire link protocol, leader election protocol, real-time Ethernet protocol, cache coherence protocol, firewire tree identification protocol, file transfer protocol, X.509 authentication protocol, Needham-Schroeder protocol, production cell case study, etc.

The evaluation was done with three explicit model checkers: Spin[2] (version 4.0.6), Murphi[3] (version 3.1), and DiVinE[4] (a prototype version).

For each of the evaluated techniques, we discuss its applicability, we summarize experimental results in other papers, and report about the effect of the reduction on our models. We also discuss the run-time overhead of the reduction and the 'complexity' of its implementation. In tables, we give results only on those models on which the particular technique has some effect. We present only the number of states in full and reduced structures. The reduction factor with respect to the number of transitions is usually very similar.

[2] `http://spinroot.com`
[3] `http://verify.stanford.edu/dill/murphi.html`
[4] `http://anna.fi.muni.cz/divine/`

Model	Full	Reduced	
rether	10,462	1,192	11.3%
synapse	13,973	1,981	14.1%
peterson	12,498	2,376	19.0%
brp	4,792	1,571	32.7%
production_cell	77,416	32,854	42.4%
iprotocol_good	29,994	12,770	42.5%
firewire	55,887	24,323	43.5%
abp	11,286	5,652	50.0%
bridge	3,186	1,676	52.6%
tip	86,556	49,082	56.7%
elevator	1,139	723	63.4%
bakery	109,144	84,517	77.4%
cambridge	8,592	6,962	81.0%
resistance	151,587	129,177	85.2%

Table 3.1: Results for dead variables reduction (DiVinE).

Dead Variables

Dead variable reduction can reduce the size of the state space up to 10% of the size of the full state space (see Table 3.1). Yorav [173] gives similar results on four software models. Bozga et al. [79] report more impressive reduction but only on one parametric model. This reduction technique is applicable to a wide class of models and it brings nearly no run-time overhead. For local variables, which are responsible for most of the reduction, we can perform the canonization by static transformation of the model. Global variables need to be canonized in run-time. Dead variables are easy to detect statically; it becomes more challenging only for arrays and more complex data types.

We have not implemented the equivalent values reduction. The manual inspection of models suggests that this technique improves over dead variable reduction only in few cases and that the improvement is not very significant. Moreover, the static analysis needed for this reduction is more complicated.

Partial Order Reduction

Notwithstanding the large body of theoretical work about partial order reduction techniques, the number of studies concerning practical results of partial order reduction is rather small. Godefroid [89] gives evaluation on four realistic models. The sizes of reduced structures are between 3% and 55% of the size of original structure. Clarke at el. [58] reports similar results on three realistic models. Table 3.2 presents results of our experiments. The size of reduced structure is between 4% and 99%. Partial order reduction is applicable mainly to models with loosely coupled processes. Many of our models use either lot of rendezvous communication or shared variables. The technique is not applicable to these models.

The run-time overhead and the complexity of static analysis depends on the quality of reduction which we want to achieve. In our experiments, we use the tool Spin. Spin uses a rather conservative approach which sacrifices some possible reduction

Model	Full	Reduced	
cambridge	146,471	6,298	4.2%
erathostenes	25,295	2,093	8.2%
snoopy	61,619	9,707	15.7%
smcs	4,634	1,196	25.8%
mobile	30,652	9,971	32.5%
pftp	144,813	47,356	32.7%
i-protocol	2,207,190	919,978	41.7%
relay	876	442	50.4%
peterson	30,432	16,720	54.9%
brp	290,174	169,208	58.3%
X.509	9,028	6,094	67.5%
sgc	299,270	293,126	97.9%
sliding	16,441	14,645	89.0%
giop	638,525	638,520	99.9%

Table 3.2: Results for partial order reduction (Spin).

for low overhead [112].

Confluence and simultaneous reachability analysis, other two techniques based on similar ideas as partial order reduction, have comparable results and domains of applicability [145, 29, 144].

Symmetry Reduction

Symmetry reduction techniques based on permutations can, in theory, achieve reduction up to $n!$ where n is the number of symmetrical entities. Table 3.3 presents practical results. Experiments were done in the tool Murphi on the same set of models as used by Dill and Ip [116]. We have just parametrized protocols by smaller values to obtain more realistic evaluation — the size of reduced structure is between 8% and 50%. Similar results were reported by Bosnacki et al. [33] in Symmetric Spin and by Iosif [115] for object oriented programs.

Symmetry reduction techniques are, of course, applicable only to models with symmetrical entities. Typical applications are cache coherence protocols, protocols over bus with several symmetrical parties, models with several symmetrical agents. The run-time overhead is non-trivial due to the computation of canonization function. Some reduction can be sacrificed for lower overhead. The detection of symmetries can be done fully automatically only in special cases. For practical purposes it is necessary to extend the modeling language with special 'symmetric constructs' (e.g., scalarset [116]).

Linear Transformations

This reduction is usable only for a restricted set of models. The effect of the technique is proportional to the size of a domain of linearly transformable variables. Table 3.3 presents results for three protocols. The run-time overhead is negligible, but the static detection of a set of linearly transformable variables is not easy. It is profitable to introduce to the modeling language a new data type for domain $\mathbb{Z}_n$.

Symmetry reduction (Murphi)

Model	Full	Reduced	
cache	67,418	5,629	8.3%
list4	8,893	1,489	16.7%
peterson3	882	172	19.5%
eadash	1,694	425	25.0%
sci	18,059	4,525	25.0%
ldash	740	372	50.2%

Linear transformations (DiVinE)

Model	Full	Reduced	
cambridge	827	222	26.8%
abp	11,286	5,958	55.1%
brp	14,720	8,121	55.2%

Table 3.3: Results for symmetry reduction and linear transformations.

Transition merging (Spin)

Model	Full	Reduced	
sgc	607,750	299,270	49.2%
erath.	47,669	25,295	53.0%
pftp	207,481	144,813	69.7%
cambridge	166,510	146,471	87.9%
snoopy	67,656	61,619	91.0%
peterson	33,434	30,432	91.0%
smcs	5,066	4,634	91.4%
X.509	9,760	9,028	92.5%
brp	309,676	290,174	93.7%
mobile	32,668	30,652	93.8%

Transition merging (DiVinE)

Model	Full	Reduced	
iprot.	29,994	6,445	21.5%
rether	10,462	6,970	66.6%
resist.	151,587	108,095	71.3%
krebs	7,869	6,027	76.6%
firewire	55,887	45,155	80.8%

Table 3.4: Results for transition merging.

Static Transformations

Our experience suggests that the effect of static transformations (merging of equivalent states, slicing, transition merging, loop acceleration) is not very dependent on the type of an application, but rather on the experience and the modeling style of a user and on possibilities of a modeling language. An experienced user, particularly an user who is acquainted with model checking algorithms, performs many static transformations manually (even unconsciously). Most of our models were crafted by experienced users in rather low-level modeling formalisms and therefore these reductions are not very efficient for them. Table 3.4. presents results for models on which the transition merging technique was applicable. Dong and Ramakrishnan [74] report much better effect of these reduction. We suspect that this is because their modeling language does not contain atomic constructs (as opposed to Spin and DiVinE). Kurshan et al. [125], Yorav [173], and Holzmann [110] report reduction effects slightly better than what we have obtained. Holzmann [110] supports our claims about the dependence on modeling style of the user as he shows that by manual re-modeling he can achieve very significant reduction.

We suppose that static transformations will be very important for models automatically generated from high level description languages and models created by naive users who are not familiar with the underlying model checking algorithms. This type of application is becoming more and more important. In order to con-

Model	Full	Reduced (BFS)		Reduced (DFS)	
jobshop	322,330	13,802	4.2%	116,769	36.2%
naive protocol	3,726	648	17.3%	923	24.7%
bridge	3,186	707	22.2%	1579	49.6%
fischer	1,670	704	42.1%	867	51.9%
abp	65,358	27,792	42.6%	30,647	47.0%
logistics	330,636	149,969	45.3%	306,776	92.7%

Table 3.5: Results for reduction based on dominating values (DiVinE).

vincingly evaluate static transformation techniques, it is necessary to perform experiments on a large set of models created by non-expert users. At this moment, it is still difficult to obtain such a set.

Static transformations do not bring any run-time overhead nor any changes to the model checker itself. This makes them very plausible.

Dominating Values

Table 3.5 presents results of the reduction technique based on dominating values. This technique is applicable only to models which contain some one-way bounded monotone variable — for our models it was either discrete time variable or counter of the number of lost messages. The automatic detection of suitable monotone variable is not easy. It is better to let a user give us some hints, e.g., by introducing special data type.

The technique involves non-trivial run-time overhead because we need to check whether the current state is simulated by some previously visited state. This check can be implemented by the following way. The identification of simulation relation is usually based only on some small part of the state vector (e.g., one monotone variable). For computation of a hash function we use only the part of the state vector which do not influence the identification of simulation relation. In this way, all possible candidate states end up in the same collision list. We search this collision list exhaustively and make a check for simulation.

As we have already noted in Section 3.3, the reduction obtained by this technique depends on the order in which states are visited. Table 3.5 shows that breadth-first order is better than depth-first order.

3.7 Summary

Finally, we summarize the insight obtained by the overview of reduction techniques and their experimental evaluation. We also draw some conclusions for developers of model checking tools.

Trade-offs

Reduction techniques are often presented as an unequivocal improvement of state space exploration. In fact, reduction techniques usually present some kind of trade-off. Reduction techniques require some precomputation (analysis of a model) and usually also reduce the speed of exploration (run-time analysis). Therefore, it is necessary to design reduction techniques carefully so that the computational requirements of the reduction do not overshadow the benefit of the reduction.

Most reduction techniques work only on some models, but their negative effects (precomputation, decrease of speed) manifest themselves on all models. Therefore, one should carefully consider whether the use of a particular reduction technique should be a default option of the tool.

There is also a trade-off between the power of a reduction technique and a preserved behavioral equivalence[5]. Instead of using one fixed reduction technique, we should choose a reduction according to the type of a verified property.

Specialization

Each technique is applicable to some class of models. There is no silver bullet — no reduction technique works really universally. A simple but important observation is that more specialized techniques yield better reduction. This is another trade-off — we trade the power of the technique and the size of the application domain.

There are few techniques which are applicable to wide range of models, e.g., dead variable reduction or transition merging. These reductions bring only moderate effect. More powerful reductions have restricted domain of application, e.g., symmetric models, loosely coupled models, acyclic models.

Today, most model checkers are specialized to some domain. Tool developers should implement only those techniques that are applicable to the application domain of their tool.

User is Important

The reduction obtained depends not only on the application domain, but also on the user's modeling style. Some reductions, particularly those based on static transformation of the model, do not bring nearly no improvement when applied to models crafted by expert users, but can be very efficient on models created by non-expert users. In general, we can expect reduction techniques to have more significant effect on novice user's models.

On the other hand, user's experience can be important for effective application of reduction techniques. Many reduction techniques can benefit from special modeling constructs (e.g., scalarsets for symmetric reduction). A correct and disciplined use of these constructs requires some experience. The user should be also able to choose

[5] This is nicely illustrated, for example, by different conditions on partial order reduction.

suitable techniques to be applied during the exploration — as discussed above, only generally applicable techniques should be used as a default option in the tool.

Unfortunately, a user's role in application and effectiveness of reduction techniques was not systematically investigated by the model checking community so far. We consider this to be an interesting topic for future work.

Good, but not Great

We can summarize our experimental evaluation as follows. On real models, no single reduction is able to reduce the size of the state space significantly under 5%. Claims about drastic reduction, which occur in some papers, are not really appropriate. If a 'drastic' reduction occurs, it is usually restricted to toy examples or to a very special type of models. However, since there are many different reduction techniques and many of them are orthogonal, most models can be reduced quite significantly and certainly there are cases, where the application of reduction techniques can change the output of the model checker from '`out of memory`' to something bit more informative.

However, from the tool developer point of view, there is another trade-off to consider. Many techniques are non-trivial to implement. Developers should consider the following question: Is it more useful to spend precious time on implementation of reduction techniques or rather on some other aspects of tool development, e.g., on improving efficiency of the search implementation, data structures? There is not a simple answer to this question — it depends on the application domain, target group of the tool, tool developers goals and priorities etc. In any case, the question should be considered carefully.

3.8 Related Work

Due to the overview nature of this chapter, most of the related work is discussed during the presentation. Here we give some more details and discuss some other works which do not fit directly into the flow of the presentation.

Partial Order Reduction

Probably the most studied reduction technique is partial order reduction and related techniques. The study of partial order reduction started with the papers by Valmari [167] and Godefroid et al.[111], summarized in Godefroid's PhD thesis [89]. Partial order reduction was promoted by Spin implementation [112]. Spin uses a conservative approach, which sacrifices some reduction for low overhead.

There are many other techniques that are based on similar ideas as partial order reduction. Most of them are mentioned in the chapter: transition compression [125, 74, 110], "next" heuristics [6], path reduction [173], simultaneous reachability analysis [144], τ-confluence [29, 145]. Another similar approach, which is not discussed in this chapter, is the net unfolding algorithm by McMillan [138].

Under-approximation refinement algorithm based on partial order reduction was described by Brim et al. [42]. This algorithm uses modified condition **CC** and refines invisible transitions.

Godefroid et al. [80] proposed dynamic partial order reduction for acyclic state spaces. This approach utilizes acyclicity in similar way as our dynamic bisimulation-based algorithm.

Bisimulation-based Reductions

Another popular reduction technique is symmetry reduction, which was promoted by the tool Murphi [116]. Later it was extended for software [133, 115, 33]. The reduction based on dead variables was described by Bozga et al. [79].

A variation of the αSEARCH algorithm was discussed by Holzman and Joshi [114] in the context of software model checking in Spin. In their approach, the abstraction function must be provided by the user (and it is the user responsibility to guarantee that the abstraction is exact).

Bisimulation Minimization

Another branch of related work is concerned with bisimulation minimization. In our setting, the goal is to compute a structure which is bisimilar to the original one and, hopefully, smaller. Bisimulation minimization algorithms, on the other hand, compute the bisimulation collapse of the structure, i.e., the smallest bisimilar structure.

General algorithm for bisimulation minimization was provided by Paige and Tarjan [146], this algorithm was improved by several authors, see for example work by Dovier at el. [76, 75]. These algorithms are not on-the-fly: one needs to work with the whole state space. Lee and Yannakakis [131] give algorithm which computes the bisimulation quotient on-the-fly and provide complexity analysis of the algorithm. We discuss this work in more detail at the end of Chapter 5.

Chapter 4

Predicate Abstraction and Refinement

"But do they know anything about A? They don't. It's just three sticks to them. But to the Educated–mark this, little Piglet–to the Educated, not meaning Poohs and Piglets, it's a great and glorious A." [47]

This chapter is mainly about a great and glorious $\mathcal{A}$ — abstraction. It introduces may/must abstractions, their variations, and particularly a predicate abstraction. We formulate techniques of several authors in a single notation which enable us to compare individual approaches. We formalize the over- and under-approximation refinement schemes based on predicate abstraction and describe three refinement strategies. We also discuss termination properties of (semi-)algorithms based on abstraction refinement. The main purpose of this chapter is to overview and relate different abstraction and refinement techniques. Another purpose of this chapter is to set the scene for the introduction of a novel under-approximation refinement algorithm in the next chapter.

4.1 Introduction

Let us revisit our Sorter example. In the second chapter we described a model checking activity in which the modeling process was purely manual: we have manually produced model of both the environment and the system (including NQC programs). Whereas the manual modeling of an environment is in most cases unavoidable, modeling of a program code can be automatized. In fact, if we want to apply model checking to systems with non-trivial amount of a program code, automatization of model construction is a necessity.

Figure 4.1 gives the code of the light sensor controller of our Sorter. It also gives a possible abstraction of this code. In this chapter we discuss techniques which perform transformation from a full code to an abstracted code automatically. The example in Figure 4.1 illustrates several issues that need to be addressed in order to make these transformations possible:

– We need to abstract away parts of the program which do not influence the computation of the program (e.g., calls to `PlaySound` function). This is done with the use of static analysis tools and it is not in our focus.

```
task light_sensor_control() {
 int x=0;
 while (true) {
  if (LIGHT > LIGHT_TRESHOLD) {
   PlaySound(SOUND_CLICK);
   Wait(30);
   x = x + 1;
  } else {
   if (x>2) {
    PlaySound(SOUND_UP);
    ClearTimer(0);
    brick = LONG;
   } else if (x>0) {
    PlaySound(SOUND_DOUBLE_BEEP);
    ClearTimer(0);
    brick = SHORT;
   }
   x = 0;
  }
 }
}
```

```
task ABS_light_sensor_control() {
 bool b = false;
 while (true) {
  if (*) {
   b = b ? true : * ;
  } else {
   if (b) {
    brick = LONG;
   } else if (b ? true : *) {
    brick = SHORT;
   }
   b = false;
  }
 }
}
```

Figure 4.1: An example of NQC code and its abstraction with respect to a predicate $b \equiv x > 2$.

– Once we have relevant variables that we want to abstract, we need to find a suitable abstraction (in the example, we abstract an integer variable `x` into a boolean variable `b` which represents `x>0`).
– For a given abstraction we need to compute an abstract transition relation (in the example, we need to find out assignments for `b`).

The abstraction serves two main purposes. At first, we need to make the program amenable to formal treatment (this encompasses for example the abstraction of `PlaySound` function). Secondly, we need to get rid of (data-oriented) sources of infinity[1] (this encompasses for example the abstraction of integer variable `x` into boolean variable `b`).

The purpose of this chapter is to give an overview of main approaches to software model checking based on predicate abstraction and to relate individual approaches. We discuss techniques for computing both over- and under-approximations with the use of predicate abstraction. We focus on the most often used approaches [95, 15, 92, 66]. We also describe relations among resulting approximations. Then we give an overview of refinement techniques and discuss their completeness properties; this part follows-up to the work by Dams [63]. We also give justification for the use of predicates obtained by computation of weakest preconditions and provide several examples which illustrate interesting behaviors of different techniques including example which points out an erroneous claim in [142]. As opposed to other

[1] From practical model checking point of view, it does not make sense to distinguish between infinity and very large numbers. Hence, we treat `int` variables as being infinite domain. These issues need to be considered more carefully if we are concerned with overflow errors.

$$
\begin{array}{llll}
(\ a, & \mathit{true} & \longmapsto & x := x + 1\ ) \\
(\ b, & \neg x > 0 & \longmapsto & x := 0\ ) \\
(\ c, & x > 0 \wedge x \leq 2 & \longmapsto & x := 0\ ) \\
(\ d, & x > 2 & \longmapsto & x := 0\ )
\end{array}
$$

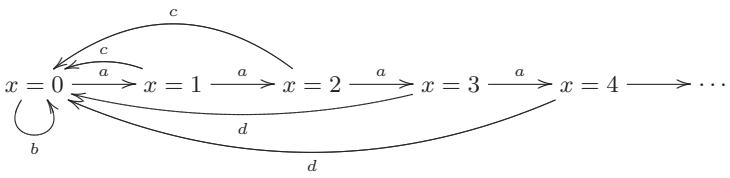


Figure 4.2: Running example: very simplified version of the light controller task and its (infinite) state space.

works on predicate abstraction, we pay attention not only to over-approximation techniques, but also to under-approximation techniques and their refinement.

Running Example

Now we introduce a simple example which is be used through this and the next chapter to illustrate different notions and techniques. The example is given in Figure 4.2. It is a very simplified version of the light controller task: we focus only on the variable `x`. As shown in the Figure, the state space of this example is infinite. Thorough the chapter we use abstraction techniques to construct different finite approximations of the structure. Note that the bisimulation quotient of the state space is finite and that it is possible to obtain a bisimilar structure using discussed abstraction techniques if we use suitable predicates. But we prefer to demonstrate results of abstraction with insufficient set of predicates — it is more illustrative.

In the chapter we consider abstraction with respect to predicates $\phi_1 \equiv x > 0, \phi_2 \equiv x > 2$. Concrete states of this system are given by the value of the variable x, we denote concrete states as $(x = k)$.

Relationship to Main Themes

– Equivalences. We study several different predicate abstractions and we use equivalences to formalize and study relations among these abstractions and the concrete state space.
– Abstractions. Abstractions, particularly predicate abstractions, are the key concept of this chapter. We define several abstraction operators on transition systems and study how to compute these abstractions directly from the model.
– Approximations and refinement. Predicate abstraction is suitable for automatic refinement. We study both over- and under-approximations, refinement strategies, and their termination properties.

4.2 May/Must Abstractions

May/must abstraction are a special instance of the framework of abstract interpretation [62] that maps a (potentially infinite state) transition system into a finite state transition system over a set of abstract states A. In the case that A is a lattice, it is possible to define may/must abstraction with the use of Galois connections [64, 15]. Our definition uses only a concretization function γ — this allows us to define abstractions even in the case that A is not a lattice. This approach is equivalent to definition via description relation [92].

4.2.1 Classical Definition

Let $T = (S, Act, \longrightarrow, s_0)$ be an LTS and γ be a function from a set of abstract states A to subsets of concrete states 2^S. We define a *must* abstraction $\mathcal{A}^{must}(A, \gamma, T) = (A, Act, \longrightarrow_{must}, a_0)$ and a *may* abstraction $\mathcal{A}^{may}(A, \gamma, T) = (A, Act, \longrightarrow_{may}, a_0)$ of a transition system T with respect to γ, where a_0 is such that $s_0 \in \gamma(a_0)$ and transition relations are defined as follows:

- $\mathfrak{a}_1 \xrightarrow{a_i}_{must} \mathfrak{a}_2$ iff $\forall s_1 \in \gamma(\mathfrak{a}_1)\ \exists s_2 \in \gamma(\mathfrak{a}_2) : s_1 \xrightarrow{a_i} s_2$
- $\mathfrak{a}_1 \xrightarrow{a_i}_{may} \mathfrak{a}_2$ iff $\exists s_1 \in \gamma(\mathfrak{a}_1)\ \exists s_2 \in \gamma(\mathfrak{a}_2) : s_1 \xrightarrow{a_i} s_2$

With the use of weakest precondition, the conditions can be reformulated as follows:

- $\mathfrak{a}_1 \xrightarrow{a_i}_{must} \mathfrak{a}_2$ iff $\gamma(\mathfrak{a}_1) \subseteq pre(a_i, \gamma(\mathfrak{a}_2))$
- $\mathfrak{a}_1 \xrightarrow{a_i}_{may} \mathfrak{a}_2$ iff $\gamma(\mathfrak{a}_1) \cap pre(a_i, \gamma(\mathfrak{a}_2)) \neq \emptyset$

Note that the must transition relation is a subset of the may transition relation.

Example 4.1 *Let us consider our running example and abstract states*[2] $\mathfrak{a}_{11}, \mathfrak{a}_{10}$ *such that:*

- $\gamma(\mathfrak{a}_{11}) = \{(x = k) \mid k > 2\}$
- $\gamma(\mathfrak{a}_{10}) = \{(x = 1), (x = 2)\}$

Now if we consider a-transitions we obtain the following:

- $\mathfrak{a}_{11} \xrightarrow{a}_{must} \mathfrak{a}_{11}$ *(and hence also* $\mathfrak{a}_{11} \xrightarrow{a}_{may} \mathfrak{a}_{11}$*),*
- $\mathfrak{a}_{10} \xrightarrow{a}_{may} \mathfrak{a}_{10}$ *and* $\mathfrak{a}_{10} \xrightarrow{a}_{may} \mathfrak{a}_{11}$ *(but there is no must a-transition from* $\mathfrak{a}_{10}$*).*

Lemma 4.1 *For any* T, A, γ *the following holds:*

- $\mathcal{A}^{must}(A, \gamma, T)$ *is simulated by* T*,*
- T *is simulated by* $\mathcal{A}^{may}(A, \gamma, T)$*.*

Proof: Let us consider the following relation R: $(\mathfrak{a}, s) \in R \overset{def}{\Longleftrightarrow} s \in \gamma(\mathfrak{a})$. It is easy to verify that R is a simulation relation in both cases. □

[2] The notation corresponds to the predicate abstraction meaning of these states that we use later in the chapter.

4.2.2 Transitions $must^-$

Must transitions, as defined above, are not very practical — under-approximations based on must transitions are often too coarse to be of any practical use. There have been several proposals how to compute better under-approximations. One of them is based on so called $must^-$ transitions [11, 13] (other proposals are discussed in related work).

The condition on a must transition $\mathfrak{a}_1 \longrightarrow_{must} \mathfrak{a}_2$ requires that each state s_1 abstracted by $\mathfrak{a}_1$ has a *successor* s_2 abstracted by $\mathfrak{a}_2$. Alternatively, we can require that each state s_2 abstracted by $\mathfrak{a}_2$ has a *predecessor* s_1 abstracted by $\mathfrak{a}_1$. In correspondence with [11, 13], let us denote the classical must transition as $must^+$ and the newly defined must transition as $must^-$:

$$\mathfrak{a}_1 \xrightarrow{a_i}_{must^-} \mathfrak{a}_2 \text{ iff } \forall s_2 \in \gamma(\mathfrak{a}_2) \exists s_1 \in \gamma(\mathfrak{a}_1) : s_1 \xrightarrow{a_i} s_2$$

Using the strongest postcondition operator, the condition on $must^-$ transitions can be expressed as:

$$\mathfrak{a}_1 \xrightarrow{a_i}_{must^-} \mathfrak{a}_2 \text{ iff } \gamma(\mathfrak{a}_2) \subseteq post(a_i, \gamma(\mathfrak{a}_1))$$

This new type of transitions is incomparable to the classical one.

Example 4.2 *Let us consider abstract states from Example 4.1 and moreover* $\mathfrak{a}_{00}$ *such that* $\gamma(\mathfrak{a}_{00}) = \{(x = k) \mid k \leq 0\}$. *Then:*

– $\mathfrak{a}_{11} \xrightarrow{a}_{must^+} \mathfrak{a}_{11}$ *but there is not a* $must^-$ *a-transition from* $\mathfrak{a}_{11}$ *to* $\mathfrak{a}_{11}$,
– $\mathfrak{a}_{00} \xrightarrow{a}_{must^-} \mathfrak{a}_{00}$ *but there is not a* $must^-$ *a-transition from* $\mathfrak{a}_{00}$ *to* $\mathfrak{a}_{00}$.

Therefore, by using $must^-$ transitions we can sometimes get better results than with $must^+$. More importantly, it is advantageous to combine $must^+$ and $must^-$ transitions in one abstract structure (this approach can be used to check general properties [13], here we focus only on reachability).

We define $\mathcal{A}^{must}_{+-}(A, \gamma, T) = (A, Act, \longrightarrow_{must^+}, \longrightarrow_{must^-}, \alpha(s_0))$ to be an LTS with two transition relations which are defined as above. For such a transition system we define the set of reachable actions as follows:

$$RA(\mathcal{A}^{must}_{+-}) = \{a_i \in Act \mid s_0 \longrightarrow^*_{must^-} s_i \longrightarrow^*_{must^+} s_n \xrightarrow{a_i} s_{n+1}\}$$

The abstraction based on both $must^+$ and $must^-$ transitions gives a (potentially) larger set of reachable actions than under-approximations based only on $must^+$ or $must^-$. At the same time, $\mathcal{A}^{must}_{+-}$ is still a correct under-approximation:

Lemma 4.2 ([11]) $RA(\mathcal{A}^{must}_{+-}(A, \gamma, T)) \subseteq RA(T)$

4.3 Predicate Abstraction

Based on the above given definition, we can, in principle, construct abstract transition system for an arbitrary (finite) abstract domain and concretization function. The main problem is the computation of may/must transitions, preferably automatically. This is a reason why we use predicate abstraction[3]. For predicate abstraction the conditions on may/must transitions can be expressed as a validity of quantifier free first order formulas. Such validity queries can be solved by decision procedures.

In the following, we suppose that we have at our disposal a function *is_valid*(φ) that returns true only if φ is a tautology, i.e., we require that *is_valid* is a sound, but not necessary complete decision procedure. Currently, there are readily available implementation of the *is_valid* function (see the discussion of related work).

Let us consider a set $\Phi = \{\phi_1, \ldots, \phi_n\}$ of predicates over the set of model variables V. We have several choices of abstract domains and concretization functions based on these predicates. Here we discuss the most common ones — domains $2^{\{0,1\}^n}$, $\{0,1\}^n$, and $\{0,1,*\}^n$. In the following we write $\mathcal{A}^{may}(A,T)$, $\mathcal{A}^{must}(A,T)$ when the concretization function γ is clear from the context.

We also use the following notation. Given a vector $\vec{b} = \langle b_1, \ldots, b_n \rangle$, where $b_i \in \{0,1,*\}$, we define a formula $[\![\vec{b}, \Phi]\!] = b_1 \cdot \phi_1 \wedge \ldots \wedge b_n \cdot \phi_n$ where $0 \cdot \phi_i = \neg\phi_i, 1 \cdot \phi_i = \phi_i, * \cdot \phi_i = \mathit{true}$. If a set of states is characterized by a predicate ϕ, then the weakest precondition with respect to transition a_i can be expressed as $\mathit{pre}(a_i, [\![\phi]\!]) = [\![g_i \Rightarrow \phi[u_i(\vec{x})/\vec{x}]]\!]$.

4.3.1 Abstract Domains

Now we overview different abstract domains and discuss how to (algorithmically) compute may/must transitions over these domains. We consider only classical *must* transitions, $must^-$ transitions can be computed analogically.

Abstract Domain $2^{\{0,1\}^n}$ [15, 95, 92]

A set $2^{\{0,1\}^n}$ with a natural ordering (set inclusion) forms a lattice and the following two functions form a Galois connection with 2^S:

$$\begin{array}{lll} \alpha^1_\Phi : 2^S \to 2^{\{0,1\}^n} & \alpha^1_\Phi(X) & = \{\vec{b} \mid X \cap \{s \mid s \models [\![\vec{b}, \Phi]\!]\} \neq \emptyset\} \\ \gamma^1_\Phi : 2^{\{0,1\}^n} \to 2^S & \gamma^1_\Phi(B) & = \{s \mid \exists \vec{b} \in B : s \models [\![\vec{b}, \Phi]\!]\} \end{array}$$

Example 4.3 *Let us consider our running example with* $\Phi = \{x > 0, x > 2\}$. *Then we get, for example:*
$\alpha^1_\Phi(\{(x = 0), (x = 37), (x = 42)\}) = \{00, 11\}$
$\gamma^1_\Phi(\{00, 11\}) = \{(x = k) \mid k \leq 0 \vee k > 2\}$

Compared to abstractions based on other abstract domains, abstractions defined by the abstract domain $2^{\{0,1\}^n}$ have some nicer theoretical properties (see,

[3] Another reason is that it is easy to do refinement of predicate abstraction.

e.g., [15, 92]). They are, however, not very practical: the number of states is doubly exponential in the number of predicates and transition relations are not easy to compute.

Abstract Domain $\{0,1\}^n$ [66, 92]

An abstract domain $\{0,1\}^n$ (bitvectors of length n) is not a lattice (with any natural ordering). Therefore, in this case we have only a concretization function:

$$\gamma^2_\Phi : \{0,1\}^n \to 2^S, \gamma^2_\Phi(\vec{b}) = \{s \mid s \models [\![\vec{b}, \Phi]\!]\}$$

Given an abstract state $\vec{b}$ we need to compute sets of successors under may/must transitions. Since we are dealing with bitvectors, sets of successors can be represented by boolean functions over $b'_1, \ldots, b'_n$. This is done by the following functions [66, 92]:

$$H_i^{may}(\vec{b}, \eta, j) = \begin{cases} (b'_j \wedge H(\vec{b}, \eta \wedge \phi_j, j+1)) \\ \vee(\neg b'_j \wedge H(\vec{b}, \eta \wedge \neg\phi_j, j+1) \quad \text{if } 0 < j \leq n \\ 0 \quad \text{if } j = n+1 \text{ and } \mathit{is\_valid}([\![\vec{b}, \Phi]\!] \Rightarrow \neg \mathit{pre}(a_i, \eta)) \\ 1 \quad \text{otherwise} \end{cases}$$

$$H_i^{must}(\vec{b}, \eta, j) = \begin{cases} (b'_j \wedge H(\vec{b}, \eta \wedge \phi_j, j+1)) \\ \vee(\neg b'_j \wedge H(\vec{b}, \eta \wedge \neg\phi_j, j+1) \quad \text{if } 0 < j \leq n \\ 1 \quad \text{if } j = n+1 \text{ and } \mathit{is\_valid}([\![\vec{b}, \Phi]\!] \Rightarrow \mathit{pre}(a_i, \eta)) \\ 0 \quad \text{otherwise} \end{cases}$$

Example 4.4 *Let us consider our running example and suppose that is_valid is complete. The value of the function H^{may} for the transition a and the abstract state $\mathfrak{a}_{10}$ represented by a bitvector 10 is computed as follows:*
$H_a^{may}(10, true, 1) =$
$(b'_1 \wedge H(10, x > 0, 2)) \vee (\neg b'_1 \wedge H(10, x \leq 0, 2)) =$
$(b'_1 \wedge (b'_2 \wedge H(10, x > 0 \wedge x > 2, 3)) \vee (\neg b'_2 \wedge H(10, x > 0 \wedge x \leq 2, 3)))\vee$
$(\neg b'_1 \wedge (b'_2 \wedge H(10, x \leq 0 \wedge x > 2, 3)) \vee (\neg b'_2 \wedge H(10, x \leq 0 \wedge x \leq 2, 3)))$

Now we compute the following:

- $H(10, x > 0 \wedge x > 2, 3) = 1$
  because $x > 0 \wedge x \leq 2 \Rightarrow \neg \mathit{pre}(x := x+1, x > 0 \wedge x > 2)$ *is not valid*
- $H(10, x > 0 \wedge x \leq 2, 3) = 1$
  because $x > 0 \wedge x \leq 2 \Rightarrow \neg \mathit{pre}(x := x+1, x > 0 \wedge x \leq 2)$ *is not valid*
- $H(10, x \leq 0 \wedge x > 2, 3) = 0$
  because $x > 0 \wedge x \leq 2 \Rightarrow \neg \mathit{pre}(x := x+1, x \leq 0 \wedge x > 2)$ *is valid*
- $H(10, x \leq 0 \wedge x > 2, 3) = 0$
  because $x > 0 \wedge x \leq 2 \Rightarrow \neg \mathit{pre}(x := x+1, x \leq 0 \wedge x \leq 2)$ *is valid*

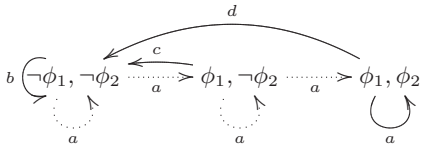


Figure 4.3: Abstraction of the program from Figure 4.2 with respect to $\{0,1\}^n$ domain. Must transitions are full lines, may transitions are dotted lines.

And together we get that: $H_a^{may}(10, true, 1) = (b_1' \wedge (b_2' \vee \neg b_2')) = b_1'$

As we see from the example, by unwinding the recursion we get that:

$$\begin{aligned} H_i^{may}(\vec{b}, true, 1) &= \{\vec{b'} \mid \neg is\_valid([\![\vec{b}, \Phi]\!] \Rightarrow \neg pre(a_i, [\![\vec{b'}, \Phi]\!]))\} \\ H_i^{must}(\vec{b}, true, 1) &= \{\vec{b'} \mid is\_valid([\![\vec{b}, \Phi]\!] \Rightarrow pre(a_i, [\![\vec{b'}, \Phi]\!]))\} \end{aligned}$$

These sets correspond to the definition of may/must transitions, i.e., we obtain a correct over-/under-approximation:

Lemma 4.3 ([92]) *Let* $\longrightarrow_{may}$, $\longrightarrow_{must}$ *be transition relations of* $\mathcal{A}^{may}(\{0,1\}^n, [\![M]\!])$, $\mathcal{A}^{must}(\{0,1\}^n, [\![M]\!])$. *Then:*

- $\mathfrak{a} \xrightarrow{a_i}_{may} \mathfrak{a}'$ *only if* $\mathfrak{a}' \in H_i^{may}(\mathfrak{a}, true, 1)$. *If the decision procedure is complete then* $\mathfrak{a} \xrightarrow{a_i}_{may} \mathfrak{a}'$ *if and only if* $\mathfrak{a}' \in H_i^{may}(\mathfrak{a}, true, 1)$.
- $\mathfrak{a}' \in H_i^{must}(a, true, 1)$ *only if* $\mathfrak{a} \xrightarrow{a_i}_{must} a'$. *If the decision procedure is complete then* $\mathfrak{a} \xrightarrow{a_i}_{must} \mathfrak{a}'$ *if and only if* $\mathfrak{a}' \in H_i^{must}(\mathfrak{a}, true, 1)$.

Example 4.5 *Using the previous lemma and the computation of functions* H^{may}, H^{must} *as illustrated in Example 4.4, we can algorithmically construct the may/must abstract transition systems — the result is given in Figure 4.3.*

Functions H^{may}, H^{must} can be computed with the use of standard BDD operations and calls to a decision procedure. With the use of functions H^{may}, H^{must}, an approximation can be computed by standard symbolic state space search. Let us denote the corresponding algorithms BitvectorMay(M, Φ) and BitvectorMust(M, Φ).

Abstract Domain $\{0,1,*\}^n$ [95, 92, 15]

An abstract domain $\{0,1,*\}^n$ (trivectors of length n) can be viewed as monomials of predicates. Monomials are naturally order by implication, thus this domain forms a lattice (formally, we need a special bottom element $\perp$, because *false* cannot be represented as trivector). The Galois connection with 2^S is defined as follows[4]:

[4]This Galois connection can be obtained by concatenating $(\alpha_\Phi^1, \gamma_\Phi^1)$ and a Cartesian abstraction [15]. Therefore, it is usually called a predicate-cartesian abstraction.

– $\alpha^3_\Phi : 2^S \to \{0,1,*\}^n$, $\alpha^3_\Phi(X) = \min\{\vec{b} \mid X \subseteq [\![\vec{b}, \Phi]\!]\}$
– $\gamma^3_\Phi : \{0,1,*\}^n \to 2^S$, $\gamma^3_\Phi(\vec{b}) = \{s \mid s \models [\![\vec{b}, \Phi]\!]\}$

We can compute the abstract transition relation in a similar way as for the domain $\{0,1\}^n$ (see [92]). Here we give more readable formulation from [95]:

$$G_i^{may}(\vec{b}) = \begin{cases} false & \text{if } is\_valid([\![\vec{b}, \Phi]\!] \Rightarrow \neg g_i) \\ \bigwedge_{j=1}^n \begin{cases} b'_j & \text{if } is\_valid([\![\vec{b}, \Phi]\!] \Rightarrow pre(u_i, \phi_j)) \\ \neg b'_j & \text{if } is\_valid([\![\vec{b}, \Phi]\!] \Rightarrow pre(u_i, \neg\phi_j)) \\ true & \text{otherwise} \end{cases} & \text{otherwise} \end{cases}$$

$$G_i^{must}(\vec{b}) = \begin{cases} \bigwedge_{j=1}^n \begin{cases} b'_j & \text{if } is\_valid([\![\vec{b}, \Phi]\!] \Rightarrow \neg pre(u_i, \neg\phi_j)) \\ \neg b'_j & \text{if } is\_valid([\![\vec{b}, \Phi]\!] \Rightarrow \neg pre(u_i, \phi_j)) \\ true & \text{otherwise} \end{cases} & \text{if } is\_valid([\![\vec{b}, \Phi]\!] \Rightarrow g_i) \\ false & \text{otherwise} \end{cases}$$

Example 4.6 *We again illustrate the computation of the function G^{may} in the same setting as in Example 4.4:*
$G_a^{may}(10) = b'_1 \wedge true = b'_1$
because:

– $x > 0 \wedge x \leq 2 \Rightarrow pre(x := x + 1, x > 0)$ *is valid*
– $x > 0 \wedge x \leq 2 \Rightarrow pre(x := x + 1, x > 2)$ *is not valid*
– $x > 0 \wedge x \leq 2 \Rightarrow pre(x := x + 1, x \leq 2)$ *is not valid*

We again obtain a correct over-/under-approximation:

Lemma 4.4 *Let $\longrightarrow_{may}, \longrightarrow_{must}$ be transition relations of $\mathcal{A}^{may}(\{0,1,*\}^n, [\![M]\!])$, $\mathcal{A}^{must}(\{0,1,*\}^n, [\![M]\!])$. Then:*

– $\mathfrak{a} \xrightarrow{a_i}_{may} \mathfrak{a}'$ *only if* $\mathfrak{a}' \in G_i^{may}(\mathfrak{a})$. *If the decision procedure is complete then* $\mathfrak{a} \xrightarrow{a_i}_{may} \mathfrak{a}'$ *if and only if* $\mathfrak{a}' \in G_i^{may}(\mathfrak{a})$.
– $\mathfrak{a}' \in G_i^{must}(\mathfrak{a})$ *only if* $\mathfrak{a} \xrightarrow{a_i}_{must} \mathfrak{a}'$. *If the decision procedure is complete then* $\mathfrak{a} \xrightarrow{a_i}_{must} \mathfrak{a}'$ *if and only if* $\mathfrak{a}' \in G_i^{must}(\mathfrak{a})$.

Using G^{may}, G^{must} for successor computation we again construct the approximation using the standard state space exploration algorithm. Let us denote these approximations TrivectorMay, TrivectorMust.

Example 4.7 *The predicate-cartesian abstraction of our running example is given in Figure 4.4.*

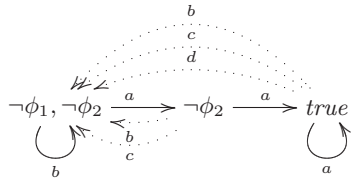


Figure 4.4: Abstraction of program from Figure 4.2 with respect to $\{0, 1, *\}^n$ domain. Must transitions are full lines, may transitions are dotted lines.

In practice it is useful to separate the abstraction process and the search [14]. In the first step, the program is abstracted into a boolean program — a program which uses only boolean variables to hold values of predicates. Updates of boolean variables are computed by boolean under-/over-approximations of weakest preconditions. In the second step, the state space of the boolean program is constructed using standard techniques.

Example 4.8 *For our running example, we can construct the following boolean program (which defines an over-approximation):*

$$
\begin{array}{llll}
(\ a, & \mathit{true} & \longmapsto & b_1 := \mathit{if}\ b_1\ \mathit{then}\ 1\ \mathit{else}\ * \\
 & & & \quad b_2 := \mathit{if}\ b_2\ \mathit{then}\ 1\ \mathit{else\ if}\ b_1\ \mathit{then}\ *\ \mathit{else}\ 0\ ) \\
(\ b, & \neg b_1 & \longmapsto & b_1 := 0, b_2 := 0\ ) \\
(\ c, & b_1 \wedge \neg b_2 & \longmapsto & b_1 := 0, b_2 := 0\ ) \\
(\ d, & b_2 & \longmapsto & b_1 := 0, b_2 := 0\ )
\end{array}
$$

4.3.2 Relations

We have mentioned several different approximations of a concrete system. Let us discuss relations among them; results are summarized in Figure 4.5. We have the following relations among may/must predicate abstractions with different abstract domains:

1. $\mathcal{A}^{may}(2^{\{0,1\}^n}, [\![M]\!])$ is simulated by $\mathcal{A}^{may}(\{0, 1, *\}^n, [\![M]\!])$
2. $\mathcal{A}^{must}(\{0, 1, *\}^n, [\![M]\!])$ is simulated by $\mathcal{A}^{must}(2^{\{0,1\}^n}, [\![M]\!])$
3. $\mathcal{A}^{may}(\{0, 1\}^n, [\![M]\!])$ is simulated by $\mathcal{A}^{may}(2^{\{0,1\}^n}, [\![M]\!])$
4. $\mathcal{A}^{may}(\{0, 1\}^n, [\![M]\!])$ is simulated by $\mathcal{A}^{may}(\{0, 1, *\}^n, [\![M]\!])$

The simulation relation for the case 1 is $(X, \mathit{cartesian}(X))$, where $\mathit{cartesian}(X)$ is cartesian abstraction of the set X [15]. The simulation relation for cases 2, 3, 4 is a natural embedding.

4.4 Refinement Schemes

In this section we deal with the problem of refining approximations. If an approximation is not good enough to either find an error or to guarantee its absence, we

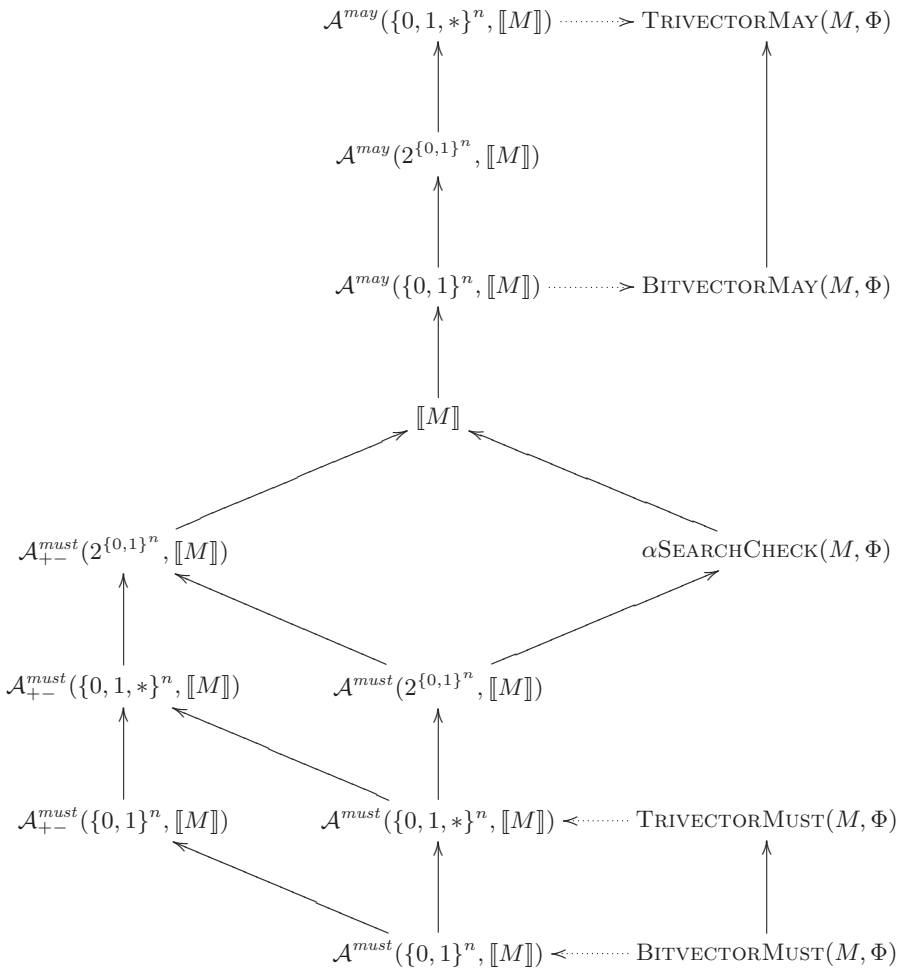


Figure 4.5: Relations among abstractions. Relations are compared with respect to reachability preorder (inclusion of sets of reachable actions). Doted inclusions are due to incompleteness of decision procedure (for complete decision procedure there would be an equivalence). For the sake of clarity, counterparts of BitvectorMust, TrivectorMust for $\mathcal{A}^{must}_{+-}$ are not shown. The algorithm αSearchCheck(M, Φ) is described and discussed in the next chapter.

need to refine it. In the case of approximations based on predicate abstraction, the refinement done by adding more predicates.

At first, we mention why it makes sense to use predicates obtained by iterated computation of weakest precondition. Secondly, we present pseudocodes of both over- and under-approximation refinement schemes and we describe three refinement strategies that use predicates obtained by weakest preconditions. Finally, we discuss some completeness and termination results and compare different refinement algorithms.

4.4.1 Predicates by Weakest Precondition

All below discussed refinement strategies use predicates from guards or predicates constructed by a repeated application of the weakest precondition operator to guards. Here we justify that this is a reasonable approach. We define the following sets:

– $Pre_1(M)$ is the set of all predicates which occur in some guard in M,
– $Pre_{i+1}(M) = Pre_i(M) \cup \{pre(a_i, \phi) \mid \phi \in Pre_i(M)\}$,
– $Pre(M) = \bigcup_{i=1}^{\infty} Pre_i(M)$.

The following lemmas hold for any algorithm (respectively for any algorithm computing under-approximation) discussed in the previous section (i.e., TrivectorMust, TrivectorMay, BitvectorMust, BitvectorMay).

Lemma 4.5 *For each k:* Approximation$(M, Pre_k(M))$ *is k-bisimilar to $[\![M]\!]$.*

Proof: The proof is done by straightforward induction with respect to k. □

Lemma 4.6 *If $[\![M]\!]$ has a finite bisimulation quotient then there exists a finite set of predicates $\Phi \subseteq_{fin} Pre(M)$ such that* Approximation(M, Φ) *is bisimilar to $[\![M]\!]$.*

Proof: If $[\![M]\!]$ has a finite bisimulation quotient then there exists k such that $\sim = \sim_k$. The result follows from Lemma 4.5. □

Lemma 4.7 *There exists a finite set of predicates $\Phi \subseteq_{fin} Pre(M)$ such that $RA($*UnderApproximation$(M, \Phi)) = RA([\![M]\!])$.*

Proof: Let k be such that each $a_i \in RA([\![M]\!])$ is reachable in at most k steps (such k exists because a set of reachable actions is finite). Due to Lemma 4.5, there exists $\Phi_k \subseteq_{fin} Pre(M)$ such that UnderApproximation(M, Φ_k) is k-bisimilar to $[\![M]\!]$. Therefore, $RA([\![M]\!]) = RA($UnderApproximation$(M, \Phi))$. □

Unfortunately, the analogy of Lemma 4.7 does not hold for over-approximations. This is illustrated by example in Figure 4.6. No finite subset of $Pre(M) = \{x = i \mid i \in \mathbb{Z}\} \cup \{y = i \mid i \in \mathbb{Z}\}$ is sufficient to rule out the transition d. To get that d is unreachable, we need the predicate $x = y$ which cannot be obtained by weakest preconditions.

$$
\begin{array}{llll}
(\ a, & pc = 0 & \longmapsto & x := x + 1, y := y + 1\ ) \\
(\ b, & pc = 0 & \longmapsto & x := x - 1, y := y - 1\ ) \\
(\ c, & pc = 0 \wedge x = 0 & \longmapsto & pc := 1\ ) \\
(\ d, & pc = 1 \wedge y \neq 0 & \longmapsto & pc := 2\ )
\end{array}
$$

Figure 4.6: Example showing non-sufficiency of predicates obtained by weakest preconditions for over-approximation techniques (the example is inspired by [149]).

```
proc RefinementOver(M)
  Φ := predicates in guards of M
  while true do
        A := OverApproximation(M, Φ)
        if e ∉ RA(A) then return true fi
        if there is a concrete path to e then return false fi
        Φ := Refine(A, Φ, M)
  od
end

proc RefinementUnder(M)
  Φ := predicates in guards of M
  old := ∅
  while old ≠ Φ do
        old := Φ
        A := UnderApproximation(M, Φ)
        if e ∈ RA(A) then return false fi
        Φ := Refine(A, Φ, M)
  od
  return true
end
```

Figure 4.7: Refinement algorithms for verification of safety properties (we are checking for the reachability of an error action e).

4.4.2 Refinement Strategies

Figure 4.7 presents the core of refinement algorithms for both over- and under-approximations. We suppose that the function $\textsc{Refine}(\mathcal{A}, \Phi, M)$ returns a set of predicates Φ' such that $\Phi \subseteq \Phi'$. For the correctness of RefinementUnder it is important that the function $\textsc{Refine}(\mathcal{A}, \Phi, M)$ returns Φ only if $RA(\mathcal{A}) = RA([\![M]\!])$. Now we discuss three strategies for realization of the function Refine. All of them are based on computation of weakest preconditions.

Non-guided [142, 63]

The simplest approach is to compute new predicates by computing weakest preconditions of all currently used predicates. The refinement algorithm can be specified

as follows:

$$\textsc{RefineN}(\mathcal{A}, \Phi, M) = \{pre(a_i, \phi) \mid \phi \in \Phi, a_i \in Act\}$$

If we start with predicates from guards, this leads to the use of sets $Pre_1, Pre_2, \ldots$. This is a safe approach since we know that we do not miss any important predicate (see Lemmas 4.7, 4.6). But it may lead to the use of predicates which are not relevant for the verification and only unnecessarily increase the size of refined approximations.

Exactness-guided

This is a novel strategy, which is an 'optimization' of the previous one. We try to use only relevant predicates. In order to determine which predicates are relevant, we employ a notion of an inexact transition. We say that abstract transition $\mathfrak{a} \xrightarrow{a_i} \mathfrak{a}'$ is *inexact* if it is a may transition but not a must transition. This can be expressed as:

$$\gamma(a) \cap pre(a_i, \gamma(a')) \neq \emptyset \wedge \gamma(a) \cap \neg pre(a_i, \gamma(a')) \neq \emptyset$$

This situation can be recognized with the use of a decision procedure in a similar way in which we computed may/must transitions. Since inexact transitions are the source of imprecision of the approximation, it makes sense, intuitively, to compute weakest preconditions only with respect to these transitions. The exactness-guided refinement can be specified as follows:

$$\textsc{RefineE}(\mathcal{A}, \Phi, M) = \{pre(a_i, \phi) \mid \phi \in \Phi, \text{ there exists inexact } \mathfrak{a} \xrightarrow{a_i} \mathfrak{a}' \text{ in } \mathcal{A}\}$$

Counterexample-guided [55, 63, 127]

Suppose that we have a path $\pi = a_1, \ldots, a_n$ which demonstrates that $\mathcal{A}$ and $[\![M]\!]$ are not equivalent (a *counterexample*). Then we can guide the refinement specifically to get rid of this discrepancy. We extend the *pre* operator to work over paths:

$$pre_{\pi,k} = \begin{cases} pre(a_k, \phi) & k = n \\ pre(a_k, pre_{\pi,k+1}) & k < n \end{cases}$$

For the refinement we use predicates computed by this operator over the spurious counterexample:

$$\textsc{RefineC}(\mathcal{A}, \Phi) = \{pre_{\pi,k}(\Phi) \mid \begin{array}{l} \pi \text{ is a counterexample to equivalence of } \mathcal{A} \text{ and } [\![M]\!], \\ 1 \leq k \leq length(\pi)\} \end{array}$$

This basic scheme can be optimized in several ways, e.g., we do not include $pre_{\pi,k}$ for all k but only for those for which a_k is not feasible in $[\![M]\!]$. Another optimization is to work with several counterexamples at once [63, 127].

How do we get a counterexample? For over-approximation techniques we get it directly from the spurious counterexample produced by the search. For under-approximations we do not have any guaranteed technique to get the counterexample. It is possible to use random walk over $[\![M]\!]$ to try to find a path which is not included in the current under-approximation.

4.4.3 Completeness

Since the problem of computing $RA([\![M]\!])$ is undecidable, we cannot have truly complete algorithm (such that it terminates on all inputs). Therefore, we study several weaker completeness properties.

Since these properties sometimes depend on an exact formulation of algorithms (e.g., details like search order), we do not provide formal results in this part and only discuss the main observations.

Semi-completeness of Reachability

The refinement algorithm eventually produces structure $\mathcal{A}$ such that $RA(\mathcal{A}) = [\![M]\!]$ (but may not recognize it).

As illustrated by example in Figure 4.6, over-approximation based algorithms which use only predicates computed by weakest precondition cannot have this property.

Refinement algorithms based on non-guided refinement and under-approximation have this property due to Lemma 4.7. Algorithms using exactness-guided refinement have this property as well. Intuitively, everything that is needed is added. Formally, it is necessary to back up this claim for each specific algorithm. We discuss a specific exactness-guided refinement algorithm and its correctness in the next chapter.

Termination for Systems with a Finite Bisimulation Quotient

If $[\![M]\!]$ has a finite bisimulation quotient then the refinement algorithm terminates.

Due to Lemma 4.6, only a finite number of predicates from $Pre(M)$ is sufficient if a finite bisimulation quotient exists. Therefore, over-approximation based algorithms have this completeness property, because they either find an error or eventually construct an abstract structure that does not contain an error state.

In the case of systems without an error, under-approximation based algorithms eventually construct a bisimilar structure. However, it may happen that the algorithm does not recognize this situations and keeps on refining (unnecessarily). For example, non-guided refinement algorithms need not terminate, e.g., on the following example:

$$\begin{array}{lll} (\ a, & pc = 0 \wedge y \geq 0 & \longmapsto y := y + 1\ ) \\ (\ b, & pc = 0 \wedge y < 0 & \longmapsto pc := 1\ ) \end{array}$$

Here the non-guided refinement keeps on introducing predicates $y \geq 1, y \geq 2, \ldots$. This is redundant because the transition $y := y + 1$ is exact with respect to predicate $y \geq 0$. Algorithms with exactness-guided refinement can recognize this and terminate.

Semi-completeness for Systems with Finite Reachable Bisimulation Quotient

If $[\![M]\!]$ has finite reachable bisimulation quotient then the refinement algorithm eventually terminates or produces a structure $\mathcal{A}$ that is bisimilar to $[\![M]\!]$ (but may not recognize it).

In [142] (Theorem 3) it is claimed that their non-guided algorithm based on may transitions does have this property. Example in Figure 4.8 shows that this claim is incorrect. In fact, for this example no may/must abstraction based on predicates from weakest preconditions produces structure bisimilar to the concrete system (the concrete system is rather trivial — it has only one state).

In the next chapter we present an under-approximation refinement algorithm which does have this semi-completeness property.

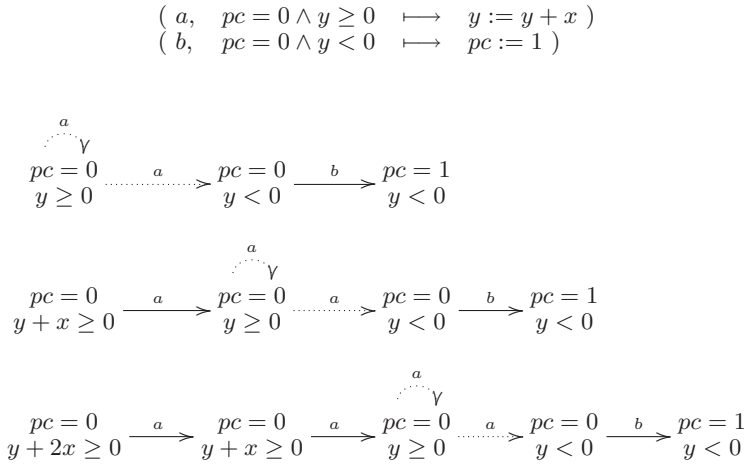


Figure 4.8: First few iterations of predicate-cartesian abstractions are illustrated. Solid lines are must transitions, dotted lines are may transitions; not all predicates are shown in states.

4.5 Related Work

Abstractions in Model Checking

The basic foundations of abstraction for analysis of programs were given by Cousot and Cousot [62] and their theory of abstract interpretations. Formal aspects of application of abstraction to model checking were first treated by Clarke, Grumberg and Long [57]. These ideas were further developed, for example, by Dams et al. [64] and implemented in the Bandera project [61] (a verification environment for Java programs). All these approaches can construct a model automatically, but an user has to choose a suitable abstraction function.

Predicate Abstraction

Predicate abstraction was introduced by Graf and Saidi [95]. Dill et al. [66] introduced more practical implementation of the approach based on symbolic repre-

sentation using BDDs. Godefroid et al. [92] provided formal treatment of different predicate abstractions using modal systems.

The SLAM project [14, 15, 17] was the first full-fledged tool based on predicate abstraction which was applicable to verification of programs in a high-level programming language. This tool is based on an automatic abstraction of C programs into booleans programs and on a refinement using counterexamples.

Extensions of Predicate Abstraction

Must transition usually provide rather poor under-approximation. This problem was addressed in several ways. One approach, suggested by Ball et al. [11, 13], is based on the use of $must^-$ transitions. This approach is discussed in the text.

Another approach is to use as a target of a must transition a set of states instead of a single state — these are so called hyper-transitions [163, 69, 65]. Formally, hyper-transitions are of the type (a, Y), where $\mathfrak{a} \in A, Y \subseteq A$. The condition on must hyper-transition is the following:

$$\mathfrak{a}_1 \xrightarrow{a_i}_{hmust} Y \text{ if } \forall s_1 \in \gamma(\mathfrak{a}_1) \exists \mathfrak{a}_2 \in Y, s_2 \in \gamma(\mathfrak{a}_2) : s_1 \xrightarrow{a_i} s_2$$

It is necessary to change appropriately definitions of simulation, reachable actions, etc. These generalized structures have nicer theoretical properties than classical may/must abstractions, e.g., completeness for branching time model checking [65] and monotonicity of refinement [163]. It is, however, not clear how to effectively compute structures with hyper-transitions and whether they can be used to efficiently compute the set of reachable actions. Therefore, we do not consider them in our comparisons.

Another direction is to improve the performance of predicate abstraction by optimizations. Henzinger et al. [104] suggested the use of lazy abstraction — newly discovered predicates are not used globally, but propagated only lazily, as needed. Another optimizations based on a more selective use of predicates were proposed by Jain et al. [118].

Automatic Refinement

Namjoshi and Kushan [142] described algorithm with a non-guided refinement. The basic idea of the counterexample-guided refinement was proposed by Clarke et al. [55]. Lakhnech [127] described a similar approach named incremental verification.

Although most of the automatic refinement algorithms address undecidable problems (i.e., they are semi-algorithms rather than algorithms), there has been some interest in termination properties and comparisons between approaches. Dams [63] provides summary of different refinement schemes in a single notation. Ball et al. [16] study completeness properties, particularly the relative completeness of algorithms.

Theorem Provers

As we described in this chapter, automatic predicate abstraction makes heavy use of theorem provers for deciding validity of queries over predicates. The first implementations of predicate abstraction tools used general theorem provers like Simplify [71]. Since validity queries generated by predicate abstraction tools are rather specific, several specialized theorem provers have been developed, e.g., Zapato [12] and Cogent [60]. Recently, Lahiri et al. [126] introduced new approach based on symbolic decision procedures which are specifically tailored towards predicate abstraction problems.

Chapter 5

Under-Approximation Refinement

'Contrariwise,' continued Tweedledee, 'if it was so, it might be; and if it were so, it would be; but as it isn't, it ain't. That's logic.' [47]

In this chapter, we propose a novel technique for exploring state spaces — an abstraction-based model checking method which relies on refinement of an under-approximation of the feasible behaviors of the model under analysis. The method is based on the αSEARCH algorithm, which was introduced in Chapter 3. As an abstraction function we use predicate abstraction. Following Tweedledee, we use logic, more specifically automated theorem provers, to automatically detect whether the abstraction is exact. The refinement is guided by the exactness of the abstraction. We study properties of this new algorithm and discuss some applications.

5.1 Introduction

In the previous chapter we discussed techniques based on the refinement of approximations based on predicate abstraction. Practice shows that the most useful of these approaches is the counterexample-guided abstraction refinement (CEGAR) of over-approximations. This approach forms the basis of some of the most popular software model checkers [15, 52, 104]. However, a strength of model checking is its ability to automate the detection of subtle errors and to produce traces that exhibit those errors. Over-approximation based abstraction techniques are not particularly well suited for this, since the detected defects may be spurious due to the over-approximation. We propose an alternative approach based on refinement of under-approximations, which effectively preserves the defect detection ability of model checking in the presence of aggressive abstractions.

The technique uses a novel combination of (explicit state) model checking, predicate abstraction and automated refinement to efficiently analyze increasing portions of the feasible behavior of a model. At each step, either an error is found, we are guaranteed no error exists, or the abstraction is refined. The core of the approach is the αSEARCH algorithm from Chapter 3. The technique traverses the concrete transitions of the model and for each explored concrete state, it stores an abstract version of the state. The abstract state, computed by predicate abstraction, is used to determine whether the model checker's search should continue or backtrack (if the abstract state has been visited before). This effectively explores

an under-approximation of the feasible behavior of the analyzed model. Hence all counter-examples to safety properties are preserved.

Refinement uses weakest precondition calculations to check, with the help of a theorem prover, whether the abstraction introduces any loss of precision with respect to each explored transition. If there is no loss of precision due to abstraction (we say that the abstraction is *exact*) the search stops and we conclude that the property holds. Otherwise, the results from the failed checks are used to refine the abstraction and the whole verification process is repeated anew. In general, the iterative refinement may not terminate. However, if a finite bisimulation quotient [131] exists for the model under analysis, then the proposed approach is guaranteed to eventually explore a finite structure that is bisimilar to the full state space.

Let us illustrate the technique on the example from Figure 4.2 (the running example introduced in the previous Chapter); see Figure 5.1. At first, let us consider the search in which we use the predicates $\phi_1 \equiv x > 0, \phi_2 \equiv x > 2$. The search starts in the initial state ($x = 0$) and we store the corresponding abstract state ($\neg\phi_1, \neg\phi_2$). Now we compute successors of the initial state: an a-successor is a state ($x = 1$) which has a corresponding abstract state ($\phi_1, \neg\phi_2$). This is a new abstract state, therefore we visit the concrete state ($x = 1$). In this case, an a-successor leads to a state ($x = 2$) which has the same corresponding abstract state ($\phi_1, \neg\phi_2$). Therefore, the search does not continue and the state ($x = 2$) is not further explored. We are, however, able to detect that this a-transition is not exact — exactness is equivalent to the validity of a formula $x > 0 \wedge \neg x > 2 \Rightarrow x + 1 > 0 \wedge \neg x + 1 > 2$. With the use of a theorem prover, it can be automatically shown that this is not a valid formula. Therefore, we need to refine, i.e., add new predicates. If we look at the failed validity query, we find a reasonable new predicate $\phi_3 \equiv x > 1$. If we add this predicate and run the algorithm again we obtain a structure with four states which is bisimilar to the full state space. In this case, all transitions are exact and the refinement algorithm terminates.

The technique can also be used in a lightweight manner, without a theorem prover, i.e., the refinement guided by the exactness checks is replaced with refinement based on syntactic substitutions (non-guided refinement) [142] or heuristic refinement. The proposed technique can be used for systematic testing, as it examines increasing portions of the model under analysis. In fact, our method extends existing approaches to testing that use abstraction mappings [96, 172], by adding support for automated abstraction refinement.

To the best of our knowledge, the presented approach is the first predicate abstraction based analysis which focuses on automated refinement of under-approximations with the goal of efficient error detection. We illustrate the application of the approach for checking safety properties in concurrent programs.

Relationship to Main Themes

– Equivalences. Equivalences are, as usual in this thesis, used to formalize correctness and properties of the studied algorithm. The correctness and behaviour

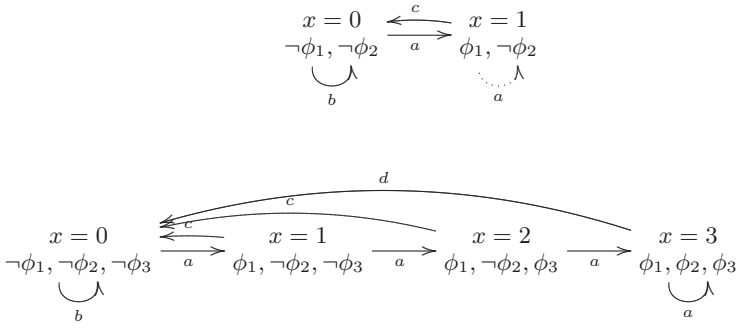


Figure 5.1: Illustration of the algorithm on example from Figure 4.2, $\phi_1 \equiv x > 0, \phi_2 \equiv x > 2, \phi_3 \equiv x > 1$. Exact transitions are full lines, inexact transitions are dotted lines.

of the algorithm is studied with respect to bisimulation. In comparison to other approaches we also make use of reachability equivalence (preorder) and simulation.
– Abstractions. Predicate abstraction function is the key ingredient of the new algorithm. Moreover, a structure computed by the algorithm is compared to other (predicate) abstractions that we studied in the previous chapter.
– Approximations and refinement. This is the key theme in this chapter — we propose a new refinement algorithm which computes series of under-approximation.

5.2 The New Algorithm

In this section we introduce the new refinement algorithm and discuss its properties.

5.2.1 The Algorithm

Our algorithm is a refinement algorithm which repeatedly explores under-approximations of the model. In each iteration we use an algorithm αSearchCheck (see Figure 5.2). This algorithm is very similar to the algorithm αSearch, which we discuss in Chapter 3. As an abstraction function we use a predicate abstraction function given by a set of predicates[1] Φ: $\alpha_\Phi : S \to \mathbb{B}_n$, where $\alpha_\Phi(s)$ is a bitvector $b_1 b_2 \ldots b_n$ such that $b_i = 1 \Leftrightarrow s \models \phi_i$. Let Φ_s be a set of all abstraction predicates that evaluate to *true* for a given state s, i.e. $\Phi_s = \{\phi \in \Phi \mid s \models \phi\}$. If the meaning is clear from the context, we sometimes write, for succinctness, $\alpha_\Phi(s)$ (or just $\alpha(s)$) instead of $[\![\alpha_\Phi(s), \Phi]\!]$.

Moreover, the algorithm performs validity checking, using a theorem prover, to determine whether the abstraction is *exact* with respect to each explored transition

[1] Note that here we use an abstraction function which works on states, whereas abstraction functions that we use in the previous chapter work on sets of states.

```
proc αSearchCheck(M, Φ)
  Φ_new := Φ; add s_0 to Wait; add α_Φ(s_0) to States
  while Wait ≠ ∅ do
        remove s from Wait
        foreach i from 1 to n do
          if s ⊨ g_i then
                        if ¬is_valid(α_Φ(s) ⇒ g_i)
                          then add g_i to Φ_new fi
                        s' := u_i(s)
                        if ¬is_valid(α_Φ(s) ⇒ α_Φ(s')[u_i(x⃗)/x⃗])
                          then add predicates in α_Φ(s')[u_i(x⃗)/x⃗] to Φ_new fi
                        if α_Φ(s') ∉ States then
                                              add s' to Wait
                                              add α_Φ(s') to States
                        fi
                        add (α_Φ(s), a_i, α_Φ(s')) to Transitions
                  else
                        if ¬is_valid(α_Φ(s) ⇒ ¬g_i)
                          then add g_i to Φ_new fi
          fi
        od
  od
  A := (States, Transitions, α_Φ(s_0))
  return (A, Φ_new)
end
```

Figure 5.2: Search procedure with checking for exact abstraction.

— see discussion below. The set Φ_{new} maintains the list of new abstraction predicates. The procedure returns the computed structure and a set of new predicates that are used for the refinement.

Figure 5.3 gives the iterative refinement algorithm for checking whether M can reach an error state described by φ. At each iteration of the loop, the algorithm invokes procedure αSearchCheck to analyze an under-approximation of the model. The under-approximation either violates the property, it is proved to be correct (if the abstraction is found to be exact with respect to all transitions), or it needs to be refined. Counterexamples are generated as usual — with depth-first search order using the stack or with breadth-first search order using parent pointers (these are standard algorithms and therefore we do not discuss this issue here).

Checking for Exact Abstraction and Refinement

Let us recall the notion of exact transition. We say that an abstraction function α is *exact* with respect to transition $s \xrightarrow{a} s'$ iff for all s_1 such that $\alpha(s) = \alpha(s_1)$ there exists s_1' such that $\alpha(s_1') = \alpha(s')$ and $s_1 \xrightarrow{a} s_1'$. In other words, α is *exact* with respect to $s \xrightarrow{a} s'$ iff $\alpha(s) \xrightarrow{a}_{must} \alpha(s')$. This definition is also related to the notion of *completeness* in abstract interpretation (see e.g. [87]), which states that no loss of precision is introduced by the abstraction.

Checking that the abstraction is *exact* with respect to concrete transition $s \xrightarrow{a_i} s'$ is equivalent to checking that $\alpha_\Phi(s) \Rightarrow pre(a_i, \alpha_\Phi(s'))$ is valid. This formula is

```
proc RefinementSearch(M, φ)
  j := 1; Φ_j := AP
  while true do
        (A_j, Φ_{j+1}) := αSearchCheck(M, Φ_i)
        if φ is reachable in A_j then return counterexample fi
        if Φ_{j+1} = Φ_j then return unreachable fi
        j := j + 1
  od
end
```

Figure 5.3: Iterative refinement algorithm.

equivalent to $\alpha_\Phi(s) \Rightarrow \alpha_\Phi(s')[u_i(\vec{x})/\vec{x}]$ when $s \models g_i$. Checking the validity for these formulas is in general undecidable. As is customary, if the theorem prover can not decide the validity of a formula, we assume that it is not valid. This may cause some unnecessary refinement, but it keeps the correctness of the approach. If the abstraction can not be proved to be exact with respect to some transition, then the new predicates from the failed formula are added to the set of new abstraction predicates. Intuitively, these predicates will be useful for proving exactness in the next iteration.

5.2.2 Correctness and Termination

Now we state correctness and termination properties of the refinement algorithm. The presentation is divided into two parts: results and a technical material. The reader may wish to skip the technical material on the first reading.

Results

Due to Lemma 3.1 from Chapter 3 we know that the αSearchCheck algorithm gives an under-approximation. We now show that if the iterative algorithm terminates, then the result is correct and moreover, if the error state is unreachable, the output structure is bisimilar to the model under analysis:

Theorem 5.1 *(Correctness of* RefinementSearch*)*
If RefinementSearch(M, φ) *terminates then:*

– *if it returns a counterexample, then it is a real error,*
– *if it returns 'unreachable', then an error state is indeed unreachable in* $[\![M]\!]$ *and moreover the computed structure is bisimilar to* $[\![M]\!]$.

In general, the proposed algorithm might not terminate (the reachability problem for our modeling language is undecidable). However, the algorithm is guaranteed to eventually find all the reachable actions of the concrete program, although it might not be able to detect that (to decide termination). Moreover, if the (reachable part of the) model under analysis has a finite bisimulation quotient, then the algorithm eventually produces a finite bisimilar structure.

Theorem 5.2 *(Termination properties of* RefinementSearch*)*
Let the αSearchCheck *algorithm use the breadth-first search order and let* A_1, A_2 ... *be a sequence of transition systems generated during iterative refinement performed by* RefinementSearch(M, φ). *Then*

- *there exits* i *such that* $RA(A_i) = RA([\![M]\!])$ *(i.e.,* RefinementSearch *has the 'semi-completeness of reachability' property),*
- *if the reachable part of the bisimulation quotient is finite, then there exists* i *such that* $A_i \sim [\![M]\!]$ *(i.e.,* RefinementSearch *has the 'semi-completeness for models with the finite reachable bisimulation quotient' property).*

The basic idea of the proof is that any two states that are in different bisimulation classes ($s \not\sim s'$) are eventually distinguished by the abstraction function, i.e., there exists j such that $\alpha_{\Phi_j}(s) \neq \alpha_{\Phi_j}(s')$. Moreover, each bisimulation class is eventually visited by RefinementSearch and the finite set of reachable actions emerge.

Technical Material

Here we provide several technical lemmas and the proofs for the two main theorems. We use the following notation: a state s is *visited* during the search if it is inserted into *Wait*; a state s is *considered* during the search if it is generated as a successor of some state in the **foreach** loop; a state s_1 is *matched* to a state s_2 if the check $\alpha_\Phi(s_1) \notin States$ fails because $\alpha_\Phi(s_1) = \alpha_\Phi(s_2)$ and s_2 was visited before. We say that transition $s \xrightarrow{a_i} s'$ is exact if α_Φ is exact with respect to it. Note that sometimes we let αSearchCheck(M, Φ) denote just the structure A computed by the algorithm and not the tuple (A, Φ_{new}).

Lemma 5.1 *If a state* s *is reachable in* $[\![M]\!]$ *via exact transitions with respect to* Φ, *then there exists* s' *such that* s' *is visited during the* αSearchCheck(M, Φ) *and* $\alpha_\Phi(s) = \alpha_\Phi(s')$.

Proof: By induction with respect to the number of exact transitions from the initial state. Basic step ($k = 0$) is trivial. For the induction step, suppose that state s is reachable via sequence of exact transitions: $s_0 \xrightarrow{a_0} \ldots \xrightarrow{a_{k-1}} s_k \xrightarrow{a_k} s_k + 1 = s$. By induction hypothesis there exists s'_k such that s'_k is visited and $\alpha_\Phi(s'_k) = \alpha_\Phi(s_k)$. Because the abstraction is exact with respect to $s_k \xrightarrow{a_k} s$, there must be s' such that $s'_k \xrightarrow{a_k} s'$ and $\alpha_\Phi(s') = \alpha_\Phi(s)$. This successor s' is considered during the visit of s'_k. There are two cases to be considered:

- s' is added to *Wait* and later visited,
- s' is matched to a previously visited s'' such that $\alpha_\Phi(s') = \alpha_\Phi(s'')$.

In both cases we get that some state with the same abstract counterpart as s is visited during the search. □

Lemma 5.2 *Let* $AP \subseteq \Phi$. *If for all reachable states* $s_1, s_2 : \alpha_\Phi(s_1) = \alpha_\Phi(s_2) \Rightarrow s_1 \sim s_2$, *then* αSearchCheck$(M, \Phi) \sim [\![M]\!]$.

Proof: Consider a relation R defined as: $s_1 R s_2$ iff $s_1 = s_2$ or s_1 was matched to s_2. It is easy to verify that R is a bisimulation relation between αSearchCheck(M, Φ) and $[\![M]\!]$. □

Lemma 5.3 *Let* $(A, \Phi_{new}) = \alpha$SearchCheck(M, Φ). *If* $\Phi_{new} = \Phi$, *then* $A \sim [\![M]\!]$.

Proof: Due to Lemma 5.2 it is sufficient to show that α_Φ induces a bisimulation relation on the reachable part of the transition system. We proceed by contradiction. Suppose that α_Φ does not induce bisimulation on the reachable part of the transition system $[\![M]\!]$. This implies that there exists reachable states s_1, s_2 such that $\alpha_\Phi(s_1) = \alpha_\Phi(s_2)$ and there exists $s_1 \xrightarrow{a_i} s_1'$ such that $s_2 \xrightarrow{a_i} s_2'$ and $\alpha(s_1') \neq \alpha(s_2')$, i.e. a set $S' = \{s$ is reachable and s is a source of an inexact transition $\}$ is nonempty.

Let us consider a state $s \in S'$ which has the shortest distance from the initial state from states in S'. This state must be reachable only via exact transitions. According to Lemma 5.1 some state s' such that $\alpha(s) = \alpha(s')$ is visited during the search. During the visit of the state s' we check whether the abstraction is exact for all transition from s'. Since s is a source of inexact transition, s' is also source of inexact transition. Therefore, some of the checked implications is not valid and Φ_{new} is updated and thus $\Phi \neq \Phi_{new}$ (which is a contradiction). □

For the following lemmas we suppose that the αSearchCheck algorithm uses the breadth-first search order.

Lemma 5.4 *Let* $\{A_j\}_{j=0}^{\infty}$ *be a sequence of transition systems generated during an infinite run of* RefinementSearch *and* $Inf_M = \{s \mid$ *there exists infinitely many* j *such that* $s \in A_j\}$. *If* $s \not\sim s'$ *and* $s \in Inf_M$ *then there exists* j *such that* $\alpha_{\Phi_j}(s) \neq \alpha_{\Phi_j}(s')$.

Proof: We prove the lemma by induction with respect to k where k is the smallest number such that $s \not\sim_k s'$. Basic step ($k = 0$): trivial. Induction step ($k + 1$): Let s_1, s_1' be such that $s \xrightarrow{a} s_1, s' \xrightarrow{a} s_1'$ and $s_1 \not\sim_k s_1'$. Since s is visited in infinitely many iterations of αSearchCheck, s_1 is considered in infinitely many iteration of αSearchCheck and therefore one of the following must hold:

1. State $s_1 \in Inf_M$. Then we can apply induction hypothesis, i.e. there exits j such that $\alpha_{\Phi_j}(s_1) \neq \alpha_{\Phi_j}(s_1')$.
2. State s_1 is matched to some state in infinitely many runs of αSearchCheck. Since we use the breadth-first search order, there are only finitely many states to which it can be matched (with breadth-first search order the state can be matched only to states with lower or equal distance from the initial state). Therefore, there exists a state s_2 such that s_1 is matched to s_2 in infinitely many runs of αSearchCheck. From induction hypothesis we get that $s_1 \sim_k s_2$ and hence $s_2 \not\sim_k s_1'$. From induction hypothesis we get that there exists m such that $\alpha_{\Phi_m}(s_2) \neq \alpha_{\Phi_m}(s_1')$. Because s_1 is matched to s_2 infinitely often we eventually get also $\alpha_{\Phi_j}(s_1) \neq \alpha_{\Phi_j}(s_1')$ for some $j \geq m$.

In both cases we get that there exists j such that α_{Φ_j} is not exact with respect to $s \xrightarrow{a_i} s_1$, therefore $pre(a_i, \alpha_{\Phi_j}(s_1))$ will be included in Φ_{j+1} and therefore $\alpha_{\Phi_{j+1}}(s_1) \neq \alpha_{\Phi_{j+1}}(s_1')$. □

Lemma 5.5 *For each reachable bisimulation class B there exists a state $s \in B$ such that s is visited by* RefinementSearch.

Proof: By induction with respect to the length of the shortest path by which state from B is reachable. Base step is obvious. Induction step: let state from B be reachable via path $s_0, \ldots, s_k, s_{k+1}$. By induction hypothesis some state $s' \sim s_k$ is reached during the search. And with the use of Lemma 5.4 we get that some $s'' \sim s_{k+1}$ is reached. □

Lemma 5.6 *Let $\{A_j\}_{j=0}^{\infty}$ be a sequence of transition systems generated during an infinite run of* RefinementSearch(M, φ). *There exits j such that $RA(A_j) = RA([\![M]\!])$.*

Proof: For each $a \in RA([\![M]\!])$ we choose some bisimulation class B such that $s \in B \Rightarrow \exists s' : s \xrightarrow{a} s'$. In this way we obtain a finite set of bisimulation classes $\{B_1, \ldots, B_k\}$ which cover all action is $RA([\![M]\!])$ (note that $RA([\![M]\!])$ is finite because AP is finite). Now we show that there exists an iteration in which at least one state from each of these classes is visited. This is done similarly to the proof of Lemma 5.5. □

Lemma 5.7 *Let $\{A_j\}_{j=0}^{\infty}$ be a sequence of transition systems generated during an infinite run of* RefinementSearch. *If the reachable part of bisimulation quotient is finite, then there exists j such that $A_j \sim [\![M]\!]$.*

Proof: By contradiction. Suppose that $\forall j : A_j \not\sim [\![M]\!]$. From Lemma 5.2 we get that there exists reachable s, s' such that $\forall j : \alpha_{\phi_j}(s) = \alpha_{\phi_j}(s')$ and $s \not\sim s'$. We show (similarly to the proof of Lemma 5.1) that there exists such s which is visited infinitely often. From Lemma 5.4 we get that eventually $\alpha_{\phi_j}(s) \neq \alpha_{\phi_j}(s')$ which is the contradiction. □

Proof: [of Theorem 5.1] The first claim follows from the fact that αSearchCheck produces an under-approximation (Lemma 3.1). The second claim follows from Lemma 5.2. □

Proof: [of Theorem 5.2] This theorem is a direct consequence of Lemmas 5.6, 5.7. □

5.2.3 Properties of the Algorithm

Having discussed the basic completeness and termination properties, we turn to other interesting properties of the algorithm.

$$
\begin{array}{lllll}
\text{A} & (\ a, & x \geq 0 & \longmapsto & x := x - 1\ ) \\
 & & & & \\
 & (\ a, & pc = 0 & \longmapsto & pc := 1, x := 1\ ) \\
 & (\ b, & pc = 0 & \longmapsto & pc := 1, x := 2\ ) \\
\text{B} & (\ c, & pc = 1 & \longmapsto & pc := 2, x := x + 1\ ) \\
 & (\ d, & pc = 2 \wedge x \geq 3 & \longmapsto & pc := 3\ ) \\
 & (\ e, & pc = 2 \wedge x < 3 & \longmapsto & pc := 3\ )
\end{array}
$$

Figure 5.4: Examples showing incomparability of under-approximations based on αSearchCheck and $\mathcal{A}^{must}_{+-}$.

Relation to Other Abstractions

The αSearchCheck produces structure which is incomparable to $[\![M]\!]$ and other approximations with respect to simulation. But it is comparable with respect to reachability:

Lemma 5.8 $RA(\mathcal{A}^{must}(2^{\{0,1\}^n}, [\![M]\!])) \subseteq RA(\alpha\text{SearchCheck}(M, \Phi)) \subseteq RA([\![M]\!])$

Proof: This is a direct consequence of Lemmas 3.1 and 5.1. □

Under-approximations based on $\mathcal{A}^{must}_{+-}$ and on αSearchCheck are incomparable. A trivial example in Figure 5.4 A illustrates that αSearchCheck can be more precise. If we consider abstraction with respect to a single predicate $x \geq 0$ we see that there is neither $must^+$ nor $must^-$ transition and therefore the RA set is empty for $\mathcal{A}^{must}_{+-}$. The αSearchCheck finds a_1 to be reachable.

On the other hand, consider an example in Figure 5.4 B and abstraction with respect to a single predicate $x \geq 3$. Due to the state matching on $(pc = 1, x = 1)$ and $(pc = 1, x = 2)$ the αSearchCheck computes structure with the set of reachable actions either $\{a, b, c, d\}$ or $\{a, b, d, e\}$ depending on the exact search order — whether a or b is traversed first from the initial state. But $RA(\mathcal{A}^{must}_{+-}(\{x \geq 3\}, [\![M]\!]))$ contains all five transitions.

Figure 4.5 shows the relation of structure produced by αSearchCheck with respect to other predicate abstractions.

Search Order and Non-Monotonicity

The search order used in αSearchCheck (depth-first or breadth-first) influences the size of the generated structure, the newly computed predicates, and even the number of iterations of the main algorithm. If there are two states s_1 and s_2 such that $\alpha_\Phi(s_1) = \alpha_\Phi(s_2)$ but $s_1 \not\sim s_2$ then, depending on whether s_1 or s_2 is visited first, different parts of the transition system will be explored.

Also note that the refinement algorithm is non-monotone, i.e., a state which is reachable in one iteration may not be reachable in the next iteration. A similar problem occurs in the context of *must* abstractions: the set of *must* transitions is not generally monotonically non-decreasing when predicates are added to refine an

$$
\begin{array}{llll}
(\ a, & pc = 0 & \longmapsto & pc := 1, x := 1\ ) \\
(\ b, & pc = 0 & \longmapsto & pc := 2\ ) \\
(\ c, & pc = 1 & \longmapsto & pc := 2\ ) \\
(\ d, & pc = 2 & \longmapsto & pc := 3\ ) \\
(\ e, & pc = 2 & \longmapsto & pc := 1, x := x + 2\ ) \\
(\ f, & pc = 3 & \longmapsto & pc := 4, x := x + 4\ ) \\
(\ g, & pc = 4 & \longmapsto & pc := 5\ )
\end{array}
$$

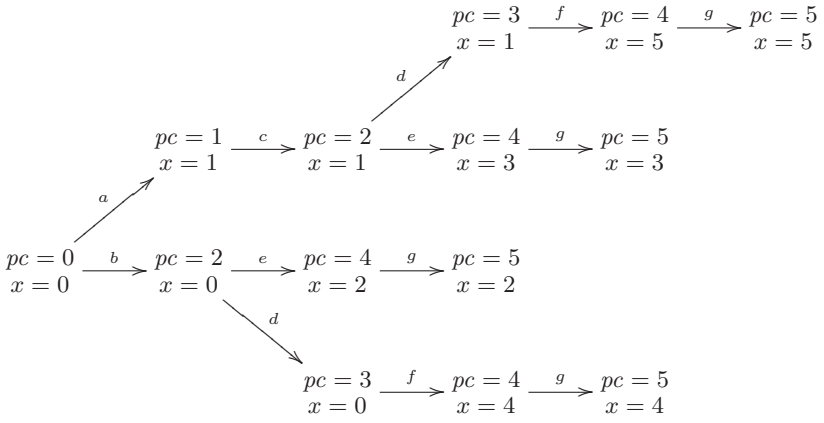


Figure 5.5: Example of a model on which the refinement can be non-monotone.

abstract system [92, 163]. However, we should note that the algorithm is guaranteed to converge to the correct answer.

The example in Figure 5.5 illustrates that the behaviour of the search is not necessary monotone in subsequent iterations. Let's consider the first iteration with predicate $x \geq 3$ and with breadth-first search order: then we visit states $(pc = 0, x = 0), (pc = 1, x = 1), (pc = 2, x = 0),\ (pc = 4, x = 2), (pc = 3, x = 0), (pc = 5, x = 2), (pc = 4, x = 4), (pc = 5, x = 4)$. If we add a predicate $x = 1$ then we visit states $(pc = 0, x = 0), (pc = 1, x = 1), (pc = 2, x = 0),\ (pc = 2, x = 1), (pc = 4, x = 2), (pc = 3, x = 0), (pc = 3, x = 1),\ (pc = 4, x = 3),\ (pc = 5, x = 2), (pc = 4, x = 5), (pc = 5, x = 3), (pc = 5, x = 5)$. The state $(pc = 5, x = 2)$ is visited during the first iteration and is not visited during the second one.

Non-termination for Finite State System

We should also note that the proposed iterative algorithm is not guaranteed to terminate even for a finite state program. This situation is illustrated by the example in Figure 4.8. Although the program is finite state (and therefore the problem can be easily solved with classical explicit model checking), it is quite difficult to solve using abstraction refinement techniques. The iterative algorithm does not terminate on this example: it keeps adding predicates $y \geq 0, y + x \geq 0, y + 2x \geq 0, \ldots$. Note that, in accordance with Theorem 5.2, it eventually produces a bisimilar structure.

However, the algorithm is not able to detect termination, and it keeps refining indefinitely. The reason is that the algorithm keeps adding predicates that refine the unreachable part of the model under analysis. Also note that the same problem occurs with over-approximation based abstraction techniques that use refinement based on weakest precondition calculations [52, 142]. Those techniques introduce the same predicates.

5.3 Extensions

In this section we propose several extensions of the algorithm.

5.3.1 Heuristics for Termination

We briefly discuss two heuristics for termination of the refinement algorithm.

Termination for Finite State Systems

To solve the problem of non-termination for finite state systems, we propose to use the following heuristic. If there is a transition for which we cannot prove that the abstraction is exact in several subsequent iterations of the algorithm, then we add predicates describing the concrete state; i.e., in the example from Figure 4.8 we would add predicates $x = 0;\ y = 0$. The abstraction eventually becomes exact with respect to each transition. And since the number of reachable transitions is finite, the algorithm must terminate.

Corollary 5.1 *If $[\![M]\!]$ is finite state then the modified algorithm terminates.*

Locally Stable Output

We discuss termination heuristic based on detecting 'local stability' of the refinement algorithm, i.e., that $A_j \sim A_{j+1}$. This heuristic is not correct. However, it is quite a natural heuristics and our experience suggest that many people tend to suggest it when they see the algorithm for the first time. Therefore, we find it useful to discuss and clarify it.

If the finite bisimulation quotient exists, then, due to Lemma 5.7, there exists n such that for each j: $A_n \sim A_{n+j} \sim [\![M]\!]$. As we see from our examples, it may, however, happen that we do not recognize this situation and keep on refining forever. Now the question is: if we detect a 'locally stable output' of the αSearchCheck algorithm, i.e., if $A_j \sim A_{j+1}$ or even $A_j = A_{j+1}$ (or even $A_j = A_{j+k}$ for several k), can we conclude anything about A_j (with respect to $[\![M]\!]$)?

The answer is, unfortunately, "no". This follows directly from undecidability of the reachability problem, but we prefer to illustrate it on a specific simple example (k is a parameter of the example):

$$\begin{array}{llll} (\ a, & pc = 0 \wedge x < k & \longmapsto & x := x + 1\ ) \\ (\ b, & pc = 0 \wedge x = k & \longmapsto & pc := 1\ ) \end{array}$$

For first $k-1$ iterations of the refinement algorithm, the output A_j is a single state with a a-transition self-loop. The algorithm needs k iterations to find the bisimilar system and terminate. Hence we see that no bound on 'local stability' is sufficient to guarantee the correctness of the output.

However, we can get use this approach to report 'conditional correctness'. In the case of locally stable output, we can report to the user a list of transition which the algorithm cannot prove to be exact, i.e., the correctness is conditioned by exactness of these transitions (in the reachable part of the state space).

5.3.2 Open Systems

Until now, we have discussed our approach in the context of "closed" systems. However, the approach can be extended to handling "open" systems (i.e. programs with inputs). In order to model open systems, we extend the guarded commands language by allowing assignments of the form $x := \mathit{input}$, which assigns to program variable x an arbitrary value from the input domain (in our case the set of integers). We can also allow the initial values of the program variables to be unspecified, in which case the transition system representing the open program has several (possibly unbounded) initial states.

In order to apply our approach, we need to compute, for each input variable, explicit concrete values that drive the concrete execution of the program. What we really want here is to pick one input value for each satisfiable valuation of the abstraction predicates. We can directly use the original algorithm — it will simply try all the possible values and continue the program execution only from values that satisfy the predicate combinations (most of the states that contain such input values will be matched if they lead to the same valuation of abstraction predicates). This "brute force" approach requires enumerating eventually the whole input domain, which is impossible for infinite input domains. Note however that the approach might still be very useful at detecting errors.

Alternatively, we can use a constraint solver for computing the input values that are solutions of the satisfiable combinations of abstraction predicates (provided that satisfiability is decidable for the abstraction predicates). The decision whether to use the "brute force" approach or the satisfiability approach depends on the number of abstraction predicates and the size of the input domain. With the brute force approach, the the whole input domain needs to be enumerated eventually. With the satisfiability approach, there are at most 2^k satisfiability queries (where k is number of predicates which depend on the input variable).

Example 5.1 *Let us consider the following simple program with input variables x, y.*

$$
\begin{array}{llll}
(\ a, & pc = 0 \wedge x < y & \longmapsto & pc := 1, z := x + y\ ) \\
(\ b, & pc = 0 \wedge x \geq y & \longmapsto & pc := 1, z := x \cdot y\ ) \\
(\ c, & pc = 1 & \longmapsto & pc := 2, max := x\ ) \\
(\ d, & pc = 2 \wedge max < y & \longmapsto & pc := 3, max := y\ ) \\
(\ e, & pc = 3 \wedge max < z & \longmapsto & pc := 4, max := z\ )
\end{array}
$$

At the beginning the only predicate on input values is $x < y$, so in first iteration we use two tuples of input values: one which satisfies the predicate (e.g., $x = 0, y = 1$) and one which does not satisfy the predicate (e.g., $x = 0, y = 0$). Using the refinement algorithm we keep on adding new predicates (in this case $x < x \cdot y, x < x+y, y < x+y, y < x \cdot y$) and new input tuples such that each satisfiable combination of predicates is covered.

5.3.3 Transition Dependent Predicates

The predicates that are generated after the validity check for one transition are used 'globally' at the next iteration. This may cause unnecessary refinement — the new predicates may distinguish states which do not need to be distinguished. To avoid this, we could use 'transition dependent' predicates. The idea is to associate the abstraction predicates with the program counter corresponding to the transition that generated them. New predicates are then added only to the set of the respective program counter. However, with this approach, it may take longer before predicates are 'propagated' to all the locations where they are needed, i.e., more iterations are needed before an error is detected or an exact abstraction is found. We need to further investigate these issues. Similar ideas are presented in [53, 103], in the context of over-approximation based predicate abstraction.

5.3.4 Light-weight Approach

As mentioned, the under-approximation and refinement approach can be used in a lightweight but systematic manner, without using a theorem prover for validity checking. This corresponds to the non-guided refinement that we discuss in the previous chapter.

We are also considering several heuristics for generating new abstraction predicates. For example, it is customary to add the predicates that appear in the guards and in the property to be checked. One could also add predicates generated dynamically, using tools like Daikon [78], or predicates from known invariants of the system (generated using static analysis techniques).

In order to extend the applicability of the proposed technique to the analysis of full-fledged programming languages, we are investigating abstractions that record information about the shape of the program heap, to be used in conjunction with the abstraction predicates. We have reported about these experiments in [147, 148].

5.4 Implementation and Applications

This section is concerned with our practical experiences with the proposed algorithm. We describe an implementation, examples to which the implementation was applied, and results of experiments.

5.4.1 Implementation

We have implemented our approach for the guarded command language. Our implementation is done in the Ocaml[2] language and it uses the Simplify theorem prover [71]. The implementation has just 590 lines of code (parsing + definition of semantics: 390 lines, αSearchCheck algorithm: 170 lines, RefinementSearch algorithm: 30 lines).

It turns out that the main practical bottleneck of the algorithm are calls to the theorem prover. Therefore, we have implemented several optimizations for reducing the number of theorem prover calls:

- When updating Φ_{new} for refinement, we add only those conjuncts of $\alpha_\Phi(s')[u_i(\vec{x})/\vec{x}]$ for which we cannot prove validity.
- We cache queries to ensure that the theorem prover is not called twice for the same query.
- All queries have the form of implication. Before calling the theorem prover for the implication, we check whether the right hand side is a tautology (in such case the implication is clearly satisfied). Results of these checks are also cached.

These optimizations reduce significantly the number of theorem prover calls and they are sufficient for the sake of application of the algorithm on small guarded command language examples. For the application to real programming languages it would be necessary to further improve these optimizations (note that significant amount of optimizations is also used in tools based on over-approximation refinement).

5.4.2 Examples

We discuss the application of our implementation for several concurrent programs. These preliminary experiments show merits of our approach. Of course, much more experimentation is necessary to really assess the practical benefits of the proposed technique and a lot more engineering is required to apply it to real programming languages.

Note that in the described experiments, we always start the first iteration of the refinement algorithm with predicates which occur in guards (particularly, we never abstract the program counter). All the reported results are for the breadth-first search order. We report the following results for each iteration of the refinement algorithm: the number of generated concrete states, the number of stored abstract

[2] `http://caml.inria.fr/`

$$
\begin{array}{lcl}
pc = 0 & \longmapsto & pc := 1, packetsOld := packets \\
pc = 1 \wedge locked = 0 & \longmapsto & pc := 2, locked := 1 \\
pc = 1 \wedge locked = 1 & \longmapsto & pc := 10 \\
pc = 2 & \longmapsto & pc := 3 \\
pc = 2 & \longmapsto & pc := 4 \\
pc = 3 \wedge locked = 1 & \longmapsto & pc := 4, locked := 0, packets := packets + 1 \\
pc = 3 \wedge locked = 0 & \longmapsto & pc := 10 \\
pc = 4 \wedge packets \neq packetsOld & \longmapsto & pc := 0 \\
pc = 4 \wedge packets = packetsOld & \longmapsto & pc := 5 \\
pc = 5 \wedge locked = 1 & \longmapsto & pc := 6, locked := 0 \\
pc = 5 \wedge locked = 0 & \longmapsto & pc := 10
\end{array}
$$

Figure 5.6: The device driver example.

states, the number of queries to the theorem prover, the number of hits to a queries cache, and newly generated predicates.

The Device Driver Example

The first example that we discuss is the "classic" predicate abstraction example of a device driver program (this example was used for the first time in [17], where the code of the program is given). Figure 5.6 gives the skeleton of the example in the guarded command language. The property of interest is the correct use of a lock — in the model it is expressed as non-reachability of a predicate $pc = 10$. In this case the first iteration (using just predicates from guards) is sufficient to prove the property: the search finds 10 concrete states, 9 abstract states and casts 3 queries to the theorem prover.

The Ticket Protocol

The next example is a ticket protocol for mutual exclusion [8] (we use the formalization of the algorithm given in [44]). The algorithm is based on simple "ticket" procedure: a process which wants to enter a critical section draws a number which is one larger than the number held by any other process. The process then waits until all processes with smaller numbers are served: this is checked by a "display" variable which shows the value of the currently smallest ticket number. The model of the protocol is given in Figure 5.7. The property of interest is a mutual exclusion in the critical section ($\neg(pc_1 = 2 \wedge pc_2 = 2 \vee pc_2 = 2 \wedge pc_3 = 2 \vee pc_1 = 2 \wedge pc_3 = 2)$). The state space is infinite state (the ticket numbers increase without any bound), but it has a finite bisimulation quotient.

The property can be proved by the tool. Intermediate results for the protocol with 3 processes are given in Table 5.1. Final results (results of the last iteration) with respect to number of processes are given in Table 5.2.

$$
\begin{array}{lcl}
pc_1 = 0 & \longmapsto & pc_1 := 1, a_1 := t, t := t+1 \\
pc_1 = 1 \wedge a_1 \leq s & \longmapsto & pc_1 := 2 \\
pc_1 = 2 & \longmapsto & pc_1 := 0, s := s+1 \\
\\
pc_2 = 0 & \longmapsto & pc_2 := 1, a_2 := t, t := t+1 \\
pc_2 = 1 \wedge a_2 \leq s & \longmapsto & pc_2 := 2 \\
pc_2 = 2 & \longmapsto & pc_2 := 0, s := s+1 \\
\\
pc_3 = 0 & \longmapsto & pc_3 := 1, a_3 := t, t := t+1 \\
pc_3 = 1 \wedge a_3 \leq s & \longmapsto & pc_3 := 2 \\
pc_3 = 2 & \longmapsto & pc_3 := 0, s := s+1
\end{array}
$$

Figure 5.7: The ticket protocol (an instance for three processes).

Iteration	Concrete states	Abstract states	TP queries	Cache hits	New predicates
1	52	25	14	18	$a_1 \leq s+1, a_2 \leq s+1, a_3 \leq s+1, t \leq s$
2	58	31	70	152	$a_1 \leq s+2, a_2 \leq s+2, a_3 \leq s+2, t \leq s+1, t+1 \leq s$
3	58	31	151	475	$t \leq s+2$
4	58	31	173	585	$t \leq s+3$
5	58	31	195	657	-

Table 5.1: The ticket protocol for three processes: intermediate results.

n	Num. of iterations	Concr. states	Abs. states	TP queries	Cache hits	Predicates
2	4	15	9	49	87	6
3	5	49	26	167	481	14
4	6	253	129	1057	4239	22
5	7	1296	651	7269	29011	32

Table 5.2: The ticket protocol: final results with respect to number of processes.

The Bakery Protocol

As a next protocol we consider the well-known bakery protocol for mutual exclusion. The protocol is based on similar idea as the ticket protocol (the ticket protocol requires special hardware instruction like Fetch-and-Add, whereas bakery protocol is applicable without any special instructions). Figure 5.8 gives the model of the protocol. The property of interest is again mutual exclusion. The state space is again infinite with a finite bisimulation quotient.

The property can be proved by the algorithm, intermediate results are given in Table 5.3.

RAX

The RAX example (illustrated in Figure 5.9) is derived from the software used within the NASA Deep Space 1 Remote Agent experiment, which deadlocked during flight [170]. We encoded the deadlock check as "$pc_1 = 4 \wedge pc_2 = 5 \wedge w_1 = 1 \wedge w_2 = 1$ is unreachable". The error is found after one iteration; the reported counterexample has 8 steps.

Note that the state space of the program is unbounded, as the program keeps incrementing the counters e_1 and e_2, when $pc_2 = 2$ and $pc_1 = 6$, respectively. We also ran our algorithm to see if it converges to a finite bisimulation quotient. Interestingly, the algorithm does not terminate for the RAX example, although it has a finite reachable bisimulation quotient. The results are shown in Table 5.4. However, if we declare the counters in the program as non-negative, i.e., we introduce two new predicates, $e1 \geq 0$, $e2 \geq 0$, then the algorithm terminates after three iterations.

The application of over-approximation based predicate abstraction to a Java version of RAX is described in detail in [170]. In that work, four different predicates were used to produce an abstract model that is bisimilar to the original program. In contrast, the work presented here allowed more aggressive abstraction to recover feasible counterexamples.

The Sorter Example

Finally, we consider our Sorter example (we use the source code given in the Appendix). The state space in this case is also infinite (the variable *request* can grow indefinitely) with a finite bisimulation quotient. All variables except $x, t1, t2, requests, timer$ are kept concrete. For listed variables we use predicates from guards: $timer > 3, timer < 7, timer > 8, timer < 8, requests > 0, requests = 0, t1 < 4, t1 = 4, x \leq 2, x > 2, t2 > 0, t2 = 0$.

5.4.3 Discussion

We see that the approach works for non-trivial infinite state systems. The approach proves to be effective in computing finite bisimilar structures of non-trivial

$$
\begin{array}{lcl}
pc_0 = 0 & \longmapsto & pc_0 := 1, choosing_0 := 1, j_0 := 0, max_0 := 0 \\
pc_0 = 1 \wedge j_0 = 0 & \longmapsto & j_0 := j_0 + 1 \\
pc_0 = 1 \wedge j_0 = 1 \wedge max_0 < num_1 & \longmapsto & max_0 := num_1, j_0 := j_0 + 1 \\
pc_0 = 1 \wedge j_0 = 1 \wedge \neg max_0 < num_1 & \longmapsto & j_0 := j_0 + 1 \\
pc_0 = 1 \wedge j_0 = 2 & \longmapsto & pc_0 := 3, num_0 := max_0 + 1, j_0 := 0, \\
 & & choosing_0 := 0 \\
pc_0 = 3 \wedge j_0 = 0 \wedge choosing_0 = 0 & \longmapsto & pc_0 := 4 \\
pc_0 = 3 \wedge j_0 = 1 \wedge choosing_0 = 0 & \longmapsto & pc_0 := 4 \\
pc_0 = 4 \wedge j_0 = 0 & \longmapsto & pc_0 := 3, j_0 := j_0 + 1 \\
pc_0 = 4 \wedge j_0 = 1 \wedge & & \\
(num_1 = 0 \vee num_1 < num_0) & \longmapsto & pc_0 := 3, j_0 := j_0 + 1 \\
pc_0 = 3 \wedge j_0 = 2 & \longmapsto & pc_0 := 5 \\
pc_0 = 5 & \longmapsto & pc_0 := 0, num_0 := 0 \\
pc_1 = 0 & \longmapsto & pc_1 := 1, choosing_1 := 1, j_1 := 0, max_1 := 0 \\
pc_1 = 1 \wedge j_1 = 0 \wedge max_1 < num_0 & \longmapsto & max_1 := num_0, j_1 := j_1 + 1 \\
pc_1 = 1 \wedge j_1 = 0 \wedge \neg max_1 < num_0 & \longmapsto & j_1 := j_1 + 1 \\
pc_1 = 1 \wedge j_1 = 1 & \longmapsto & j_1 := j_1 + 1 \\
pc_1 = 1 \wedge j_1 = 2 & \longmapsto & pc_1 := 3, num_1 := max_1 + 1, j_1 := 0, \\
 & & choosing_1 := 0 \\
pc_1 = 3 \wedge j_1 = 0 \wedge choosing_1 = 0 & \longmapsto & pc_1 := 4 \\
pc_1 = 3 \wedge j_1 = 1 \wedge choosing_1 = 0 & \longmapsto & pc_1 := 4 \\
pc_1 = 4 \wedge j_1 = 0 \wedge & & \\
(num_0 = 0 \vee num_0 \leq num_1) & \longmapsto & pc_1 := 3, j_1 := j_1 + 1 \\
pc_1 = 4 \wedge j_1 = 1 & \longmapsto & pc_1 := 3, j_1 := j_1 + 1 \\
pc_1 = 3 \wedge j_1 = 2 & \longmapsto & pc_1 := 5 \\
pc_1 = 5 & \longmapsto & pc_1 := 0, num_1 := 0
\end{array}
$$

Figure 5.8: The bakery example for two processes.

Iteration	Concrete states	Abstract states	TP queries	Cache hits
1	429	223	90	277
New predicates:	$0 < num_0, num_0 < 0, num_1 < 0, 0 < num_1, max_0 < 0, max_1 < 0,$ $max_0 + 1 = num_1, max_0 + 1 < num_1, max_0 + 1 = 0, num_0 = max_1 + 1,$ $max_1 + 1 < num_0, max_1 + 1 = 0, max_0 < max_1 + 1, max_1 < max_0 + 1$			
2	565	291	620	2023
New predicates:	$max_1 < num_1 + 1, num_1 < max_1 + 1, num_0 < max_0 + 1, max_0 < num_0 + 1,$ $1 < num_1, 1 = num_1, 1 < num_0, num_0 = 1, max_0 < 1, max_1 < 1$			
3	791	410	1271	4412
New predicates:	-			

Table 5.3: The bakery protocol: intermediate results.

Iter.	Concr. states	Abs. states	TP queries	Cache hits	New predicates
1	69	44	10	10	$e_1 = 0, e_2 = 0$
2	101	65	20	44	$e_1 = -1, e_2 = -1$
3	101	65	26	64	$e_1 = -2, e_2 = -2$
4	101	65	32	84	...

Table 5.4: The RAX example: intermediate results.

$$
\begin{array}{lcl}
pc_1 = 1 & \longmapsto & c_1 := 0, pc_1 := 2 \\
pc_1 = 2 \wedge c_1 = e_1 & \longmapsto & pc_1 := 3 \\
pc_1 = 3 & \longmapsto & w_1 := 1, pc_1 := 4 \\
pc_1 = 4 \wedge w_1 = 0 & \longmapsto & pc_1 := 5 \\
pc_1 = 2 \wedge \neg(c1 = e_1) & \longmapsto & pc_1 := 5 \\
pc_1 = 5 & \longmapsto & c_1 := e_1, pc_1 := 6 \\
pc_1 = 6 & \longmapsto & e_2 := e_2 + 1, w_2 := 0, pc_1 := 2 \\
\\
pc_2 = 1 & \longmapsto & c_2 := 0, pc_2 := 2 \\
pc_2 = 2 & \longmapsto & e_1 := e_1 + 1, w_1 := 0, pc_2 := 3 \\
pc_2 = 3 \wedge c_2 = e_2 & \longmapsto & pc_2 := 4 \\
pc_2 = 4 & \longmapsto & w_2 := 1, pc_2 := 5 \\
pc_2 = 5 \wedge w_2 = 0 & \longmapsto & pc_2 := 6 \\
pc_2 = 3 \wedge not(c2 = e_2) & \longmapsto & pc_2 := 6 \\
pc_2 = 6 & \longmapsto & c_2 := e_2, pc_2 := 2
\end{array}
$$

Figure 5.9: The RAX example.

Iteration	Concrete states	Abstract states	TP queries	Cache hits
1	2025	1682	380	446
New predicates:	$timer > 6, timer > 2, timer = 7, timer < 7,$ $x > 1, x <= 1, t2 = 1, t2 > 1$			
2	4805	2495	1613	3114
New predicates:	$timer > 5, timer < 6, timer = 6, timer > 1,$ $t1 = 3, t1 < 3, x \leq 0, x > 0$			
4	6334	4799	4130	11567
New predicates:	$t1 = 1, t1 < 1, timer > -1$			
5	6334	4799	4472	13029
New predicates:	$t1 = 0, t1 < 0$			
6	6334	4799	4593	13657
New predicates:	-			

Table 5.5: The Sorter: intermediate results.

infinite-state systems and in finding errors using under-approximation based predicate abstraction.

The main limiting factor is the number of theorem prover calls. We believe that this is not such an issue as it may seem from our results:

- In our examples, we work with manually constructed models. These models, naturally, contain only relevant variables. In applications to real programming languages we can expect much larger portion of variables which can be abstracted away.
- The number of theorem prover calls can be significantly reduced by optimizations. Even the simple optimizations that we implemented reduced the number of queries by order of magnitude.

In general, we believe that the technique presented here is complementary to over-approximation abstraction techniques and it can be used in conjunction with such techniques, as an efficient way of discovering feasible counterexamples. We view the integration of the two approaches as an interesting topic for future research. Our technique explores transitions that are guaranteed to be feasible in the state space bounded by the abstraction predicates. In contrast, the over-approximation based techniques may also explore transitions that are spurious and therefore could require additional refinement before reporting a real counterexample. Hence, our technique can potentially finish in fewer iterations and it can use fewer predicates (which enable more state space reduction), while retaining the model checker's capability of finding real errors.

5.5 Related Work

The most closely related work to ours is the work of Grumberg et al. [98] where a refinement of an under-approximation is used to improve analysis of multi-process systems. The procedure in [98] checks models with an increasing set of allowed interleavings of the given processes, starting from a single interleaving. It uses SAT-based bounded model checking for analysis and refinement, whereas here we focus on explicit model checking and predicate abstraction, and we use weakest precondition calculations for abstraction refinement.

Another closely related work is that of Lee and Yannakakis [131], which proposes an on-the-fly algorithm for computing the bisimulation quotient of an (infinite state) transition system. Similar to our approach, the algorithm from [131] traverses concrete transitions while computing *blocks* of equivalent states; if some transition is found to be *unstable* the block is *split* into sub-blocks . There are some important differences between our approach and the work presented in [131]. We use refinement globally while the block splitting in [131] is local. This makes the approach in [131] more efficient in the number of visited states, but less efficient in the treatment of states/blocks and more difficult to realize — the work [131] provides realization only

for restricted systems of affine transition systems. As a consequence of this global refinement, our algorithm may not compute *the* bisimulation quotient (as in [131]) but rather just *a* bisimilar structure (due to extra refinement). Moreover, unlike [131], our algorithm is formulated in terms of predicate abstraction and it provides a clear separation between state exploration and refinement. As a result, we have a simple algorithm that we believe it is easy to understand and implement and it allows for combination with other abstraction approaches (e.g. over-approximation techniques based on predicate abstraction).

Our approach can be contrasted with the work on predicate abstraction for modal transition systems [92, 163], used in the verification and refutation of branching time temporal logic properties. An abstract model for such logics distinguishes between *may* transitions, which over-approximate transitions of the concrete model, and *must* transitions, which under-approximate the concrete transitions (see also [11, 65, 69]). The technique presented here explores and generates a structure which is *more precise* (contains more feasible behaviors) than the model defined by the *must* transitions, for the same abstraction predicates. The reason is that the model checker explores transitions that correspond not only to *must* transitions, but also to *may* transitions that are feasible.

Moreover, unlike [92, 163] and over-approximation based abstraction techniques [15, 52], the under-approximation and refinement approach does not require the a priori construction of the abstract transition relation, which involves exponentially many theorem prover calls (in the number of predicates), regardless of the size of (the reachable portion of) the analyzed system. In our case, the model checker executes concrete transitions and a theorem prover is only used during refinement, to determine whether the abstraction is exact with respect to each executed transition. Every such calculation makes at most two theorem prover calls, and it involves only the *reachable* state space of the system under analysis. Another difference with previous abstraction techniques is that the refinement process is not guided by the spurious counterexamples, since no spurious behavior is explored. Instead, the refinement is guided by the failed exactness checks for the explored transitions.

Păsăreanu et al. [159] developed a technique for finding guaranteed feasible counterexamples in abstracted programs. The technique essentially explores an under-approximation defined by the *must* abstract transitions (although the presentation is not formalized in these terms). The work presented here explores an under-approximation which is more precise than the abstract system defined by the *must* transitions. Hence it has a better chance of finding errors while enabling more aggressive abstraction and therefore more state space reduction.

Holzmann and Joshi [114] advocates the use of abstraction mappings during concrete model checking in a way similar to what we present here (they called the approach "model-driven software verification"). In their approach, the abstraction function needs to be provided by the user. The CMC model checking tool [141] also attempts to store state information in memory using aggressive compressing techniques (which can be seen as a form of abstraction), while the detailed state information is kept on the stack. These techniques allow the detection of subtle errors

which can not be discovered by classical model checking, using e.g., breadth-first search or by state-less model checking [90]. While these techniques use abstractions in an ad-hoc manner, our work contributes the automated generation and refinement of abstractions.

Chapter 6

Sampled Semantics of Timed Systems

He looked up at his clock, which had stopped at five minutes to eleven some weeks ago. "Nearly eleven o'clock," said Pooh happily. "You're just in time for a little smackerel of something." [139]

Now we turn to the study of timed systems, i.e., systems with clocks (and also stopped clocks). In this chapter we study some theoretical aspects of timed systems. In the next chapter we turn our attention to a practical verification problem.

Timed systems can be considered with two types of semantics — dense time semantics and discrete time semantics. The most typical examples are *real* semantics and *sampled* semantics (i.e., discrete semantics with a fixed time step ϵ). We investigate the relations between real semantics and sampled semantics with respect to different behavioral equivalences. We also overview different non-emptiness problems for timed systems and we fill two missing results: non-emptiness of stopwatch automata with fixed sampling period and ω-language non-emptiness of timed automata with an unknown sampling period. Decidability of the latter problem is our main technical contribution (this problem was previously wrongly classified as undecidable). For the proof we employ a novel characterization of reachability relations between configurations of a timed automaton.

6.1 Introduction

Let us review the Sorter example. In previous chapters we discussed techniques suitable for dealing with software aspects of the system. The Sorter example, however, has also real time aspects which are important for a correct functionality of the system.

Our modeling of the system in Chapter 2 illustrates two possible approaches to dealing with real time aspects. The guarded command language model uses discretization — time is sampled, all events occur in discrete time intervals. The timed automata model uses dense time — events can occur in any moment. In this chapter we study some questions about these two approaches to time.

The semantics of models of timed systems can be defined over various time domains. The usual approach is to use *dense* time semantics, particularly *real* time semantics (time domain is $\mathbb{R}_0^+$). From many points of view, this semantics is very

plausible. One does not need to care about the granularity of time during the modeling phase. This semantics leads to an uncountable structure with a finite quotient (for timed automata) and thus it is amenable to verification with finite state methods. Moreover, theoretical and often also practical complexity of problems for dense time semantics is usually the same as for various discrete semantics.

Discrete semantics, particularly *sampled* semantics with fixed time step $\epsilon \in \mathbb{Q}^+_{>0}$ (time domain is $\{k \cdot \epsilon \mid k \in \mathbb{Z}^+_0\}$), is also often considered, e.g., in [39, 38, 27]. One of the advantages of the sampled time domain is a wider choice of representations for sets of clock valuations, e.g., explicit representation or symbolic representation using decision diagrams. Another important issue is implementability. If a system is realized on a hardware then there is always some granularity of time (e.g., clock cycle, sampling period). Therefore, sampled semantics is closer to the implementation then more abstract dense time semantics.

Dense time semantics can even lead to misleading verification results. Assume that we have a model of a timed system such that it satisfies some property in dense time semantics. Now the question is whether there is an implementation (realized on a discrete time hardware) such that it preserves this property. Dense time semantics allows behaviors which are not realizable in any real system. If satisfaction of the property depends on these behaviors then there might not be an implementation satisfying the property (we discuss such examples in this chapter).

Previous chapter suggests another motivation for sampled semantics — under-approximation refinement. For real-time systems, sampled semantics gives a natural under-approximation. This under-approximation can be moreover easily refined by decreasing ϵ.

The sampled semantics, on the other hand, brings the problem of determining a fixed sampling period. Therefore, we also consider non-emptiness problems with an unknown sampling period — for these problems the goal is to decide whether there exists some sampling period ϵ such that the language with respect to ϵ is non-empty.

Verification problems can be stated as (ω-)language non-emptiness problems. Non-emptiness problems are the main focus of this chapter. Our main result concerns ω-language non-emptiness problem with an unknown period, i.e., the problem of deciding whether there exists ϵ such that $L^\epsilon_\omega(A) \neq \emptyset$. We show that this problem is decidable for timed automata. This problem was previously wrongly classified as undecidable [5]. The same problem was studied in the control setting with a slightly different sampled behavior in [48]. In their setting, the automaton is a model of a controller with the periodic control loop which always gets a sampled data and performs a control action. Therefore, the automaton has to perform an action at every sampled time point. This is a difference from our definition of sampled semantics – the automaton can idle for several sampling periods. The fact that the automaton has to react at every sampled time point makes the problem undecidable even for (finite word) language non-emptiness.

Our proof uses a novel characterization of reachability relations in timed automata. Representations of reachability relations were studied before: using additive theory of real numbers [59] and $2n$-automata [73]. Our novel representation is based on sim-

ple linear (in)equalities (comparisons of clock differences). This representation is of independent interest, since it is simpler and more specific then previously considered characterizations and it gives a better insight into reachability relations.

We also study non-emptiness problems for stopwatch automata. Particularly, we provide a new undecidability proof for the non-emptiness problem in sampled semantics for stopwatch automata with diagonal constraints and one stopwatch.

Finally, we systematically study relations between dense time semantics and sampled semantics for different timed systems. We study these relations in terms of behavioral equivalences, as it is well known which verification results are preserved by which equivalence (see Chapter 2). All considered equivalences are "untimed" — the only important information for an equivalence are actions performed and not precise timepoints at which these actions are taken.

Relationship to Main Themes

- Equivalences. We study equivalences between different semantics of timed/stopwatch automata. These results provide insight into the applicability of discrete methods to real time verification. They are also used in the study of non-emptiness problems.
- Abstractions. We do not explicitly talk about abstraction in this chapter. Anyway, sampled semantics may be seen as a must abstraction of the dense semantics — it would be sufficient to use an abstraction function which maps each dense valuation into its nearest "sampled point".
- Approximation and refinement. We do not explicitly consider refinement algorithms here, but approximation and refinement serve as a motivation. The under-approximation refinement algorithm could be naturally applied to unknown period non-emptiness problems. But in this way we would not be able to obtain decidability results.

6.2 Region Graph

Before we discuss our results, we introduce a classical timed automata technique — a region graph construction (this construction is used in several of our proofs). Although we use a slightly nonstandard definition of a region graph, the reader who is familiar with time automata may wish to skip this section on the first reading.

For any $\delta \in \mathbb{R}$, $\mathsf{int}(\delta)$ denotes the integral part of δ and $\mathsf{fr}(\delta)$ denotes the fractional part of δ. Let k be an integer constant. We define the following relations on the valuations. The equivalence $\cong_k$ is a standard region equivalence (its equivalence classes are *regions*), the equivalence $\sim_k$ is an auxiliary relation which allows us to forget about the clocks whose values are above k.

- $\nu \cong_k \nu'$ iff all the following conditions hold:
  - for all $x \in \mathcal{C} : \mathsf{int}(\nu(x)) = \mathsf{int}(\nu'(x))$ or $\nu(x) > k \wedge \nu'(x) > k$,

  - for all $x, y \in \mathcal{C}$ with $\nu(x) \leq k$ and $\nu(y) \leq k : \mathsf{fr}(\nu(x)) \leq \mathsf{fr}(\nu(y))$ iff $\mathsf{fr}(\nu'(x)) \leq \mathsf{fr}(\nu'(y))$,
  - for all $x \in \mathcal{C}$ with $\nu(x) \leq k : \mathsf{fr}(\nu(x)) = 0$ iff $\mathsf{fr}(\nu'(x)) = 0$;
- $\nu \sim_k \nu'$ iff for all $x \in \mathcal{C} : \nu(x) = \nu'(x)$ or $\nu(x) > k \wedge \nu'(x) > k$.

Note that $\sim_k$ is refinement of $\cong_k$, $\cong_k$ has a finite index for all semantics, $\sim_k$ has a finite index for sampled semantics.

Lemma 6.1 ([4]) *Let A be a diagonal-free timed automaton and K be a maximal constant which occurs in some guard in A. For each location $l \in L : \nu \cong_K \nu' \Rightarrow (l, \nu) \sim (l, \nu')$.*

In the following (and in all subsequent uses of region graph) we suppose that the timed automaton is diagonal-free and that each transition resets at most one clock. Each timed automaton can be transformed into an automaton which satisfies these constraints and which is equivalent to the original one with respect to simulation equivalence.

Given a region D (an equivalence class of $\cong_K$) we define:

- *integral*(D) is a set of clocks x such that $x \leq K$ and $\mathsf{fr}(x) = 0$ in D,
- *fractional*(D) is a set of clocks x such that $x \leq K$ and $\mathsf{fr}(x) \neq 0$ in D,
- *maxfractional*(D) is a set of clocks x such that $x \leq K$ and the fractional part of x is maximal in D,
- *minfractional*(D) is a set of clocks x such that $x \leq K$ and the fractional part of x is minimal in D,
- *above*(D) is a set of clocks x such that $x > K$ in D,
- $D \models g, D' = D[Y := 0]$ are defined naturally.

A region D' is an *immediate time successor* of a region D iff one of the following holds:

- *integral*$(D) \neq \emptyset$: *integral*$(D') = \emptyset$; the ordering of fractional parts is the same in D' as in D; and *above*(D') is the same as *above*(D) except that it contains $x \in$ *integral*(D) such that $D(x) = K$, or
- *integral*$(D) = \emptyset$: *integral*(D') = *maxfractional*(D), integer values of these clocks are incremented; ordering of fractional parts in D' is the same as in D except for clocks in *maxfractional*(D) which become the smallest; *above*(D') = *above*(D).

A *region graph* of a timed automaton A is defined as follows[1]:

- states are tuples (l, D) where l is a location and D is a region such that *integral*$(D) \neq \emptyset$,
- there is an transition $(l, D) \rightarrow (l', D')$ iff one of the following applies

[1] Note that we use non-standard definition, because for the proof we need to work only with regions such that *integral*$(D) \neq \emptyset$ and we want to have 'as small steps as possible'.

	closed TA	TA	SWA
rational semantics	bisimilar	bisimilar	trace eq.
sampled semantics	similar	reachability eq.	reachability eq.

Table 6.1: Summary of equivalences between semantics: each field gives the relation to real semantics.

- **Time**: $l = l'$ and there exists D'' such that D'' is an immediate time successor of D and D' is an immediate time successor of D'',
- **Reset**: there exists a transition $(l, a, g, Y, l') \in E$ such that $D \models g$ and $D' = D[Y := 0]$,
- **Time&Reset**: there exists a region D'' and a transition $(l, a, g, Y, l') \in E$ such that D'' is a time successor of D, $D'' \models g$ and $D' = D''[Y := 0]$.

The region graph construction is finite and bisimilar to $[\![A]\!]_{\mathbb{R}_0^+}$ [4]. Thus we can use it to answer model checking problems, particularly the non-emptiness problem:

Theorem 6.1 ([4]) *Let A be a timed automaton. The problem of deciding whether $L(A)$ is non-empty is PSPACE-complete.*

6.3 Dense vs. Sampled Semantics

In this section, we present a set of results about relations between dense time semantics and sampled semantics of timed systems. This shows the limits of using discrete time verification methods for the dense time problems.

All the results are summarized in Table 6.1. For the sampled semantics, a given result means that there exists an ϵ such that the given equivalence is guaranteed. In a case of closed TA, such ϵ can be easily constructed from the syntax of the automaton (as the greatest common divisor of all constants). For general TA, such ϵ can be constructed, but it requires to explore the region graph corresponding to the automaton. For SWA, such ϵ cannot be constructed algorithmically (because we do not know which actions are reachable).

6.3.1 Real versus Rational

We start with relations between real and rational semantics as it creates a connection between dense and sampled semantics.

Lemma 6.2 *Let A be a timed automaton. Then $[\![A]\!]_{\mathbb{Q}_0^+}$ is bisimilar to $[\![A]\!]_{\mathbb{R}_0^+}$.*

Proof: This follows directly from the region construction since each region contains at least one rational valuation. □

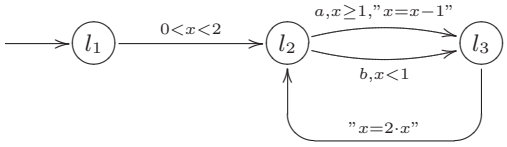


Figure 6.1: Stopwatch automaton for binary expansion. Clock x is stopped in locations l_2 and l_3.

For stopwatch automata, however, we can guarantee only trace equivalence — we show that there exists a SWA which has infinite traces realizable in real semantics, but not in rational one.

Lemma 6.3 *Let A be an SWA. Then $[\![A]\!]_{\mathbb{Q}_0^+}$ is trace equivalent to $[\![A]\!]_{\mathbb{R}_0^+}$.*

Proof: Let us consider a run π in $[\![A]\!]_{\mathbb{R}_0^+}$. We can consider the delays on this run as parameters $\delta_1, \ldots, \delta_n$. The set of values of these parameters, which enable execution through the same sequence of location and over the same trace is described by a system of linear inequalities in $\delta_1, \ldots, \delta_n$ — these inequalities are obtained by substituting sums of $\delta_1, \ldots, \delta_n$ for $\nu(x)$ in guards. The set of solutions of this system of linear inequalities is a non-empty convex polyhedron and it has a rational solution. Therefore, there exists a run π' in $[\![A]\!]_{\mathbb{Q}_0^+}$ over the same trace as π. □

Lemma 6.4 *There exist an SWA A such that $[\![A]\!]_{\mathbb{Q}_0^+}$ is* not *infinite trace equivalent to $[\![A]\!]_{\mathbb{R}_0^+}$.*

Proof: [sketch] A skeleton of the example illustrating this observation is given in Figure 6.1. The operations $x = x - 1$ and $x = 2 \cdot x$ are not valid operations of stopwatch automata, but can be simulated using several locations and (stopwatch) clocks (see e.g., [105]). The automaton in the first step nondeterministically chooses a value between 0 and 2 and then it accepts a sequence of actions a, b corresponding to a binary expansion of the chosen value. □

6.3.2 Real versus Sampled

Now we study relations between dense time and sampled semantics. We show that for a closed TA we can guarantee the simulation equivalence (i.e., that there is an ϵ such that $[\![A]\!]_{\mathbb{R}_0^+}$ is simulation equivalent to $[\![A]\!]_\epsilon$), but not bisimilarity. For general TA (as well as for SWA) the best what we can guarantee is the reachability equivalence (i.e., that there is an ϵ such that $[\![A]\!]_{\mathbb{R}_0^+}$ is reachability equivalent to $[\![A]\!]_\epsilon$).

Lemma 6.5 *Let A be a* closed *TA and ϵ be the greatest common divisor of constants in A. Then $[\![A]\!]_\epsilon$ is simulation equivalent to $[\![A]\!]_{\mathbb{R}_0^+}$.*

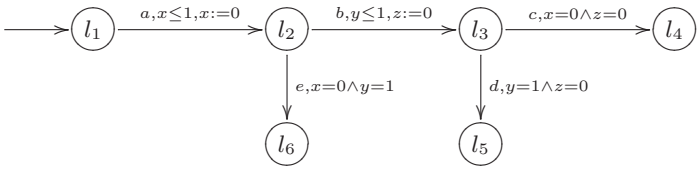


Figure 6.2: An automaton for which there is no ϵ such that discrete and dense semantics are bisimilar.

Proof: Let A' is an automaton obtained from A by dividing all constants by ϵ. Then $[\![A]\!]_\epsilon$ is bisimilar to $[\![A']\!]_1 = [\![A']\!]_{\mathbb{Z}_0^+}$. Therefore, it is sufficient to prove the claim for $\epsilon = 1$.

Let ν be a clock valuation and $[\nu]$ denote a valuation such that $[\nu](x) = \lfloor \nu(x) \rfloor$ or $[\nu](x) = \lceil \nu(x) \rceil$ for all clocks x. Consider the following relation:

$$S = \{(\nu, [\nu]) \mid \forall x, y.(([\nu](x) = \lfloor \nu(x) \rfloor) \wedge ([\nu](y) = \lceil \nu(y) \rceil)) \Rightarrow \mathsf{fr}(\nu(x)) < \mathsf{fr}(\nu(y))\}$$

Informally, $(\nu, \nu') \in S$ if and only if ν' is a corner point of a region containing ν. If A is closed then S is a simulation relation on $[\![A]\!]_{\mathbb{R}_0^+}$ and $[\![A]\!]_{\mathbb{Z}_0^+}$.

We show that S is a simulation relation. If a guard is enabled for ν and $(\nu, \nu') \in S$ then this guard is enabled also for ν', since A is closed. If A performs an action step resetting clocks in Y, then obviously $(\nu[Y := 0], \nu'[Y := 0]) \in S$. We show that S is preserved by two special delay steps (immediate time successor) such that any delay step can be composed of them.

The first delay step changes a valuation where some clocks have fractional part equal to zero to a valuation where all clocks have nonzero fractional parts, but the integral parts are the same. If A performs such a delay step in dense time semantics, we do not do anything in sampled time semantics.

The second delay step changes a valuation where all fractional parts are nonzero and x has a greatest fractional part to a valuation ν_{delay} where $\nu_{delay}(x) = \lceil \nu(x) \rceil$. If $\nu'(x) = \lceil \nu(x) \rceil$ then $\nu'_{delay} = \nu'$, otherwise $\nu'_{delay} = \nu' + 1$. Note, that $(\nu_{delay}, \nu' + 1) \in S$ because x had a greatest fractional part and thus for all other clocks y it holds that $\nu'(y) = \lfloor \nu(y) \rfloor$.

□

Lemma 6.6 *There exits a closed TA A such that $[\![A]\!]_{\mathbb{R}_0^+}$ is* not *bisimilar to $[\![A]\!]_\epsilon$ for any ϵ.*

Proof: Figure 6.2 shows an automaton for which there is no ϵ such that dense time and sampled semantics are bisimilar. For the proof we use a characterization of bisimulation in terms of a game between Challenger and Defender [166]. Consider the following play of the bisimulation game. We suppose that ϵ is a divisor of 1 (otherwise we choose ϵ' such that $\epsilon' \mid 1 \wedge \epsilon' \mid \epsilon$). Challenger plays with sampled semantics, delays for $1 - \epsilon$ and then takes a transition. Defender can delay for

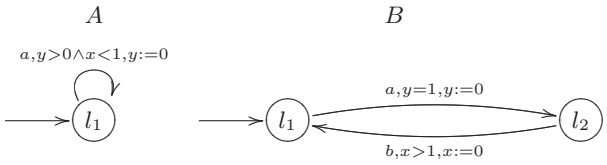


Figure 6.3: Difference between dense and sampled semantics (example (b) is taken from [5]).

$0 \leq \delta < 1$ and then take a transition. If Defender delays for 1 time unit then Challenger takes e transition, which Defender cannot take.

Now Challenger plays with dense time semantics, delays for $(1 - \delta)/2$ and then takes b transition. Defender can either delay for 0 or for ϵ and then take b transition. Challenger plays with sampled semantics again in the next step, delays for 0 and takes a transition according to the previous move of Defender. If Defender delayed for 0 then Challenger takes d transition, otherwise he takes c transition. Defender has no answer. □

Lemma 6.7 *Let A be an SWA. Then there exists ϵ such that $[\![A]\!]_{\mathbb{R}_0^+}$ is reachability equivalent to $[\![A]\!]_\epsilon$.*

Proof: From Lemma 6.3 we have that for each reachable action a there is a finite run π_a which contains action a and which has only rational delays. Let ϵ_a be the greatest common divisor of all delays on π_a. Let ϵ be the greatest common divisor of all ϵ_a where a is a reachable action. Then, clearly, each action is reachable in $[\![A]\!]_\epsilon$ if and only if it is reachable in $[\![A]\!]_{\mathbb{R}_0^+}$. □

Lemma 6.8 *There exists a TA A such that $[\![A]\!]_{\mathbb{R}_0^+}$ is* not *trace equivalent to $[\![A]\!]_\epsilon$ for any ϵ.*

Such an automaton could be easily obtained by enabling *zeno* behavior (arbitrary number of events in finite time), see Figure 6.3 A. Zeno behavior cannot be obtained in the sampled semantics. But there are also non-zeno automata which are not trace equivalent in dense and sampled domains, see Figure 6.3 B.

6.4 Reachability Relations

In this section we study reachability relations and their efficient representations. Reachability relations describe which valuations can be reached from a given valuation. Representations of reachability relations were studied before[2]: using additive

[2]Definition of reachability relations in these works is slightly different from ours. Our definition is geared towards application in the proof of Theorem 6.3. Nevertheless, the difference is not significant with respect to results about representations.

theory of real numbers [59] and $2n$-automata [73].

We present a novel simple representation for reachability relations (called clock difference relations). We use this characterization in the next section to prove our main result (Theorem 6.3). The fact that such simple (in)equalities are sufficient to capture the reachability relations and that these relations can be computed effectively is of independent interest and can be used in other applications which are beyond the scope of this thesis (see [59, 73]).

Let $(l, D), (l, D')$ be two states in a region graph. Then *reachability relation* of the tuple $(l, D), (l, D')$ is a relation on valuation $C_{(l,D)(l',D')} \subseteq D \times D'$ such that for each $\nu \in D, \nu' \in D'$:

$$(\nu, \nu') \in C_{(l,D)(l',D')} \Longleftrightarrow \exists \nu'' \sim_K \nu' : (l, \nu) \rightarrow^+ (l, \nu'')$$

Clock difference relations (CDR) structure over a set of clocks $\mathcal{C}$ is a set of (in)equalities of the following form:

– $x' - y' \bowtie u - v$
– $x' - y' \bowtie 1 - (u - v)$

where $\bowtie \in \{<, >, =\}$, $x, y, u, v \in \mathcal{C}$. The semantics of a CDR B is defined as follows. We say that a pair of valuations (ν, ν') satisfies B ($(\nu, \nu') \vDash B$) if and only if:

– if $x' - y' \bowtie u - v \in B$ then $\mathsf{fr}(\nu'(x)) - \mathsf{fr}(\nu'(y)) \bowtie \mathsf{fr}(\nu(u)) - \mathsf{fr}(\nu(v))$,
– if $x' - y' \bowtie 1 - (u - v) \in B$ then $\mathsf{fr}(\nu'(x)) - \mathsf{fr}(\nu'(y)) \bowtie 1 - (\mathsf{fr}(\nu(u)) - \mathsf{fr}(\nu(v)))$,

Theorem 6.2 *Reachability relations $C_{(l,D)(l',D')}$ are effectively definable as a finite unions of clock difference relations.*

The proof of this theorem is based on the following key facts:

– If there is an immediate transition $(l, D) \rightarrow (l', D')$ then the reachability relation can be directly expressed as a CDR.
– If $(l, D) \rightarrow^+ (l'', D'') \rightarrow (l', D')$ and the reachability relation over (l, D), (l'', D'') is expressed as a union of CDRs then the reachability relation over (l, D), (l', D') can be expressed as a union of CDRs as well.
– Using these two steps, reachability relations can be computed by a standard dynamic programming algorithm. Termination is guaranteed, because there is only a finite number of CDRs over a fixed set of clocks. Correctness is proved by induction with respect to the length of a path between (l, D) and (l', D').

The formal proof of this theorem is rather technical and complicated. Since reachability relations are not directly connected with main topics of this thesis, we do not present the proof here. An interested reader may find the full proof in a technical report [121].

6.5 Non-emptiness Problems

Most verification problems can be reduced to some non-emptiness problem, i.e., non-emptiness problems are at the core of verification. As we show below, for timed systems, there is quite a large number of non-emptiness problems to consider (each of them answers different verification question). In the verification effort, we have to be careful to solve the right one.

Let us, for example, consider examples in Figure 6.3. For these examples $L_\omega(A)$ is non-empty whereas for all ϵ the language $L^\epsilon_\omega(A)$ is empty. Non-emptiness implies existence of a behavior which violates a given liveness property. These examples demonstrates that all infinite traces may be non-realizable. Therefore, on a real system the property would be satisfied.

On the other hand, it may be difficult to choose the right sampling period ϵ. Therefore, it may be useful to consider the *unknown period non-emptiness problem*, i.e., the question whether there exists an ϵ such that $L^\epsilon_\omega(A)$ is non-empty.

Let us summarize all the parameters of non-emptiness problems in our setting:

- type of the automaton[3]:
  - timed automata,
  - diagonal-free stopwatch automata,
  - stopwatch automata;
- semantics:
  - dense semantics[4],
  - sampled semantics - a fixed sampling period,
  - sampled semantics - an unknown sampling period;
- type of language:
  - finite words (non-emptiness is equivalent to reachability in the semantics[5]),
  - infinite words (non-emptiness is equivalent to cycle detection in the semantics).

By combining all these cases we obtain 18 different non-emptiness problems. In the rest of this section we provide results for cases which were not considered before (or were considered erroneously) and then we give a brief summary of all 18 cases.

6.5.1 Timed Automata, Infinite Words, Unknown Period

The most interesting case is for languages of infinite words for timed automata with an unknown sampling period. We show that the problem of deciding whether there exists an ϵ such that $L^\epsilon_\omega(A)$ is non-empty is decidable. This problem was considered in a survey paper [5] where it is claimed that the problem is undecidable, with a

[3] For non-emptiness problems, it is not important whether the automaton is closed and for timed automata it is also not important whether the automaton is diagonal-free.

[4] It is not important whether we consider real or rational semantics.

[5] In order to have meaningful non-emptiness problem for finite words, it is necessary to introduce accepting states. This extension is classical and straightforward and we gloss over this issue here.

reference to [48]. The work [48], however, deals with a slightly different problem: it is required that the timed automaton performs action step after *every* discrete time step (this requirement is motivated by control theory). In that setting, the problem is undecidable even for finite words. In our setting, the problem is decidable:

Theorem 6.3 *Let A be a timed automaton. The problem of deciding whether there exists ϵ such that $L^{\epsilon}_{\omega}(A) \neq \emptyset$ is decidable.*

Our proof is based on the region construction and on the application of Theorem 6.2 (representation of reachability relations by linear constraints). The region graph can be directly used for ω-language emptiness checking in dense semantics — the ω-language is non-empty if and only if there is a cycle in the region graph. This is, however, not true for sampled semantics, as illustrated by examples in Figure 6.3.

Intuitively, the problem is the following. Existence of a cycle in the region graph from a region (l, D) to itself means that there exists some valuations $\nu, \nu' \in D$ such that $(l, \nu) \rightarrow^{+} (l, \nu')$. These valuations may be constrained, e.g., in example in Figure 6.3(b) the constraint on paths from state $(l_1, [x = 0, 0 < y < 1])$ to itself is that $1 > \nu'(y) > \nu(y) > 0$. In dense semantics we can have an infinite run which satisfies this constraint, but in sampled semantics we cannot. In sampled semantics we need a path $(l, \nu) \rightarrow^{+} (l, \nu')$ such that $\nu \sim_k \nu'$ (valuations may differ only in clocks above constants).

Lemma 6.9 *There exists an ϵ such that $L^{\epsilon}_{\omega}(A)$ is non-empty if and only if there exists a reachable state (l, D) in the region graph of A such that the following condition is satisfiable:*

$$\exists \nu, \nu' \in D : (\nu_0, \nu) \in C_{(l_0, D_0)(l,D)} \wedge (\nu, \nu') \in C_{(l,D)(l,D)} \wedge \nu \sim_K \nu'$$

Proof: At first, suppose that the condition is satisfiable. Due to Theorem 6.2, the condition can be expressed as boolean combination of linear inequalities. The set of solutions is an union of convex polyhedrons and therefore there must exist a rational solution ν, ν'. From the definition of reachability relations we get that there exists $\nu'' \sim_K \nu$ such that $(l_0, \nu_0) \rightarrow^{+} (l, \nu) \rightarrow^{+} (l, \nu'')$ in the real semantics. Since real and rational semantics are bisimilar, there exists such a path in rational semantics as well. We take ϵ as the greatest common divisor of time steps on this path. Thus the path $(l_0, \nu_0) \rightarrow^{+} (l, \nu) \rightarrow^{+} (l, \nu'')$ is executable in $[\![A]\!]_{\epsilon}$ and since ν'' is bisimilar to ν (because $\nu \sim_K \nu''$) we can construct an infinite run. Therefore $L^{\epsilon}_{\omega}(A)$ is non-empty.

On the other hand, if $L^{\epsilon}_{\omega}(A)$ is non-empty then there exists an infinite run $(l_0, \nu_0) \rightarrow (l_1, \nu_1) \rightarrow (l_2, \nu_2) \rightarrow \ldots$. Since $\sim_K$ has a finite index (over sampled semantics) there must exists i, j such that $l_i = l_j, \nu_i \sim_K \nu_j$. These valuation demonstrate the satisfiability of the condition. □

Now, we can easily prove the main result :

Proof: [of Theorem 6.3] The result now directly follows from Lemma 6.9, since the condition given in this lemma can be expressed by linear constraints (due to Theorem 6.2) and satisfiability of such constraint can be decided (it is a special case of linear programming). □

6.5.2 Stopwatch Automata, Finite Words, Fixed Period

The second interesting case is for finite words and stopwatch automata with a fixed period. We show that the choice of the time domain and the type of constraints are important. With dense semantics, the problem is known to be undecidable even for diagonal-free constraints and one stopwatch [105]. We show that in sampled semantics the problem is decidable for diagonal-free constraints. However, if we allow diagonal constraints, the non-emptiness problem is again undecidable. We have to use a different reduction than in the dense case, but, surprisingly, only one stopwatch suffices even in the case of sampled semantics.

Lemma 6.10 *Let A be a diagonal-free SWA and ϵ a given sampling period. Then the finite words non-emptiness problem in sampled semantics $[\![A]\!]_\epsilon$ is PSPACE-complete.*

Proof: We use a standard extrapolation approach[6] — it is easy to check that the relation $\sim_K$ induces bisimulation on $[\![A]\!]_\epsilon$ (K is the largest constant occurring in guards) for a diagonal-free SWA. We can easily obtain unique representant of each bisimulation class (by extrapolating all clock values larger than K to the value $K+1$) and thus we can easily perform the search over the bisimulation collapse.

Complexity: PSPACE-membership follows from the algorithm (search in an exponential graph can be done in polynomial space), PSPACE-hardness follows from PSPACE-hardness for timed automata. □

Lemma 6.11 *Let A be an SWA with one stopwatch. Then the finite words non-emptiness problem in sampled semantics $[\![A]\!]_\epsilon$ is undecidable.*

Proof: We show the undecidability by reduction from the halting problem for a two counter machine M. Since this is a usual approach in this area (see e.g., [105, 48, 36]), we just describe the main idea — how to encode counter values and perform increment/decrement.

The value of a counter i is represented as the difference of two clocks: $x_i - y_i$. Before the start of the simulation of M the simulating SWA nondeterministically guesses the maximal value c of counters during a computation of M and sets the stopwatch to the value $c+1$. From this moment, the stopwatch is stopped for the rest of the computation with the value $c+1$.

Values of clocks are kept in the interval $[0, c+1]$ all the time. Whenever the value of a clock reaches $c+1$, the clock is reseted. Testing the value of a counter for zero is straightforward: just testing $x_i = y_i$. Decrementing a counter i is performed by postponing the reset of a clock x_i by 1 time unit. Incrementing a counter i is performed by postponing the reset of a clock y_i by 1 time unit. During the increment we have to check for an 'overflow' — if the difference $x_i - y_i$ equals to c and we should perform an increment then it means that the initial nondeterministic guess was wrong and the simulation should not continue.

[6] We discuss extrapolation in more detail in the next chapter.

	timed automata	diagonal-free stopwatch automata	stopwatch automata
fixed period	PSPACE-compl.	PSPACE-compl.	undec
unknown period	PSPACE-compl./dec	undec	undec
dense	PSPACE-compl.	undec	undec

Table 6.2: Summary of non-emptiness problems: in all cases but one the result is the same for languages of finite, and infinite words. Only in the case of timed automata, an unknown period, and infinite words the complexity of the problem is not known.

Note that the stopwatch is used in a very limited fashion: it is stopped once and then keeps a constant value. □

6.5.3 Summary of Results

We are ready to present summary of all 18 non-emptiness problems. All problems, with potential exception of one for which the exact complexity is unknown, are either PSPACE-complete or undecidable. In all cases the (un)decidability result is the same for languages of finite and infinite words. Only in one case the justification is principally different. The summary is given in Table 6.2. Justification for these results is as follows:

1. timed automata, dense time: proved in [4],
2. timed automata, fixed period: proved in [4] (the same proof as for 1.),
3. timed automata, unknown period:
   – finite words: due to Lemma 6.7 this problem is equivalent to 1.,
   – infinite words: Theorem 6.3,
4. diagonal-free stopwatch automata, dense time: proved in [105],
5. diagonal-free stopwatch automata, fixed period: Lemma 6.10 (for languages of infinite words it is an easy extension),
6. diagonal-free stopwatch automata, unknown period:
   – finite words: due to Lemma 6.7 this problem is equivalent to 4.,
   – infinite words: easy reduction from the finite words case,
7. stopwatch automata, dense time: follows from 4.,
8. stopwatch automata, fixed period: Lemma 6.11,
9. stopwatch automata, unknown period: follows from 8.

6.6 Related Work

Timed automata were introduced by Alur and Dill [4], who also proved the basic results. Stopwatch automata and their expressive power was studied by Cassez and Larsen [49]. They showed that stopwatch automata are expressively equivalent to

linear hybrid automata. Abdeddaim and Maler [3] and Krčál and Yi [122] studied usefulness of stopwatch automata for modeling scheduling problems. Henzinger et al. [105] studied reachability problems for different types of hybrid automata, including stopwatch automata.

There has been a considerable amount of work related to discretization issues and verifying dense time properties using discrete time methods, e.g., [106, 128, 143, 10]. The main difference to our work is that usually only a fixed sampling period and a trace equivalence are considered.

Implementability issues are discussed in a connection with robust semantics of timed automata in [171, 160]. Goal of these works is to decide whether a set of bad states is reachable if the clock rates drift a little bit or the guards are enlarged a little bit. It is argued that a real hardware can never produce synchronized clocks and measure them with infinite precision.

Gollu et al. [94] studied discretization of timed automata preserving reachability. They present an elaborate discretization scheme which preserves reachability (and in fact even bisimilarity) for any timed automaton. Our discretization scheme is just sampling with a fixed period.

Practical aspects of verification with the use of sampled semantics are discussed in [39, 38, 27, 32, 9]. These works are concerned mainly with data structures for representing sets of discrete valuations (e.g., different types of decision diagrams). They do not consider theoretical problems concerning relations between dense and sampled semantics in greater depth.

Reachability relations were studied by Comon and Jurski [59] and by Dima [73]. Comon and Jurski [59] showed that reachability relations are definable by additive theory of real numbers (this theory is decidable). Their construction proceeds through timed automata without nested loops. Dima [73] showed that reachability relations are definable by $2n$-automata, a novel representation technique. The proof is based on closure of the representation under union, composition and star.

Chapter 7

Zone Based Abstractions of Time Systems

"Are we nearly there?" Alice managed to pant out at last. "Nearly there!" the Queen repeated. "Why, we passed it ten minutes ago! Faster!" [47]

In this chapter we consider the classical reachability problem for timed automata with dense time semantics. We propose a novel technique which improves the performance of state-of-the-art model checkers for timed automata.

In verification of timed automata we usually use zone based abstractions with respect to the maximal constants to which clocks of the timed automaton are compared. In this chapter we show that by distinguishing maximal lower and upper bounds we can obtain significantly coarser abstractions. We show the correctness of the new abstractions and we experimentally demonstrate their advantages.

7.1 Introduction

In the previous chapter we study fundamental questions about timed automata semantics and non-emptiness problems. In this chapter we turn to more practical aspects: given a timed automaton we want to solve a particular problem (reachability in dense time semantics) as quickly as possible, despite the fact that it is a PSPACE-complete problem. Techniques presented in this chapter make use of special use of clocks in practical timed automata models. More specifically, we distinguish between lower and upper bounds when consider maximal bounds to which clocks are compared. For example, in our Sorter example the clock `Timer` is compared only to lower bounds.

By their very definition timed automata describe (uncountable) infinite state-spaces. Thus, algorithmic verification relies on existence of exact finite abstractions. In the original work by Alur and Dill the so-called region-graph construction provided a "universal" such abstraction (we discuss this construction in the previous chapter). However, whereas well-suited for establishing decidability of problems related to timed automata, the region-graph construction is highly impractical from a tool-implementation point of view. Instead, most real-time verification tools apply abstractions based on so-called zones, which in practice provide much coarser (and hence smaller) abstractions.

To insure finiteness, it is essential that the given abstraction (region- or zone-based) takes into account the actual constants with which clocks are compared. In particular, the abstraction could identify states which are identical except for the clock values which exceed the *maximum* such constants. Obviously, the smaller we may choose these maximum constants, the coarser the resulting abstraction will be. Allowing clocks to have different (maximum) constants is an obvious first step in this direction, and in [20] this idea has been (successfully) taken further by allowing the maximum constants not only to depend on the particular clock but also of the particular location of the timed automata. In all cases the *exactness* is established by proving that the abstraction respects *bisimilarity* (i.e., states identified by the abstraction are bisimilar).

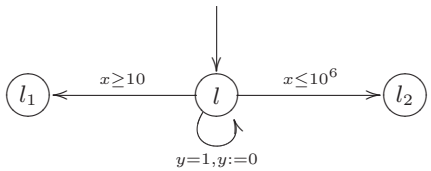


Figure 7.1: A small timed automaton.

Consider the timed automaton of Figure 7.1. Clearly 10^6 is the maximum constant for x and 1 is the maximum constant for y. Thus abstraction based on maximum constants will distinguish all states where $x \leq 10^6$ and $y \leq 1$. In particular, a forward computation of the full state space will – regardless of the search-order – create an excessive number of symbolic states including all symbolic states of the form $(l, x - y \leq k)$ where $0 \leq k \leq l0^6$. However, assuming that we are only interested in *reachability* properties the application of downwards closure with respect to *simulation* will lead to an exact abstraction which could potentially be substantially coarser than closure under bisimilarity. Observing that 10^6 is an *upper* bound on the edge from l to l_2 in Figure 7.1, it is clear that for any state where $x > 10$ increasing x will only lead to "smaller" states with respect to simulation preorder. In particular, applying this downward closure results in the radically smaller collection of abstract states, namely $(l, x - y \leq k)$ where $0 \leq k \leq 10$ and $(l, x - y \geq 11)$.

The fact that 10^6 is an *upper* bound in the example of Figure 7.1 is crucial for the reduction we obtained above. In this paper we present new, substantially coarser yet still exact abstractions which are based on *two* maximum constants obtained by distinguishing upper and upper bounds. In all cases the exactness is established by proving that the abstraction respects downwards closure with respect to simulation (i.e., for each state in the abstraction there is an original state simulating it). The variety of abstractions comes from the additional requirements to *effective* representation and *efficient* computation and manipulation. In particular we insist that zones can form the basis of our abstractions; in fact the suggested abstractions

are defined in terms of low-complexity operations on the difference bound matrix (DBM) representation of zones. We also experimentally demonstrate the significant speedups obtained by our new abstractions. Here, the distinction between lower and upper bounds is combined with the orthogonal idea of location-dependency of [20].

In [21] we also discuss how to use the idea of distinguishing upper and lower bounds to accelerate successor computation and we describe an application of the technique to a jobshop scheduling problem. We do not discuss these issues here since they are not directly related to our main themes.

Relationship to Main Themes

– Equivalences. We study, which equivalences are preserved by given extrapolation operation: classical extrapolation operation preserve bisimulation equivalence, whereas our new extrapolation preserves only simulation equivalence. Consideration of a weaker equivalence enables us to obtain a more powerful reduction technique.
– Abstractions. The symbolic semantics, which is the basic approach used in this section, is an (exact) abstraction of concrete semantics. Moreover, extrapolation operation are formalized in terms of abstraction.
– Approximations and refinement. These are not used in this chapter as our techniques produce exact results. We just compare (experimentally) the new techniques to convex-hull over-approximation.

7.2 Symbolic Semantics

For readability, let us denote the set of all real valuations $Val = \mathbb{R}^{\mathcal{C}}_{\geq 0}$. The symbolic semantics of a timed automaton $A = (L, Act, \mathcal{C}, l_0, E)$ is based on the abstract transition system $[\![A]\!]_S = (S, s_0, \Longrightarrow)$, where $S = L \times 2^{Val}$, and '$\Longrightarrow$' is defined by the following two rules:

– time step: $(l, W) \overset{delay}{\Longrightarrow} (l, W')$ if $W' = \{\nu + d \mid \nu \in W \wedge d \geq 0\}$,
– action step: $(l, W) \overset{act(a)}{\Longrightarrow} (l', W')$ if there exists a transition $(l, a, g, Y, l') \in E$ such that $W' = \{\nu[Y := 0] \mid \nu \in W \wedge \nu \models g\}$.

In a similar way to concrete semantics, we define the transition relation of $[\![A]\!]_S$ by concatenating these two types of transitions: $(l, W) \overset{a}{\Longrightarrow} (l', W')$ iff there exists (l'', W'') such that $(l, W) \overset{delay}{\Longrightarrow} (l'', W'') \overset{act(a)}{\Longrightarrow} (l', W')$.

Symbolic semantics induces a transition system which is countable (as opposed to uncountable $[\![A]\!]_{\mathbb{R}^+_0}$) but it still may be infinite and hence not directly usable for exploration. To obtain a finite graph one may, as suggested in [20], apply some abstraction $\alpha : 2^{Val} \mapsto 2^{Val}$, such that $W \subseteq \alpha(W)$. The abstract transition system $\mathcal{A}(\alpha, [\![A]\!]_S) = (S, s_0, \Longrightarrow_\alpha)$ is then given by the following inference rule:

$$\frac{(l, W) \Longrightarrow (l', W')}{(l, W) \Longrightarrow_\alpha \big(l', \alpha(W')\big)} \quad \text{if } W = \alpha(W)$$

To make the presentation more readable, we sometimes write in this chapter 'abstraction α' instead of '$\mathcal{A}(\alpha, [\![A]\!]_S)$'.

A simple way to assure that the reachability graph induced by '$\Longrightarrow_\alpha$' is finite is to establish that there are only finitely many abstractions of sets of valuations; that is, the set $\{\alpha(W) \mid \alpha \text{ defined on } W\}$ is finite. In such a case, α is a finite abstraction.

Of course, if α and α' are two abstractions such that for any set of valuations W, $\alpha(W) \subseteq \alpha'(W)$, we prefer to use abstraction α', because the graph induced by it, is *a priori* smaller than the one induced by α. Our aim is thus to propose an abstraction which is finite, as coarse as possible, and which induces an exact abstract transition system.

The abstraction traditionally used in real-time model-checkers such as Uppaal [129] and Kronos [37], is based on the idea that the behaviour of an automaton is only sensitive to changes of a clock if its value is below a certain constant. That is, for each clock there is a maximum constant such that once the value of a clock has passed this constant, its exact value is no longer relevant — only the fact that it is larger than the maximum constant matters. Transforming a DBM to reflect this idea is often referred to as *extrapolation* [35, 20] or *normalisation* [67]. In the following we choose the term *extrapolation*.

7.2.1 Classical Maximal Bounds

The classical abstraction for timed automata is based on maximal bounds, one for each clock of the automaton. Let $A = (L, Act, \mathcal{C}, l_0, E)$ be a timed automaton. The *maximal bound* of a clock $x \in \mathcal{C}$, denoted $M(x)$, is the maximal constant k such that there exists a guard containing $x \bowtie k$ in A. Let ν and ν' be two valuations. We define the following relation:

$$\nu \equiv_M \nu' \overset{\text{def}}{\Longleftrightarrow} \forall x \in \mathcal{C} : \text{either } \nu(x) = \nu'(x) \text{ or } \big(\nu(x) > M(x) \text{ and } \nu'(x) > M(x)\big)$$

.

Lemma 7.1 *The relation $R = \{((l, \nu), (l, \nu')) \mid \nu \equiv_M \nu'\}$ is a bisimulation relation.*

Definition 7.1 ($\alpha_{\equiv_M}$ w.r.t. $\equiv_M$) *Let W be a set of valuations. We define the abstraction w.r.t. $\prec_M$ as $\alpha_{\equiv_M}(W) = \{\nu \mid \exists \nu' \in W, \nu' \equiv_M \nu\}$.*

Lemma 7.2 *Let A be a timed automaton. Then the abstraction $\mathcal{A}(\alpha_{\equiv_M}, [\![A]\!]_S)$ is reachability equivalent to $[\![A]\!]_{\mathbb{R}_0^+}$ and finite.*

The above given lemmas come from [20]. They are, moreover, consequences of the below given results.

7.2.2 Lower and Upper Bounds

The new abstractions introduced in the following is substantially coarser than $\alpha_{\equiv_M}$. It is no longer based on a single maximal bound per clock but rather on two maximal bounds per clock allowing lower and upper bounds to be distinguished.

Definition 7.2 *Let $A = (L, Act, \mathcal{C}, l_0, E)$ be a timed automaton. The* maximal lower bound *denoted $L(x)$, (resp.* maximal upper bound *$U(x)$) of a clock $x \in \mathcal{C}$ is the maximal constant k such that there exists a constraint $x > k$ or $x \geq k$ (resp. $x < k$ or $x \leq k$) in a guard of some transition of A. If such a constant does not exist, we set $L(x)$ (resp. $U(x)$) to $-\infty$.*

Let us fix for the rest of this section a timed automaton A and bounds $L(x)$, $U(x)$ for each clock $x \in \mathcal{C}$ as above. The idea of distinguishing lower and upper bounds is the following: if we know that the clock x is between 2 and 4, and if we want to check that the constraint $x \leq 5$ can be satisfied, the only relevant information is that the value of x is greater than 2, and not that $x \leq 4$. In other terms, checking the emptiness of the intersection between a non-empty interval $[c, d]$ and $]-\infty, 5]$ is equivalent to checking whether $c > 5$; the value of d is not useful. Formally, we define the LU-preorder as follows.

Definition 7.3 (LU-preorder $\prec_{LU}$) *Let ν and ν' be two valuations. Then $\nu' \prec_{LU} \nu$ if and only if for each clock x:*

- *either $\nu'(x) = \nu(x)$,*
- *or $L(x) < \nu'(x) < \nu(x)$,*
- *or $U(x) < \nu(x) < \nu'(x)$.*

Lemma 7.3 *The relation $R = \{((l, \nu), (l, \nu')) \mid \nu' \prec_{LU} \nu\}$ is a simulation relation.*

Proof: The only non-trivial part in proving that R indeed satisfies the properties of a simulation relation is to establish that if g is a clock constraint, then "$\nu \models g$ implies $\nu' \models g$". Consider the constraint $x \leq c$. If $\nu(x) = \nu'(x)$, then we are done. If $L(x) < \nu'(x) < \nu(x)$, then $\nu(x) \leq c$ implies $\nu'(x) \leq c$. If $U(x) < \nu(x) < \nu'(x)$, then it is not possible that $\nu \models x \leq c$ (because $c \leq U(x)$). Consider now the constraint $x \geq c$. If $\nu(x) = \nu'(x)$, then we are done. If $U(x) < \nu(x) < \nu'(x)$, then $\nu(x) \geq c$ implies $\nu'(x) \geq c$. If $L(x) < \nu'(x) < \nu(x)$, then it is not possible that ν satisfies the constraint $x \geq c$ because $c \leq L(x)$. □

Using the above LU-preorder, we can now define a first abstraction based on the lower and upper bounds.

Definition 7.4 ($\alpha_{\prec_{LU}}$, abstraction w.r.t. $\prec_{LU}$) *Let W be a set of valuations. We define the abstraction w.r.t. $\prec_{LU}$ as $\alpha_{\prec_{LU}}(W) = \{\nu \mid \exists \nu' \in W,\ \nu' \prec_{LU} \nu\}$.*

Before going further, we illustrate this abstraction in Figure 7.2. There are several cases, depending on the relative positions of the two values $L(x)$ and $U(x)$ and of the valuation we are looking at. We represent with a plain line the value of $\alpha_{\prec_{LU}}(\{\nu_1\})$

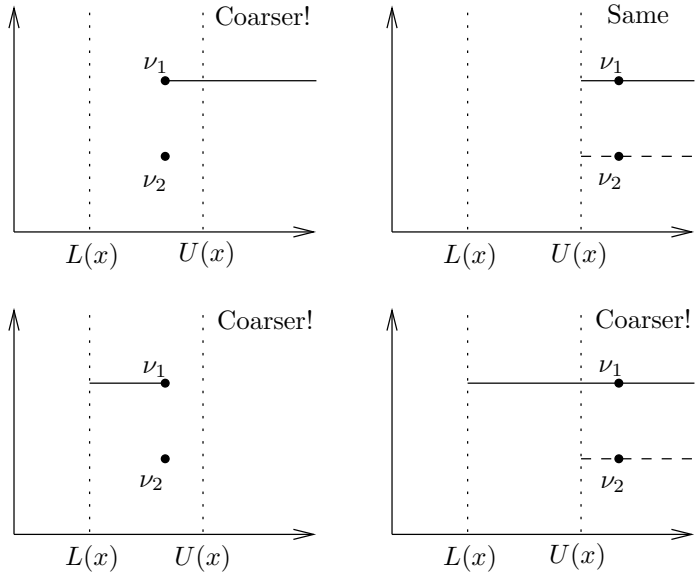


Figure 7.2: Quality of $\alpha_{\prec_{LU}}$ (full line) compared with $\alpha_{\equiv_M}$ (dashed line) for $M = \max(L, U)$.

and with a dashed line the value of $\alpha_{\equiv_M}(\{\nu_2\})$, where the maximal bound $M(x)$ corresponds to the maximum of $L(x)$ and $U(x)$. In each case, we indicate the "quality" of the new abstraction compared with the "old" one. We notice that the new abstraction is coarser in three cases and matches the old abstraction in the fourth case.

Lemma 7.4 *Let A be a timed automaton. Then the abstraction $\mathcal{A}(\alpha_{\prec_{LU}}, [\![A]\!]_S)$ is reachability equivalent to $[\![A]\!]_{\mathbb{R}_0^+}$ and coarser or equal to $\mathcal{A}(\alpha_{\equiv_M}, [\![A]\!]_S)$.*

Proof: Completeness is obvious, and soundness comes from Lemma 7.3. Definitions of $\alpha_{\prec LU}$ and $\alpha_{\equiv_M}$ give the last result because for each clock, x, we have $M(x) = \max(L(x), U(x))$. □

This result could suggest to use $\alpha_{\prec_{LU}}$ in real time model-checkers. However, we do not yet have an efficient method for computing the transition relation '$\Longrightarrow_{\alpha_{\prec_{LU}}}$'. Indeed, even if W is a zone, it might be the case that $\alpha_{\prec_{LU}}(W)$ is not even convex. For effectiveness and efficiency reasons we prefer abstractions which transform zones into zones because we can then use the DBM data structure.

7.3 Extrapolation Using Zones

In this section we present DBM-based extrapolation operators that gives abstractions which are exact, finite and also effective.

7.3.1 Zones and Difference Bound Matrices

A first step in finding an effective abstraction is realizing that W is always a zone whenever $(l^0, \{\nu_0\}) \Longrightarrow^* (l, W)$. A *zone* is a conjunction of constraints on the form $x \prec c$ or $x - y \prec c$, where x and y are clocks and $c \in \mathbb{Z}$. Zones can be represented using *Difference Bound Matrices* (DBM). We briefly recall the definition of DBMs, and refer to [72, 24, 34] for more details. A DBM is a square matrix $D = \langle c_{i,j}, \prec_{i,j} \rangle_{0 \leq i,j \leq n}$ such that $c_{i,j} \in \mathbb{Z}$ and $\prec_{i,j} \in \{<, \leq\}$ or $c_{i,j} = \infty$ and $\prec_{i,j} = <$. The DBM D represents the zone $[\![D]\!]$ which is defined by $[\![D]\!] = \{\nu \mid \forall 0 \leq i, j \leq n,\ \nu(x_i) - \nu(x_j) \prec_{i,j} c_{i,j}\}$, where $\{x_i \mid 1 \leq i \leq n\}$ is the set of clocks, and x_0 is a clock which is always 0, (*i.e.* for each valuation ν, $\nu(x_0) = 0$). DBMs are not a canonical representation of zones, but a normal form can be computed by considering the DBM as an adjacency matrix of a weighted directed graph and computing all shortest paths. In particular, if $D = \langle c_{i,j}, \prec_{i,j} \rangle_{0 \leq i,j \leq n}$ is a DBM in normal form, then it satisfies the *triangular inequality*, that is, for every $0 \leq i, j, k \leq n$, we have that $(c_{i,j}, \prec_{i,j}) \leq (c_{i,k}, \prec_{i,k}) + (c_{k,j}, \prec_{k,j})$ where comparisons and additions are defined in a natural way (see [34]). All operations needed to compute '$\Longrightarrow$' can be implemented by manipulating the DBMs.

7.3.2 Extrapolation Operations

The exact symbolic transition relations induced by abstractions considered so far, unfortunately do not preserve convexity of sets of valuations. In order to allow for sets of valuations to be represented *efficiently* as zones, we consider slightly finer abstractions α_{Extra} such that for every zone Z, $Z \subseteq \alpha_{Extra}(Z) \subseteq \alpha_{\equiv_M}(Z)$ (resp. $Z \subseteq \alpha_{Extra}(Z) \subseteq \alpha_{\prec_{LU}}(Z)$) (this ensures correctness) and $\alpha_{Extra}(Z)$ is a zone (this gives an effective representation). These abstractions are defined in terms of *extrapolation* operators on DBMs. If *Extra* is an extrapolation operator, it defines an abstraction on zones α_{Extra} such that for every zone Z, $\alpha_{Extra}(Z) = [\![Extra(D_Z)]\!]$ where D_Z is the DBM in normal form which represents the zone Z.

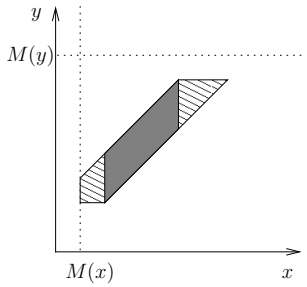
In the remainder, we consider a timed automaton A over a set of clocks $\mathcal{C} = \{x_1, \ldots, x_n\}$ and we suppose that we are given another clock x_0 which is always zero. For all these clocks, we define the constants $M(x_i)$, $L(x_i)$, $U(x_i)$ for $i = 1, ..., n$. For x_0, we set $M(x_0) = U(x_0) = L(x_0) = 0$ (x_0 is always equal to zero, so we assume we are able to check whether x_0 is really zero). In our framework, a zone is represented by DBMs of the form $\langle c_{i,j}, \prec_{i,j} \rangle_{i,j=0,\ldots,n}$.

We now present several extrapolations starting from the classical one and improving it step by step. Each extrapolation is illustrated by a small picture representing a zone (in black) and its corresponding extrapolation (dashed).

Classical Extrapolation Based on Maximal Bounds $M(x)$.

If D be a DBM $\langle c_{i,j}, \prec_{i,j}\rangle_{i,j=0\ldots n}$, $Extra_M(D)$ is given by the DBM $\langle c'_{i,j}, \prec'_{i,j}\rangle_{i,j=0\ldots n}$ defined and illustrated below:

$$(c'_{i,j}, \prec'_{i,j}) = \begin{cases} \infty & \text{if } c_{i,j} > M(x_i) \\ (-M(x_j), <) & \text{if } -c_{i,j} > M(x_j) \\ (c_{i,j}, \prec_{i,j}) & \text{otherwise} \end{cases}$$



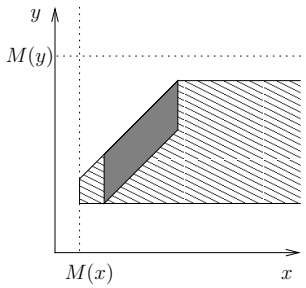
This is the extrapolation operator used in the real-time model-checkers Uppaal and Kronos. This extrapolation removes bounds that are larger than the maximal constants. The correctness follows from $\alpha_{Extra_M}(Z) \subseteq \alpha_{\equiv_M}(Z)$ and is proved in [35] and for the location-based version in [20].

In the remainder, we propose several other extrapolations that improve the classical one, in the sense that zones obtained with the new extrapolations are larger than zones obtained with the classical extrapolation.

Diagonal Extrapolation Based on Maximal Constants $M(x)$.

The first improvement consists in noticing that if the whole zone is above the maximal bound of some clock, then we can remove some of the diagonal constraints of the zones, even if they are not themselves above the maximal bound. More formally, if $D = \langle c_{i,j}, \prec_{i,j}\rangle_{i,j=0,\ldots,n}$ is a DBM, $Extra^+_M(D)$ is given by $\langle c'_{i,j}, \prec'_{i,j}\rangle_{i,j=0,\ldots,n}$ defined as:

$$(c'_{i,j}, \prec'_{i,j}) = \begin{cases} \infty & \text{if } c_{i,j} > M(x_i) \\ \infty & \text{if } -c_{0,i} > M(x_i) \\ \infty & \text{if } -c_{0,j} > M(x_j), i \neq 0 \\ (-M(x_j), <) & \text{if } -c_{i,j} > M(x_j), i = 0 \\ (c_{i,j}, \prec_{i,j}) & \text{otherwise} \end{cases}$$

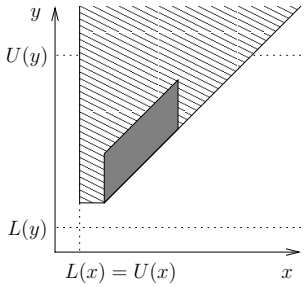


For every zone Z it then holds that $Z \subseteq \alpha_{Extra_M}(Z) \subseteq \alpha_{Extra_M^+}(Z)$.

Extrapolation Based on LU-bounds $L(x)$ and $U(x)$.

The second improvement uses the two bounds $L(x)$ and $U(x)$. If $D = \langle c_{i,j}, \prec_{i,j} \rangle_{i,j=0,\ldots,n}$ is a DBM, $Extra_{LU}(D)$ is given by $\langle c'_{i,j}, \prec'_{i,j} \rangle_{i,j=0,\ldots,n}$ defined as:

$$(c'_{i,j}, \prec'_{i,j}) = \begin{cases} \infty & \text{if } c_{i,j} > L(x_i) \\ (-U(x_j), <) & \text{if } -c_{i,j} > U(x_j) \\ (c_{i,j}, \prec_{i,j}) & \text{otherwise} \end{cases}$$

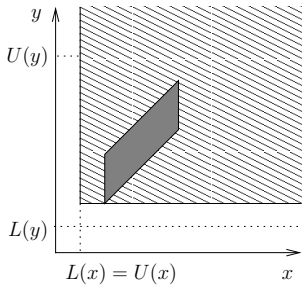


This extrapolation benefits from the properties of the two different maximal bounds. It does generalise the operator α_{Extra_M}. For every zone Z, it holds that $Z \subseteq \alpha_{Extra_M}(Z) \subseteq \alpha_{Extra_{LU}}(Z)$.

Diagonal Extrapolation Based on LU-bounds $L(x)$ and $U(x)$.

This last extrapolation is a combination of the extrapolation based on LU-bounds and the improved extrapolation based on maximal constants. It is the most general one. If $D = \langle c_{i,j}, \prec_{i,j} \rangle_{i,j=0,\ldots,n}$ is a DBM, then $Extra^+_{LU}(D)$ is given by the DBM $\langle c'_{i,j}, \prec'_{i,j} \rangle_{i,j=0,\ldots,n}$ defined as:

$$(c'_{i,j}, \prec'_{i,j}) = \begin{cases} \infty & \text{if } c_{i,j} > L(x_i) \\ \infty & \text{if } -c_{0,i} > L(x_i) \\ \infty & \text{if } -c_{0,j} > U(x_j), i \neq 0 \\ (-U(x_j), <) & \text{if } -c_{0,j} > U(x_j), i = 0 \\ (c_{i,j}, \prec_{i,j}) & \text{otherwise} \end{cases}$$



7.3.3 Correctness

We know that all the above extrapolations are complete abstractions as they transform a zone into a clearly larger one. Finiteness also comes immediately, because we can do all the computations with DBMs and the coefficients after extrapolation can only take a finite number of values. Effectiveness of the abstraction is obvious as extrapolation operators are directly defined on the DBM data structure. The only difficult point is to prove that the extrapolations we have presented are correct. To prove the correctness of all these abstractions, due to the inclusions shown in Figure 7.3, it is sufficient to prove the correctness of the largest abstraction, *viz* $\alpha_{Extra^+_{LU}}$.

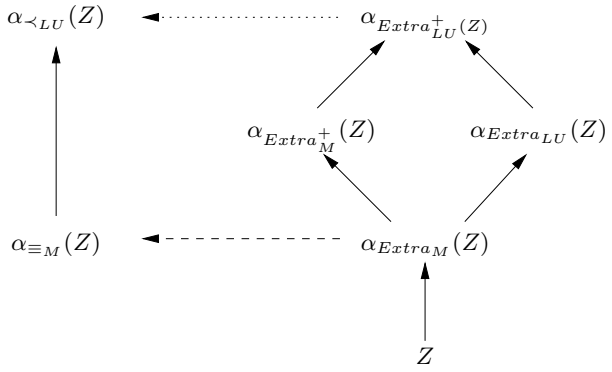


Figure 7.3: For any zone Z, we have the inclusions indicated by the arrows. The sets $\alpha_{Extra^+_M}(Z)$ and $\alpha_{Extra_{LU}}(Z)$ are incomparable. The α_{Extra} operators are DBM based abstractions whereas the other two are semantic abstractions. The dashed arrow was proved in [20] whereas the dotted arrow is the main result of this paper.

Lemma 7.5 *Let Z be a zone. Then $\alpha_{Extra^+_{LU}}(Z) \subseteq \alpha_{\prec_{LU}}(Z)$.*

The proof of this lemma is quite technical (notice, however, that it is a key result). Before we turn to this proof, we give the main theorem which states that $\alpha_{Extra^+_{LU}}$ is an abstraction which can be used in the implementation of timed automata.

Theorem 7.1 *Let A be a timed automaton. Then the abstraction $\mathcal{A}(\alpha_{Extra^+_{LU}}, [\![A]\!]_S)$ is reachability equivalent to $[\![A]\!]_{\mathbb{R}^+_0}$, finite and effectively computable.*

This theorem is a consequence of Lemma 7.5 and above stated results. Let us turn to the proof of Lemma 7.5 (the reader may wish to skip this proof on the first reading).

Proof: [of Lemma 7.5] Let $D = \langle c_{i,j};\ \prec_{i,j}\rangle_{i,j=0...n}$ be a DBM in normal form representing a non-empty zone. We note $D' = \langle c'_{i,j};\ \prec'_{i,j}\rangle_{i,j=0...n}$ be the DBM $Extra^+_{LU}(D)$. Let us fix $\nu \in [\![Extra^+_{LU}(D)]\!]$. We want to prove that $\nu \in \alpha_{\prec_{LU}}([\![D]\!])$. For this, we define the set P_ν as $\{\nu' \in [\![D]\!] \mid \nu' \prec_{LU} \nu\}$, and we prove that P_ν is not empty. This is sufficient as, by definition of $\alpha_{\prec_{LU}}$, we have $\nu \in \alpha_{\prec_{LU}}([\![D]\!]) \iff P_\nu \neq \emptyset$. The set P_ν is defined by the constraints:

$$\begin{aligned}
&\{x_i - x_j \prec_{i,j} c_{i,j} \mid i,j \in \{0,1,...,n\}\}\\
\cup\ &\{x_i > L(x_i) \mid \nu(x_i) > L(x_i) \text{ and } i \in \{1,...,n\}\}\\
\cup\ &\{x_i \leq \nu(x_i) \mid \nu(x_i) \leq U(x_i) \text{ and } i \in \{1,...,n\}\}\\
\cup\ &\{x_i \geq \nu(x_i) \mid \nu(x_i) \leq L(x_i) \text{ and } i \in \{1,...,n\}\}
\end{aligned}$$

The first set of constraints represents the constraints of D. The other lines represent the constraints due to $\cdot \prec_{LU} \nu$. Indeed, it is easy to check that for each i, the constraint $(x_i = \nu(x_i)) \vee (L(x_i) < x_i < \nu(x_i)) \vee (U(x_i) < \nu(x_i) < x_i)$ which is the direct definition of $\prec_{LU}$ is equivalent to the constraint $(\nu(x_i) > L(x_i) \implies x_i > L(x_i)) \wedge (\nu(x_i) \leq U(x_i) \implies x \leq \nu(x_i)) \wedge (\nu(x_i) \leq L(x_i) \implies x_i \geq \nu(x_i))$. The three last lines above correspond to this set of constraints.

We simplify the constraints defining P_ν. For this, we need the following three lemmas.

Lemma 7.6 *If $c_{j,0} < +\infty$, then $(c'_{j,0}, \prec'_{j,0}) = \infty$ implies $c_{j,0} > L(x_j)$.*

Lemma 7.7 *If $(c'_{0,i}, \prec'_{0,i}) \neq (c_{0,i}, \prec_{0,i})$ then $-c_{0,i} > U(x_i)$ and $(c'_{0,i}, \prec'_{0,i}) = (-U(x_i), <)$.*

Lemma 7.8 *Let $\nu \in [\![Extra^+_{LU}(D)]\!]$. Then*

1. *If $\nu(x_i) \leq U(x_i), L(x_i)$, then $\nu(x_i) \prec_{i,0} c_{i,0}$ and therefore $(\nu(x_i), \leq) \leq (c_{i,0}, \prec_{i,0})$.*
2. *If $\nu(x_i) \leq L(x_i), U(x_i)$, then $-c_{0,i} \prec_{0,i} \nu(x_i)$ and therefore $(-\nu(x_i), \leq) \leq (c_{0,i}, \prec_{0,i})$.*

Proof:

1. If $c_{i,0} > L(x_i)$, then we have $\nu(x_i) \prec_{i,0} c_{i,0}$. If $c_{i,0} \leq L(x_i)$, then $(c'_{i,0}, \prec'_{i,0}) = (c_{i,0}, \prec_{i,0})$ and we are done for the first inequality.
2. If $-c_{0,i} > U(x_i)$, then it is not possible as $(c'_{0,i}, \prec'_{0,i}) = (-U(x_i), <)$. Otherwise, $(c'_{0,i}, \prec'_{0,i}) = (c_{0,i}, \prec_{0,i})$. Thus we are also done for the second inequality.

□

Applying the previous lemmas, we get that P_ν is represented by the DBM $\langle p_{i,j}, \subset_{i,j} \rangle_{i,j=0\ldots n}$ where

$$
\begin{array}{rl}
(p_{i,0}, \subset_{i,0}) = & \begin{cases} (\nu(x_i), \leq) & \text{if } \nu(x_i) \leq L(x_i), U(x_i) \\ \min((\nu(x_i), \leq), (c_{i,0}, \prec_{i,0})) & \text{if } L(x_i) < \nu(x_i) \leq U(x_i) \\ (c_{i,0}, \prec_{i,0}) & \text{if } \nu(x_i) > U(x_i) \end{cases} \\
(p_{0,i}, \subset_{0,i}) = & \begin{cases} (-\nu(x_i), \leq) & \text{if } \nu(x_i) \leq L(x_i), U(x_i) \\ \min((c_{0,i}, \prec_{0,i}), (-\nu(x_i), \leq)) & \text{if } U(x_i) < \nu(x_i) \leq L(x_i) \\ \min((c_{0,i}, \prec_{0,i}), (-L(x_i), <)) & \text{if } \nu(x_i) > L(x_i) \end{cases} \\
(p_{i,j}, \subset_{i,j}) = & (c_{i,j}, \prec_{i,j}) \quad \text{if } i, j \neq 0
\end{array}
$$

We need to prove that P_ν is non-empty. If it is not the case, it means that there is a negative cycle in any DBM representing P_ν. As the above DBM only differs from D (which is non-empty and in normal form) by coefficients $(i, 0)$ and $(0, i)$ (for all i's), we get that there exist some i and j (with potentially $i = j$) such that:

$$(p_{i,0}, \subset_{i,0}) + (p_{0,j}, \subset_{0,j}) + (c_{j,i}, \prec_{j,i}) < (0, \leq) \tag{7.1}$$

We want to prove that this is not possible. We have to distinguish several cases, depending on the values of $p_{i,0}$ and $p_{0,j}$.

Case $(p_{i,0}, \subset_{i,0}) = (c_{i,0}, \prec_{i,0})$ (hyp not used).

We can simplify inequality (7.1) by applying the triangular inequality and we get that

$$(c_{j,0}, \prec_{j,0}) + (p_{0,j}, \subset_{0,j}) < (0, \leq)$$

1. *Case* $(p_{0,j}, \subset_{0,j}) = (c_{0,j}, \prec_{0,j})$.
   In this case, we get
   $$(c_{j,0}, \prec_{j,0}) + (c_{0,j}, \prec_{0,j}) < (0, \leq)$$
   which implies that D is empty. Contradiction.
2. *Case* $(p_{0,j}, \subset_{0,j}) = (-\nu(x_j), \prec)$. (hyp: $\nu(x_j) \leq L(x_j)$)
   We then get that
   $$(c_{j,0}, \prec_{j,0}) + (-\nu(x_j), \leq) < (0, \leq)$$
   which implies that $\nu(x_j) \not\prec_{j,0} c_{j,0}$. In particular we have, $(c'_{j,0}, \prec'_{j,0}) > (c_{j,0}, \prec_{j,0})$, which is possible only if $c_{j,0} > L(x_j)$ (see Lemma 7.6). However, in this case, we have that $\nu(x_j) \leq L(x_j)$, which is a contradiction.
3. *Case* $(p_{0,j}, \subset_{0,j}) = (-L(x_j), <)$. (hyp: $(-L(x_j), <) \leq (c_{0,j}, \prec_{0,j})$ and $\nu(x_j) > L(x_j)$)
   We get that
   $$(c_{j,0}, \prec_{j,0}) + (-L(x_j), <) < (0, \leq)$$

and thus that

$$c_{j,0} \leq L(x_j) < \nu(x_j)$$

As $c_{j,0} \leq L(x_j)$, we get that $(c'_{j,0}, \prec'_{j,0}) = (c_{j,0}, \prec_{j,0})$ and thus there is a contradiction (because $\nu(x_j) \prec'_{j,0} c'_{j,0}$).

Case $(p_{i,0}, \subset_{i,0}) = (\nu(x_i), \leq)$ (hyp: $\nu(x_i) \leq U(x_i)$).

In this case, we get that

$$(\nu(x_i), \leq) + (p_{0,j}, \subset_{0,j}) + (c_{j,i}, \prec_{j,i}) < (0, \leq)$$

1. *Case* $(p_{0,j}, \subset_{0,j}) = (c_{0,j}, \prec_{0,j})$. (hyp not used)
   Using the triangular inequality, we can simplify and we get
   $$(\nu(x_i), \leq) + (c_{0,i}, \prec_{0,i}) < (0, \leq)$$
   If $(c_{0,i}, \prec_{0,i}) = (c'_{0,i}, \prec'_{0,i})$, this is not possible. Otherwise, $-c_{0,i} > U(x_i)$ and $(c'_{0,i}, \prec'_{0,i}) = (-U(x_i), <)$ (see Lemma 7.7). Thus $\nu(x_i) > U(x_i)$, which is indeed a contradiction.
2. *Case* $(p_{0,j}, \subset_{0,j}) = (-\nu(x_j), \leq)$. (hyp: $\nu(x_j) \leq L(x_j)$)
   We get that
   $$(\nu(x_i) - \nu(x_j), \leq) + (c_{j,i}, \prec_{j,i}) < (0, \leq)$$
   If $(c'_{j,i}, \prec'_{j,i}) = (c_{j,i}. \prec_{j,i})$, then this is not possible. Thus, $\infty = (c'_{j,i}, \prec'_{j,i}) > (c_{j,i}, \prec_{j,i})$ and there are three cases:
   - $c_{j,i} > L(x_j)$, thus $(c_{j,i}, \prec_{j,i}) > (\nu(x_j), \leq)$ which implies that $(\nu(x_i), \leq) < (0, \leq)$, impossible. We thus assume that $c_{j,i} \leq L(x_j)$.
   - $-c_{0,j} > L(x_j)$, thus $-c_{0,j} > \nu(x_j)$. From Lemma 7.8 (point *2.*), this is not possible (as $\nu(x_j) \leq L(x_j)$) if $\nu(x_j) \leq U(x_j)$. Assume thus that $\nu(x_j) > U(x_j)$. In this case, $-\nu(x_j) \leq c_{0,j}$, *i.e.* $\nu(x_j) \geq -c_{0,j}$, which leads to a contradiction.
   - $-c_{0,i} > U(x_i)$, contradiction with $\nu(x_i) \leq U(x_i)$
3. *Case* $(p_{0,j}, \subset_{0,j}) = (-L(x_j), <)$. (hyp: $\nu(x_j) > L(x_j)$ and $(c_{0,j}, \prec_{0,j}) \geq (-L(x_j), <)$)
   We get that
   $$(\nu(x_i), \leq) + (-L(x_j), <) + (c_{j,i}, \prec_{j,i}) < (0, \leq)$$
   If $(c_{j,i}, \prec_{j,i}) > (L(x_j), <)$, then we get $(\nu(x_i), \leq) < (0, \leq)$ which is not possible. Assume now that $(c_{j,i}, \prec_{j,i}) \leq (L(x_j), <)$. If $(c'_{j,i}, \prec'_{j,i}) = (c_{j,i}, \prec_{j,i})$, then
   $$(\nu(x_i) - \nu(x_j), \leq) + (c'_{j,i}, \prec'_{j,i}) < (0, \leq)$$
   This is not possible. The only possibility is thus to have $\infty = (c'_{j,i}, \prec'_{j,i}) > (c_{j,i}. \prec_{j,i})$. Can we have $-c_{0,j} > L(x_j)$ or $-c_{0,i} > U(x_i)$? By hypothesis, the first case is not possible. If $-c_{0,i} > U(x_i)$, then $(c'_{0,i}, \prec'_{0,i}) = (-U(x_i), <)$ which contradicts the fact that $\nu(x_i) \leq U(x_i)$.

In all cases, there is a contradiction with inequality 7.1, P_ν is thus non-empty. This concludes the proof. □

7.4 Experiments

We have implemented a prototype of a location based variant of the $Extra^{+}_{LU}$ operator in Uppaal 3.4.2. Uppaal supports networks of communicating timed automata. Maximum lower and upper bounds for clocks are found for each automaton using a simple fixed point iteration. Given a location vector, the maximum lower and upper bounds are found by taking the maximum of the bounds in each location, similar to the approach taken in [20]. For storing visited states, we rely on the *minimal constraint form* representation of a zone described in [129], which does not store $+\infty$ entries.

As expected, experiments with the model in Figure 7.1 show that with LU extrapolation, the computation time for building the complete reachable state space does not depend on the value of the constants, whereas the computation time grows with the constant when using the classical extrapolation. We have also performed experiments with models of various instances of the CSMA/CD protocol and Fischer's protocol for mutual exclusion. Finally, experiments using a number of industrial case studies were made. For each model, Uppaal was run with four different options: (-n1) classic non-location based extrapolation (without active clock reduction), (-n2) classic location based extrapolation (active clock reduction is a side-effect of this), (-n3) LU location based extrapolation, and (-A) classic location based extrapolation with convex-hull approximation. In all experiments the minimal constraint form for zone representation was used [129] and the complete state space was generated. All experiments were performed on a 1.8GHz Pentium 4 running Linux 2.4.22, and experiments were limited to 15 minutes of CPU time and 470MB of memory. The results can be seen in Table 7.1.

Looking at the table, we see that for both Fischer's protocol for mutual exclusion and the CSMA/CD protocol, Uppaal scales considerably better with the LU extrapolation operator. Comparing it with the convex hull approximation (which is an over-approximation), we see that for these models, the LU extrapolation operator comes close to the same speed, although it still generates more states. Also notice that the runs with the LU extrapolation operator use less memory than convex hull approximation, due to the fact that in the latter case DBMs are used to represent the convex hull of the zones involved (in contrast to using the minimal constraint form of [129]). For the three industrial examples, the speedup is less dramatic. These models have a more complex control structure and thus little can be gained from changing the extrapolation operator. This is supported by the fact that also the convex hull technique fails to give any significant speedup (in the last example it even degrades performance). During the course of our experiments we also encountered examples where the LU extrapolation operator does not make any difference: the token ring FDDI protocol and the B&O protocols found on the Uppaal website[1] are among these.

[1] `http://www.uppaal.com`

	-n1			-n2			-n3			-A		
Model	Time	States	Mem	Time	States	Mem	Time	States	Mem	Time	States	Mem
f5	4.02	82,685	5	0.24	16,980	3	0.03	2,870	3	0.03	3,650	3
f6	597.04	1,489,230	49	6.67	158,220	7	0.11	11,484	3	0.10	14,658	3
f7				352.67	1,620,542	46	0.47	44,142	3	0.45	56,252	5
f8							2.11	164,528	6	2.08	208,744	12
f9							8.76	598,662	19	9.11	754,974	39
f10							37.26	2,136,980	68	39.13	2,676,150	143
f11							152.44	7,510,382	268			
c5	0.55	27,174	3	0.14	10,569	3	0.02	2,027	3	0.03	1,651	3
c6	19.39	287,109	11	3.63	87,977	5	0.10	6,296	3	0.06	4,986	3
c7				195.35	813,924	29	0.28	18,205	3	0.22	14,101	4
c8							0.98	50,058	5	0.66	38,060	7
c9							2.90	132,623	12	1.89	99,215	17
c10							8.42	341,452	29	5.48	251,758	49
c11							24.13	859,265	76	15.66	625,225	138
c12							68.20	2,122,286	202	43.10	1,525,536	394
bus	102.28	6,727,443	303	66.54	4,620,666	254	62.01	4,317,920	246	45.08	3,826,742	324
philips	0.16	12,823	3	0.09	6,763	3	0.09	6,599	3	0.07	5,992	3
sched	17.01	929,726	76	15.09	700,917	58	12.85	619,351	52	55.41	3,636,576	427

Table 7.1: Results for Fischer protocol (f), CSMA/CD (c), a model of a buscoupler, the Philips Audio protocol, and a model of a 5 task fixed-priority preemptive scheduler. -n1 is with classical maximum bounds extrapolation, -n2 is with location based maximum bounds extrapolation, -n3 is with location based LU extrapolation, and -A is with convex hull over-approximation. Times are in seconds, states are the number of generated states and memory usage is in MB.

7.5 Related Work

Extrapolation

The classical abstraction technique for timed automata takes into account the maximum constants to which the various clocks are compared. This technique — termed extrapolation or normalization — was described by Daws and Tripakis [67] and rigorously proved by Bouyer [35]. Most of the zone based abstractions (including the ones examined in this chapter) require that guards are restricted to conjunctions of simple lower or upper bounds (strict or non-strict) on individual clocks. Thus, constraints on clock differences are generally not allowed. Bouyer [35] gives more details about intricacies of zone based abstractions in the presence of difference constraints. Bengtsson and Yi [25] provide a solution for this case which is based on zone splitting.

Uppaal

Uppaal [7] is an integrated tool environment for modeling, validation and verification of real-time systems modeled as networks of timed automata, extended with data types (bounded integers, arrays, etc.). It is a state-of-the-art tool supporting temporal logic verification, storage reductions [129, 23] and other reduction techniques. There are many extension of the tool, e.g., Uppaal Cora for cost optimal reachability analysis, Times for schedulability analysis and synthesis of schedules, or Uppaal Tron for testing of real-time systems. The basic search algorithm of Uppaal is based on zones. The technique presented in this chapter is applicable even in these extensions of the Uppaal tool.

Improvements of Zone Based Approach

The basic extrapolation technique has been extended before. Daws and Tripakis [67] describe a number of additional abstraction techniques including active clock reduction and the over-approximating convex hull abstraction. Behrmann et al. [20] use approach in which the maximum constraints used in the abstraction do not only depend on the clocks but also on the particular locations of the timed automata. They show that active clock reduction is obtained as a special case of the location-dependent abstraction. In this paper we compare the performance of our new (exact) abstraction technique to that of convex hull approximation.

Successor Computation

The computation of successors over DBMs is a nontrivial operation. It requires several steps: intersection with guards, reseting clocks, elapse of time, extrapolation operation, and canonization (computation of normal form). The most time consuming is the canonization function which has complexity cubic with respect to the number of clocks. When applying guard, resetting clocks, or computing the delay

successors, the normal form can be recomputed much more efficiently [162] — this leads to acceleration of the successor computation. In [22] we show, that by distinguishing lower and upper bounded clocks, we can accelerate the computation even further.

Jobshop Scheduling

The jobshop scheduling problem consists in scheduling a finite set of jobs on a finite set of machines under given restrictions (two jobs can not use the same machine at the same time) such that all jobs are completed in the shortest amount of time. As shown by Abdeddaim and Maler [2], jobs can be represented by simple timed automata and the jobshop scheduling problem can be encoded as a (minimal cost) reachability. Since obtained timed automata are quite special, [2] proposed a specialized technique, called domination test, for accelerating the computation. In [22] we show that inclusion checking with the new LU-extrapolation algorithm is more general than this domination test technique.

Chapter 8

Conclusions

"Anyhow," he said, "it is nearly Luncheon Time." [139]

The final chapter provides a summary of the thesis and its contribution, some critique of the work and several directions for future work.

8.1 Summary

The thesis is concerned with formal verification of computer systems, particularly with the model checking method. Techniques studied in the thesis are geared towards an application in embedded system verification — our main interest lies in techniques suitable for verification of systems with software and real-time aspects.

The basic aim of our effort is to alleviate the effect of state space explosion. To this end, we study different techniques that try to reduce the size of the state space, particularly abstraction and reduction techniques. Although we study techniques of different flavour, all of them are connected by three main themes:

– Equivalences. Equivalences are used to study the effect of reductions and abstractions and to formalize their correctness. The main equivalences that are employed are reachability equivalence, simulation equivalence, and bisimulation.
– Abstraction. Abstraction serves in different ways as a powerful reduction technique. We work with abstraction functions both on the level of states (i.e., for a given concrete state a function produces an abstract state) and on the level of transition systems (i.e., for a given concrete transition system a function produces an abstract transition system). In this way, abstraction can be employed both to discuss theoretical reasoning about techniques and their correctness and to describe the realization of reduction techniques.
– Approximation and refinement. In order to reduce the size of a state space significantly, we often end up with (only) approximation of the full state space. Approximations are often sufficient to find errors or prove properties. If the approximation is not good enough, an iterative refinement is used to get better approximations.

In Chapter 2 we describe a simple Lego© Mindstorms system for sorting bricks. Let us review the content of the thesis on this example and discuss how different parts of the thesis could be useful for verification of (more realistic version of) such a system.

Chapter 3 is concerned with on-the-fly reductions. These reductions aim at reducing the size of the state space during the exploration by omitting some parts of the state space. We preform the reduction in such a way that the reduced structure is equivalent to the full state space. For the Sorter model, this technique can eliminate redundancy introduced by the symmetry of bricks. If we had considered more realistic model of the system, it would be useful to employ partial order reduction, which reduces some interleaving due to concurrency[1].

Chapter 4 is concerned with predicate abstraction and refinement techniques. These techniques would be particularly useful if we considered a more realistic version of the Sorter with a more complex control program. Predicate abstraction techniques are suitable for abstracting away (data) aspects of the program which do not directly influence the critical behaviour of the system. For example, the `PlaySound` function or the exact value of `request` variable are not important in the Sorter example. The key feature of these techniques is that the abstraction is performed automatically using an automatic refinement.

Chapter 5 aims at similar goals as Chapter 4. The main difference is that instead of over-approximations, which are used and refined in Chapter 4, we use under-approximations and their refinement. This approach is more suitable for falsification rather then verification. Thus the approach would be particularly useful during the construction of the Sorter system, as a bridge between usual debugging techniques and classical verification techniques.

Chapter 6 turns to real-time aspects of embedded systems and studies the nature of time, particularly the relation between dense time and sampled time. In Chapter 2, we provide two models of the Sorter system: one with sampled time, another with dense time. This demonstrates the usefulness of understanding the relations between these two semantics. The chapter also studies different non-emptiness problems. Non-emptiness problems form a fundamental part of the verification process.

Chapter 7 is also concerned with real-time features of systems. It is more specific and practical then Chapter 6 — it focuses on reduction technique for a particular algorithm for timed automata state space exploration. The technique is based on the observation that practical examples often use clocks in restricted ways. For example, the `Timer` clock in the Sorter is compared only to lower bounds, i.e., we never check whether `Timer` is smaller than some bound. This observation enables us to do an abstraction step during the exploration more aggressively and thus to obtain smaller state space.

8.2 Contribution and Critique

In the review of contributions of the thesis, we also consider a possible critique and provide arguments to defend the thesis.

[1] In our simple model we use token-based synchronization so there is no interleaving.

The most important contribution of the thesis lies in the under-approximation refinement algorithm for software verification. The algorithm presents a novel approach to application of predicate abstraction in software verification. We prove the correctness of the approach and discuss formal termination properties.

At the moment, it is not completely clear whether the approach will be useful for realistic programs — the approach is demonstrated only by small examples and the implementation for real programming languages is not straightforward. Although we believe that the approach will scale, we have to admit that at the moment we are not able to demonstrate it. An engineering effort required to address the implementation issues is well beyond the scope of the thesis. Tools based on over-approximation refinement, despite being intensively developed during last years, are also still applied to rather small examples.

Moreover, the contribution of this approach is also in the attitude to the problem: it provides a new point of view on refinement algorithms. Maybe the proposed algorithm will not turn out to be the practically most useful one, but it can inspire another work based on under-approximation refinement. It also suggests new possibilities for combining under- and over-approximations (as we discuss below).

The thesis also contains several important technical results. The most interesting one is the decidability of the non-emptiness problem of timed automata ω-language in sampled semantics with an unknown sampling period. A straightforward objection is that this problem is so specialized and complicatedly formulated that nobody is really interested in it. There are, however, quite good reasons why to study this problem:

- This problem can be more practically relevant than another, more simply formulated non-emptiness problems. The formulation of this problem overcomes difficulties of both dense time semantics (the problem of non-realizable runs) and classical sampled semantics (the problem of determining a fixed sampling period).
- This problem was formulated before by Alur and Madhusudan [5] who provided wrong classification of the problem.
- The decidability proof is interesting. Particularly, we provide an interesting characterization of reachability relations between valuations. This characterization can be useful in other contexts as well.

The contribution of the thesis with the most straightforward practical outcome is the novel extrapolation technique for zone based exploration algorithm for timed automata. This technique is based on distinguishing between lower and upper bounds in guards. Although the provided experimental results for this technique are not very thorough, the new technique clearly improves on previously used extrapolation techniques: in some cases it brings clear improvement and at the same time it does not bring any additional overhead. This alone justifies its usage in the model checker implementation.

Finally, the thesis provides overviews of on-the-fly reduction techniques, predicate abstraction techniques, and results about dense and sampled semantics of timed

automata. The presented results are collected from many different sources which use different notations. Our overviews are presented in a single framework and fill in some missing results. This presentation greatly facilitates understanding of the results and enables better comparison. For on-the-fly reductions we also provide an experimental evaluation.

However, for the predicate abstraction the experimental comparison is absent and even for on-the-fly reductions it would be useful to have a more thorough comparison, particularly with models taken from "real users" and not only "polished" models from academics.

This objection is definitely pertinent. Reasons why the thesis does not contain more experimental results are practical: it would be very difficult to perform such experiments at the time. Examples used in the evaluation of on-the-fly reductions are state-of-the-art examples. Compared to usual experimental works in the model checking field, the set of used examples is rather large. Concerning predicate abstraction techniques, here the comparison is even more problematical. At first, the usefulness of these techniques is based on a significant amount of an engineering work (heuristics, implementation details, etc.). It would be very difficult to implement different techniques in a single setting. Secondly, interesting benchmark examples are hard to find — even practically oriented papers on predicate abstraction usually contain experiments on only few classical examples.

To conclude, we believe that the thesis present several interesting contributions. Some of the results need to be further extended, improved, or evaluated, but this work is beyond the scope of the thesis and we leave it as a future work.

8.3 Future Work

There exist many possible direction for future work, some of them are directly proposed by the above given critique. Here we discuss the most interesting ones.

Evaluation of Under-approximation Refinement for Finite State Systems

In Chapter 3 we briefly outline possible under-approximation refinement approaches for finite state systems (e.g., based on non-exact abstraction functions or on approximate techniques for reduction of interleaving). The merit of these approaches is, at the moment, disputable. From theoretical point of view, we cannot say much about these techniques — in the worst case they need to do the same work as exact techniques. These techniques can be advantageous only for error detection. The question is, whether they are really advantageous, i.e., whether we can find interesting errors in early iterations of the refinement algorithm.

As we argue above, it is difficult to perform testifying evaluation. For evaluation of these techniques we need models with realistic, non-trivial errors. At the moment, there is quite a large number of finite state systems case studies. But most of these

case studies present only a final, correct model. For the evaluation of error-detecting techniques we need semi-finished models with errors.

Evaluations over Non-expert Users' Models

In Chapter 3 we argue that there can be a significant difference between models created by experts and models created by non-expert users. Since model checking aims at automatization of the verification process it should be applicable by non-expert users. Unfortunately, most of the currently performed experimental evaluation is done on models crafted by researches. It is important to create benchmarks of examples created by ordinary users and evaluate techniques (particularly reduction techniques) over these models.

Under-approximation Refinement for Full Programming Languages

One of the straightforward directions for further work is the extension of the under-approximation refinement algorithm given in Chapter 5 to full programming languages. This represents several nontrivial engineering challenges, for example: dealing with pointers and complex data structures, heuristics for optimization of theorem prover calls, heuristics for guiding of the refinement. Note that many of these issues are very similar to those encountered in implementation of over-approximation refinement techniques and we expect that they can be solved in similar manner. There are, however, certain issues which are specific for this approach and which need to be solved (e.g., guiding of the refinement).

Combination of Over- and Under-Approximations

Up to now, the combination of over- and under-approximations based on predicate abstraction has been advocated only in theoretical works as a technique for proving general properties (involving both universal and existential quantification) [92, 13]. In practice, the combination has not been exploited to a larger extent. This is caused by impracticality of must-abstractions[2]. We believe that the new under-approximation based on αSearchCheck is more practical.

This opens possibilities of practical combinations of over- and under-approximations. Such a combination has several plausible advantages. These techniques are complementary, each has its own strength: over-approximation based techniques are better for proving programs correct, whereas under-approximation based techniques are better for finding errors. By using combination, we can get earlier termination. In the case that the search does not terminate, we can give user more information. By having both over- and under-approximation it should also be possible to better choose predicates for the refinement.

[2]Note that whereas there exist several tools based on may-abstractions [14, 104, 52], there is none based on must-abstractions.

Heuristic Search for a Right Abstraction

Software model checking consists of two main steps: searching for a right abstraction and searching in the state space of the abstract model. Whereas the second step is automatic and quite optimized, the first step is still mainly manual.

Current refinement algorithms perform automated search for abstraction, but only in a limited fashion. They start with a set of predicates and keep on adding new predicates according to some strategy — so the search is one-way, linear. It may be useful to view the search for a right abstraction as a standalone problem and consider also other approaches then a one-way search.

Let us consider a 'meta state space' of abstractions. Each node in this meta state space is given by a tuple (A, Φ) where A is an approximation technique based on predicate abstraction and Φ is a set of predicates. Each node in this meta state space defines an approximation of an input program. Given an input program P and a property φ, the goal of the meta-search is to find a tuple (A, Φ) such that the corresponding approximation produced by (A, Φ) is sufficient to determine whether P satisfy φ.

Since we are dealing with undecidable problems, we cannot hope to get some deterministic optimal algorithm for the meta-search. So the meta-search is a heuristic search and we should concentrate on finding practical heuristics for guiding this search.

The meta-search setting lends itself to a distributed solution. Each tuple (A, Φ) defines a stand-alone, non-trivial problem — generation of the approximation and checking of the property over this approximation. These problems can be easily distributed among many workstations. The manager coordinates the meta-search — it collects answers and determines the strategy, i.e., it decides which approximations should be computed. This use of distribution can be much more efficient than the usual use of a distributed computation in model checking, when several workstation compute together a single state space and thus have to communicate intensively.

Algorithms for the Non-emptiness Problems with an Unknown Period

We provide decidability result for the ω-language non-emptiness problem in sampled semantics with an unknown sampling period and we give some arguments, why this problem can be practically important. So the natural questions to ask are:

– What is the complexity of the problem?
– Is there any practically usable algorithm for the problem?

Distinguishing Lower/Upper Bounds in Timed Automata Verification

In Chapter 7 we show that distinguishing lower and upper bounds can be beneficial for zone based algorithms for timed automata verification. There exist several other approaches to timed automata verification which are not based on the use of zones, e.g., symbolic representation of sets of valuations using decision diagrams (BDDs

and similar structures) or SAT techniques based on bounded model checking ideas. An interesting question is whether these other approaches can also benefit from distinguishing between lower and upper bounds.

Bibliography

[1] FDIV replacement program. Intel White Paper, November 1994.

[2] Y. Abdeddaim and O. Maler. Job-shop scheduling using timed automata. In *Proc. of Computer Aided Verification (CAV 2001)*, volume 2102 of *LNCS*, pages 478–492. Springer, 2001.

[3] Y. Abdeddaïm and O. Maler. Preemptive job-shop scheduling using stopwatch automata. In *Proc. of Tools and Algorithms for Construction and Analysis of Systems (TACAS02)*, volume 2280 of *LNCS*, pages 113–126. Springer, 2002.

[4] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[5] R. Alur and P. Madhusudan. Decision problems for timed automata: A survey. In *Formal Methods for the Design of Real-Time Systems*, volume 3185 of *LNCS*, pages 1–24. Springer, 2004.

[6] R. Alur and B.-Y. Wang. "Next" heuristic for on-the-fly model checking. In *International Conference on Concurrency Theory*, volume 1664 of *LNCS*, pages 98–113. Springer, 1999.

[7] T. Amnell, G. Behrmann, J. Bengtsson, P. R. D'Argenio, A. David, A. Fehnker, T. Hune, B. Jeannet, Kim g. Larsen, O. Mller, P. Pettersson, C. Weise, and W. Yi. Uppaal – now, next, and future. In *Proc. Modelling and Verification of Parallel Processes (Movep2k)*, volume 2067 of *LNCS*, pages 99–124. Springer, 2001.

[8] G. R. Andrews. *Concurrent Programming, Principles and Practice.* Addison-Wesley Publishing Company, 1991.

[9] E. Asarin, M. Bozga, A. Kerbrat, O. Maler, A. Pnueli, and A. Rasse. Data-structures for the verification of timed automata. In *Proc. of Hybrid and Real-Time Systems (HART'97)*, volume 1201 of *LNCS*, pages 346–360. Springer, 1997.

[10] E. Asarin, O. Maler, and A. Pnueli. On discretization of delays in timed automata and digital circuits. In *Proc. of Conference on Concurrency Theory (CONCUR'98)*, volume 1466 of *LNCS*, pages 470–484. Springer, 1998.

[11] T. Ball. A theory of predicate-complete test coverage and generation. In *Proc. Formal Methods for Components and Objects (FMCO 2004)*, volume 3188 of *LNCS*. Springer, 2004.

[12] T. Ball, B. Cook, S. K. Lahiri, and L. Zhang. Zapato: Automatic theorem proving for predicate abstraction refinement. In *Proc. of Computer Aided Verification (CAV'04)*, volume 3114 of *LNCS*, pages 457–461. Springer, 2004.

[13] T. Ball, O. Kupferman, and G. Yorsh. Abstraction for falsification. In *Proc. of Computer Aided Verification (CAV 2005)*, LNCS. Springer, 2005.

[14] T. Ball, R. Majumdar, T. D. Millstein, and S. K. Rajamani. Automatic predicate abstraction of c programs. In *Proc. of Programming Language Design and Implementation (PLDI 2001)*, pages 203–213. ACM Press, 2001.

[15] T. Ball, A. Podelski, and S. K. Rajamani. Boolean and cartesian abstraction for model checking C programs. In *Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2001)*, volume 2031 of *LNCS*, pages 268–283. Springer, 2001.

[16] T. Ball, A. Podelski, and S. K. Rajamani. Relative completeness of abstraction refinement for software model checking. In *Proc. of Tools and Algorithms for the Construction and Analysis of Systems (TACAS '02)*, number 2280 in LNCS, pages 158–172. Springer, 2002.

[17] T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In *Proc. of SPIN workshop*, volume 2057 of *LNCS*, pages 103–122. Springer, 2001.

[18] T. Basten, D. Bosnacki, and M.C.W. Geilen. Cluster-based partial-order reduction. *Automated Software Engineering*, 11(4):365–402, 2004.

[19] J. Baumgartner, T. Heyman, V. Singhal, and A. Aziz. Model checking the IBM gigahertz processor: An abstraction algorithm for high-performance netlists. In *Proc. of Computer Aided Verification (CAV 1999)*, volume 1633 of *LNCS*, pages 72–83. Springer, 1999.

[20] G. Behrmann, P. Bouyer, E. Fleury, and Kim G. Larsen. Static guard analysis in timed automata verification. In *Proc. of Tools and Algorithms for the Construction and Analysis of Systems (TACAS'2003)*, volume 2619 of *LNCS*, pages 254–277. Springer, 2003.

[21] G. Behrmann, P. Bouyer, K. G. Larsen, and R. Pelánek. Lower and upper bounds in zone based abstractions of timed automata. In *Proc. of Tools and Algorithms for Construction and Analysis of Systems (TACAS 2004)*, volume 2988 of *LNCS*, pages 312–326. Springer, 2004.

[22] G. Behrmann, P. Bouyer, K. G. Larsen, and R. Pelánek. Lower and upper bounds in zone-based abstractions of timed automata. *International Journal on Software Tools for Technology Transfer*, 2005.

[23] G. Behrmann, K. G. Larsen, and R. Pelánek. To store or not to store. In *Proc. of Computer Aided Verification (CAV 2003)*, volume 2725 of *LNCS*. Springer, 2003.

[24] J. Bengtsson. *Clocks, DBMs ans States in Timed Systems*. PhD thesis, Department of Information Technology, Uppsala University, Uppsala, Sweden, 2002.

[25] J. Bengtsson and W. Yi. On clock difference constraints and termination in reachability analysis of timed automata. In *Proc. of International Conference on Formal Engineering Methods (ICFEM'03)*, number 2885 in LNCS. Springer, 2003.

[26] C. Bernardeschi, A. Fantechi, S. Gnesi, S. Larosa, G. Mongardi, and D. Romano. A formal verification environment for railway signaling system design. *Formal Methods in System Design: An International Journal*, 12(2):139–161, March 1998.

[27] D. Beyer. Improvements in BDD-based reachability analysis of timed automata. In *Proc. of Formal Methods Europe (FME 2001)*, volume 2021 of *LNCS*, pages 318–343. Springer, 2001.

[28] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Proc. of Tools and Algorithms for Construction and Analysis of Systems (TACAS 1999)*, volume 1579 of *LNSC*, pages 193–207. Springer, 1999.

[29] S. Blom and J. van de Pol. State space reduction by proving confluence. In *Proc. of Computer Aided Verification (CAV 2002)*, number 2404 in LNCS, pages 596–609. Springer, 2002.

[30] B. Boigelot and P. Godefroid. Symbolic verification of communication protocols with infinite state spaces using QDDs. In *Proc. of Computer Aided Verification (CAV 1996)*, volume 1102 of *LNCS*, pages 1–12. Springer, 1996.

[31] B. Boigelot and P. Wolper. Symbolic verification with periodic sets. In *Proc. of Computer Aided Verification (CAV 1994)*, volume 818 of *LNCS*, pages 55–67. Springer, 1994.

[32] D. Bosnacki. Digitization of timed automata. In *Proc. of Formal Methods for Industrial Critical Systems (FMICS'99)*, pages 283–302, 1999.

[33] D. Bosnacki, D. Dams, and L. Holenderski. Symmetric spin. In *Proc. of SPIN Workshop*, volume 1885 of *LNCS*, pages 1–19. Springer, 2000.

[34] P. Bouyer. Timed automata may cause some troubles. Research Report LSV–02–9, Laboratoire Specification et Verification, ENS de Cachan, France, 2002.

[35] P. Bouyer. Untameable timed automata! In *Proc. of Symposium on Theoretical Aspects of Computer Science (STACS'03)*, volume 2607 of *LNCS*, pages 620–631. Springer, 2003.

[36] P. Bouyer, C. Dufourd, E. Fleury, and A. Petit. Are timed automata updatable? In *Proc. of Computer Aided Verification (CAV 2000)*, volume 1855 of *LNCS*, pages 464–479. Springer, 2000.

[37] M. Bozga, C. Daws, O. Maler, A. Olivero, S. Tripakis, and S. Yovine. Kronos: a model-checking tool for real-time systems. In *Proc. of Computer Aided Verification (CAV'98)*, volume 1427 of *LNCS*, pages 546–550. Springer, 1998.

[38] M. Bozga, O. Maler, A. Pnueli, and S. Yovine. Some Progress in the Symbolic Verification of Timed Automata. In *Proc. of Computer Aided Verification (CAV'97)*, volume 1254 of *LNCS*, pages 179–190. Springer, 1997.

[39] M. Bozga, O. Maler, and S. Tripakis. Efficient verification of timed automata using dense and discrete time semantics. In *Proc. of Correct Hardware Design and Verification Methods (CHARME'99)*, volume 1703 of *LNCS*, pages 125–141. Springer, 1999.

[40] L. Brim, I. Černá, P. Krčál, and R. Pelánek. Distributed LTL model checking based on negative cycle detection. In *Proc. Foundations of Software Technology*

and Theoretical Computer Science (FST TCS 2001), volume 2245 of *LNCS*, pages 96–107. Springer, 2001.

[41] L. Brim, I. Černá, P. Krčál, and R. Pelánek. How to employ reverse search in distributed single-source shortest paths. In *Proc. SOFSEM'01*, volume 2234 of *LNCS*, pages 191–200. Springer, 2001.

[42] L. Brim, I. Černá, P. Moravec, and J. Šimša. Under-approximation generation using partial order reduction. Technical Report FIMU-RS-2005-04, Masaryk University Brno, 2005.

[43] M. C. Browne, E. M. Clarke, and O. Grumberg. Characterizing finite Kripke structures in propositional temporal logic. *Theor. Comput. Sci.*, 59(1-2):115–131, 1988.

[44] T. Bultan, R. Gerber, and W. Pugh. Symbolic model checking of infinite state systems using presburger arithmetic. In *Proc. of Computer Aided Verification (CAV'97)*, volume 1254 of *LNCS*, pages 400–411. Springer, 1997.

[45] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, 1992.

[46] R. Cardell-Oliver. Conformance test experiments for distributed real-time systems. In *ISSTA '02: Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, pages 159–163, New York, NY, USA, 2002. ACM Press.

[47] L. Carroll. *Alice in Wonderland and Through the Looking Glass*. 1865.

[48] F. Cassez, T. A. Henzinger, and J.-F. Raskin. A comparison of control problems for timed and hybrid systems. In *Proc. of the Hybrid Systems: Computation and Control*, volume 2289 of *LNCS*, pages 134–148. Springer, 2002.

[49] F. Cassez and K. G. Larsen. The impressive power of stopwatches. In *Proc. of Conference on Concurrency Theory (CONCUR 2000)*, number 1877 in LNCS, pages 138–152. Springer, 2000.

[50] I. Černá and R. Pelánek. Distributed explicit fair cycle detection. In *Proc. SPIN workshop*, volume 2648 of *LNCS*, pages 49–73. Springer, 2003.

[51] I. Černá and R. Pelánek. Relating hierarchy of temporal properties to model checking. In *Proc. of Mathematical Foundations of Computer Science (MFCS 2003)*, volume 2747 of *LNCS*, pages 318–327. Springer, 2003.

[52] S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. *ACM Trans. Computer Systems*, 30(6):388–402, 2004.

[53] S. Chaki, E. Clarke, A. Groce, and O. Strichman. Predicate abstraction with minimum predicates. In *Proc. Correct Hardware Design and Verification Methods (CHARME 2003)*, volume 2860 of *LNCS*. Springer, 2003.

[54] S. Christensen, L.M. Kristensen, and T. Mailund. A Sweep-Line Method for State Space Exploration. In *Proc. of Tools and Algorithms for Construction and Analysis of Systems (TACAS 2001)*, volume 2031 of *LNCS*, pages 450–464. Springer, 2001.

[55] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Proc. of Computer Aided Verification (CAV 2000)*, volume 1855 of *LNCS*, pages 154–169. Springer, 2000.

[56] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, 1999.

[57] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, 1994.

[58] E.M. Clarke, O. Grumberg, M. Minea, and D. Peled. State space reduction using partial order techniques. *International Journal on Software Tools for Technology Transfer (STTT)*, 2(3):279–287, November 1999.

[59] H. Comon and Y. Jurski. Timed automata and the theory of real numbers. In *Proc. of Conference on Concurrency Theory (CONCUR'99)*, volume 1664 of *LNCS*, pages 242–257. Springer, 1999.

[60] B. Cook, D. Kroening, and N. Sharygina. Cogent: Accurate theorem proving for program verification. In *Proc. of Computer Aided Verification (CAV'05)*, volume 3576 of *LNCS*, pages 296–300. Springer, 2005.

[61] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, Corina S. Păsăreanu, Robby, and H. Zheng. Bandera: extracting finite-state models from java source code. In *Proc. of International Conference on Software Engineering (ICSE 2000)*, pages 439–448. ACM Press, 2000.

[62] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. of Principles of Programs Languages (POPL 1977)*, pages 238–252, 1977.

[63] D. Dams. Comparing abstraction refinement algorithms. In *Proc. of Workshop on Software Model Checking*, volume 89(3) of *ENTCS*. Elsevier, 2003.

[64] D. Dams, R. Gerth, and O. Grumberg. Abstract interpretation of reactive systems. *ACM Trans. Program. Lang. Syst.*, 19(2):253–291, 1997.

[65] D. Dams and K. S. Namjoshi. The existence of finite abstractions for branching time model checking. In *Proc. of Logic in Computer Science (LICS 2004)*, pages 335–344. IEEE Computer Society, 2004.

[66] S. Das, D. L. Dill, and S. Park. Experience with predicate abstraction. In *Proc. of Computer Aided Verification (CAV '99)*, pages 160–171. Springer, 1999.

[67] C. Daws and S. Tripakis. Model-checking of real-time reachability properties using abstractions. In *Proc. of Tools and Algorithms for the Construction and Analysis of Systems (TACAS'98)*, volume 1384 of *LNCS*, pages 313–329. Springer, 1998.

[68] C. Daws and S. Yovine. Reducing the number of clock variables of timed automata. In *Proc. of Real-Time Systems Symposium (RTSS '96)*, pages 73–81. IEEE Computer Society, 1996.

[69] L. de Alfaro, P. Godefroid, and R. Jagadeesan. Three-valued abstractions of games: Uncertainty, but with precision. In *Proc. of Logic in Computer Science (LICS 2004)*, pages 170–179. IEEE Computer Society, 2004.

[70] L. de Alfaro, T. A. Henzinger, and F. Y. C. Mang. Detecting errors before reaching them. In *Proc. of Computer Aided Verification (CAV 2000)*, volume 1855 of *LNCS*, pages 186–201, 2000.

[71] D. L. Detlefs, K. R. M. Leino, G. Nelson, and J. B. Saxe. Extended static checking. *Research Report 159, Compaq Systems Research Center*, 1998.

[72] D. Dill. Timing assumptions and verification of finite-state concurrent systems. In *Proc. of Automatic Verification Methods for Finite State Systems*, volume 407 of *LNCS*, pages 197–212. Springer, 1989.

[73] C. Dima. Computing reachability relations in timed automata. In *Proc. of Symp. on Logic in Computer Science (LICS 2002)*. IEEE Computer Society Press, 2002.

[74] Y. Dong and C. R. Ramakrishnan. An optimizing compiler for efficient model checking. In *Proc. of Formal Description Techniques for Distributed Systems and Communication Protocols (FORTE XII) and Protocol Specification, Testing and Verification (PSTV XIX)*, pages 241–256. Kluwer, B.V., 1999.

[75] A. Dovier, R. Gentilini, C. Piazza, and A. Policriti. Rank-based symbolic bisimulation (and model checking). In *Proc. of Workshop on Logic, Language, Information and Computation (WoLLIC 2002)*, volume 67 of *ENTCS*. Elsevier, 2002.

[76] A. Dovier, C. Piazza, and A. Policriti. A fast bisimulation algorithm. In *Proc. of Computer Aided Verification (CAV'01)*, volume 2102 of *LNCS*, pages 79–90. Springer, 2001.

[77] B. Dutertre and V. Stavridou. Formal requirements analysis of an avionics control system. *Software Engineering*, 23(5):267–278, 1997.

[78] M. D. Ernst, A. Czeisler, W. G. Griswold, and D. Notkin. Quickly detecting relevant program invariants. In *Proc. 22nd International Conference on Software Engineering (ICSE'00)*, pages 449–458. ACM Press, 2000.

[79] J. C. Fernandez, M. Bozga, and L. Ghirvu. State space reduction based on live variables analysis. *Journal of Science of Computer Programming (SCP)*, 47(2-3):203–220, 2003.

[80] C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *Proc. of Principles of programming languages (POPL'05)*, pages 110–121. ACM Press, 2005.

[81] C. Flanagan and S. Qadeer. Thread-modular model checking. In *Proc. of SPIN Workshop*, volume 2648 of *LNCS*. Springer, 2003.

[82] H. Garavel, R. Mateescu, and I. Smarandache. Parallel state space construction for model-checking. In *Proc. SPIN Workshop*, volume 2057 of *LNCS*, pages 217–234. Springer, 2001.

[83] J. Geldenhuys. State caching reconsidered. In *Proc. of SPIN Workshop*, volume 2989 of *LNCS*, pages 23–39. Springer, 2004.

[84] J. Geldenhuys and P. J. A. de Villiers. Runtime efficient state compaction in SPIN. In *Proc. of SPIN Workshop*, volume 1680 of *LNCS*, pages 12–21. Springer, 1999.

[85] R. Gerth. Model checking if your life depends on it: A view from Intel's trenches. In *Proc. SPIN workshop*, volume 2057 of *LNCS*. Springer, 2001.

[86] R. Gerth, R. Kuiper, D. Peled, and W. Penczek. A partial order approach to branching time model checking. *Information and Computation*, 150(2):132–152, 1999.

[87] R. Giacobazzi and E. Quintarelli. Incompleteness, counterexamples and refinements in abstract model checking. In *Proc. 8th Static Analysis Symposium (SAS'01)*, volume 2126 of *LNCS*, pages 356–373. Springer, 2001.

[88] J. Gleick. A bug and a crash - sometimes a bug is more than a nuisance. *New York Times Magazine*, December 1996.

[89] P. Godefroid. *Partial-order methods for the verification of concurrent systems: an approach to the state-explosion problem*, volume 1032 of *LNCS*. Springer, 1996.

[90] P. Godefroid. Software model checking: The VeriSoft approach. Technical Memorandum ITD-03-44189G, Bell Labs, March 2004.

[91] P. Godefroid, G. J. Holzmann, and D. Pirottin. State space caching revisited. In *Proc. of Computer Aided Verification (CAV 1992)*, volume 663 of *LNCS*, pages 178–191. Springer, 1992.

[92] P. Godefroid, M. Huth, and R. Jagadeesan. Abstraction-based model checking using modal transition systems. In *Proc. of Conference on Concurrency Theory (CONCUR '01)*, pages 426–440. Springer, 2001.

[93] P. Godefroid and S. Khurshid. Exploring very large state spaces using genetic algorithms. In *Proc. of Tools and Algorithms for Construction and Analysis of Systems (TACAS 2002)*, volume 2280 of *LNCS*, pages 266–280. Springer, 2002.

[94] A. Gollu, A. Puri, and P. Varaiya. Discretization of timed automata. In *Proc. of Conference on Decision and Control*, pages 957–958. IEEE Computer Society, 1994.

[95] S. Graf and H. Saidi. Construction of abstract state graphs with pvs. In *Proc. of Computer Aided Verification (CAV 1997)*, pages 72–83. Springer, 1997.

[96] W. Grieskamp, Y. Gurevich, W. Schulte, and M. Veanes. Generating finite state machines from abstract state machines. In *Proc. of International symposium on Software testing and analysis (ISSTA'02)*, pages 112–122. ACM Press, 2002.

[97] A. Groce and W. Visser. Heuristics for model checking Java programs. *Software Tools for Technology Transfer (STTT)*, 6(4):260–276, 2004.

[98] O. Grumberg, F. Lerda, O. Strichman, and M. Theobald. Proof-guided underapproximation-widening for multi-process systems. In *Proc. of Principles of programming languages (POPL '05)*, pages 122–131, New York, NY, USA, 2005. ACM Press.

[99] O. Grumberg and D. E. Long. Model checking and modular verification. *ACM Transactions on Programming Languages and Systems*, 16(3):843–871, 1994.

[100] P. Haslum. Model checking by random walk. In *Proc. of ECSEL Workshop*, 1999.

[101] J. Hatcliff, M. B. Dwyer, and H. Zheng. Slicing software for model construction. *Higher-Order and Symbolic Computation*, 13(4):315–353, 2000.

[102] M. Hendriks and K. G. Larsen. Exact acceleration of real-time model checking. In *Workshop on Theory and Practice of Timed Systems (TPTS'02)*, volume 65 of *ENTCS*. Elsevier, 2002.

[103] T. A. Henzinger, R. Jhala, R. Majumdar, and K. McMillan. Abstractions from proofs. In *Proc. 31st Symposium on Principles of Programming Languages (POPL'04)*, pages 232–244. ACM Press, 2004.

[104] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Proc. of Principles of Programming Languages (POPL 2002)*, pages pp. 58–70. ACM Press, 2002.

[105] T. A. Henzinger, P. W. Kopke, A. Puri, and P. Varaiya. What's decidable about hybrid automata? In *Proc. of ACM symposium on Theory of computing (STOC'95)*, pages 373–382. ACM Press, 1995.

[106] T. A. Henzinger, Z. Manna, and A. Pnueli. What good are digital clocks? In *Proc. of Colloquium on Automata, Languages and Programming (ICALP'92)*, volume 623 of *LNCS*, pages 545–558. Springer, 1992.

[107] G. J. Holzmann. Algorithms for automated protocol verification. *AT&T Technical Journal*, 69(2):32–44, 1990.

[108] G. J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, 1991.

[109] G. J. Holzmann. An analysis of bitstate hashing. In *Proc. of Protocol Specification, Testing, and Verification*, pages 301–314. Chapman & Hall, 1995.

[110] G. J. Holzmann. The engineering of a model checker: the GNU i-protocol case study revisited. In *Proc. of SPIN Workshop*, volume 1680 of *LNCS*. Springer, 1999.

[111] G. J. Holzmann, P. Godefroid, and D. Pirottin. Coverage preserving reduction strategies for reachability analysis. In *Proc. of Protocol Specification, Testing, and Verification*, 1992.

[112] G. J. Holzmann and D. Peled. An improvement in formal verification. In *Proc. of Formal Description Techniques VII*, pages 197–211. Chapman & Hall, Ltd., 1995.

[113] G. J. Holzmann and A. Puri. A minimized automaton representation of reachable states. *Software Tools for Technology Transfer (STTT)*, 3(1):270–278, 1998.

[114] G.J. Holzmann and R. Joshi. Model-driven software verification. In *Proc. of SPIN Workshop*, volume 2989 of *LNCS*, pages 77–92. Springer, 2004.

[115] R. Iosif. Symmetric model checking for object-based programs. Technical Report TR 2001-5, Kansas State University, 2001.

[116] C. N. Ip and D. L. Dill. Better verification through symmetry. *Formal Methods in System Design*, 9(1–2):41–75, 1996.

[117] T. K. Iversen, M. J. Kristoffersen, K. J., Larsen, K. G., M. Laursen, R. G. Madsen, S. K. Mortensen, P. Pettersson, and C. B. Thomasen. Model-checking

real-time control programs — Verifying Lego mindstorms systems using Uppaal. In *Proc. of 12*th *Euromicro Conference on Real-Time Systems*, pages 147–155. IEEE Computer Society Press, June 2000.

[118] H. Jain, F. Ivancic, A. Gupta, and M. K. Ganai. Localization and register sharing for predicate abstraction. In *Proc. of Tools and Algorithms for the Construction and Analysis of Systems (TACAS'05)*, volume 3440 of *LNCS*, pages 397–412. Springer, 2005.

[119] M.D. Jones and J.Sorber. Parallel random walk search for LTL violations. In *Proc. of Parallel and Distributed Model Checking (PDMC 2002)*, volume 68 of *ENTCS*, pages 156–161. Elsevier, 2002.

[120] J.P. Krimm and L. Mounier. Compositional state space generation from Lotos programs. In *Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS 1997)*, volume 1217 of *LNCS*, pages 239–258. Springer, 1997.

[121] P. Krčál and R. Pelánek. Reachability relations and sampled semantics of timed systems. Technical Report FIMU-RS-2005-09, Masaryk University Brno, 2005.

[122] P. Krčál and W. Yi. Decidable and undecidable problems in schedulability analysis using timed automata. In *Proc. of Tools and Algorithms for Construction and Analysis of Systems (TACAS'04)*, volume 2988 of *LNCS*, pages 236–250. Springer, 2004.

[123] A. Kuehlmann, K. L. McMillan, and R. K. Brayton. Probabilistic state space search. In *Proc. of Computer-Aided Design (CAD 1999)*, pages 574–579. IEEE Press, 1999.

[124] R. Kurshan, V. Levin, M. Minea, D. Peled, and H. Yenigun. Static partial order reduction. In *Proc. Tools and Algorithms for Construction and Analysis of Systems (TACAS 1998)*, volume 1384 of *LNCS*, pages 345 – 357. Springer, 1998.

[125] R. P. Kurshan, V. Levin, and H. Yenigün. Compressing transitions for model checking. In *Proc. of Computer Aided Verification (CAV 2002)*, volume 2404 of *LNCS*, pages 569–581. Springer, 2002.

[126] S. K. Lahiri, T. Ball, and B. Cook. Predicate abstraction via symbolic decision procedures. In *Proc. of Computer Aided Verification (CAV'05)*, volume 3576 of *LNCS*, pages 24–38. Springer, 2005.

[127] Y. Lakhnech, S. Bensalem, S. Berezin, and S. Owre. Incremental verification by abstraction. In *Proc. of Tools and Algorithms for the Construction and Analysis of Systems TACAS 2001*, volume 2031 of *LNCS*, pages 98–112. Springer, 2001.

[128] K. G. Larsen and W. Yi. Time-abstracted bisimulation: Implicit specifications and decidability. *Information and Computation*, 134(2):75–101, 1997.

[129] Kim G. Larsen, F. Larsson, P. Pettersson, and W. Yi. Efficient verification of real-time systems: Compact data structure and state-space reduction. In *Proc. of Real-Time Systems Symposium (RTSS'97)*, pages 14–24. IEEE Computer Society Press, 1997.

[130] M. Laursen, R. Madsen, and S. Mortensen. Verifying distributed Lego RCX programs using Uppaal, 1999. Technical report, Institute of Computer Science, Aalborg University.

[131] D. Lee and M. Yannakakis. Online minimization of transition systems. In *Proc. of Symposium on Theory of Computing (STOC 1992)*, pages 264–274. ACM Press, 1992.

[132] F. Lerda and R. Sisto. Distributed-memory model checking with SPIN. In *Proc. of SPIN workshop*, volume 1680 of *LNCS*. Springer, 1999.

[133] F. Lerda and W. Visser. Addressing dynamic issues of program model checking. In *Proc. of SPIN Workshop*, volume 2057 of *LNCS*, pages 80–102. Springer, 2001.

[134] N. G. Leveson and C. S. Turner. Investigation of the Therac-25 accidents. *IEEE Computer*, 26(7):18–41, 1993.

[135] F. Lin, P. Chu, and M. Liu. Protocol verification using reachability analysis: the state space explosion problem and relief strategies. *Computer Communication Review*, 17(5):126–134, 1987.

[136] M. R. Lowry. Software construction and analysis tools for future space missions. In *Proc. Tools and Algorithms for Construction and Analysis of Systems (TACAS 2002)*, volume 2280 of *LNCS*, pages 1–19. Springer, 2002.

[137] M. Mihail and C. H. Papadimitriou. On the random walk method for protocol testing. In *Proc. Computer Aided Verification (CAV 1994)*, volume 818 of *LNCS*, pages 132–141. Springer, 1994.

[138] K. L. McMillan. A technique of state space search based on unfolding. *Formal Methods in System Design: An International Journal*, 6(1):45–65, 1995.

[139] A. A. Milne. *Winnie-the-Pooh*. 1926.

[140] M. O. Möller. Parking can get you there faster - model augmentation to speed up real-time model-checking. In *Electronic Notes in Theoretical Computer Science*, volume 65. Elsevier, 2002.

[141] M. Musuvathi, D. Y. W. Park, A. Chou, D. R. Engler, and D. L. Dill. CMC: A pragmatic approach to model checking real code. In *Proc. of Symposium on Operating Systems Design and Implementation (OSDI'02)*, 2002.

[142] K. S. Namjoshi and R. P. Kurshan. Syntactic program transformations for automatic abstraction. In *Proc. of Computer Aided Verification (CAV 2000)*, pages 435–449. Springer, 2000.

[143] J. Ouaknine and J. Worrell. Revisiting digitization, robustness, and decidability for timed automata. In *Proc. of IEEE Symp. on Logic in Computer Science (LICS 2003)*, pages 198–207. IEEE Computer Society Press, 2003.

[144] K. Ozdemir and H. Ural. Protocol validation by simultaneous reachability analysis. *Computer Communications*, 20:772–788, 1997.

[145] G. J. Pace, F. Lang, and R. Mateescu. Calculating tau-confluence compositionally. In *Proc. Computer Aided Verification (CAV 2003)*, volume 2725 of *LNCS*, pages 446 – 459. Springer, 2003.

[146] R. Paige and R. E. Tarjan. Three partition refinement algorithms. *SIAM Journal of Computing*, 16(6):973–989, 1987.

[147] C. Pasareanu, R. Pelánek, and W. Visser. Concrete search with abstract matching and refinement. In *Proc. of Computer Aided Verification (CAV 2005)*, volume 3576 of *LNCS*, pages 52–66. Springer, 2005.

[148] C. Pasareanu, R. Pelánek, and W. Visser. State matching for efficient test input generation. In *Proc. of Automated Software Engineering (ASE'05)*, pages 414–417. ACM Press, 2005.

[149] C. Pasareanu and W. Visser. Verification of java programs using symbolic execution and invariant generation. In *Proc. of SPIN Workshop 2004*, volume 2989 of *LNCS*. Springer, 2004.

[150] R. Pelánek. Typical structural properties of state spaces. In *Proc. of SPIN Workshop*, volume 2989 of *LNCS*, pages 5–22. Springer, 2004.

[151] R. Pelánek. On-the-fly state space reductions. Technical Report FIMU-RS-2005-03, Masaryk University Brno, 2005.

[152] R. Pelánek, T. Hanžl, I. Černá, and L. Brim. Enhancing random walk state space exploration. In *Proc. of Formal Methods for Industrial Critical Systems (FMICS'05)*, pages 98–105. ACM Press, 2005.

[153] R. Pelánek and J. Strejček. Deeper connections between LTL and alternating automata. In *Proc. of Conference on Implementation and Application of Automata(CIAA 2005)*, volume 3845 of *LNCS*, pages 241–252. Springer, 2005.

[154] D. Peled. Combining partial order reductions with on-the-fly model-checking. In *Proc. of Computer Aided Verification (CAV 1994)*, volume 818 of *LNCS*, pages 377–390. Springer, 1994.

[155] W. Penczek, R. Gerth, R. Kuiper, and M. Szreter. Partial order reductions preserving simulations. In *Proc. of CSP'99 Workshop*, pages 153–171, 1999.

[156] W. Penczek, M. Szreter, R. Gerth, and R. Kuiper. Improving partial order reductions for universal branching time properties. *Fundamenta Informaticae*, 43(1-4):245–267, 2000.

[157] G. D. Penna, B. Intrigila, I. Melatti, E. Tronci, and M. V. Zilli. Exploiting transition locality in automatic verification of finite state concurrent systems. *Software Tools for Technology Transfer (STTT)*, 6(4):320–341, 2004.

[158] A. Pnueli. In transition from global to modular temporal reasoning about programs. *Logics and models of concurrent systems*, pages 123–144, 1985.

[159] C. S. Păsăreanu, M. B. Dwyer, and W. Visser. Finding feasible abstract counter-examples. *Software Tools for Technology Transfer (STTT)*, 5(1):34–48, 2003.

[160] A. Puri. Dynamical properties of timed automata. *Discrete Event Dynamic Systems*, 10(1-2):87–113, 2000.

[161] K. Qian and A. Nymeyer. Guided invariant model checking based on abstraction and symbolic pattern databases. In *Proc. of Tools and Algorithms for Construction and Analysis of Systems (TACAS 2004)*, number 2988 in LNCS, pages 487–511. Springer, 2004.

[162] T. Rokicki. *Representing and Modeling Digital Circuits*. PhD thesis, Stanford University, Stanford, USA, 1993.

[163] S. Shoham and O. Grumberg. Monotonic abstraction-refinement for CTL. In *Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, volume 2988 of *LNCS*, pages 546–560. Springer, 2004.

[164] U. Stern and D. L. Dill. Parallelizing the Murφ verifier. In *Proc. of Computer Aided Verification (CAV 1997)*, volume 1254 of *LNCS*, pages 256–267. Springer, 1997.

[165] U. Stern and D. L. Dill. Using magnetic disk instead of main memory in the Murphi verifier. In *Proc. of Computer Aided Verification (CAV'98)*, volume 1427 of *LNCS*, pages 172–183. Springer, 1998.

[166] C. Stirling. Local model checking games. In *Proc. of Conference on Concurrency Theory (CONCUR'95)*, volume 962 of *LNCS*, pages 1–11. Springer, 1995.

[167] A. Valmari. A stubborn attack on state explosion. In *Proc. of Computer Aided Verification (CAV'91)*, volume 531 of *LNCS*, pages 156–165. Springer, 1991.

[168] H. van der Schoot. Partial-order verification in spin can be more efficient. In *Proc. of SPIN Workshop*. Twente University, 1997.

[169] W. Visser. Memory efficient state storage in SPIN. In *Proc. of SPIN Workshop*, pages 21–35, 1996.

[170] W. Visser, S. Park, and J. Penix. Applying predicate abstraction to model check object-oriented programs. In *Proc. of Workshop on Formal Methods in Software Practice*. ACM Press, 2000.

[171] M. De Wulf, L. Doyen, and J.-F. Raskin. Almost ASAP semantics: From timed models to timed implementations. In *Proc. of Hybrid Systems: Computation and Control (HSCC'04)*, volume 2993 of *LNCS*, pages 296–310. Springer, 2004.

[172] T. Xie, D. Marinov, and D. Notkin. Rostra: A framework for detecting redundant object-oriented unit tests. In *Proc. 19th Automated Software Engineering*, 2004.

[173] K. Yorav. *Exploiting Syntactic Structure for Automatic Verification*. PhD thesis, The Technion – Israel Institute of Technology, 2002.

Index

Appendix A

Notation

M	guarded command language model
A	timed/stopwatch automaton
$\mathcal{C}$	set of clocks of timed automata
V	set of variables of guarded command language model
x, y	variables or clocks in a model
l	location in timed automaton
ν	valuation of clocks
$[\![\cdot]\!]$	(concrete) semantics of a given entity
T	labeled transition system
S	set of all states in LTS
s	a state in LTS
a, b, c, a_i	action name
$\xrightarrow{a}$	transition in state space
τ	invisible action
$\sim$	bisimulation relation
$[s]_\sim$	equivalence class of bisimulation
α	abstraction function on states (or set of states)
$\mathcal{A}$	abstraction function on transition systems
$\mathfrak{a}$	state in abstract transition system
ϕ	predicate
Φ	set of predicates
$\vec{b}$	bitvector
ϵ	sampling period
i, j, k	indexes
Wait	data structure holding states to be explored during the exploration algorithm
States	data structure holding visited states during the exploration algorithm
Transitions	data structure holding visited transitions
pre, *post*	weakest precondition, strongest postcondition
RA	set of reachable actions

Appendix B

Sorter Example

Here we provide some details about our main running example: the NQC source code of the Lego© Sorter example and both guarded command language and timed automata models.

B.1 NQC Source Code

The source code is given in the NQC language[1].

```
#define BELT    OUT_A
#define ARM     OUT_C
#define BELT2   OUT_B
#define LIGHT   SENSOR_1
#define BUTTON  SENSOR_3
#define ROTATION SENSOR_2
#define LONG 1
#define SHORT 2
#define LIGHT_TRESHOLD  45

int brick = 0;
int requests = 0;

task main() {
        int x=1;
        SetSensor(LIGHT, SENSOR_LIGHT);
        SetSensor(BUTTON, SENSOR_TOUCH);
        SetSensor(ROTATION, SENSOR_ROTATION);
        SetPower(BELT, 1);
        SetPower(ARM, 1);
        Rev(BELT);
        Rev(ARM);
        On(BELT);
        SetPower(BELT2, 3);
        start arm_controller;
        start light_sensor_controller;
        start belt2_controller;
        start button_controller;
        while (true) {}
}

task light_sensor_controller() {
        int x=0;
        while (true) {
```

[1] http://bricxcc.sourceforge.net/nqc/

```
                if (LIGHT > LIGHT_TRESHOLD) {
                        PlaySound(SOUND_CLICK);
                        Wait(30);
                        x = x + 1;
                } else {
                        if (x>2) {
                                PlaySound(SOUND_UP);
                                ClearTimer(0);
                                brick = LONG;
                        } else if (x>0) {
                                PlaySound(SOUND_DOUBLE_BEEP);
                                ClearTimer(0);
                                brick = SHORT;
                        }
                        x = 0;
                }
        }
}

task arm_controller() {
        while (true) {
                if (Timer(0)>15 && brick == LONG) {
                        ClearSensor(ROTATION);
                        Off(BELT);
                        On(ARM);
                        while (ROTATION < 12) { }
                        brick = 0;
                        Off(ARM);
                        On(BELT);
                }
        }
}

task belt2_controller() {
        while (true) {
                if (Timer(0)>25 && brick == SHORT) {
                        brick = 0;
                        if (requests > 0) {
                                OnRev(BELT2);
                                requests = requests - 1;
                        } else {
                                OnFwd(BELT2);
                        }
                        Wait(200);
                        Off(BELT2);
                }
        }
}

task button_controller() {
        while (true) {
                if (BUTTON == 1) {
                        requests = requests + 1;
                        Wait(100);
                }
        }
}
```

B.2 Guarded Command Language Model

Now we give a guarded command language model. To make it more readable, we divide the model into several parts corresponding to different tasks in the source code and to the environment. Note that this division is artificial and that formally the guarded command language model is 'monolithic'.

Light Sensor Controller

$$
\begin{array}{lcl}
LC = 0 \wedge token = 3 \wedge LightSensorLevel = 0 & \longmapsto & token := 4; \\
LC = 0 \wedge token = 3 \wedge LightSensorLevel = 1 & \longmapsto & LC := 1, x := 1, token := 4; \\
LC = 1 \wedge token = 3 \wedge LightSensorLevel = 1 & \longmapsto & x := x + 1, token := 4; \\
LC = 1 \wedge token = 3 \wedge LightSensorLevel = 0 & \longmapsto & LC := 2, timer := 0; \\
LC = 2 \wedge x <= 2 & \longmapsto & LC := 0, brick := 3, token := 4; \\
LC = 2 \wedge x > 2 & \longmapsto & LC := 0, brick := 4, token := 4;
\end{array}
$$

Arm Controller

$$
\begin{array}{lcl}
AC = 0 \wedge token = 1 \wedge \neg(brick = 4 \wedge timer > 3) & \longmapsto & token := 2; \\
AC = 0 \wedge token = 1 \wedge brick = 4 \wedge timer > 3 & \longmapsto & AC := 1, brick := 0, \\
 & & token := 2, ArmKicking := 1, \\
 & & Belt1Moving := 0; \\
AC = 1 \wedge token = 1 & \longmapsto & AC := 0, ArmKicking := 0, \\
 & & Belt1Moving := 1, token := 2;
\end{array}
$$

Belt2 Controller

$$
\begin{array}{lcl}
BC2 = 0 \wedge token = 2 \wedge & & \\
\quad \neg(brick = 3 \wedge timer > 7) & \longmapsto & token := 3; \\
BC2 = 0 \wedge token = 2 \wedge & & \\
\quad (brick = 3 \wedge timer > 7) & \longmapsto & BC2 := 1, t1 := 0; \\
BC2 = 1 \wedge requests > 0 & \longmapsto & BC2 := 2, Belt2Moving := 2, token := 3; \\
BC2 = 1 \wedge requests = 0 & \longmapsto & BC2 := 2, Belt2Moving := 1, token := 3; \\
BC2 = 2 \wedge token = 2 \wedge t1 < 4 & \longmapsto & BC2 := 0, t1 := t1 + 1, token := 3; \\
BC2 = 1 \wedge token = 2 \wedge t1 = 4 & \longmapsto & BC2 := 0, token := 3;
\end{array}
$$

Button Controller

$$
\begin{array}{lcl}
token = 0 \wedge ButtonPressed = 0 & \longmapsto & token := 1; \\
token = 0 \wedge ButtonPressed = 1 & \longmapsto & ButtonPressed := 0, \\
 & & requests := requests + 1, token := 1;
\end{array}
$$

Timer

$$
\begin{array}{lcl}
token = 4 \wedge timer < 8 & \longmapsto & timer := timer + 1, token := 5; \\
token = 4 \wedge timer = 8 & \longmapsto & token := 5;
\end{array}
$$

User

$$
\begin{array}{lcl}
User = 0 \wedge token = 5 & \longmapsto & token := 6; \\
User = 0 \wedge token = 5 & \longmapsto & ButtonPressed := 1, token := 6; \\
User = 0 \wedge token = 5 & \longmapsto & User := 1, PutShortBrick := 1, \\
 & & t2 := 1, token := 6; \\
User = 0 \wedge token = 5 & \longmapsto & User := 0, PutLongBrick := 1, \\
 & & t2 := 3, token := 6; \\
User = 1 \wedge token = 5 \wedge t2 > 0 & \longmapsto & t2 := t2 - 1, token := 6; \\
User = 1 \wedge token = 5 \wedge t2 = 0 & \longmapsto & User := 0, token := 6;
\end{array}
$$

Brick

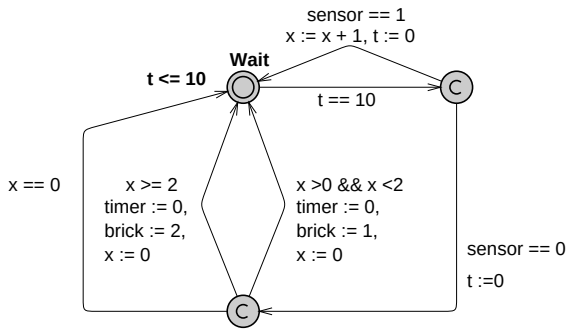
If there are more bricks, we need more (suitably modified) copies of this fragment of the code.

$$
\begin{array}{lcl}
token = 6 \wedge location1 = 0 \wedge PutShortBrick = 0 & \longmapsto & token := 0; \\
token = 6 \wedge location1 = 0 \wedge PutShortBrick = 1 & \longmapsto & PutShortBrick := 0, \\
 & & location1 := 1, \\
 & & token := 0; \\
token = 6 \wedge location1 = 1 \wedge & & \\
(position1 < 3 \vee position1 = 4 \vee & & \\
(position1 > 5 \wedge position1 < 10)) \wedge & & \\
Belt1Moving = 1 & \longmapsto & position1 := position1 + 1, \\
 & & token := 0; \\
token = 6 \wedge location1 = 1 \wedge Belt1Moving = 1 \wedge & & \\
position1 = 3 & \longmapsto & LightSensorLevel := 1, \\
 & & position1 := position1 + 1, \\
 & & token := 0; \\
token = 6 \wedge location1 = 1 \wedge Belt1Moving = 1 \wedge & & \\
position1 = 5 & \longmapsto & LightSensorLevel := 0, \\
 & & position1 := position1 + 1, \\
 & & token := 0; \\
token = 6 \wedge location1 = 1 \wedge position1 = 10 \wedge & & \\
ArmKicking = 0 \wedge Belt1Moving = 1 & \longmapsto & position1 := position1 + 1, \\
 & & token := 0; \\
token = 6 \wedge location1 = 1 \wedge position1 = 11 \wedge & & \\
ArmKicking = 0 \wedge Belt1Moving = 1 & \longmapsto & location1 := 2, \\
 & & position1 := 3, \\
 & & token := 0; \\
token = 6 \wedge location1 = 1 \wedge Belt1Moving = 0 \wedge & & \\
position1 < 10 & \longmapsto & token := 0; \\
token = 6 \wedge location1 = 1 \wedge Belt1Moving = 0 \wedge & & \\
(position1 = 10 \vee position1 = 11) \wedge & & \\
ArmKicking = 0 & \longmapsto & token := 0; \\
token = 6 \wedge location1 = 1 \wedge & & \\
(position1 = 10 \vee position1 = 11) \wedge & & \\
ArmKicking = 1 & \longmapsto & location1 := 3, \\
 & & token := 0; \\
token = 6 \wedge location1 = 2 \wedge Belt2Moving = 1 \wedge & & \\
position1 < 5 & \longmapsto & position1 := position1 + 1, \\
 & & token := 0; \\
token = 6 \wedge location1 = 2 \wedge Belt2Moving = 2 \wedge & & \\
position1 > 0 & \longmapsto & position1 := position1 - 1, \\
 & & token := 0; \\
token = 6 \wedge location1 = 2 \wedge Belt2Moving = 0 & \longmapsto & token := 0; \\
token = 6 \wedge location1 = 2 \wedge Belt2Moving = 1 \wedge & & \\
position1 = 5 & \longmapsto & location1 := 4, token := 0; \\
token = 6 \wedge location1 = 2 \wedge Belt2Moving = 2 \wedge & & \\
position1 = 0 & \longmapsto & location1 := 5, token := 0; \\
location1 = 5 \vee location1 = 4 \vee location1 = 3 & \longmapsto & token := 0;
\end{array}
$$

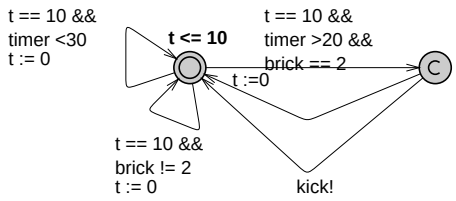
B.3 Timed Automata Model

Finally, we present Uppaal templates of the timed automata model.

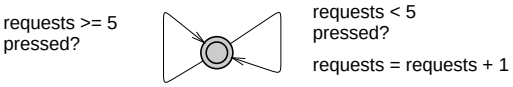
Light Sensor Controller



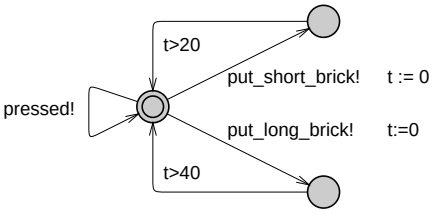
Arm Controller



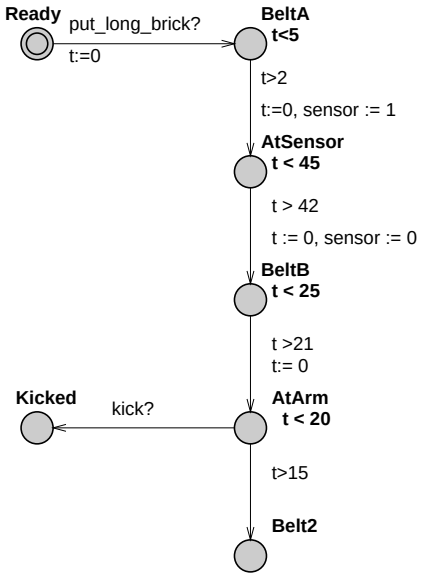
Button Controller



User



Long Brick



Short Brick