Sebastian Kupferschmid

Directed Model Checking for Timed Automata

Doktorarbeit

Institut für Informatik
Technische Fakultät
Albert-Ludwigs-Universität Freiburg

November 2009

Tag der Disputation:
18. Dezember 2009

Dekan:
Prof. Dr. Hans Zappe, *Albert-Ludwigs-Universität Freiburg*

Gutachter:
Prof. Dr. Bernhard Nebel, *Albert-Ludwigs-Universität Freiburg*
Prof. Dr. Andreas Podelski, *Albert-Ludwigs-Universität Freiburg*

To my family

Zusammenfassung

Die vorliegende Dissertation mit dem Titel "Directed Model Checking for Timed Automata" befasst sich mit der gerichteten Modellprüfung für Realzeitsysteme. Die Arbeit gliedert sich in zwei einführende Teile und einen inhaltlichen Teil. Der erste Teil führt in das Gebiet der Modellprüfung ein. Hier werden grundlegende Konzepte wie z. B. Kripke Struktur, temporale Logik und das Problem der Modellprüfung vorgestellt. Der Teil endet mit einer kurzen Beschreibung existierender Modellprüfungsverfahren.

Der zweite Teil behandelt Realzeitsysteme und gerichtete Modellprüfung. Er enthält die Definitionen, die zum Verständnis dieser Dissertation nötig sind. Zuerst wird die Syntax und die Semantik von Realzeitautomaten eingeführt. Da Zeit in diesem Modell durch reelle Zahlen modelliert wird, ist der Zustandsraum eines Realzeitautomaten ein überabzählbar großes Transitionssystem. Deshalb scheinen Realzeitsysteme ungeeignet für die Modellprüfung zu sein. Das ist allerdings nicht der Fall, da sich diese Zustandsräume endlich partitionieren lassen. Im Anschluss wird die gerichtete Modellprüfung für Realzeitsysteme eingeführt. Neben der Vorstellung eines allgemeinen Algorithmus für die gerichtete Modellprüfung werden hier auch existierende Ansätze für die gerichtete Modellprüfung diskutiert.

Der dritte Teil bildet den Hauptteil der Arbeit. Es werden verschiedene Heuristiken und Verbesserungen für die gerichtete Modellprüfung eingeführt. In Kapitel 5 wird die erfolgreichste Relaxierung aus dem Gebiet der Handlungsplanung auf die gerichtete Modellprüfung für Realzeitautomaten adaptiert und erweitert. Mit den resultierenden Heuristiken ist es möglich, erreichbare Fehlerzustände in Systemen zu finden, die mit zuvor vorgeschlagenen Heuristiken nicht entdeckt werden können. Im darauf folgenden Kapitel wird Prädikatenabstraktion verwendet, um sogenannte Musterdatenbanken zu erzeugen. Durch die Kombination von bekannten Techniken aus den Bereichen der Modellprüfung und der künstlichen Intelligenz erhält man eine Familie von Heuris-

tiken, die mit dem aktuellen Stand der Technik mithalten können. In Kapitel 7 wird eine weitere Musterdatenbank-Heuristik, die auf dem Prinzip der russischen Puppen basiert präsentiert. Der Ansatz zielt darauf ab, diejenigen Teile des Systems so gut wie möglich zu erhalten, die unmittelbar relevant für die zu überprüfende Eigenschaft sind. Mit der resultierenden Heuristik lassen sich beweisbar kürzeste Fehlerpfade in den größten unserer Fallbeispiele finden. Mit der Technik, die in Kapitel 8 präsentiert wird, lässt sich heuristische Suche im Allgemeinen deutlich beschleunigen. Dieser Ansatz kann mit vielen Heuristiken kombiniert werden und skaliert oft deutlich besser als gierige Suche. Gleichzeitig liefert das Verfahren erheblich kürzere Fehlerpfade als gierige Suche. Im letzten Kapitel wird zuerst eine auf Gegenbeispielen basierende Abstraktionsverfeinerung für Realzeitsysteme präsentiert. Danach wird anhand einiger Beispiele demonstriert, dass gerichtete Modellprüfung bei fehlerhaften Systemen oft deutlich performanter als Abstraktionsverfeinerung ist.

Die Dissertation schließt mit einer Diskussion der wesentlichen Ergebnisse des dritten Teils und einem Ausblick auf zukünftige Forschungsaufgaben in diesem Gebiet.

Acknowledgments

I wrote the thesis at hand while I was a member of the research group on the Foundations of Artificial Intelligence at Albert-Ludwigs-Universität Freiburg, headed by Bernhard Nebel. It has been a long while from the very beginning of my PhD studies to the completion of the thesis. During that period, many people directly or indirectly contributed to it. Now, I would like to take the opportunity to thank them.

First of all, I want to thank my adviser Bernhard Nebel. Bernhard gave me the necessary freedom to pursue my own ideas. At the same time he gave me valuable advice and pushed me at the right time to get this thesis finally done. I am really thankful for that. Andreas Podelski served as the second reviewer for this thesis. I not only want to thank Andreas for this service, but also for many fruitful discussions and a successful collaboration.

Special thanks go to Jörg Hoffmann. Especially in the beginning of my PhD studies, Jörg helped me a lot to gain ground in heuristic search. For me, working with Jörg was like learning from him. Many thanks for that. Special thanks also go to Malte Helmert, for being a friend and an exceptional officemate. Whenever I had a scientific question Malte almost always had a brilliant answer to it. I thank Malte for the great time we had, too much coffee and listening to Helge Schneider. And of course, special thanks also go to Martin Wehrle. I want to thank Martin for an intensive and very productive collaboration and for providing valuable feedback on several drafts of this thesis. I really enjoyed our pair programming sessions in which we created MCTA. It was always fun to work with Martin.

I also want to thank the German Research Foundation (DFG) that partly supportedy this work as part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS, http://www.avacs.org/). The many inspiring discussions and collaborations we had in the R3 subproject were a valuable contribution to this thesis.

In addition to Bernhard, Andreas, Jörg, and Martin, I particularly want to thank Henning Dierks, Klaus Dräger, Bernd Finkbeiner, and Jan-Georg Smaus. I also want to thank Kim G. Larsen and Gerd Behrmann from Aalborg University. Following my visit in Aalborg in 2004, they greatly supported me in getting acquainted with the implementation aspect of model checking timed automata.

This thesis was proofread very carefully by different people over and over again. I am very thankful to those who undertook this laborious task: Malte Helmert, Robert Mattmüller, Gabi Röger, Jan-Georg Smaus, Martin Wehrle, and Stefan Wölfl. I also want to thank all my former and current colleagues at the research group on the Foundations of Artificial Intelligence. Thank you all for the many helpful discussions we had, for the nice Oberseminars, and for playing table soccer. It was always fun to work in such a pleasant working environment.

Last but definitely not least, I also want to thank my family and friends for their moral support. My heartfelt thanks go to all of them. I especially want to thank Birte Krüger, Anne Bongers, Tobias Sing, Stefan Hugger, Malte Helmert and my parents Angelika and Hans-Peter Kupferschmid.

Freiburg, November 2009 Sebastian Kupferschmid

Contents

Part I Model Checking: Motivation, Temporal Logic, Methodology

1 Introduction 3
1.1 Real-time Systems, Verification and Heuristics 4
1.2 An Illustrative Example 5
1.3 Outline 8

2 Model Checking 11
2.1 Motivation, Goals and History 11
2.2 Temporal Logic 12
2.2.1 Kripke Structure 13
2.2.2 A Branching Time Logic 14
2.2.3 The Model Checking Problem 16
2.3 Model Checking Techniques 17
2.3.1 Abstraction-based Methods 17
2.3.2 Symbolic Model Checking 18
2.3.3 Bounded Model Checking 19
2.3.4 Directed Model Checking 20

Part II Real-time Systems

3 Model Checking for Real-time Systems 23
3.1 Timed Automata 23
3.1.1 Syntax and Semantics 24
3.2 Timed Automata Systems 26
3.3 The Region Automaton 28
3.4 Our Formalism 31
3.4.1 Bounded Integer Variables 32

3.4.2 The Zone Automaton 32
3.4.3 Reachability Analysis for Timed Automata 34

4 Directed Model Checking for Real-time Systems 37
4.1 The General Idea 37
4.2 Directed Model Checking 38
4.2.1 A Basic Directed Model Checking Algorithm 39
4.2.2 Obtaining Heuristic Functions 39
4.3 Related Work on Directed Model Checking 40
4.3.1 Approaches for Untimed Systems 41
4.3.2 Approaches for Timed Systems 43
4.4 Our Model Checking Tools 45
4.4.1 UPPAAL/DMC 46
4.4.2 MCTA 46

Part III Heuristics for Model Checking Timed Automata

5 Adapting an AI Planning Heuristic for Directed Model Checking 49
5.1 The Monotonicity Abstraction 49
5.1.1 The General Idea 50
5.2 The Monotonicity Abstraction for Timed Automata 51
5.2.1 Abstract Semantics 52
5.2.2 Properties of the Monotonicity Abstraction 53
5.3 Approximating h^+ 55
5.3.1 Remarks on Linear Arithmetic 57
5.3.2 The h^L Heuristic 59
5.3.3 The h^U Heuristic 59
5.4 Evaluation 63
5.4.1 The Heuristic Functions 63
5.4.2 The Benchmark Set 64
5.4.3 Experimental Results 65
5.5 Exploiting Automata Locations 69
5.5.1 The General Idea 69
5.5.2 The h^L_X Heuristic 70
5.5.3 Admissibility of the h^L_X Heuristic 72
5.5.4 Evaluation 74
5.6 Conclusion 75

6 Heuristic Functions Based on Predicate Abstraction 77
6.1 Predicate Abstraction 77
6.1.1 The General Idea 78
6.1.2 Predicate Abstraction for Timed Automata 79
6.1.3 Building the Abstract State Space 79
6.1.4 Encoding Transitions 80
6.2 Pattern Database Heuristics 81
6.2.1 Predicate Abstraction Pattern Databases 81
6.2.2 Implementation Details 82
6.2.3 Selecting Predicates 83
6.2.4 Combining Multiple Pattern Databases 85
6.3 Evaluation 88
6.3.1 Experimental Setup 88
6.3.2 Experimental Results 89
6.4 Discussion 89
6.5 Conclusion 93

7 Fast Directed Model Checking via Russian Doll Abstraction 95
7.1 Russian Doll Abstraction 95
7.1.1 The General Idea 96
7.1.2 Obtaining Abstractions 96
7.1.3 Pattern Databases 100
7.2 Choosing Abstraction Sets 102
7.2.1 A Cone-of-influence-based Method 102
7.2.2 A Method Based on the Monotonicity Abstraction 103
7.2.3 Comparison of the Methods 106
7.3 Evaluation 107
7.3.1 Experimental Setup 108
7.3.2 Experimental Results 108
7.3.3 Discussion 109
7.4 Conclusion 112

8 Useless Transitions are Useful 113
8.1 Evaluating State Transitions 113
8.1.1 An Illustrating Example 114
8.1.2 Existing Methods 115
8.2 Transition-based Directed Model Checking 115
8.2.1 Useless and Relatively Useless Transitions 116
8.2.2 Directed Model Checking with Relatively Useless Transitions 120
8.2.3 Discussion 121

8.3 Evaluation 122
8.3.1 Experimental Setup 123
8.3.2 Experimental Results 123
8.4 Conclusion 127

9 Automatic Abstraction Refinement for Timed Automata 129
9.1 Counterexample-guided Abstraction Refinement 129
9.1.1 Existing Methods for Timed Automata 130
9.2 From PLC Automata to Timed Automata 131
9.2.1 PLC Automata 132
9.2.2 Semantics of PLC Automata 133
9.2.3 Generating Abstractions 135
9.3 Abstraction Refinement for Timed Automata 135
9.3.1 Counterexample Analysis 135
9.3.2 Refining Abstractions 139
9.4 Evaluation 143
9.5 Discussion 145
9.6 Conclusion 148

10 Discussion 149
10.1 Conclusion 149
10.2 Future Work 151

A Case Studies and Benchmarks 153
A.1 The Fischer Protocol 153
A.1.1 The Models 153
A.2 The Single-tracked Line Segment Case Study 154
A.2.1 The Models 155
A.3 The Mutual Exclusion Case Study 157
A.3.1 The Models 158
A.4 The Arbiter Tree Case Study 159
A.4.1 The Models 160

B Determining Good Parameters for $h^{\mathcal{P}}_{syn}$ and $h^{\mathcal{P}}_{AR}$ 163
B.1 Experimental Results for the Syntax-Based Abstractions 163
B.2 Experimental Results for the Predicates Selected via ARMC 165
B.3 Runtime Results for the Generation of PDBs 167

C Preprocessing for the h^{coi} and $h^{\mathcal{A}}$ Heuristics 171
C.1 Preprocessing for the h^{coi} Heuristic 171
C.2 Preprocessing for the $h^{\mathcal{A}}$ Heuristic 173

References 175

Part I

Model Checking:
Motivation, Temporal Logic, Methodology

1

Introduction

Modern life without embedded systems is hardly imaginable. In 2002, only 2% of all produced microprocessors were built in PCs. The vast majority can be found in embedded systems. The application areas of these systems range from MP3 players or automobiles to aircraft and factory controllers used in nuclear power plants. Malfunctions, like the prominent Pentium FDIV bug or the first test flight of the Ariane 5, can cause enormous loss of money. Even more severe are bugs in safety-critical systems, as they can result in loss of human lives. Therefore, correct functioning of these devices is mandatory.

Model checking is an appropriate means to achieve this. In a nutshell, model checking is a standard technique that supports system engineers in designing correct systems. The main advantage of model checking is that it is mainly an *automatic* method to verify user defined properties that a system has to meet. The disadvantage of model checking is the state explosion problem. As embedded devices grow both in complexity and functionality, new methods to tackle the state explosion problem have to be investigated.

The approach presented in this thesis, namely directed model checking, has its origin in Artificial Intelligence (AI). Especially during the last decade, AI planning has made tremendous progress in solving larger and larger search problems. This progress was mainly driven by the development of heuristic functions. In this thesis we propose methods that alleviate the state explosion problem by adapting and extending heuristic functions coming from the area of AI planning to the context of model checking. The main advantage of these functions is that they are computed fully automatically and do not need any user input. Therefore, directed model checking, as presented in this thesis, remains an automatic method.

1.1 Real-time Systems, Verification and Heuristics

Most embedded systems are real-time systems. In general, a real-time system is a computer system that interacts with the real world and the correct functioning of the device depends on timing requirements on these interactions. As an example of such a system consider an anti-lock braking system of a car. It consists of sensors that measure the rotational speed of each wheel and a controller that can influence the brake force for each wheel. If, during a braking operation, the speed difference of the slowest and the fastest wheel is, for a certain time, above some threshold, then the brake force for that wheel is not increased any more. If this is not enough to compensate the speed difference, then the brake force of that wheel is decreased. This process is continuously repeated. Depending on how the anti-lock braking system is modeled, it can possibly consist of several components. For such composed systems, a distinction can be drawn between synchronous and asynchronous systems. A typical representative of the former class are digital circuits. Here, the system's components are all driven by one common clock and the smallest amount of time between successive events is known a priori. In contrast, a characteristic attribute of asynchronous systems is that the time between successive events can be arbitrarily small. These systems are best modeled using *timed automata*. Roughly speaking, a timed automaton is a finite state automaton equipped with a set of real-valued clocks. The values of these clocks increase with the same constant pace over time.

On the one hand, timed automata are well-suited to model real-time systems, but on the other hand, the state explosion problem for real-time systems is even worse. The reason for this is that model checking timed automata suffers from two sources of state explosion: one stems from the control part (parallel composition) and is due to the interleaving semantics of the automata, the other source are the real-valued clocks. Because of the state explosion problem it is often not feasible to enumerate the entire reachable state space of practically relevant systems, i. e., it is often not possible to check if the model under consideration satisfies the given property or not.

One possibility to cope with that problem is to use *directed model checking*. In a nutshell, directed model checking is the application of heuristic search to model checking. In directed model checking, instead of *verifying* a property, one tries to prove the opposite, i. e., to *falsify* the given property. This can be easier, because an error state may be found by exploring only a small fraction of the entire search space. Algorithms that are good at detecting error states can be used for debugging purposes, which is one of the main purposes of model checking. By applying heuristic search methods to model checking, directed model checking accelerates the detection of reachable error states. In heuristic search, the traversal of the state space is guided ("directed") with a heuristic

function. Such a function assigns to each state that is encountered during the state space traversal a heuristic value. States with lower heuristic values are explored first.

In many cases, the use of directed model checking methods enormously accelerates the detection of error states. It also improves the quality of the found error traces, i. e., reported error traces are much shorter and thus easier to understand. We will see this in the next section.

1.2 An Illustrative Example

In this section we want to provide the reader with an example of a model checking task for a timed automata system. We further demonstrate the state explosion problem and how it can be alleviated with directed model checking.

A mutual exclusion algorithm, or mutex for short, ensures that a shared resource is never accessed by more than one process of a concurrent system simultaneously. For example, in a multi-threaded computer program like a database system, a mutex can be used to grant write access to the hard disk.

Suppose a software developer has to implement a mutex algorithm for a program with two threads. Further suppose that the execution time of each instruction (each line of code) is between 1 and 2 ms. The developer comes up with the algorithm shown in Fig. 1.1. The pseudo-code shows the implementation of the mutex algorithm for the ith thread. In the algorithm, n is a shared integer variable and $pid_i = i$ is the unique id of thread i. The meaning of line 4 is that the thread has to wait for at least 2 ms. If a thread reaches line 6 it has access to the shared resource. If the thread no longer needs the resource it releases it and resets n to 0.

1 **function** mutex():
2   **if** $n \neq 0$ **then**: **goto** 2
3     $n := pid_i$
4   wait for ≥ 2 ms
5   **if** $n \neq pid_i$ **then**: **goto** 2
6   *access resource*
7   *release resource*
8     $n := 0$

Fig. 1.1. The mutex algorithm for the ith thread

Although the algorithm only consists of a few lines, it is not easy to see whether it satisfies the mutex property or not. In fact, the algorithm is a flawed version of the Fischer protocol for mutual exclusion [46, 73]. Due to the interleaving of process execution, which can cause an exponential blow-up of the

system's state space, it is difficult for humans to detect such errors. In the example, an error state is reachable, because the idle time of the wait instruction in line 4 has to be *strictly* greater than 2 ms.

Since each instruction of the mutex function can be executed at an arbitrary point in *real* time, this algorithm is best modeled using timed automata. Figure 1.2 gives a graphical representation of such an automaton model that represents the mutex function.

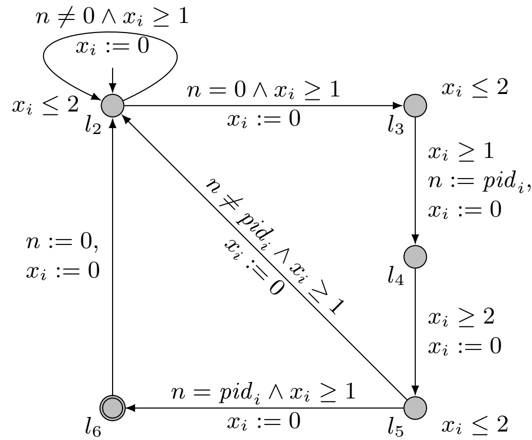


Fig. 1.2. A timed automaton model for the mutex algorithm

In the figure, the nodes l_2 to l_6 represent the locations of the automaton. They model the value of the program counter, i. e., if the program is in the jth line of the pseudocode, then the automaton is in location l_j. The location l_2 is the initial location of all automata. This is indicated with a small arrow. The variables pid_i and n are defined as in the algorithm from Fig. 1.1 and x_i is a *clock* variable. A clock variable is a special variable that measures the real time between successive events. The value of a clock is equal to the elapsed time since the clock was last reset. An edge, e. g. $l_2 \xrightarrow[x_i:=0]{n=0 \wedge x_i \geq 1} l_3$, is enabled if the current state satisfies the edge's guard ($n = 0 \wedge x_i \geq 1$). The values of the variables are changed according to the edge's effect (in this case, $x_i := 0$). It is also possible for the automaton to idle in its current state, but the idling time can be bounded by a constraint on the clock values. For instance, the automaton must not idle for more than two time units in location l_3. The location l_6 is the critical location, i. e., if an automaton is in this location it has access to the shared resource. For this reason it is double circled.

To verify that the algorithm satisfies the mutex property, it is sufficient to prove that always at most one automaton is in its double circled location. Di-

rected model checking, however, checks if there is a reachable error state, i. e., a state where at least two automata are in their critical locations. To check if the mutex property holds, model checkers like UPPAAL [7] or MCTA [72] can be used. UPPAAL is a very popular tool suite for designing and model checking real-time systems. The tool exclusively uses uninformed search methods like breadth-first or depth-first search. MCTA is also a model checker for real-time systems, which was mainly developed by the author of this thesis and Martin Wehrle. In contrast to UPPAAL, MCTA additionally features directed model checking techniques. MCTA uses a fraction of UPPAAL's input language and comes with several heuristics and search enhancements. We will explain them later in this thesis.

Table 1.1. Comparison of UPPAAL and MCTA. The results are computed on an AMD Opteron system with 2.3 GHz. Legend: *Aut.*: number of parallel automata, *explored states*: the number of explored states, before an error state was found, *trace length*: length of the found error trace

	explored states		runtime in s		trace length	
Aut.	UPPAAL	MCTA	UPPAAL	MCTA	UPPAAL	MCTA
2	92	9	0.0	0.0	24	8
3	408	11	0.0	0.0	122	10
4	4628	14	0.0	0.0	874	13
5	16770	16	0.1	0.0	2314	15
6	7002	18	0.0	0.0	841	17
7	79212	20	0.8	0.0	4475	19
8	541392	22	8.1	0.0	19039	21
9	942777	24	16.1	0.0	19723	23
10	542160	26	16.0	0.0	10625	25
11	2909478	28	116.5	0.0	23559	27
12	10565796	30	705.3	0.0	36881	29
13	7932379	32	1652.6	0.0	24338	31
14	7568414	34	1799.3	0.0	12742	33
15	12999765	35	9774.0	0.0	15574	35

Table 1.1 provides a first impression of the potential of directed model checking. Here, we compared UPPAAL's fastest search method with MCTA's on the faulty mutex algorithm. The results impressively demonstrate the benefit of directed model checking methods for the analysis of incorrect timed automata systems. In comparison with UPPAAL, our tool finds much shorter error traces much faster. While UPPAAL takes more than two and a half hours to detect an error trace in one of the larger examples, MCTA instantly finds an error trace that is orders of magnitude shorter and thus is much easier to understand. As already mentioned, with short error traces it is easier to fix the bug in the algo-

rithm. Wading through error traces with up to 36000 transitions as provided by UPPAAL in order to see what went wrong is not feasible.

1.3 Outline

The thesis at hand consists of three parts. The remainder of this part is mainly a short introduction to model checking. It presents basic concepts like Kripke structures and temporal logic and defines the model checking problem. This part is concluded by some brief sketches of prevailing model checking techniques. Part II focuses on real-time systems and directed model checking. It provides the necessary notations and definitions to understand this thesis. First, the syntax and semantics of timed automata are introduced. As the non-negative reals are used to model time, the semantics of timed automata is an uncountable transition system. At first glance, this seems not to be feasible for model checking. However, there are finite, exact abstractions of the semantics, namely the region and the zone automaton. Afterwards, directed model checking for timed automata is introduced. This includes a basic directed model checking algorithm as well as the discussion of some previous work in this area. Part III is the main part of this thesis. In Chap. 5, we adapt and extend the most successful heuristic function from the area of AI planning to the context of model checking timed automata. This allows us to detect reachable error states in systems that are beyond the scope of other previously proposed heuristics for directed model checking. In Chap. 6, we use predicate abstraction to generate pattern database heuristics. Here we obtain a family of heuristics by combining well-known techniques from the areas of model checking and artificial intelligence. The obtained heuristics are comparable with the state of the art in directed model checking. In Chap. 7, we present another pattern database heuristic that is based on a Russian doll principle. Our approach homes on preserving a precise representation of those parts of the system that are of immediate relevance to the property under consideration. The resulting heuristic allows us to find provable shortest error traces in our largest benchmarks in a matter of seconds. The technique presented in Chap. 8 is a generic search enhancement with which general heuristic search can be significantly accelerated. The framework can be applied to a wide range of heuristics, and often scales much better than greedy search and yields almost shortest error traces. This is achieved by not only prioritizing states, but also state transitions. In the last Chap. 9 of Part III, we first present a counterexample-guided abstraction refinement loop for timed automata. Hereinafter, we empirically demonstrate that in the presence of reachable error states, directed model checking often outperforms counterexample-guided abstraction refinement. Chapter 10 concludes. This thesis is partially based on the following papers.

- S. Kupferschmid, J. Hoffmann, H. Dierks, and G. Behrmann. Adapting an AI planning heuristic for directed model checking. In *Proceedings of the 13th International SPIN Workshop (SPIN 2006)*, pages 35–52. Springer-Verlag, 2006.
- J. Hoffmann, J.-G. Smaus, A. Rybalchenko, S. Kupferschmid, and A. Podelski. Using predicate abstraction to generate heuristic functions in Uppaal. In *Proceedings of the 4th Workshop on Model Checking and Artificial Intelligence (MOCHART 2006)*, pages 51–66. Springer-Verlag, 2007.
- H. Dierks, S. Kupferschmid, and K. G. Larsen. Automatic abstraction refinement for timed automata. In *Proceedings of the 5th International Conference on Formal Modelling and Analysis of Timed Systems (FORMATS 2007)*, pages 114–129. Springer-Verlag, 2007.
- S. Kupferschmid, K. Dräger, J. Hoffmann, B. Finkbeiner, H. Dierks, A. Podelski, and G. Behrmann. UPPAAL/DMC — Abstraction-based heuristics for directed model checking. In *Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2007)*, pages 679–682. Springer-Verlag, 2007.
- S. Kupferschmid, M. Wehrle, B. Nebel, and A. Podelski. Faster than UPPAAL? In *Proceedings of the 20th International Conference on Computer Aided Verification (CAV 2008)*, pages 552–555. Springer-Verlag, 2008.
- S. Kupferschmid, J. Hoffmann, and K. G. Larsen. Fast directed model checking via Russian doll abstraction. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2008)*, pages 203–217. Springer-Verlag, 2008.
- M. Wehrle, S. Kupferschmid, and A. Podelski. Transition-based directed model checking. In *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2009)*, pages 186–200. Springer-Verlag, 2009.

2

Model Checking

Model checking is a method to verify concurrent systems. Its main advantage over other approaches like theorem proving is that model checking is an automatic method. As a consequence system engineers do not need expert knowledge in logic to verify their systems. Once a suitable representation of the important parts of a design is translated into the input language of a model checker (which is often a simple compilation step), model checking is a fully automatic method. Another important characteristic of model checking is that temporal logic is used to specify the properties that a system has to meet. Temporal logic is very well-suited to describe the behavior of a system over time. Last but not least, if the system violates a property, a model checker generates a counterexample, i. e., a trace of instructions that led to the violation.

The beginning of model checking dates back to the early 1980s [24, 88]. At that time model checking was more a theoretical technique than a practical method. Now, 25 years later, model checking is a highly efficient means for the verification of concurrent systems and is used in many companies. Recently, Clarke, Emerson and Sifakis received the 2007 Turing Award for their pioneering and ongoing work on model checking.

2.1 Motivation, Goals and History

The original motivation for model checking was concurrent program verification. In such programs it is typically hard to find errors, since the reachability of an error state may depend on a particular order in which the programs execute their instructions. In the 1970s, the prevailing verification technique for this kind of programs were manual proofs, constructed in some Hoare-style logic based on formal axioms and inference rules. Beside the fact that this approach requires expert skills in logic and is very time consuming, the main disadvantage of this approach is that is not feasible to verify large systems. In the late 1970s, Pnueli

[84] and Owicki and Lamport [81] proposed to use temporal logic to specify concurrent systems, but proofs at that time were still handmade. At the beginning of the 1980s, things changed dramatically. Clarke and Emerson [24] and independently Queille and Sifakis [88] combined state space exploration techniques with temporal logic to verify concurrent programs. This was the birth of model checking. Their idea was to transform a concurrent program into a finite state transition system. In order to check (verify) if the program is a model of the property, which is given in temporal logic, their algorithm explored the entire reachable state space. Around 1990, McMillan introduced *symbolic model checking* [77]. Here, the states of the state space are not stored explicitly, but symbolically by representing sets of states with Boolean functions. Such sets can be efficiently represented using *binary decision diagrams* [17, 77], henceforth called BDDs. With this approach it was possible to verify extremely large systems, orders of magnitude more than with explicit state model checking [17]. In 1999, *bounded model checking* was introduced by Biere [12]. The main idea of this approach is to search for counterexamples in execution traces that consist of k or fewer steps. If no error is found, k is increased until k reaches some predefined upper bound. Instead of BDDs this approach normally uses propositional satisfiability testing (SAT). Recently, Yang and Dill [97] and Edelkamp et al. [43] introduced what today is known as *directed model checking*. In directed model checking, instead of verifying a given safety property one tries to *falsify* it, i. e., to search for reachable error states. Especially when the system is erroneous this is often much easier than the exhaustive enumeration of the entire state space. The reason for this is that an error state can be encountered by only exploring a small fraction of the entire reachable state space. To achieve this, heuristic search methods are used to quickly guide the search toward short error traces.

All the different model checking approaches have in common that they tackle the state explosion problem and thus make it possible to model check larger and larger systems. The progress in model checking can therefore also be seen as the progress on tackling the state explosion problem.

2.2 Temporal Logic

Conceptually, temporal logic is a formalism to describe the behavior of a system over time. It implicitly models the ordering of events without introducing time explicitly. Originally, temporal logic was developed by philosophers and linguists to describe the chronological ordering of events. In the late 1970s, Pnueli [84] and Owicki and Lamport [81] proposed to use temporal logic to specify the temporal and sequential behavior of systems. Probably the most important

temporal properties are *safety properties* and *liveness properties*. While a safety property describes that something bad will never happen, a liveness property expresses that something good will eventually happen. A typical instance of a safety property is mutual exclusion, typical liveness properties are request response properties.

2.2.1 Kripke Structure

Temporal logic is used to describe the behavior of a system. Such a system is usually modeled as some kind of *Kripke structure*. A Kripke structure is a transition system which is defined over a set of atomic propositions. A Kripke structure consists of a set of states, a transition relation and a labeling function that assigns to each state a set of atomic propositions that hold in that state.

Definition 2.1 (Kripke structure). *Let P be a set of atomic propositions. A Kripke structure over P is a tuple $M = (S, s_0, T, L)$, where S is a finite set of states, $s_0 \in S$ is the initial state, $T \subseteq S \times S$ is a transition relation, and $L : S \to 2^P$ is a labeling function which assigns to each state a set of propositions that hold in that state. Here, 2^P denotes the power set of P.*

Note that there are also definitions of the term where s_0 is a subset of S rather than an element of S. In this thesis, we never deal with systems that have more than one initial state, hence s_0 is always an element of S and not a subset thereof.

A particular behavior of a Kripke structure corresponds to a sequence of states where each state of the sequence can be reached from its immediate predecessor state via a transition.

Definition 2.2 (Computation path). *Let $M = (S, s_0, T, L)$ be a Kripke structure. A computation path, also called a trace, of M is a possibly infinite sequence $\pi = s_0, s_1, \ldots$ of states with $(s_i, s_{i+1}) \in T$ for all $i \geq 0$.*

The set of all computation paths of a Kripke structure starting from its initial state can be obtained by unfolding the Kripke structure. This results in an infinite tree, the so-called *computation tree*. Every path in this tree, starting from s_0, gives one possible behavior of the system. Figure 2.1 shows on the left a small Kripke structure M, defined over the set of atomic propositions $P = \{p, q\}$. States are depicted by nodes, the initial state is marked with a small arrow. For each pair of states in the transition relation, there is a directed edge in the graph connecting these states. Each state is labeled with the set of atomic propositions that hold in that state. The right part of the figure shows a part of the corresponding computation tree of M. The root of the tree (marked with a small arrow) corresponds to the initial state of the Kripke structure. The immediate successors of a node s correspond to the set of nodes $\{s' \in S \mid T(s, s')\}$.

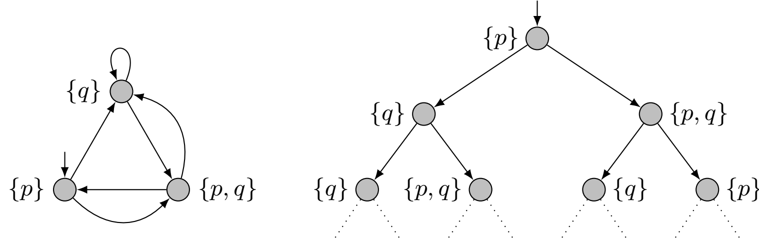


Fig. 2.1. A Kripke structure and its computation tree

2.2.2 A Branching Time Logic

Based on how a temporal logic deals with branching in the computation tree, it is either classified as a linear or branching time temporal logic. In a branching-time temporal logic the temporal operators quantify over *all* computation paths that start with a particular state. A linear-time temporal logic provides operators to describe events along a *single* computation path. In this thesis, we use *computation tree logic* (CTL) which is a branching time logic. Computation tree logic was introduced by Emerson and Clarke [45] in the early 1980s. For example CTL can be used to describe that there is a certain behavior of the example Kripke structure from Fig. 2.1, so that eventually a state is reached in which p and q are true. In CTL this can be expressed by $\exists\,\mathcal{F}(p \wedge q)$. CTL formulas are composed of path quantifiers $\exists$ (for at least one computation path) or $\forall$ (for all computation paths) and temporal operators $\mathcal{G}$, $\mathcal{F}$, $\mathcal{X}$ and $\mathcal{U}$ (globally, finally, next and until).

In CTL there are two types of formulas, namely *state formulas* and *path formulas*. While a state formula is interpreted on a state, i. e., the current values of the system's variables, a path formula is evaluated on a particular computation path.

Definition 2.3 (Syntax of CTL). *Let P be a set of atomic propositions. The syntax of a state formula is then defined by the following rules.*

1. *If $p \in P$, then p is a state formula.*
2. *If φ and ψ are state formulas, then $\neg\varphi$, $\varphi \vee \psi$ and $\varphi \wedge \psi$ are state formulas.*

The syntax of path formulas is defined as follows.

1. *If φ and ψ are state formulas, then $\mathcal{X}\,\varphi$, $\mathcal{F}\,\varphi$, $\mathcal{G}\,\varphi$ and $\varphi\,\mathcal{U}\,\psi$ are path formulas.*
2. *If φ is a path formula, then $\exists\varphi$ and $\forall\varphi$ are state formulas.*

Definition 2.4 (Semantics of CTL). *Let $M = (S, s_0, T, L)$ be a Kripke structure, defined over the set of atomic propositions P, $s \in S$ and π be a computation path of M. We will write $\pi[k]$ to refer to the kth state of that sequence. The semantics of CTL state formulas is defined as follows.*

$$
\begin{array}{lll}
M, s \models p & \textit{iff} & p \in L(s) \\
M, s \models \neg\varphi & \textit{iff} & M, s \not\models \varphi \\
M, s \models (\varphi \vee \psi) & \textit{iff} & M, s \models \varphi \textit{ or } M, s \models \psi \\
M, s \models (\varphi \wedge \psi) & \textit{iff} & M, s \models \varphi \textit{ and } M, s \models \psi \\
M, s \models \exists\varphi & \textit{iff} & M, \pi \models \varphi \textit{ for some path } \pi \textit{ with } \pi[0] = s \\
M, s \models \forall\varphi & \textit{iff} & M, \pi \models \varphi \textit{ for all paths } \pi \textit{ with } \pi[0] = s
\end{array}
$$

The semantics of CTL path formulas is defined as follows.

$$
\begin{array}{lll}
M, \pi \models \mathcal{X}\,\varphi & \textit{iff} & M, \pi[1] \models \varphi \\
M, \pi \models \mathcal{F}\,\varphi & \textit{iff} & M, \pi[k] \models \varphi \textit{ for some } k \geq 0 \\
M, \pi \models \mathcal{G}\,\varphi & \textit{iff} & M, \pi[k] \models \varphi \textit{ for all } k \geq 0 \\
M, \pi \models \varphi\,\mathcal{U}\,\psi & \textit{iff} & M, \pi[k] \models \psi \textit{ for some } k \geq 0 \textit{ and} \\
 & & M, \pi[i] \models \varphi \textit{ for all } 0 \leq i < k
\end{array}
$$

Figure 2.2 illustrates the basic CTL operators. The trees in the figure are computation trees of some Kripke structure. The grayed nodes represent states in which the formula φ holds, in black nodes the formula ψ holds. Intuitively $\exists\,\mathcal{G}\,\varphi$ means that there is a path on which φ holds in every state. The formula $\forall\,\mathcal{G}\,\varphi$ expresses that the property φ holds for all states of the system. Such an expression is called an invariant. The meaning of $\exists\,\mathcal{F}\,\varphi$ is that there is a reachable state in which φ holds. The formula $\forall\,\mathcal{F}\,\varphi$ states that on every path eventually φ becomes true. Intuitively, the meaning of $\exists\,\mathcal{X}\,\varphi$ is that there is an immediate successor state in which φ holds. The meaning of $\exists(\varphi\,\mathcal{U}\,\psi)$ is that there is some computation path π such that on the last state of its prefix ψ holds and on all intermediate states φ holds. The formula $\forall(\varphi\,\mathcal{U}\,\psi)$ expresses the same for all computation paths.

Note that $\forall\,\mathcal{G}\,\varphi$ is dual to $\exists\,\mathcal{F}\,\varphi$. More precisely, let $M = (S, s_0, T, L)$ be a Kripke structure and $s \in S$. $M, s \models \forall\,\mathcal{G}\,\varphi$ holds iff $M, s \not\models \exists\,\mathcal{F}\,\neg\varphi$. Also $M, s \models \forall\,\mathcal{F}\,\varphi$ and $M, s \not\models \exists\,\mathcal{G}\,\neg\varphi$ are equivalent. Also worth mentioning, the set $\{\textit{false}, \rightarrow, \exists\,\mathcal{X}, \exists\,\mathcal{U}, \exists\,\mathcal{G}\}$, where $\rightarrow$ denotes the propositional implication, is a minimum set of operators: all CTL formulas can be normalized to only use

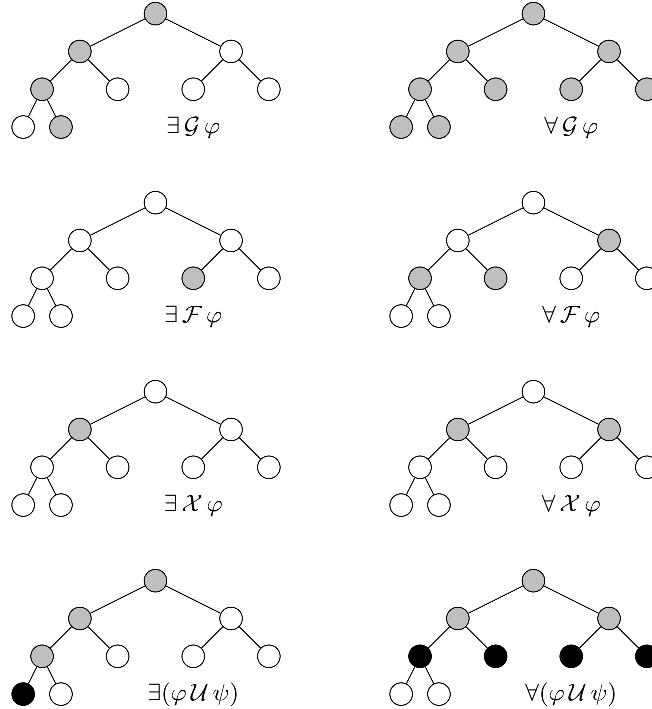


Fig. 2.2. Illustration of the basic CTL operators: grayed nodes represent states where φ holds, in the black nodes ψ holds

these operators (cf. [27]). In this thesis, we will also use $\Box$ for $\mathcal{G}$ and $\Diamond$ instead of $\mathcal{F}$.

2.2.3 The Model Checking Problem

With the definitions of Kripke structure and temporal logic, we can now formally define the model checking problem. Let M be a Kripke structure of the system of interest and φ be a temporal logic formula which describes the desired temporal behavior of the system. The model checking problem is to decide whether M satisfies φ. This is summarized in the next definition.

Definition 2.5 (Model checking problem). *Let $M = (S, s_0, T, L)$ be a Kripke structure and φ be some temporal formula. The model checking problem is defined as the decision problem whether*

$$M, s_0 \models \varphi.$$

Note that, depending on whether s_0 is a set of states or not, there is also another widely used definition of the model checking problem. Let M and φ be defined as above, then the model checking problem is $\{s \in S \mid M, s \models \varphi\} \cap s_0 \neq \emptyset$. Especially for regression-based state space traversal this may be more appropriate. In this thesis however, we will only deal with forward search.

2.3 Model Checking Techniques

All different approaches to model checking face one common problem: the state explosion problem. Roughly speaking, the state space of a practically relevant system is normally extremely huge because of a combinatorial blowup. This is due to the fact that the size of a system's state space tends to grow exponentially in the number of parallel automata and the number of variables. For example consider a parallel system with n automata. Each of these automata has k locations. For this system the size of the state space is k^n. The size of the state space also depends on how independently the automata can perform transitions. Strong dependency limits the number of possible states.

There are several approaches to deal with the state explosion problem. For example, distributed and external model checking make use of "extra" computation resources. In distributed model checking the model checking problem is distributed over several computers or processors [21, 38, 76]. In external model checking, secondary memory (e. g. hard disks) is used to store the state space. By doing so, much larger systems can be analyzed [13, 44, 65]. In the next section we will have a closer look at some approaches to tackle the state explosion problem.

2.3.1 Abstraction-based Methods

The aim of abstraction is to verify a property on an *abstract*, i. e., simplified, version of the system under consideration. The abstract system has to be significantly smaller than the original one, otherwise verification remains as hard as for the original system. Another property the abstract system has to meet is that it preserves the behavior of the original system. In general, the abstract system exhibits more behavior than the original one and thus normally does not satisfy the same properties as the original one. However, universal properties that are proved on the abstract system also hold on the original one. There are several different methods how abstractions can be used for model checking. Here we will briefly sketch two of them.

Predicate abstraction is the combination of model checking techniques and theorem proving. It was pioneered by Graf and Saïdi in the late 1990s [50]. To

check if a given system satisfies a certain property, the following is done. Based on a set of abstraction predicates (logical formulas), an over-approximation of the system under consideration is generated. The transition relation of the abstract system is typically computed using a theorem prover. The abstract system is then model checked. If the given property is verified then it also holds in the original system. Otherwise an abstract counterexample is generated. Especially in the latter case, predicate abstraction is often combined with *abstraction refinement*.

Abstraction refinement was first proposed by Clarke et al. [26] in the early 1990s. Since then many researchers began to automate this process starting with the work of Balarin and Sangiovanni-Vincentelli [4]. In abstraction refinement, a safety property is checked on a very coarse abstraction of the system under consideration. If the safety property does not hold on the abstract system, then it has to be checked if the abstract counterexample is not spurious, i. e., if it also holds on the original system. Otherwise the abstraction is refined and the process is repeated. In counterexample guided abstraction refinement (CEGAR) the refinement step depends on analyzing the counterexample, i. e., the abstraction is refined so that the spurious counterexample disappears in the next iteration.

2.3.2 Symbolic Model Checking

The first symbolic model checker was McMillan's *Symbolic Model Verifier* (SMV) [77] which is nowadays one of the most popular model checkers. The main idea of symbolic model checking (SMC) is to avoid to explicitly build the state space of the system under consideration. In SMC a Boolean encoding is used to represent the transition relation of the system and also to represent sets of states. The use of binary decision diagrams (BDDs) [15] for SMC was made popular by McMillan [77]. Independently, Pixley [83] and Couderd et al. [28] developed similar algorithms. By the use of BDDs, the original model checking algorithm, proposed by Emerson and Clarke [45], can solve orders of magnitude larger problems.

In a nutshell, a symbolic forward reachability algorithm can be implemented like this. Let $init : S \rightarrow \{false, true\}$ be a Boolean function (encoded as a BDD) that represents the system's initial state. Further let $T : S \times S \rightarrow \{false, true\}$ be another BDD that encodes the transition relation of the system. The traversal algorithm can then be formulated as a fix point iteration, which only uses basic BDD operations.

$$F_0 = \{s \in S \mid init(s) = true\}$$
$$F_{i+1} = \{s' \in S \mid s' \in F_i \vee \exists s \in F_i : T(s, s')\}$$

A fix point is reached if $F_{i+1} = F_i$, which can again be checked with basic BDD operations. The combination of, mainly hand tailored, abstractions and symbolic model checking made this approach even more successful. Due to this, for the first time it was possible to apply model checking to industrial designs. There are success stories on the exhaustive exploration of systems that contain 10^{20} states [18]. As SMC is so successful, many hardware companies started to use this approach to validate their designs. Nowadays SMC is a method that is used in many industrial strength model checkers.

2.3.3 Bounded Model Checking

In 1999, Biere et al. [12] introduced *bounded model checking* (BMC). The main idea of bounded model checking is to search for a counterexample in all computation paths of the system that consists of less than k execution steps. If no counterexample is found, k is increased and the process is repeated until k exceeds some *predefined* upper bound, or a counterexample is found. This is somewhat related to iterative deepening depth-first search. Since the user has to provide the upper bound for k, which is also called the *completeness threshold* of the system, BMC is not a complete method if the bound is not high enough. A BMC problem can efficiently be reduced to a propositional satisfiability problem. In contrast to SMC, where BDDs are used in order to represent sets of states, BMC utilizes SAT procedures. This leverages the success of SAT to the context of model checking.

In general, the structure of a SAT formula that is generated in the kth iteration of invariant checking looks like this: $s_0 \wedge \bigwedge_{i=0}^{k-1} R(i, i+1) \wedge \neg p_k$. Here s_0 is a formula describing the initial state of the system, $R(i, i+1)$ is an encoding of the transition relation from iteration i to iteration $i+1$ and p_i is a formula describing the invariant in iteration i. Note that termination of this algorithm cannot be checked as this can be done for BDD-based methods. For BDD-based methods termination can be checked as BDDs which represent the same sets of states are equal. The bottleneck of BDD-based methods is that the size of a BDD can be exponential in the number of states the BDD represents. This can even be true for systems which can be verified using an explicit state representation. SAT solvers that are especially tailored to the special structure of the formulas resulting from BMC improve this method even more [93]. An interesting property of BMC methods and SMC methods is that there is only little correlation on the problems that are hard for SMC and the problems that are hard for BMC [11]. In this context, BMC methods can be seen as a complementary approach to SMC. In recent years many hardware companies employ BMC methods to debug their designs.

2.3.4 Directed Model Checking

Directed model checking (DMC) is a very young research direction. The papers by Yang and Dill [97] and Edelkamp et al. [43] are the first contributions to this area. Both papers appeared around 2000. The term directed model checking was coined by Edelkamp. In contrast to SMC or BMC, states in DMC are typically represented explicitly. The main idea of DMC is to explore those parts of the state space first that show promise to contain reachable error states. As a consequence, it is possible to detect error states in systems whose entire state space is too huge for brute force methods. In directed model checking, the state space traversal is guided ("directed") towards error states based on specific criteria. Ideally, these guidance criteria are automatically extracted from the system under consideration by taking an abstraction thereof. Based on such an abstraction, a heuristic function h is computed that typically approximates a state's distance to its nearest error state. During the search process, h is used to assign each encountered state s a heuristic value $h(s)$. These values are used in order to determine which state to explore next. Directed model checking methods mainly differ in the way how they define and compute their underlying heuristic functions. As directed model checking is the central topic of this thesis, we will give a more detailed introduction to this research area in Chap. 4. In the main part of this thesis (Part III), we will present our contribution to this research direction.

Part II

Real-time Systems

3

Model Checking for Real-time Systems

Synchronous systems, like digital circuits, where all components are driven by one common clock, are best modeled using discrete time. In a discrete-time model, integers are used to model time. Asynchronous systems cannot be modeled appropriately using a discrete-time model. The reason for this is that in asynchronous systems, the interaction of processes can happen at arbitrary time points. To be able to use integers for modeling real-time, one has to determine a fixed minimum time quantum δ, i. e., the smallest time-span of two consecutive events. All other time-spans can then be expressed by integer multiples of δ. On the one hand, if δ cannot be chosen precise enough, subtle bugs can be missed. On the other hand, the smaller δ is, the larger is the induced state space of the system and thus model checking becomes infeasible. Brzozowski et al. [16] showed that the reachability problem for synchronous circuits with bounded delays cannot be solved correctly if a discrete time model is used, no matter how fine the resolution is chosen. To model asynchronous systems correctly, the application of real time is preferable. Instead of using integers to model time, the non-negative reals $\mathbb{R}_{\geq 0}$ are used.

In this chapter we introduce timed automata, an automata model that is appropriate to model real-time systems. Further, we will give a model checking algorithm with which these automata can be checked.

3.1 Timed Automata

Timed automata have first been proposed by Alur and Dill [2, 3]. Nowadays, they are the most used formalism to model real-time systems. In this thesis we define timed automata as they are used in UPPAAL, a state-of-the-art model checker for timed automata systems [7]. We use these definitions because much of our work is implemented in this tool. The definitions in this chapter are mainly based on the paper of Bengtsson and Yi [10].

3.1.1 Syntax and Semantics

A timed automaton is a finite automaton equipped with a set of real-valued *clocks*. The value of a clock gives the elapsed time since the clock was reset the last time. All clocks synchronously advance with the same constant pace.

Before we give the formal definition of timed automata, we will first provide an illustrative example. Figure 3.1 shows a timed automaton that is a very simple model of a one-button computer mouse. The initial location of the automaton is *init*. When the user clicks the mouse button, the automaton takes the edge from *init* to *single*. The edge is annotated with the clock reset $x := 0$. This resets the clock x to 0. Afterwards, the value of x increases with the pace of time. If the user does not click the mouse button again, then the automaton has to leave the *single* location after one time unit is passed and goes back to *init*.

This is modeled by labeling the *single* location with the location invariant $x \leq 1$. The value of x must satisfy the invariant while the automaton is in that location. If the invariant is not satisfied, the location has to be left. If the user was fast enough to double click the mouse button within less than 1 time unit, then the automaton would take the transition to *double*.

In the following, X is a finite set of clocks. Let $x, y \in X$ be clocks and let $c \in \mathbb{Z}$ be an integer constant. We refer to expressions of the form $x \bowtie c$ and $x - y \bowtie c$, where $\bowtie \in \{<, >, =, \leq, \geq\}$, as *clock constraints*. We define $\mathcal{B}(X)$ as the set of conjunctions over clock constraints.

Definition 3.1 (Syntax of timed automata). *A timed automaton is a tuple $A = \langle L, l^0, E, \Sigma, X, I\rangle$ where*

1. *L is a finite set of locations,*
2. *$l^0 \in L$ is the initial location,*
3. *$E \subseteq L \times \mathcal{B}(X) \times (\Sigma \cup \{\tau\}) \times 2^X \times L$, where 2^X denotes the power set of X, is a set of edges,*
4. *Σ is a finite set of synchronization labels, and $\tau \notin \Sigma$ denotes a special internal label,*
5. *X is a finite set of clocks and*
6. *$I : L \to \mathcal{B}(X)$ is a function that assigns* invariants *to locations.*

For an edge $\langle l, g, a, r, l'\rangle$ we will also write $l \xrightarrow[r]{g,a} l'$. The label $g \in \mathcal{B}(X)$ is called the *guard* of the edge and serves as an enabling condition. The automaton can only take this edge if the current clock values satisfy g. The label $r \subseteq 2^X$ represents the *clock resets* that are executed if the edge is taken. Instead of writing $r = \{x, y\}$ we will often write $x := 0, y := 0$, which is more intuitive. The purpose of synchronization labels is explained later in Sec. 3.2.

As much of our work is implemented in UPPAAL [7], we also restrict the form of location invariants to conjunctions over clock constraints of the form

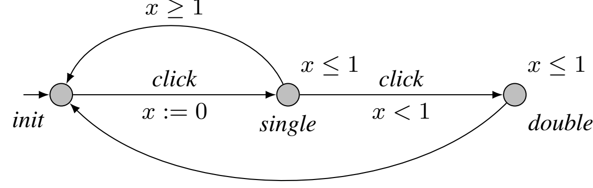


Fig. 3.1. A simple timed automaton modeling a computer mouse

$x \prec c$, where x is a clock variable, c a natural number and $\prec \in \{<, \leq\}$. For the sake of presentation we will leave out "default" labels in figures showing timed automata. The default location invariant and the default guard always evaluates to true, the default clock reset affects no clock and the default synchronization label is τ. Before we define the semantics of timed automata, we first have to introduce the notion of *clock valuation*.

Let X be a set of clocks. A clock valuation is a function that maps each clock in X to the non-negative reals $\mathbb{R}_{\geq 0}$. The set of clock valuations over X is denoted with $\mathcal{V}(X)$. Let $u, v \in \mathcal{V}(X)$ be two clock valuations and $g \in \mathcal{B}(X)$. For $d \in \mathbb{R}_{\geq 0}$, let $u + d$ denote the clock valuation that maps each $x \in X$ to $u(x) + d$. For a set $r \subseteq X$, let $[r \mapsto 0]u$ denote the clock valuation that maps all clocks $x \in r$ to zero and all other clocks $y \in X \setminus r$ to $u(y)$. The expression $u \models g$ means that the clock valuation u satisfies g. Formally, the satisfaction relation $\models$ is inductively defined as follows. Let $x, y \in X$ be two clocks, $c \in \mathbb{Z}$ be an integer constant, $u \in \mathcal{V}(X)$ and $\bowtie \in \{<, >, =, \leq, \geq\}$.

$$
\begin{array}{lll}
u \models x \bowtie c & \text{iff} & u(x) \bowtie c \\
u \models x - y \bowtie c & \text{iff} & u(x) - u(y) \bowtie c \\
u \models \varphi_1 \wedge \varphi_2 & \text{iff} & u \models \varphi_1 \text{ and } u \models \varphi_2
\end{array}
$$

Definition 3.2 (Semantics of timed automata). *Let $A = \langle L, l^0, E, \Sigma, X, I \rangle$ be a timed automaton. The semantics of a timed automaton is a labeled transition system $\mathcal{T}(A) = (S, s_0, T)$, where $S \subseteq L \times \mathcal{V}(X)$ is a set of states and $T \subseteq S \times (\mathbb{R}_{\geq 0} \cup \Sigma) \times S$ is a transition relation. A state $s = \langle l, u \rangle \in S$ is a pair consisting of an automaton location $l \in L$ and a clock valuation $u \in \mathcal{V}(X)$. The initial state of $\mathcal{T}(A)$ is $s_0 = \langle l^0, v^0 \rangle$, where v^0 is the clock valuation that assigns 0 to every clock in X. The transition relation T is defined as follows. For the sake of readability, we will write $s \xrightarrow{\lambda} s'$ for transitions $\langle s, \lambda, s' \rangle \in T$.*

1. *$\langle l, u \rangle \xrightarrow{d} \langle l, u + d \rangle \in T$ for $d \in \mathbb{R}_{\geq 0}$ if for all $d' \in \mathbb{R}_{\geq 0}$ with $0 \leq d' \leq d$ it holds that $u + d' \models I(l)$.*

2. $\langle l, u\rangle \xrightarrow{a} \langle l', u'\rangle \in T$ *if there exists an edge* $e = l \xrightarrow[r]{g,a} l' \in E$ *such that,* $u \models g$, $u' = [r \mapsto 0]u$ *and* $u' \models I(l')$.

A transition of the first type is called a *timed transition*. The automaton idles in its current state and only lets time pass. While idling, the automaton has to respect the location invariant. Transitions of the second type are called *discrete transitions*. The execution of a discrete transitions takes no time. It is only possible to take such a transition if the corresponding edge guard is satisfied.

Figure 3.2 illustrates the semantics of the automaton modeling the one-button mouse from Fig. 3.1. The chart below shows one possible behavior of the timed automaton when the user performs a double click. Here, the user clicks the mouse at absolute time points 1.7 and 2.3. The x-coordinate of the chart gives the absolute time, the y-coordinate gives the current value of x. The horizontal bold line segments in the chart show the location in which the automaton is at each point in time. Before time point 1.7, the automaton is in location *init*, after that time point, the automaton's current location is *single*. After time point 2.3 the automaton is at *double* before it is in *init* again after time point 2.7. The diagonal line segments represent the current value of the clock x. The value of x is initially 0 and constantly increases over time. At time point 1.7, when the user performs the first click, x is reset. This is because the corresponding edge from *init* to *single* is labeled with the clock reset $x := 0$. After that time point, x increases again. After pressing the mouse button a second time, the current location is *double*. This location is labeled with the invariant $x \leq 1$, which means that the automaton must leave this location before the invariant is violated. The behavior of the automaton as depicted in the figure consists of four timed transitions and three discrete transitions. Each timed transition corresponds to one of the intervals where x is growing monotonically. The discrete transitions happen at the absolute time points 1.7, 2.3 and 2.7. Recall that discrete transitions happen instantaneously.

Every parallel line to the y-axis corresponds to a state of the timed automaton. For instance, the vertical line, labeled with s in the figure, represents the following state. The intersection with the bold line gives the current location, the intersection with the diagonal lines gives the current value of the clock. Thus the vertical line s represents the state $\langle \mathit{init}, u\rangle$, where u is a clock valuation with $u(x) = 0.6$.

3.2 Timed Automata Systems

Concurrent systems consist of several parallel components. To be able to model a concurrent real-time system, several timed automata can be composed into

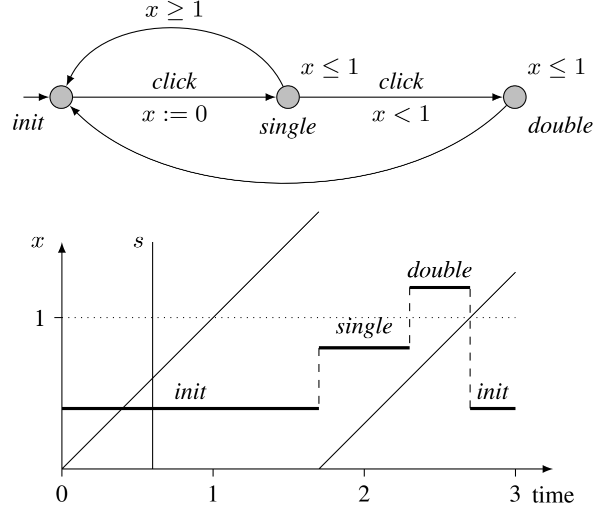


Fig. 3.2. A possible behavior of a timed automaton

an automata system. Let $N = \{1, \ldots, n\}$ and let $A_i = (L_i, l_i^0, E_i, \Sigma_i, X_i, I_i)$ be a timed automaton for $i \in N$. A timed automata system $\mathcal{S}$ is the parallel composition $A_1 \parallel \ldots \parallel A_n$. A system $\mathcal{S} = A_1 \parallel \ldots \parallel A_n$ of timed automata can also be seen as a single timed automaton, namely as the product automaton of $A_1, \ldots, A_n$. For timed automata systems, the synchronization labels play an important role. They are used for so-called hand-shake synchronization of two automata. Two automata of the system can simultaneously take two discrete transitions if the corresponding edges of the involved automata are labeled with inverse synchronization labels. From now on, we will use the following naming convention to indicate that two labels are inverse. All the names referring to synchronization labels either end with a "!" or a "?". Two labels are inverse, if their names only differ in the last sign. For instance, the two synchronization labels $a!$ and $a?$ are inverse. Alternatively, we also use $\overline{a}$ to denote the inverse synchronization label to a. Note that there is no inverse synchronization label to τ. Edges that are labeled with τ cannot synchronize, they are taken individually. Next we will give the semantics of timed automata systems.

Definition 3.3 (Semantics of timed automata systems). *Let $\mathcal{S}$ be the timed automata system $A_1 \parallel \ldots \parallel A_n$, where $A_i = (L_i, l_i^0, E_i, \Sigma_i, X_i, I_i)$ is a timed automaton for $1 \leq i \leq n$. The semantics of a timed automata system is a labeled transition system $\mathcal{T}(\mathcal{S}) = (S, s_0, T)$, where S is a set of states, $s_0 \in S$ is the initial state and T is a transition relation. States are pairs of location vectors $\overline{l} \in L_1 \times \cdots \times L_n$ and clock valuations. The initial state s_0 is $\langle (l_1^0, \ldots, l_n^0), v^0 \rangle$, where $v^0 : \bigcup_{i \in N} X_i \rightarrow \mathbb{R}_{\geq 0}$ is the clock valuation that maps every clock to 0.*

For a location vector $\bar{l}$ *we denote the location vector where the* i*th component is replaced with* l'_i *with* $\bar{l}[l'_i/l_i]$*. By* $I(\bar{l})$*, we denote the composed invariant function* $\bigwedge_{i=1}^{n} I_i(l_i)$*. The transition relation* T *is defined as follows.*

1. $\langle \bar{l}, u \rangle \xrightarrow{d} \langle \bar{l}, u + d \rangle \in T$ *if* $u + d' \models I(\bar{l})$ *for all* $0 \leq d' \leq d$*.*
2. $\langle \bar{l}, u \rangle \xrightarrow{\tau} \langle \bar{l}[l'_i/l_i], u' \rangle \in T$ *if* $l_i \xrightarrow[r]{g,\tau} l'_i \in E_i$*,* $u \models g$*,* $u' = [r \mapsto 0]u$ *and* $u' \models I(\bar{l}[l'_i/l_i])$*.*
3. $\langle \bar{l}, u \rangle \xrightarrow{\tau} \langle \bar{l}[l'_i/l_i][l'_j/l_j], u' \rangle$ *if there exists* $i, j \in N$ *with* $i \neq j$ *such that*
   i) $l_i \xrightarrow[r_i]{g_i,a} l'_i \in E_i$*,* $l_j \xrightarrow[r_j]{g_j,\overline{a}} l'_j \in E_j$ *and* $u \models g_i \wedge g_j$ *and*
   ii) $u' = [r_i \cup r_j \mapsto 0]u$ *and* $u' \models I(\bar{l}[l'_i/l_i][l'_j/l_j])$*.*

Again we refer to transitions of the first type as timed transitions. Transitions of the two latter types are called discrete transitions. As the semantics of timed transitions and the first type of discrete transitions are defined as for single timed automata, we will only comment on the second type of discrete transitions. Since these transitions are induced by two edges with inverse synchronization labels, we will call these *synchronized transitions*. The purpose of synchronization is to restrict the behavior of the system. Revisit the automaton from Fig. 3.1. Suppose this automaton is part of a system that additionally consists of an automaton with a single location and a self-loop edge, which is labeled with the inverse synchronization label to *click*. In this example the use of synchronization labels ensures that the mouse automaton can only proceed from *init* to *single* and from *single* to *double*, if the automaton modeling the user simultaneously performs a discrete transition.

We conclude this section with the following definition. Let A be a timed automaton and $\mathcal{T}(A) = (S, s_0, T)$ the transition system representing A's semantics. A *trace* of A is a, possibly infinite, alternating sequence of states and transitions

$$\pi = s_0 \xrightarrow{\lambda_0} s_1 \xrightarrow{\lambda_1} \ldots \xrightarrow{\lambda_{i-1}} s_i \xrightarrow{\lambda_i} \ldots,$$

where $s_i \in S$ and $\lambda_i \in \mathbb{R}_{\geq 0} \cup \{\tau\}$ for $i \geq 0$. The label λ_i is the real number d if the corresponding transition $s_{i-1} \xrightarrow{\lambda_i} s_i \in T$ is a timed transition with duration d. If the corresponding transition is a discrete transition, then λ_i is τ. We call a state $s \in S$ reachable if there is a trace π for which $s_i = s$ holds for at least one $i \geq 0$.

3.3 The Region Automaton

As the values of clocks are real numbers, the transition system induced by the semantics of a timed automaton is *infinite*. Model checking is a verification tech-

nique for *finite* transition systems, therefore it seems that model checking is not appropriate for timed automata. The reason why model checking timed automata is decidable is due to the fact that there exists an exact, finite abstraction of timed automata. Alur and Dill [3] first showed that the infinite state space of timed automata can be finitely partitioned. This is widely regarded as the major break-through in the verification of timed systems.

The main idea behind this finite partitioning is the following. Given a timed automata system, it is not important to know the exact value of a clock variable, as long as it is possible to determine, for all clock constraints that occur in the system, whether the clock values satisfy them or not. As a consequence thereof, system states with the same discrete part, i. e., states with the same location vector, can be grouped together, if their clock valuations satisfy the same clock constraints. Such a group of states is called a symbolic state.

Let $\mathcal{S}$ be a system of timed automata and let X be the set of all clocks that appear in $\mathcal{S}$. The function $k : X \to \mathbb{Z}$ that maps each $x \in X$ to the largest absolute value $|c_x|$, such that $x \bowtie c_x$ or $x - y \bowtie c_x$ is a clock constraint that either is a subformula of some edge guard or some location invariant, is called the *clock ceiling* of $\mathcal{S}$. Clock ceilings play an important role in the finite partitioning of state spaces of timed automata. We will see this in the next definition.

Definition 3.4 (Region equivalence). *Let X be a non-empty set of clocks. Further, for $d \in \mathbb{R}_{\geq 0}$, let $\{d\}$ denote the fractional part of d and $\lfloor d \rfloor$ denote the integer part of d. Two clock valuations u and v are region equivalent (denoted by $u \sim_k v$) iff the following conditions hold for every pair of clocks $x, y \in X$:*

1. $u(x) > k(x)$ *iff* $v(x) > k(x)$
2. *if* $u(x) \leq k(x)$ *then*
   a) $\lfloor u(x) \rfloor = \lfloor v(x) \rfloor$ *and*
   b) $\{u(x)\} = 0$ *iff* $\{v(x)\} = 0$
3. *if* $u(x) \leq k(x)$ *and* $u(y) \leq k(y)$ *then*
   $\{u(x)\} \leq \{u(y)\}$ *iff* $\{v(x)\} \leq \{v(y)\}$

It is not difficult to prove that region equivalence is an equivalence relation. An equivalence class $[u]_k$ induced by $\sim_k$ is the set of all clock valuations that are region equivalent to u. These equivalence classes are called *clock regions*. The idea behind this partitioning is the following. First, if the value of a clock is greater than its ceiling, it does not matter how large this value is. Second, under certain conditions, the fraction of a clock value is not important, because clocks are only compared to integers. Figure 3.3 illustrates this. Let $X = \{x, y\}$ be a set of two clocks and let the clock ceilings $k(x) = 3$ and $k(y) = 2$. The figure shows the clock regions for this configuration. It consists of the following 60

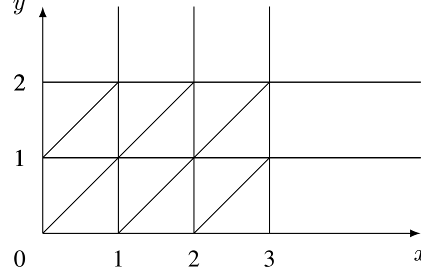


Fig. 3.3. Regions for a system with two clocks

clock regions: 18 open regions, e. g., $0 < x < y < 1$, 30 open line segments, e. g., $1 < x = y < 2$ and 12 intersection points, e. g., $x = y = 1$.

Note that, for a finite number of clocks, there is only a finite number of equivalence classes. Alur and Dill [3] proved that the number of clock regions for a set of n clocks is exponential in the number of clocks and the maximum clock ceilings.

With the notion of clock regions, we can now define the region automaton. Given a timed automaton A, the corresponding region automaton is a *finite* transition system that behaves like A. More precisely, the region automaton is a sound and complete symbolic representation of the semantics of A.

Definition 3.5 (Region automaton). *Let $A = \langle L, l^0, E, \Sigma, X, I\rangle$ be a timed automaton. The region automaton $\mathcal{R}(A) = (Q, q_0, \Delta)$ is a transition system, where Q is a finite set of states. The states are pairs of automaton locations and clock regions. The initial state is $q_0 = \langle l^0, [v^0]\rangle$, where $v^0 \in \mathcal{V}(X)$ is the clock valuation that maps every clock in X to 0. Let $u, v \in \mathcal{V}(X)$ be clock valuations. Further, let $\mathcal{T}(A) = (S, s_0, T)$ be the labeled transition system that represents the semantics of A. The transition relation Δ is defined as follows.*

1. $\langle l, [u]\rangle \to \langle l, [v]\rangle \in \Delta$ *if* $\langle l, u\rangle \xrightarrow{d} \langle l, v\rangle \in T$ *for* $d \in \mathbb{R}_{\geq 0}$
2. $\langle l, [u]\rangle \to \langle l', [u]\rangle \in \Delta$ *if* $\langle l, u\rangle \xrightarrow{a} \langle l', u\rangle \in T$

To illustrate the semantics of region automata, consider Fig. 3.4. The figure shows the region automaton of the timed automaton model of the computer mouse from Fig. 3.1. The initial state of the region automaton is $\langle \mathit{init}, x = 0\rangle$, states that are not reachable from that state are not shown.

Let A be a timed automaton and $\mathcal{R}(A) = (Q, q_0, \Delta)$ its region automaton. Further let $\mathcal{T}(A) = (S, s_0, T)$ be the transition system that represents A's semantics. As already stated, the region automaton is a finite representation of the semantics of A. Alur et al. [1] proved that $\mathcal{R}(A)$ and $\mathcal{T}(A)$ are *bisimilar*. This means, if we want to check if $\mathcal{T}(A), s_0 \models \varphi$, where φ is a CTL formula, it is

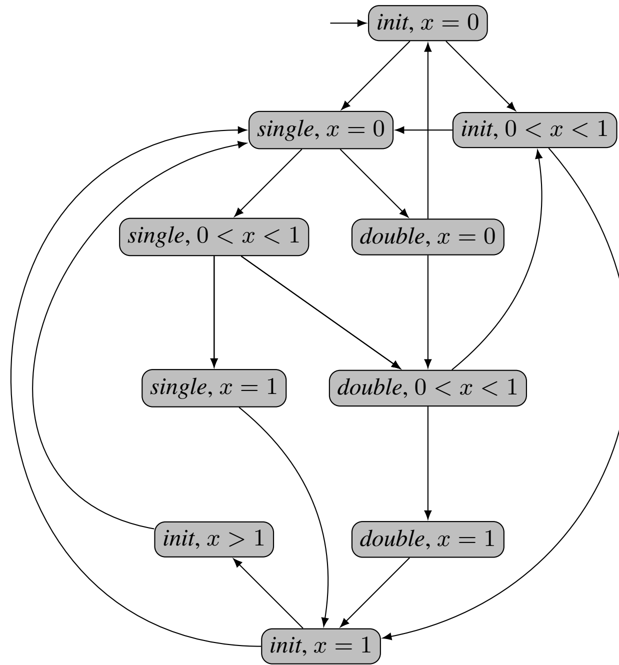


Fig. 3.4. The region automaton for the automaton from Fig. 3.1

sufficient to check if $\mathcal{R}(A), q_0 \models \varphi$ and vice versa. As the number of states in the region automaton is finite, the model checking problem for timed automata is decidable, too. Alur et al. [1] proved that the model checking problem for timed automata is **PSPACE** complete.

3.4 Our Formalism

Clock regions are a useful means to partition the infinite state space of timed automata, but they are not efficient. For instance, suppose that every constant that appears in the description of the timed automaton model of the one button mouse (see Fig. 3.1) is multiplied by 10. This causes the region automaton to consist of 64 states (compared to 10 states).

A more efficient representation of the continuous part of timed automata states is based on the notion of *zones*. Let X be a set of clocks and $g \in \mathcal{B}(X)$ be a conjunction of clock constraints. A zone Z is the maximal set of clock valuations that satisfies g, i. e., $Z = \{u \in \mathcal{V}(X) \mid u \models g\}$. As every conjunction of clock constraints induces a zone, we identify such conjunctions with zones. From the definition of zones it follows that a zone corresponds to a set of clock regions. Hence, the induced *zone automaton* is in most cases much coarser than

the corresponding region automaton and thus states are represented more compactly. This computation model is used in the UPPAAL and the MCTA model checkers and is also the fundamental computation model of this thesis.

It is well-known that zones can be efficiently represented using a data structure which is called *difference bound matrices* [37]. We do not want to describe this data structure in detail here, since understanding the details is not necessary for this thesis. In a nutshell, difference bound matrices can be used to represent and manipulate zones. There is also a canonical, i. e., unique, representation of zones, which is convenient, because the same conjunction of clock constraints can be represented by different zones. For an algorithmic introduction to difference bound matrices, the interested reader is referred to the paper of Behrmann et al. [6].

3.4.1 Bounded Integer Variables

Before we give the formal definition of the zone automaton, let us mention that the timed automata systems that we consider in this thesis also feature bounded integer variables. Every automaton A_i of a timed automata system $\mathcal{S}$ is augmented with a finite set of such variables V_i. A state of a timed automata system is a triple $\langle \bar{l}, \bar{v}, Z \rangle$, where $\bar{l}$ is a location vector, $\bar{v}$ is a variable value vector and Z is the zone of the state. Every integer variable has an initial value. The values of integer variables can be used to restrict the behavior of the automaton. Therefore, edges can be annotated with *integer effects* and *integer guards*. In the following, we define the syntax and the semantics thereof. Let V be a finite set of bounded integer variables. An *integer assignment* is an expression of the form $v := c_0 + \sum_{i=1}^{n} c_i \cdot v_i$, where $c_i \in \mathbb{Z}$ and $v, v_i \in V$. An integer effect is a set of integer assignments, where each integer variable v occurs on the left hand side of at most one assignment. An *integer constraint* is a comparison of the form $c_0 \bowtie \sum_{i=1}^{n} c_i \cdot v_i$, where $c_i \in \mathbb{Z}, v_i \in V$ and $\bowtie \in \{<, >, =, \neq, \leq, \geq\}$. An integer guard is a conjunction over such constraints.

The semantics is defined as obvious: an edge $l \xrightarrow[f,r]{g_I \wedge g_X, a} l'$ can be taken if both its integer guard g_I and its clock guard g_X are satisfied. By taking the edge, the values of the clocks and integers are changed according to the clock resets r and the integer effect f.

3.4.2 The Zone Automaton

As already mentioned, zone automata are the fundamental computation model used in this thesis. In this section we formally define this notion. Actually, the definition of zone automata is much like the definition of region automata but hard to read. Therefore we also give a detailed discussion afterwards.

Definition 3.6 (Zone Automaton). *Let $\mathcal{S} = A_1 \parallel \ldots \parallel A_n$ be a timed automata system where $A_i = \langle L_i, l_i^0, E_i, \Sigma_i, X_i, V_i, I_i\rangle$ for $1 \leq i \leq n$. The zone automaton $\mathcal{Z}(\mathcal{S}) = (S, s_0, T)$ is a transition system, where S is a finite set of states. A state is a tuple, consisting of a location vector, a variable value vector and a zone. The initial state is $s_0 = \langle \bar{l}_0, \overline{v}_0, Z_0\rangle$, where $\bar{l}_0$ is the vector consisting of the initial locations, $\overline{v}_0$ is the vector containing the initial variable values and $Z_0 = I(\bar{l}_0) \wedge \bigwedge_{x \in X} x \geq 0$. Further, let $\mathcal{T}(A) = (Q, q_0, R)$ be a labeled transition system that represents the semantics of $\mathcal{S}$. The transition relation T is defined as follows.*

1. $\langle \bar{l}, \overline{v}, Z\rangle \rightarrow \langle \bar{l}, \overline{v}, \|up(Z \wedge I(\bar{l})) \wedge I(\bar{l})\|_{\mathcal{S}}\rangle \in T$
   if $\langle \bar{l}, \overline{v}, u\rangle \xrightarrow{\delta} \langle \bar{l}, \overline{v}, u + \delta\rangle \in R$ *for* $\delta \in \mathbb{R}_{\geq 0}$.
2. $\langle \bar{l}, \overline{v}, Z\rangle \rightarrow \langle \bar{l}', f_i(\overline{v}), r_i(g_i \wedge Z \wedge I(\bar{l})) \wedge I(\bar{l}')\rangle \in T$ *with* $\bar{l}' = \bar{l}[l_i'/l_i]$
   if there exists $\langle \bar{l}, \overline{v}, u\rangle \xrightarrow{\tau} \langle \bar{l}[l_i'/l_i], f_i(\overline{v}), u'\rangle \in R$,
   induced by $l_i \xrightarrow[f_i, r_i]{g_i, \tau} l_i' \in E_i$ *for* $i \in \{1, \ldots, n\}$.
3. $\langle \bar{l}, \overline{v}, Z\rangle \rightarrow \langle \bar{l}', f_i(f_j(\overline{v})), r_i(r_j(g_i \wedge g_j \wedge Z \wedge I(\bar{l}))) \wedge I(\bar{l}')\rangle \in T$
   with $\bar{l}' = \bar{l}[l_i'/l_i][l_j'/l_j]$
   if there exists $\langle \bar{l}, \overline{v}, u\rangle \xrightarrow{\tau} \langle \bar{l}[l_i'/l_i][l_j'/l_j], f_i(f_j(\overline{v}), u'\rangle \in R$,
   induced by $l_i \xrightarrow[f_i, r_i]{g_i, a} l_i' \in E_i$ *and* $l_j \xrightarrow[f_j, r_j]{g_j, \overline{a}} l_j' \in E_j$
   for $i, j \in \{1, \ldots, n\}$ *with* $i \neq j$.

   We additionally require that integer variables affected by f_i are not affected by f_j and vice versa.

For a zone Z, $\mathrm{up}(Z)$ is defined as $\{u + d \mid u \in Z, d \in \mathbb{R}_{\geq 0}\}$, $\|Z\|_{\mathcal{S}}$ normalizes Z with respect to the clock ceilings of the timed automaton system $\mathcal{S}$ and $r(Z)$ is an abbreviation for $\{[r \mapsto 0]u \mid u \in Z\}$.

Again, the first type of transitions is referred to as timed transitions. A timed successor s' of a state s only differs from s in the zone. The zone of the successor state has to respect the invariant $I(s)$. The resulting zone is then extended by replacing all clock valuations u with $u + d$, where d is a non-negative real number. Again, the resulting zone has to respect $I(s)$. Afterwards the zone is normalized. The normalization is necessary, otherwise the number of zones can be unbounded. For a more detailed discussion, we again refer to the paper of Behrmann et al. [6].

There are two types of discrete successors, successor states reached via τ transitions, and successors reached via synchronized transitions. Let us have a closer look at discrete successor states, reached via a τ transition induced by the

edge $e = l_i \xrightarrow[f_i,r_i]{g_i,\tau} l'_i \in E_i$. We assume that e is applicable in the current state. The location vectors of the successor state are obtained by replacing the source location l_i with the target location l'_i of the edge. The variable value vector of the successor state is obtained by applying the edge's integer effect f_i to the variable value vector of the predecessor state. To apply the transition, the zone of s has to respect the edge's guard and the invariant $I(s)$. Afterwards the clock resets are applied. The resulting zone then has to satisfy the invariant of the successor state $I(s')$.

Let us finally give some remarks concerning discrete successor of two synchronized edges $e_i = l_i \xrightarrow[f_i,r_i]{g_i,a} l'_i \in E_i$ and $e_j = l_j \xrightarrow[f_j,r_j]{g_j,\overline{a}} l'_j \in E_j$. Again, we assume that e_i and e_j are applicable in the current state. The successor state is obtained by applying the effects of both edges to the current state. To rule out ambiguity, we require that, regardless of the order in which the edges e_i and e_j are applied, the successor state is unique. Therefore every integer variable that occurs on the left hand side of an integer assignment of e_i must not occur in any integer assignment of e_j and vice versa.

3.4.3 Reachability Analysis for Timed Automata

We conclude this chapter by providing an algorithm for deciding reachability for timed automata systems. The algorithm from Fig. 3.5 takes as input a system of timed automata $\mathcal{S}$ and a formula φ, where φ is conjunction over location predicates, integer constraints and clock constraints. A location predicate is an expression of the form $A_i = l$ and evaluates to true in a state s if A_i's current location is l. The zone of the initial state is $Z_0 = (\bigwedge_{x \in X} x \geq 0) \wedge I(\overline{l_0})$. In line 9, the algorithm checks whether the current state was already explored, i. e., it checks if there is an already explored state that subsumes the current state. For instance, a state $\langle \bar{l}, \overline{v}, Z \rangle$ is already explored, if there is a state $\langle \bar{l}, \overline{v}, Z' \rangle$ in the closed list with $Z \subseteq Z'$. For example, this is the case if Z is induced by $x \leq 2 \wedge y \geq 3$ and $Z' = \{u \in \mathcal{V}(X) \mid u \models x \leq 3 \wedge y \geq 1\}$. The *succ*$(s)$ function returns the symbolic successor states for the state s. In the algorithm, s is of the form $\langle \bar{l}, \overline{v}, Z \rangle$.

The successor generation function *succ* operates as follows. Given a state $s = \langle \bar{l}, \overline{v}, Z \rangle$, the function first computes all discrete successor states, i. e., all states that are reached by applying one discrete transition to s. Afterwards, it computes for every such discrete successor the time successor state.

```
1  function reachability(𝒮, φ):
2      open = ∅
3      closed = ∅
4      open.insert(s_0)
5      while open ≠ ∅ do:
6          s = ⟨l̄, v̄, Z⟩ = open.pop()
7          if s ⊨ φ then:
8              return True
9          if Z ⊈ Z'' for all s'' = ⟨l̄, v̄, Z''⟩ ∈ closed then:
10             closed = closed ∪{s}
11             for each s' ∈ succ(s) do:
12                 if s' ∉ closed then:
13                     open.insert(s')
14     return False
```

Fig. 3.5. Reachability analysis for timed automata

4

Directed Model Checking for Real-time Systems

In this chapter we introduce the research area in which this thesis is settled, namely directed model checking. After a general introduction to this area, we will discuss directed model checking for timed automata. Directed model checking can be seen as a special technique for the verification of safety properties, especially when the state space of the system at hand is too huge to be enumerated exhaustively. In directed model checking the state space traversal is focused, in some sense, on those parts of the search space that show promise to contain reachable error states. This dramatically increases the chance that an error state can be found before the memory and time resources are exhausted.

4.1 The General Idea

When model checking safety properties, the ultimate goal is to prove the absence of error states. However, to do so one has to explore the entire state space of the system under consideration. It is therefore essential to use an efficient representation and implementation of its state space. Prominent examples of such implementations are the SPIN [64] and the UPPAAL [7] model checkers. SPIN handles the Promela language, describing systems of communicating processes. UPPAAL handles timed automata systems.

Due to the state explosion problem, enumerating the entire state space is often not feasible in practice. A potentially much easier task is to only try to *detect* error states, i. e., to *falsify* the safety property. This is easier, because an error state may be found by exploring only a small fraction of the entire search space. Research on directed model checking deals with the development of algorithms that accelerate the detection of states violating a given safety property. As a consequence, directed model checking algorithms are especially tailored to debugging purposes. They can even be good for proving an application error-free,

because such algorithms can handle the intermediate iterations in the abstraction refinement life cycle, i. e., those iterations in which spurious error states exist.

In order to quickly detect error states, two main issues have to be addressed: first, the *search space size*, i. e., the number of search states that need to be considered before an error state is found; and second, the length of the detected *trace* to the error state. The search space size determines the scalability of the search. Short error traces are preferred, because for debugging, they are easier to understand; in abstraction refinement, they provide better information about what aspects of the abstraction should be refined. Ideally, one wants an *optimal*, i. e., a shortest possible, error trace.

4.2 Directed Model Checking

As already mentioned, directed model checking is the application of heuristic search to model checking. In Artificial Intelligence (AI), especially in AI planning, heuristic search has been overwhelmingly successful in the past decade, in particular winning all the satisficing planning competitions (e. g. [20, 49, 56, 61, 89]). Heuristic search addresses *both*, the number of explored search states and the error trace length by influencing the *order* in which the search states are explored. The application of heuristic search to model checking was pioneered a few years ago by Edelkamp et al. [42, 43], christening this research direction *directed model checking*.

In directed model checking, the search for a particular state that satisfies some property, e. g. a state that violates a safety property, is guided with a *heuristic function*. A heuristic function h is a function that maps states to integers, estimating the state's distance to a nearest error state. The search then gives preference to states with lower h value. There are many different ways of doing the latter, of which we consider the wide-spread methods A^* [53, 54] and *greedy search* (cf. [82]). In the former, search nodes s are explored by increasing value of $c(s) + h(s)$, where $c(s)$ is the length of the search path on that s was reached. If h is *admissible*, i. e., if it never overestimates the real distance to a nearest error state, then A^* is guaranteed to return a shortest possible error trace. In greedy search, search nodes are explored by increasing value of $h(s)$. This gives no guarantee on the length of the detected error trace, but tends to explore fewer search states in practice.

Before we give a directed model checking algorithm, we first need the notion of reachability problem.

Definition 4.1 (Reachability problem). *Let $\mathcal{S} = \langle S, s_0, T\rangle$ be a transition system, where S is a set of states, $s_0 \in S$ is the initial state of the system and*

$T \subseteq S \times S$ is a transition relation. A reachability problem is given by a tuple $\langle \mathcal{S}, \varphi \rangle$, where φ is a propositional formula, the so-called target formula*. A reachability problem is the problem to decide if there is a solution, i. e., a trace starting from s_0 to a state $s \in S$ which satisfies φ.*

Model checking invariants can also be stated as a reachability problem. Here, one wants to prove that there is no reachable error state, i. e., a state that violates an invariant φ. In CTL this is given by $\mathcal{S}, s_0 \models \forall\Box\varphi$. This is equivalent to $\mathcal{S}, s_0 \not\models \exists\Diamond\neg\varphi$, which is a reachability problem. A solution in this context is called a *counterexample* or an *error trace*.

4.2.1 A Basic Directed Model Checking Algorithm

Figure 4.1 shows a basic directed model checking algorithm. Given a reachability problem $\langle \mathcal{S}, \varphi \rangle$ and a heuristic function h, the algorithm returns True if there is a state that satisfies φ, otherwise it returns False. The initial state of $\mathcal{S}$ is s_0. The algorithm maintains a priority queue, called open, which contains visited but not yet explored states. When open.getMinimum is called, open returns a minimum element, i. e., one of its elements with minimal priority value. States that have been expanded are stored in closed. Every state encountered during search is first checked if it is an error state. If this is not the case, its successors are computed. Every successor that has not been visited before is inserted into open according to its priority value. The evaluate function depends on the applied version of directed model checking, i. e., if applied with A^* or greedy search. For A^*, evaluate(s, h) returns $h(s) + c(s)$, where $c(s)$ is the length of the path on which s was reached for the first time. For greedy search, it simply evaluates to $h(s)$. When every successor has been computed and prioritized, the process continues with the next state from the open queue with lowest priority value. Every state stores information about how it has been reached, i. e., its immediate predecessor state and transition. Therefore, if an error state s is finally reached, the corresponding error trace is generated by tracing back from s.

The remaining question in this context is, how do we obtain the main ingredient for directed model checking, namely a heuristic function.

4.2.2 Obtaining Heuristic Functions

Our approach to directed model checking has its origins in the area of AI planning. The heuristics presented in this thesis are based on what AI people call a *relaxation*, which is the same as the model checking term *abstraction*: an overapproximation. However, the usage of abstractions in AI planning and directed model checking differs from how they are usually used in model checking. For

```
 1 function dmc(S, φ, h):
 2     open = empty priority queue
 3     closed = ∅
 4     priority = evaluate(s_0, h)
 5     open.insert(s_0, priority)
 6     while open ≠ ∅ do:
 7         s = open.getMinimum()
 8         if s ⊨ φ then:
 9             return True
10         if s ∉ closed then:
11             closed = closed ∪ {s}
12             for each s' ∈ succs(s) do:
13                 if s' ∉ closed then:
14                     priority = evaluate(s', h)
15                     open.insert(s', priority)
16     return False
```

Fig. 4.1. A basic directed model checking algorithm

instance, in abstraction refinement (cf. Sec. 2.3.1), abstractions are used to prove the absence of reachable error states. In directed model checking, the abstracted problem is used to approximate real error distances: the heuristic value for a state is obtained by solving an abstract reachability problem and taking the length of the abstract solution as the heuristic estimate for that state. This has to be done for each state that is encountered during the state space traversal. To be able to solve such an abstract problem in every search state, the granularity of the abstraction has to be chosen very carefully. On the one hand, it is desirable to have heuristic functions that are as informative as possible. On the other hand, the computation must not be too expensive. In this theses, the abstract problems are *fully* automatically generated, based on the declarative description of the original reachability problem. As a consequence, our directed model checking approaches are also fully automatic.

4.3 Related Work on Directed Model Checking

Directed model checking is a by now well-established technique that has found its way in many state-of-the-art tools like UPPAAL (UPPAAL/DMC [69]), SPIN (HSF-SPIN [43]), or JAVA PATHFINDER [94]. While it is obvious that model checking, i. e., the state space traversal, can be profitably guided ("directed") based on specific criteria, it is perhaps less obvious whether this is still true for criteria that are extracted automatically from the model itself. Research on directed model checking is exactly about that question.

The application of heuristic search to model checking was pioneered a few years ago by Yang and Dill [97] and Edelkamp et al. [42, 43]. Although

Edelkamp et al.'s work was not the first contribution to this new research direction, the term *directed model checking* was actually coined by them. Today, there are various other approaches of this sort. In the remainder of this section, we will review some of them. The main difference between all these approaches is how they define and compute the heuristic function. Different definitions make all the difference because no heuristic can work well in *all* examples. The best one can hope to do is to define a range of heuristics that cover (work well in) an as large as possible range of examples.

4.3.1 Approaches for Untimed Systems

Heuristics Based on Graph-distance

Edelkamp et al. [42, 43] work in the context of SPIN. They propose to base the distance estimation on the graph-distances within each single automaton. Their algorithm is tailored to location reachability. Let S be a system of n parallel automata $A_1 \parallel \ldots \parallel A_n$, let $J \subseteq \{1, \ldots, n\}$ and let $\varphi = \exists\Diamond(\bigwedge_{j \in J} A_j = l_{j'})$ be a target formula. The locations $l_{j'}$ that occur in φ are called *target locations*. To validate the safety property φ, Edelkamp et al. propose the following heuristic. For a system state s, let $d_{A_j}(s)$ be the distance of A_j's current location to its target location. If A_j has no target location, i. e., $A_j = l_{j'}$ is not a subformula of φ, then $d_{A_j}(s)$ is set to 0. They define an admissible heuristic function as $\max_i d_{A_j}(s)$, where s is a system state. A non-admissible heuristic function is defined as $\sum_i d_{A_j}(s)$. These heuristics are then used in A^* and iterative deepening A^*. Note that both heuristics are rather crude approximations of the system semantics. But nevertheless they reported good results on a number of benchmarks.

Guiding Based on Hints and Target Enlargement

In 1998, Yang and Dill [97] proposed several heuristics for directed model checking which they implemented in MURφ++. This is probably the first application of heuristic search to model checking. The authors did not call it directed model checking, but *prioritized model checking*. As far as we know this term is nearly never used in the literature. Yang and Dill introduced three heuristics: two user-definable heuristics and one automatically generated heuristic, namely the *Hamming distance* heuristic [52]. This heuristic is based on the bit string representation of states. The heuristic value of a state is the minimum number of bits that need to be flipped in order to turn it into a state violating the invariant. The hamming distance heuristic is based on a very crude approximation of the system and is not very discriminating. In many cases, greedy search with this heuristic behaves just like ordinary breadth-first search.

The second heuristic is based on *target enlargement*. Target enlargement, as the name suggests, aims at increasing the set of error states and thus makes it easier to reach one. This is done prior to search, by iteratively computing the preimage of the set of error states, i. e., the set of their predecessors. The authors propose to overapproximate target enlargement by abstracting the preimage operator. This abstraction is based on ignoring a set of user defined system variables. During search, whenever a state is encountered that is contained in this enlarged target, the number of abstract preimage computations to reach this state is taken as the heuristic estimate. The approximation of the preimage operator is up to the user, however the authors give a guideline that often yields a useful abstraction. Note that this heuristic can also be seen as a *pattern database heuristic*. We will discuss pattern database heuristics below.

The third heuristic is called *guideposts*. To guide the search, the user has to define a set of guideposts. A guidepost is a condition that is interesting or a required precondition in order to reach an assertion violation. During search, states whose history has visited more guidepost are preferred. It is interesting to see that guideposts are similar to landmarks in AI planning [62]. The main difference of these approaches is that in the latter landmarks have to be visited in a particular order before a goal state is reached, while Yang and Dill just use it as a heuristic estimate.

Structural Heuristics

Groce and Visser [51] introduce two heuristics inspired by the area of testing. These heuristics are integrated in JAVA PATHFINDER, a tool for model checking multi-threaded Java programs [94]. The heuristics do not try to target an error formula but instead drive the search based on a branch covering metric and on structural properties like the branching structure of the program or thread interdependency.

They propose to use an amalgam of best first search and beam search in order to detect errors. They report that their search algorithm can find deep error states that can only be found by beam search with a very small queue limit.

Their first heuristic is based on a metric for code coverage. In the area of testing a huge number of different code covering metrics have been proposed. The authors follow a branch coverage metric that requires that at every branching point in the program all possible branches are visited at least once. The intuition behind this metric is that the higher the code coverage is, the higher is the confidence that the code is correct.

During search, states covering yet unexplored branches receive the highest priority. States that cover an already visited branch receive a priority reflecting the number of times this branch was visited before. The resulting heuristic is not admissible and is reported to get easily stuck in local minima.

The second heuristic exploits thread interleavings. It has turned out that testing metrics based on this are especially useful in detecting subtle deadlocks. Search prioritizes states that are reached on execution traces that involve many thread switching operations.

Pattern Database Heuristics

Qian and Nymeyer [86] introduced the use of *pattern database heuristics* [29] based on abstractions generated by ignoring some of the variables of the system variables. Roughly spoken, a pattern database contains the entire state space of an abstraction of the system under consideration. The heuristic value of a state is the minimum distance of the corresponding abstract state to an abstract error state. Typically, pattern databases are precomputed and stored in hashtables. Qian and Nymeyer work in the context of symbolic invariant model checking and have implemented their techniques in NuSMV [22]. They also provide a technique to automatically derive abstractions of the system at hand [87]. The technique is based on the cone of influence abstraction. Starting with the set of variables that occur in the property, the algorithm iteratively increases this set by all variables that directly influence the already selected variables. The number of iterations is an input parameter of the heuristic.

4.3.2 Approaches for Timed Systems

When model checking timed automata systems, one has to deal with two sources of state explosion. The first source is due to the *discrete* part of the system, i. e., the number of system states is exponential in the number of parallel automata and the number of variables. The second source stems from the *continuous* part of the system. Since the clock variables range over the infinite domain of non-negative reals, the size of the continuous part is also infinite. Even for the zone automaton, the number of zones is still exponential in the number of clock variables. So far, there is no verification method for real-time systems that scales in both the number of parallel automata and the number of clock variables. Clarke regards efficient model checking for timed automata systems as one of the most important challenges in model checking [23].

The aim of subproject R3 of the AVACS project [1] is to tackle this problem by combining abstraction techniques and directed model checking. While directed model checking mainly tackles the state explosion coming from the discrete part of the system, abstraction is a suitable technique to deal with the second source, i. e., the continuous part. Our work in this project is focused on directed model

[1] Automatic Verification and Analysis of Complex Systems (AVACS) is a transregional collaborative research center funded by the German Research Foundation (DFG), see http://www.avacs.org for more information.

checking for real-time systems. As far as we are aware of, Dräger et al.'s work [39] and our own work [70] are the first applications of directed model checking with automatically generated heuristics to real-time systems. Both papers were published in 2006.

User Definable Heuristics

UPPAAL CORA [8, 75] is an extension of UPPAAL that supports *priced timed automata* [9]. In a nutshell, a priced timed automaton is a timed automaton whose locations and transitions are additionally labeled with prices. The total costs of an execution trace is the sum of the transition costs plus the costs for idling in the visited locations of the trace. UPPAAL CORA offers various mechanisms for guiding and pruning the search for optimal reachability in terms of accumulated costs. To achieve this, the UPPAAL input language is extended by two special user defined functions *heur* and *remaining*. The first function is used to assign priorities to search states. The search then gives preference to states with lower priorities. The *remaining* function estimates the remaining costs of reaching a solution. The *remaining* function is used in order to prune the search space: if the current costs of a state plus its remaining costs are greater than the total costs of the best known solution, than this state is pruned from the search. Note that, in the definition of these functions, it is only possible to refer to system variables.

Behrmann et al. [8], Larsen et al. [75] and Dierks et al. [34] achieved good results in applications for which they hand-coded the heuristics. The basic common idea of these contributions is to transform either scheduling problems or planning tasks to a reachability problem that can then be solved using UPPAAL CORA. Neither of them provides an automatic method to compute heuristic values. To manually design a good heuristic is a tedious and time-consuming job. A deep understanding of the system at hand is absolutely necessary to carefully tune the heuristic.

Heuristics Based on Automata-theoretic Abstractions

Dräger et al. [39, 40] developed a heuristic that aims at a close representation of the process synchronization required to reach the error. Each process is represented as a finite-state automaton. The heuristic h^{aa} estimates the error distance $d(s)$ of a system state s as the error distance of the corresponding abstract state $\alpha(s)$ in an abstraction that approximates the full product of all process automata.

The approximation of the product of a set of automata is computed incrementally by repeatedly selecting two automata from the current set and replacing them with an abstraction of their product. To avoid state space explosion, the size of these intermediate abstractions is limited by a preset bound N: to

reach a reduction to N states, the abstraction first merges bisimilar states and then states whose error distance is already high in the partial product.

In this way, the precision of the heuristic is guaranteed to be high in close proximity to the error, and can, by setting N, be fine-tuned for states further away from the error. Dräger et al. experimentally found out that fairly low values of N, such as $N = 50$ or $N = 100$, already significantly speed up the search for the error, and therefore represent a good trade-off between cost and precision. Recently this approach has been successfully adapted to the area of AI planning by Helmert et al. [57].

Heuristics Based on Predicate Abstraction

Recently Smaus and Hoffmann [91] continued our work on using predicate abstraction to generate heuristic functions in UPPAAL [63]. In our work [63], it has been investigated how to construct pattern database heuristics using predicate abstraction. A pattern database heuristic is typically computed in a preprocessing step.

Therefore, the entire state space of an abstraction of the original system is explored in a backward manner. During search, the abstract state space serves as a lookup table for the heuristic values, i. e., the heuristic value of a state s is the length of an abstract error trace that starts with an abstract state that corresponds to s. In both papers [63, 91], the abstract state space is obtained by applying a predicate abstraction to the original system. The main difference of these two papers is, how a set of predicates is derived that results in a well-informed pattern database heuristic.

The generation of predicates in our work was mainly syntactical, i. e., they were directly obtained from the textual representation of the system. Smaus and Hoffmann use abstraction refinement to refine initial abstractions that are additionally seeded with randomly generated predicates. The use of abstraction refinement in their work differs from the traditional use in verification. Instead of excluding all possible error traces, they aim at getting a small set of predicates that is informative enough to closely describe the original state space in the surrounding of error states. Therefore they base a refinement step on several abstract error paths, starting at arbitrary abstract states.

4.4 Our Model Checking Tools

We developed two directed model checking tools for timed automata. To the best of our knowledge, these tools are the only tools of that kind that are equipped with automatically generated heuristics. The only other tool that incorporates heuristic guidance is UPPAAL CORA, which was discussed above. All directed

model checking techniques discussed in this thesis are implemented in at least one of these tools.

4.4.1 UPPAAL/DMC

UPPAAL/DMC [69] is the first tool that we have developed. As the name suggests, UPPAAL/DMC is an extension of UPPAAL that provides generic heuristics for directed model checking. The development of the tool was joint work with Gerd Behrmann and Kim G. Larsen from Aalborg University. UPPAAL/DMC incorporates many directed model checking techniques, for example the heuristics proposed by Edelkamp et al. [43] and Dräger et al. [40]. Moreover, the following heuristics are also implemented in this tool: h^L and h^U, two heuristics based on the monotonicity abstraction (see Chap. 5), $h^{\mathcal{P}}_{syn}$ and $h^{\mathcal{P}}_{AR}$, pattern database heuristics based on predicate abstraction (see Chap. 6) and $h^{\mathcal{A}}$, another pattern database heuristic based on variable abstractions (see Chap. 7). Precompiled Linux binaries of our tool are freely available at http://www.informatik.uni-freiburg.de/~kupfersc/uppaal_dmc/.

4.4.2 MCTA

MCTA [72] is our second model checking tool for real-time specifications modeled as timed automata. Although the tool can be used for verification, MCTA is rather optimized for falsification, i. e., detecting violations of safety properties fast and returning short error traces. In a nutshell, this tool provides nearly the same functionality as UPPAAL/DMC. It comes with the same heuristics, except the pattern database heuristics. In addition, MCTA features a special global search method (see Chap. 8). The main difference is that MCTA's architecture is especially tailored for directed model checking. As a consequence, it is easy to implement new heuristics or new search algorithms such as *multi-heuristic best-first search* proposed by Helmert [56].

MCTA accepts input models in the form of the UPPAAL input language [7]. So far only a fraction thereof is supported, e. g. there is no support for urgent channels, arrays, etc. yet. Internally, MCTA uses UPPAAL's timed automata parser library. For the representation of zones, MCTA uses UPPAAL's difference bound matrices library. Both libraries are released under the terms of the LGPL or GPL, respectively, and are freely available at http://www.uppaal.com/. All other data structures and all algorithms (and their implementation) used are genuine to MCTA.

MCTA is also free software and released under the terms of the GPL. Precompiled Linux executables and the source code of MCTA are freely available at http://mcta.informatik.uni-freiburg.de/.

Part III

Heuristics for Model Checking Timed Automata

5

Adapting an AI Planning Heuristic for Directed Model Checking

Directed model checking is an emerging field, which improves the falsification of safety properties by providing heuristics that can guide the state space traversal *quickly* towards *short* error traces. Heuristic search is also very successful in the area of AI planning. Our main contribution in this chapter is the adaptation and the extension of the most successful heuristic function from AI planning to directed model checking. The heuristic is based on solving an abstracted problem in every search state. We adapt the abstraction and its solution to systems of timed automata. Our empirical evaluation revealed that, compared to both blind search and some heuristics proposed by Edelkamp et al. [42], we consistently obtain significant, sometimes dramatic, search space reductions. This results in likewise strong reductions of runtime and memory requirements.

5.1 The Monotonicity Abstraction

In this section we introduce the *monotonicity abstraction*, the abstraction method underlying our implemented heuristic function. We first give a high-level description of the abstraction in a generic way, then we define it in the context of systems of timed automata.

Before we start, let us remark that the monotonicity abstraction was first invented in AI planning for the STRIPS fragment of PDDL 2.1 [47]. There it is known under the name *ignoring delete lists* [14, 61]. In STRIPS, the *delete lists* are effect instructions that make a Boolean variable *false*. Ignoring delete lists simplifies the problem because, in STRIPS, variables are only required to be *true*. The monotonicity abstraction we describe below is a generalization of this abstraction approach. We remark that the generalization is *not* published in the AI planning literature; it is, in spirit, somewhat similar to Edelkamp's framework on generalizing the relaxed planning heuristic [41].

5.1.1 The General Idea

The monotonicity abstraction is based on the simplifying assumption that *every state variable, once it obtained a value, keeps that value forever*. The value of a variable is no longer an element, but a *subset* of its domain. That subset grows monotonically over transition applications, hence the name of this abstraction.

In a little more detail, in general a reachability problem consists of a transition system and a target formula. The transition system, e. g. a planning task, a system of timed automata, a piece of program code, etc., can be seen as given by a set of state variables, a set of transition rules and an initial state. The transition rules have a guard, i. e., a formula out of some class of (non-temporal) formulas, and an effect, i. e., an instruction how the variable values change when the rule is applied. States are value assignments to the variables, the target formula is a formula. A solution is a path of transitions that, when applied to the start state, ends in a state that satisfies the target formula.

Under the monotonicity abstraction, the semantics of a transition system as above is changed as follows. States now map each variable to a subset of its domain. The initial assignment contains the single value assigned by the initial state. A formula evaluates to true in a state if there *exists* a variable value vector in the state so that the formula evaluates to true when inserting these values. Executing an effect instruction becomes a *set union* operation, where the new value of each variable v is its old value (a domain subset) plus the new value assigned by the effect. If the effect outcome depends on variables, then all possible value vectors for these variables are used, each yielding a value for v.

Figure 5.1 shows a simple automaton A with two locations l_1 and l_2 and one integer variable v. The initial state of the system is $\langle A = l_1, v = 0\rangle$. Suppose we want to check if there is a reachable state s in which $v = 2$ holds. Obviously, there is no such state. However, in the abstraction there is such a state, i. e., an abstract state s^+ that satisfies the abstract target formula $\varphi^+ = \exists c \in s^+(v) : (c = 2)$. The initial value of v in the abstraction is $\{0\}$. After taking the edge from l_1 to l_2, the abstract value of v becomes $\{0, 1\}$. Since $s^+(A) = \{l_1, l_2\}$ and since the guard of this edge is abstracted to $\exists c \in s^+(v) : (c = 0)$, the edge can be applied a second time. Afterwards v's abstract value becomes $\{0, 1, 2\}$. The new values obtained for v are 1 and 2. The value 1 is obtained by inserting 0 into the effect's right hand side and the value 2 is obtained by inserting 1. In this state the abstract formula φ^+ evaluates to *true*.

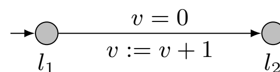


Fig. 5.1. A simple automaton with two locations

It is not difficult to see that the monotonicity abstraction induces an overapproximation of the real transition system. As a consequence, every solution in the real system corresponds to a solution in the abstract system. We will state this formally below, for our abstraction of timed automata. In many cases, deciding solution existence is a polynomial-time problem under the abstraction, making it feasible to solve the abstract problem in every search state. Under certain conditions, checking satisfaction of a formula becomes **NP**-hard in the abstraction, due to the additional existential quantification. In particular, this is the case in our context of timed automata. We make an additional simplification to get around this. We will later come back to this.

5.2 The Monotonicity Abstraction for Timed Automata

Before we give our definitions, consider from a higher point of view what happens if we apply the above abstraction to a system of timed automata. Under the abstraction, each automaton will potentially be in several locations in a state. The integer variables will have several possible values in a state. The clock variables will only accumulate new values. Transitions will be applicable as soon as one of the possible value vectors satisfies the guard.

Thinking a little more about the clocks, one sees that they are likely to trivialize very quickly under the monotonicity abstraction. The reason for this are the *timed* transitions. As time passes, the clocks accumulate all the passing time points. After waiting from time point u to time point $u + d$, the new clock value subsets contain the entire interval $[u, u + d]$. So, in a location l with invariant $I(l)$, the clock value subsets immediately gather all values up to the upper bound specified by $I(l)$. Initially, all clock values are 0. Since time passes continually, the clock value subsets will always have the form $[0, \mathit{max}]$, where *max* is the latest time point yet reached, containing no information other than *max*. As soon as a location l' is reached with $I(l') = \mathit{true}$, then *max* will become infinite. As a consequence the clock value subsets will be $\mathbb{R}_{\geq 0}$ and thus every clock constraint will henceforth evaluate to *true*.

Therefore, reasoning about clock values under the monotonicity abstraction is not likely to contribute useful information, unless additional techniques are used. We present an idea for such additional techniques in Sec. 5.5. For now, we do not include clocks in the computation of heuristic values. While this is undesirable, as said our empirical results demonstrate that taking (abstract) account of automaton locations, synchronization, and integer variables can yield useful search guidance.

5.2.1 Abstract Semantics

Our definitions are straightforward and read as follows. We denote abstract constructs with a superscribed $^+$ to indicate the additivity of the abstraction. Let $\mathcal{S} = A_1 \parallel \ldots \parallel A_n$ be a system of timed automata, where $A_i = \langle L_i, l_i^0, E_i, \Sigma_i, C_i, I_i, V_i \rangle$ is a timed automaton for $i \in N = \{1, \ldots, n\}$. An abstract state s^+ assigns each automaton A_i a location subset $s^+(A_i) \subseteq L_i$. Each integer variable $v \in \bigcup_{i \in N} V_i$ is assigned a value set $s^+(v) \subseteq \mathrm{dom}(v)$. Formulas (conjunctions over integer constraints) are abstracted by *locally*, existentially quantifying the variables *in each condition separately*. For instance, a formula $(v \bowtie_1 v') \wedge (v \bowtie_2 c)$ is abstracted to $\exists c_1 \in s^+(v), \exists c_1' \in s^+(v') : (c_1 \bowtie_1 c_1') \wedge \exists c_2 \in s^+(v) : (c_2 \bowtie_2 c)$. That is, we allow achievement of each condition in separate. We will later explain why we do not use global quantification. An assignment of the form $v := c$, where $v \in V$ and $c \in \mathbb{Z}$, results in $s^+(v) := s^+(v) \cup \{c\}$. An assignment $v := w$, where $v, w \in V$ results in $s^+(v) := s^+(v) \cup s^+(w)$. A linear assignment $v := c_0 + \sum_{i=1}^{n} c_i \cdot v_i$, where $c_i \in \mathbb{Z}$ and $v, v_i \in V$ results in $s^+(v) := s^+(v) \cup \{c_0 + \sum_{i=1}^{n} c_i \cdot e_{v_i} \mid e_{v_i} \in s^+(v_i)$ for all $1 \leq i \leq n\}$. Values not contained in $\mathrm{dom}(v)$ are removed from the result. A τ transition of automaton A_i from location l_i to l_i' is enabled if $l_i \in s^+(A_i)$ and the respective abstract edge guard holds in s^+. The effect assignments are executed as above and $s^+(A_i) := s^+(A_i) \cup \{l_i'\}$ is set. A synchronous transition of automaton A_i from location l_i to l_i' and of automaton A_j from location l_j to l_j' is enabled if $l_i \in s^+(A_i), l_j \in s^+(A_j)$ and both respective abstract edge guards hold in s^+. The effect assignments are executed as above and $s^+(A_i) := s^+(A_i) \cup \{l_i'\}$ as well as $s^+(A_j) := s^+(A_j) \cup \{l_j'\}$ are set. If s_0 is the initial state of $\mathcal{S}$, then the abstract initial state s_0^+ is given by $s_0^+(A_i) = \{l_i^0\}$ for every automaton A_i, and $s_0^+(v) = \{s_0(v)\}$ for every variable $v \in V$. A path of successively enabled transitions from s_0 is an *abstract solution* if it ends in a state s^+ in which the abstract target formula holds.

Let us come back to the question why we evaluate guards by *locally*, existentially quantifying the variables in each condition separately. Quantifying the variables over the entire formula *globally* yields an **NP**-complete constraint problem. So, there is no way around making further abstractions. This is stated in the next proposition.

Proposition 5.1. *Let V_i for $1 \leq i \leq n$ be a set of integer values and let $\varphi = \exists e_1 \in V_1, \ldots, \exists e_n \in V_n : \bigwedge_{j=1}^{m} c_j$, where c_j is a comparison of the form $e_{1j} \bowtie_j e_{2j}$ and $\bowtie_j \in \{<, >, =, \leq, \geq, \neq\}$. To decide whether φ is satisfiable is **NP**-hard.*

Proof (Proposition 5.1). We will prove the claim by a reduction from the graph 3-colorability problem, which is **NP**-complete (cf. [48, GT4]). Let $G = (V, E)$

be a graph, where $V = \{v_1, \ldots, v_n\}$ is the set of vertices and $E \subseteq V \times V$ is the set of edges. There is a conjunct $e_{1j} \neq e_{2j}$ and corresponding existential quantifications $\exists e_{1j} \in V_l$ and $\exists e_{j2} \in V_k$ in φ iff there is an edge between v_l and v_k in G. Let $V_i = \{1, 2, 3\}$ for $1 \leq i \leq n$. From the construction it follows that φ is satisfiable iff G is 3-colorable. □

We chose to do local quantification mainly because it is very simple and can be implemented efficiently. It also comes in handy for linear arithmetic. When dealing with linear combinations of integer variables, even checking a single condition $\exists \overline{v} : f(\overline{v}) = c$, where f is a linear function that depends on the variables $\overline{v}$ and c is an integer constant, is **NP**-hard. This is not usually a problem since the number of variables in the expressions is typically small. Note that it is also possible to handle expressions in an incremental way. We will come back to this is in Sec. 5.3. As the total number of variables in a conjunction of expressions can become quite large, it is convenient to address the single expressions in separate.

5.2.2 Properties of the Monotonicity Abstraction

In this section we will prove some theoretical properties of the monotonicity abstraction. First we will show that every solution of a reachability problem $\langle \mathcal{S}, \varphi \rangle$ corresponds to an abstract solution of the corresponding abstract problem, induced by the monotonicity abstraction. Afterwards we show that, if the assignments and guards that occur in the system $\mathcal{S}$ comply with a certain form, it can be decided in polynomial time if there is an abstract solution. Finally we prove that solving the abstract reachability problem optimally, i. e., finding an abstract solution of minimal length, is computationally hard, i. e., **NP**-hard.

Proposition 5.2. *Let $\langle \mathcal{S}, \varphi \rangle$ be a reachability problem, where $\mathcal{S}$ is a system of timed automata and φ a target formula, i. e., a conjunction over location predicates and integer constraints. If $t_1, \ldots, t_n$ is a solution then $t_1, \ldots, t_n$ is also an abstract solution.*

Proof (Proposition 5.2). Let s_i, for $0 \leq i \leq n$, denote the reached state after the execution of transitions $t_1, \ldots, t_i$. We show by induction over i that $s_i(A_j) \in s_i^+(A_j)$ for all automata A_j, and $s_i(v) \in s_i^+(v)$ for all integer variables v. This suffices to prove the claim.

It is obvious for $i = 0$. If it holds for i, then transition t_{i+1} is enabled in s_i^+. The new location and integer variable values resulting from executing t_{i+1} are, by definition, inserted into the respective s_{i+1}^+ value subsets. Each of t_{i+1}'s assignments is of the form $v := c_0 + \sum_{j=1}^{n} c_j \cdot v_j$, where $v, v_j \in V$ and $c_j \in \mathbb{Z}$. For all these assignments, it holds that $s_{i+1}(v) = c_0 + \sum_{j=1}^{n} c_j \cdot s_i(v_j)$. By

induction it holds that $s_i(v_j) \in s_i^+(v_j)$ and $c_0 + \sum_{j=1}^{n} c_j \cdot s_i(v_j) \in \{c_0 + \sum_{j=1}^{n} c_j \cdot e_{v_j} \mid e_{v_j} \in s^+(v_j)\} \subseteq s_{i+1}^+$. □

By Proposition 5.2, every solution in the real search space is also contained in the abstract search space. So the length of an optimal abstract solution is an admissible heuristic function. We will come back to this below. It can be decided in polynomial time if there exists an abstract solution or not. This is formally stated by the next theorem.

Theorem 5.3. *Let $\langle \mathcal{S}, \varphi \rangle$ be a reachability problem, where $\mathcal{S}$ is a system of timed automata and φ is a target formula. Let TASolEx$^+$ denote the decision problem if there is an abstract solution of the reachability problem. If all assignments that occur in $\mathcal{S}$ are either of the form $v := c$ or $v := v'$, where v, v' are integer variables and c is an integer constant, then*

*TASolEx$^+$ is in **P**.*

Proof (Theorem 5.3). We will describe a polynomial solution algorithm in Sec. 5.3. □

The polynomial solution algorithm forms the basis of our heuristic functions. For a heuristic function, what we want to know is not primarily if there is an abstract solution, but what the *length* of an abstract solution is (if there is one). Abstract solutions may contain arbitrarily many redundant transitions, and we want to know what an *optimal* abstract solution is. We call the length of such a solution, for a state s, the heuristic value for s. It is denoted with $h^+(s)$. Unfortunately, computing h^+ is still hard. This is formally stated in the following proposition.

Proposition 5.4. *Let $\langle \mathcal{S}, \varphi \rangle$ be a reachability problem, where $\mathcal{S}$ is a system of timed automata and φ a target formula and let $l \in \mathbb{N}$. Let TASolMin$^+$ denote the problem to decide if there is an abstract solution of length at most l.*

*TASolMin$^+$ is **NP**-hard.*

Proof (Proposition 5.4). We prove the theorem by a reduction from 3SAT. Let $V = \{v_1, \ldots, v_n\}$ be a set of n Boolean variables and $\varphi = \bigwedge_{j=1}^{m} (c_{j1} \vee c_{j2} \vee c_{j3})$ a 3-CNF formula, where $c_{ji} \in \{v_1, \neg v_1, \ldots, v_n, \neg v_n\}$. Further let $\mathcal{S}$ be the automata system that consists of the following $n + m$ automata. There is an automaton A_{v_i} for every variable $v_i \in V$ and an automaton A_{c_j} for every clause $c_j = (c_{j1} \vee c_{j2} \vee c_{j3})$. The automata are depicted in Fig. 5.2. Every automaton has a local error location, the double circled location. An error state is reached if all automata are in their local error locations.

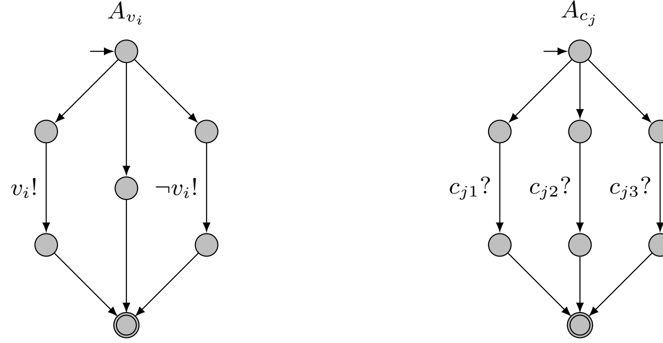


Fig. 5.2. Automata encoding variables v_i and clauses c_j

Assume that $f : V \rightarrow \mathbb{B}$ is a function that assigns each variable $v \in V$ a truth value so that φ evaluates to true. An abstract error trace of length $2n + 3m$ can then be constructed as follows. For every automaton A_{c_j} select the three transitions from the initial location to the local error location that contains a transition that synchronizes on a label $c_{jl}?$ that evaluates to true with respect to f. The synchronization partner for this transition is the automaton A_{v_i}, where v_i is the variable of the literal c_{jl}. In A_{v_i} we select the path needed for the synchronization. For the A_{v_i} automata that are not needed in order to synchronize with any of the A_{c_j} automata, select the two τ transitions leading from the initial state to the local error location. The total length of this abstract error trace is $2n + 3m$.

Assume there is an abstract error trace of length $2n + 3m$. For each A_{v_i} at least two τ transitions are needed to reach a local error state. For every automaton A_{c_j} also two τ transitions are needed plus one synchronized transition. This sums up to $2n+3m$. From this abstract error trace we construct $f : V \rightarrow \mathbb{B}$ such that f satisfies φ as follows. Set $f(v_i)$ to *true* iff automaton A_{v_i} synchronizes on $v_i!$. □

Note that one does not even need integer variables in the proof to Proposition 5.4. The desired admissible heuristic function h^+, based on our abstraction, cannot be computed efficiently. So, in practice, we will have to *approximate* h^+. We introduce two approximation techniques in the next section, one computes a lower bound, the other one computes an upper bound.

5.3 Approximating h^+

Our heuristic functions map search states to integers. For each state s that is encountered during search, we are facing the following situation. We are given a system of timed automata, a target formula, and a state s. The task is to approximate the length of an optimal abstract solution that starts with s.

These approximations are based on a forward-chaining algorithm that generalizes algorithms proposed in the context of numeric planning [59]. The algorithm is a forward fixpoint computation. It determines in polynomial time whether there is an abstract solution, by building a data structure called the *abstract transition graph*, or ATG for short. The ATG is a layered graph encoding the reachability information. Pseudocode is given in Fig. 5.3. The algorithm takes as input a system of timed automata $\mathcal{S} = A_1 \parallel \ldots \parallel A_n$, where A_i is a timed automaton, a target formula φ and the state s for which we want to compute a heuristic value.

```
1  function build-atg(S, φ, s):
2      for each A_i do: L_0(A_i) := {s(A_i)}
3      for each v do: V_0(v) := {s(v)}
4      k := 0
5      while L_k, V_k ⊭ φ^+ do:
6          for each A_i do: L_{k+1}(A_i) := L_k(A_i)
7          for each v do: V_{k+1}(v) := V_k(v)
8          for each enabled τ edge l_i --(g_i,τ / f_i,r_i)--> l'_i of all automata A_i do:
9              L_{k+1}(A_i) := L_{k+1}(A_i) ∪ {l'_i}
10             for each assignment v := c_0 + Σ_{l=1}^{n} c_l · v_l in f_i do:
11                 V_{k+1}(v) := V_{k+1}(v) ∪ {c_0 + Σ_{i=1}^{n} c_i · e_{v_i} | e_{v_i} ∈ V_k(v_i)}
12         for each pair of enabled edges l_i --(g_i,a / f_i,r_i)--> l'_i, l_j --(g_j,ā / f_j,r_j)--> l'_j of all A_i ≠ A_j do:
13             L_{k+1}(A_i) := L_{k+1}(A_i) ∪ {l'_i}
14             L_{k+1}(A_j) := L_{k+1}(A_j) ∪ {l'_j}
15             for each assignment v := c_0 + Σ_{l=1}^{n} c_l · v_l in f_i ∪ f_j do:
16                 V_{k+1}(v) := V_{k+1}(v) ∪ {c_0 + Σ_{i=1}^{n} c_i · e_{v_i} | e_{v_i} ∈ V_k(v_i)}
17         if L_{k+1}(A_i) = L_k(A_i) for all A_i and V_{k+1}(v) = V_k(v) for all v then:
18             minlayer := ∞
19             return
20         k := k + 1
21     minlayer := k
```

Fig. 5.3. Building an abstract transition graph (ATG)

The ATG is a sequence of location sets $L_k(A_i)$ and of variable value sets $V_k(v)$, the *layers* of the graph. The algorithm from Fig. 5.3 builds these in an incremental way, so that their contents increase monotonically over k. As already said, the algorithm is a forward fixpoint computation. The initial location and variable value sets L_0 and V_0 contain the current locations and variable values of the state s. While the abstract target formula φ^+ is not satisfied in the abstraction, i. e., if there are no locations in L_k and no variable values in V_k such that φ^+ evaluates to true, then the following is done. We call a transition t enabled if the source locations (there are two locations if t is a synchronized transition) of the transition are contained in L_k and there are variable values in

V_k such that the guard of the transition is satisfied. For every enabled transition t the target locations are added to the corresponding location sets. The variable value sets are updated according to the assignments of t. A fixpoint is reached if, after processing each enabled transition, the location and variable value sets L_k, V_k and L_{k+1}, V_{k+1} are equal. In this case there is no abstract solution and the iteration terminates, setting *minlayer* to infinity.

Lemma 5.5. *If for a reachability problem $\langle \mathcal{S}, \varphi \rangle$ there is an abstract solution of length n, then the built-atg algorithm successfully stops in at most n steps.*

Proof (Lemma 5.5). Say $t_1, \ldots, t_n$ is an abstract solution. We prove that, when running the algorithm without the stopping criterion, in iteration n the target formula will be satisfied. This suffices because, obviously, if the stopping criterion holds in an earlier iteration $m < n$ then there is a fixpoint: $L_k(A_i) = L_m(A_i)$ for all A_i and $k > m$, and $V_k(v) = V_m(v)$ for all v and $k > m$, yielding a contradiction.

Let s_k^+, for $0 \leq k \leq n$, denote the state after abstract execution of transitions $t_1, \ldots, t_k$ in the start state. We show by induction over k that s_k^+ is contained in L_k and V_k, i. e., for all i it holds that $s_k^+(A_i) \subseteq L_k(A_i)$ and for all v it holds that $s_k^+(v) \subseteq V_k(v)$. This suffices to prove the overall claim. The induction base case is obvious. If the induction hypothesis holds for k, then t_{k+1} is enabled by L_k and V_k, so it is processed in the inner loop. The locations added by t_{k+1} are inserted into s_{k+1}^+. The values added by its effects $v := c_0 + \sum_{i=1}^{n} c_i \cdot v_i$ are inserted into $V_{k+1}(v)$, because $s_k^+(v_i) \subseteq V_k(v_i)$ by induction hypothesis. So, the values inserted into L_{k+1} and V_{k+1} in particular contain the values inserted by t_{k+1} into s_{k+1}^+ with these effects. □

In particular, if the procedure for constructing the ATG terminates unsuccessfully, then there is no abstract solution. It is easy to see that, if the targets are reached in layer *minlayer*, then an abstract solution can be constructed as the sequence, for $k = 0, \ldots, \mathit{minlayer} - 1$, of all transitions enabled by L_k and V_k. So altogether the algorithm for constructing the ATG is a polynomial procedure deciding existence of an abstract solution, and Theorem 5.3 follows.

5.3.1 Remarks on Linear Arithmetic

Before we introduce two heuristics based on the monotonicity abstraction, we give some remarks on linear arithmetic.

Note that, if we allow linear expressions over unbounded integer variables, then the build-atg algorithm from Fig. 5.3 does not necessarily terminate. To see this, consider the automaton A depicted in Fig. 5.4. Suppose the initial state of the system is given by $\langle A = l_1, v = 1 \rangle$ and the error state is $\langle A = l_2, v = 0 \rangle$.

Of course the error state is neither reachable in the real search space nor in the abstract search space. The construction algorithm for the ATG, however, does not terminate. This is because an abstract error state is never reached and thus the stopping criterion of the while loop is never satisfied. Also the fixpoint test (line 14 in Fig. 5.3) always fails, because $V_k \neq V_{k+1}$ for all layers k.

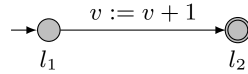


Fig. 5.4. An example where build-atg would not terminate for unbounded variables

Of course, since we only consider bounded integer variables, termination is guaranteed. This is because there are only finitely many variable values that can be added to the variable value sets. In general the variable value sets V_k cannot be represented in size polynomial of the reachability problem. To see this consider Fig. 5.5 as an example. It depicts an automaton with $n+1$ assignments and $n+1$ variables. Suppose that the initial state is given by l_0 and the values of all variables v_i are 0. During the construction of the ATG, as soon as the assignment of the edge from l_n to l_{n+1} is executed, v_n will contain 2^n different values, which is exponential in the number of assignments.

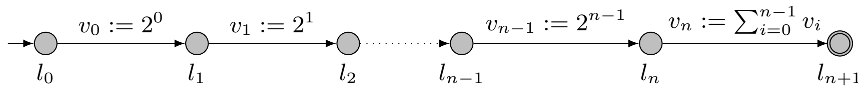


Fig. 5.5. Abstract variables can contain an exponential number of values.

In order to overcome this situation, it is possible to apply an additional abstraction on the linear arithmetic. We refer the reader to Hoffmann's article about the METRIC-FF planning system [59]. Note that this additional abstraction results in a coarser abstraction of the system's semantics and thus the resulting heuristic is less informed.

Some final notes about the handling of linear expressions in the abstraction. As said, testing $\exists \overline{v} : f(\overline{v}) = c$ is **NP**-hard for linear expressions $f(\overline{v})$, but the number of variables in $\overline{v}$ is typically small. Our main algorithmic trick to deal with the expressions efficiently is an *incremental* computation. If, at some point during the construction of the ATG, we want to know whether $\exists \overline{v} : f(\overline{v}) = c$ is true based on the current value subsets V_k, then we can refer back to the last time we asked that same question, and just take into account how the value subsets have changed since then. In fact, we just keep a flag at each expression occurring in the input, indicating whether the expression can be satisfied yet.

Every time the value subset of a variable occurring in the expression changes (grows), we see whether that change serves to satisfy the expression; if so, we set the flag. Checking guard satisfaction in the ATG then simply means to refer to the flags. Similarly, one can deal with linear expression effect right hand sides, $v := f(\overline{v'})$. We just enumerate the set of value tuples for $\overline{v'}$, referring back to the previous version of that set. Typically, just one or two variables in $f(\overline{v'})$ have gathered new values since the last evaluation of $f(\overline{v'})$. It suffices to enumerate these changes and extend the old tuple set correspondingly. Note that this incremental approach can, in fact, be implemented for (almost) arbitrarily complicated expressions, not only linear ones.

5.3.2 The h^L Heuristic

Let us focus again on how to approximate h^+. As said, we compute a lower bound as well as an upper bound. We call the lower bound h^L, and the upper bound h^U. By Proposition 5.5, a lower bound on h^+ is the *minlayer* value determined by the ATG algorithm. We set $h^L(s)$ to that value as computed by the procedure for building the ATG for the state s (see Fig. 5.3). Note that h^L returns infinity for s if there is no abstract solution. From Proposition 5.2 it follows that in such cases there is also no real solution and thus the state can be excluded from further examinations. Pseudocode for the h^L heuristic is given in Fig. 5.6.

```
1 function h^L(S, φ, s):
2     build-atg(S, φ, s)
3     return minlayer
```

Fig. 5.6. The h^L heuristic

5.3.3 The h^U Heuristic

Regarding an upper bound, note that, with the above, the number of all transitions enabled at layers $k = 0, \ldots, minlayer-1$ provides such a bound. However, this bound is likely to be far too generous, counting transitions that are reachable but not needed to achieve the targets. We therefore use a more involved method to determine our upper bound h^U. The method basically selects, at each layer $k = 0, \ldots, minlayer - 1$, a *subset* of the enabled transitions, so that the sequence of the selected transitions is still an abstract solution. This is done by a backward-chaining procedure on the ATG. We set h^U to the length of the selected abstract solution. Note that this abstract solution is not necessarily optimal. Like the h^L heuristic, the h^U heuristic assigns infinity to a state from which the abstract error state is not reachable.

From a successfully built ATG, one can select an abstract solution with the algorithm given in Fig. 5.7.

1 **function** extract-solution($\mathcal{S}$, φ, s):
2 **for** $k := 0, \ldots,$ *minlayer* **do**:
3 **for each** A_i **do**: $TL_k(A_i) := \emptyset$
4 **for each** v **do**: $TV_k(v) := \emptyset$
5 make-target(*minlayer*, φ)
6 **for** $k :=$ *minlayer*$, \ldots, 1$ **do**:
7 **for each** A_i **do**:
8 **for each** $l' \in TL_k(A_i)$ **do**:
9 select transition t enabled at $k-1$ that ends in l'
10 make-target($k-1$, t's start locations, t's guard formulas)
11 **for each** v **do**:
12 **for each** $c \in TV_k(v)$ **do**:
13 select t enabled at $k-1$ with effect $v := c_0 + \sum_{i=1}^{n} c_i v_i$ such that $c \in \{c_0 + \sum_{i=1}^{n} c_i e_{v_i} \mid e_{v_i} \in V_{k-1}(v_i)\}$
14 **for each** v_i that occurs in the effect **do**:
15 select $e_{v_i} \in V_{k-1}(v_i)$ such that $c = c_0 + \sum_{i=1}^{n} c_i e_{v_i}$
16 $TV_{k-1}(v_i) := TV_{k-1}(v_i) \cup \{e_{v_i}\}$
17 make-target($k-1$, t's start locations, t's guard formulas)

Fig. 5.7. Extracting an abstract solution from an abstract transition graph

1 **function** $h^U(\mathcal{S}, \varphi, s)$:
2 build-atg($\mathcal{S}$, φ, s)
3 **if** *minlayer* $= \infty$ **then**:
4 **return** ∞
5 extract-solution($\mathcal{S}$, φ, s)
6 **return** number of selected transitions

Fig. 5.8. The h^U heuristic

The algorithm makes use of location sets $TL_k(A_i)$ and of variable value sets $TV_k(v)$. At ATG layer k, these sets contain the current target locations and target variable values at layer k. A location l is a target location, if the target formula contains a location constraint of the form $A = l$. A target value for a variable v_i is a, possibly empty, subset V_i of the variable's domain $\mathrm{dom}(v_i)$, so that the integer part of the abstract target formula φ^+ is satisfied by $V_1, \ldots, V_n$ The *TL* and *TV* sets are initialized as empty, then the target locations and formula of the overall task are inserted by the make-target function. That function is explained in detail below. Transitions supporting the targets in the $TL_k(A_i)$ and $TV_k(v)$ sets are selected during a backwards loop over k from the top ATG layer to layer 1. The start locations and guard formulas of the selected transitions are

inserted into the *TL* and *TV* sets; by construction of the ATG, these new targets will appear at layers below k (see also below). It is ensured that the transition effects achieve the desired target.

The make-target function takes as arguments a number m, a set of locations, and a set of formulas (the sets contain 1 or 2 elements each). The function first determines, for each location l in automaton A_i, the lowest k such that $l \in L_k(A_i)$ and sets $TL_k(A_i)$ to $TL_k(A_i) \cup \{l\}$. Note that $k \leq m$ will hold by construction, i. e., the new location targets are inserted at layers below in the graph (otherwise the transition would not be enabled at m).

The interesting part is the selection of the linear guard. As we know that every guard of any transition in the ATG is satisfied, we have to find a value for each variable that occurs in the guard such that the guard is satisfied. For each chosen value c of variable v, the lowest k with $c \in V_k(v)$ is determined, and $TV_k(v)$ is set to $TV_k(v) \cup \{c\}$. Note again that $k \leq m$ will hold. For every such variable value pair we add a transition that assigns this value to the variable to the relaxed plan. Note again that $k \leq m$ will hold.

Proposition 5.6. *Let $\langle \mathcal{S}, \varphi \rangle$ be a reachability problem, where $\mathcal{S}$ is a timed automata system and φ a target formula and s a state of $\mathcal{S}$, so that the algorithm in Fig. 5.3 stops with success. Then the transitions selected by the algorithm in Fig. 5.7 form an abstract solution.*

Proof (Proposition 5.6). First, note that by construction of the abstract transition graph, the algorithm cannot fail, i. e., there is always a transition sufficient to support a target location or variable value. We form the abstract solution by arranging the transitions in inverse order of selection, i. e., in particular from bottom $k = 0$ to top $k = \mathit{minlayer} - 1$. We show by induction over k that the start locations, and the guards, of the transitions are satisfied in the respective abstract state of execution. For $k = 0$ this is obvious. For $k > 0$ it follows because the start locations and guards of the transitions were posted as targets at layers below, and thus achieved by the respective selected supporting transitions (which are enabled by induction hypothesis). So all selected transitions will be enabled in the sequence, thus achieving the (global) target locations and formula that were posted at the start of the algorithm. □

Consider Fig. 5.9 as an example. It depicts a system consisting of two automata Q and B that can synchronize via the synchronization label a. The initial state of the system is given by $s_0 = \langle Q = q_1, B = b_1 \rangle$. Suppose that an error state is reached when both automata are in their double circled locations.

The ATG for this automata system is given in Fig. 5.10, it can be constructed in n steps as follows. Automaton Q takes the edge to q_2 once, and can then take its edge from q_2 to q_1 exactly $n - 1$ times in sequence. This is possible since

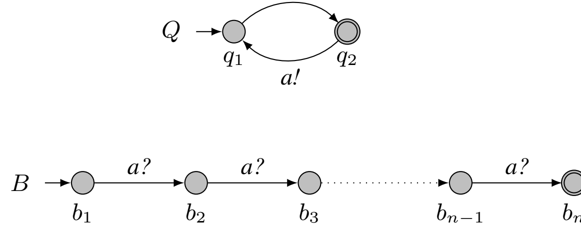


Fig. 5.9. A simple example where h^L and h^U deliver bad heuristic values

location q_2 remains in the reached location subset $L_k(Q)$, for each value of k. At layer n the abstract target formula is satisfied and the algorithm stops.

Note that, in order to reach an error state in the real search space, Q needs to go through repeated cycles. More precisely, as B has n locations, the real solution takes $2(n-1)$ transitions, half of which are synchronized between both automata. Especially if we replaced the edge from q_1 to q_2 with a chain of m non-synchronized edges, the example illustrates that the h^L and h^U estimates can be arbitrarily bad.

Layer k	$L_k(B)$	$L_k(Q)$
0	$\{b_1\}$	$\{q_1\}$
1	$\{b_1\}$	$\{q_1, q_2\}$
$2+i$	$\{b_1, \ldots, b_{2+i}\}$	$\{q_1, q_2\}$

Fig. 5.10. The ATG for the initial state of the system from Fig. 5.9

Figure 5.11 gives another example. In the start state, all automata are in the bottom location. An error state is reached if all automata are in the double circled locations. In each automaton except the first one, one has two choices, one of which leads into a dead end (a state from which the error cannot be reached), since the required synchronization will not be available anymore. Built for the start state, each layer k of the ATG corresponds exactly to the locations that can be reached within k steps — in particular, the double circled location in the kth automaton. So *minlayer* $= n$, and $h^L = h^U = n$ is the precise error state distance. If, during search, a wrong decision was made in automaton A_i, then the top left location in A_i does not appear in the ATG, and the heuristic value is ∞. Another example where h^L and h^U are precise is, e. g., a situation that requires (only) to repeatedly increment an integer variable. Intuitively, h^L and h^U are good at detecting long sequences of transitions that build upon each other to achieve some target, and at finding out that such a sequence is not available. What they are *not* good at is to see that *the same thing has to be*

done multiple times[1] – under the monotonicity abstraction, everything needs to be done at most once. A bad situation was given earlier in Fig. 5.9, where the top automaton needs to go through repeated cycles, while h^L and h^U act as if a single cycle is sufficient.

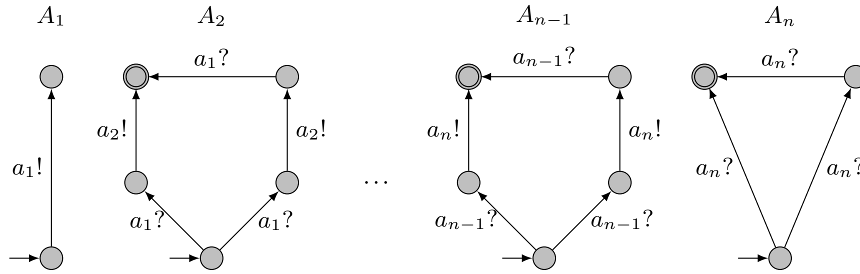


Fig. 5.11. A simple example where h^L and h^U deliver the precise error distance

5.4 Evaluation

We evaluated our heuristics by comparing them to other heuristics and other uninformed search methods in two different settings.

In the optimal setting, we are interested in finding shortest possible error traces. The configurations of this setting are UPPAAL's breadth-first search (BFS) and A^* search with the h^L and d^L heuristics. The latter heuristic is a heuristic proposed by Edelkamp et al. [42, 43], which we will explain in the next section.

In the suboptimal setting, we are interested in finding any solution. The configurations of this setting are UPPAAL's randomized depth-first search (rDFS) and greedy search with any of h^L, h^U, d^L and d^U. Note that UPPAAL's rDFS is by far the most efficient uninformed search method across many examples, including ours. The d^U heuristic, also proposed by Edelkamp et al., is explained in the next section.

5.4.1 The Heuristic Functions

Edelkamp et al. [42, 43] work in the context of SPIN. They propose to base the distance estimation on the graph-distances within each single automaton. For automaton A_i, let $d(A_i)$ be the distance of A_i's start location to its target location, when ignoring all edge guards. Recall that a target location is a location

[1] When repeatedly incrementing a variable, every increment has a *different* effect.

that appears in a location constraint in the target formula. If there is no target location, then $d(A_i)$ is set to 0. An admissible heuristic function, called d^L, is defined as $\max_{A_i} d(A_i)$, and a non-admissible heuristic function, called d^U, is defined as $\sum_{A_i} d(A_i)$. We implemented these heuristic functions, taking the $d(A_i)$ to be the graph distances in the individual automata.

The underlying abstraction of the d^L and d^U heuristics is a rather crude approximation of the system semantics. It completely ignores synchronization and integer variables. Examine Fig. 5.11 again. Recall that for this example the h^L and h^U heuristics return the precise error distance and are able to detect all dead ends. In contrast, the heuristic value for the initial state returned by d^L is 2 and that of d^U is $2n - 1$, and no dead ends are detected.

Note that our heuristics are computationally more expensive than d^L and d^U, i. e., computing the heuristic function takes more runtime than what is needed for d^L and d^U. However, as we will see, this often pays off in terms of much smaller explored search spaces.

5.4.2 The Benchmark Set

All our benchmarks stem from the AVACS benchmark suite and can be obtained at http://www.avacs.org.

The C_i examples, where $1 \leq i \leq 9$, come from a case study called Single-tracked Line Segment. It models a distributed real-time controller for a segment of tracks where trams share a piece of track. It was originally modeled in terms of PLC automata [32, 68], an automata-like notation for real-time programs. The PLC automata were translated into timed automata with the tool MOBY/RT [80]. The property to be checked requires that never both directions are given permission to enter the shared segment simultaneously. This property is ensured by three PLC automata of the whole controller. We injected an error by manipulating a delay such that the asynchronous communication between these automata is faulty. The given set of PLC automata had eight input variables. We used MOBY/RT to constructed nine models with decreasing complexity by abstracting more and more of these inputs.

The M_i and N_i examples, for $1 \leq i \leq 4$, come from a study called Mutual Exclusion [32]. It models a real-time protocol to ensure mutual exclusion in a distributed system via asynchronous communication. We flawed the model by increasing an upper time bound. By applying various abstraction techniques, we got models of different complexity. The resulting models do not have many automata but a non-trivial number of clocks and variables.

The F_i^A and F_i^B examples are two different versions of the Fischer protocol. The index i gives the number of parallel automata. An error state is reached if at least two automata are simultaneously in a certain location. We made error

states reachable by weakening one of the temporal conditions in the automata. The variants differ in the way how they encode the error condition. Variant A adds additional automata with synchronization. Variant B specifies two of the automata for the error condition.

The A_i examples, where $2 \leq i \leq 6$, model an arbiter tree that establishes mutual exclusion between 2^k client processes. The processes are arranged in a binary tree of height k, where each leaf node is a client and each internal node is an arbiter that ensures mutual exclusion between its two children. The examples $A_2, \ldots, A_6$ contain arbiter trees of height 2–6, with an exponentially growing number of processes. An error state is reached if two clients simultaneously have access to the shared resource. Due to a faulty client, such a state is reachable.

A more detailed description of our benchmarks can be found in the Appendix of this thesis.

5.4.3 Experimental Results

We implemented all heuristics mentioned in this chapter in MCTA. The tool can be freely downloaded from http://mcta.informatik.uni-freiburg.de/. All experiments in this section were obtained on an AMD Opteron 2.3 GHz system with 4 GByte of memory.

As said, our configurations finding optimal error paths are UPPAAL's standard breadth-first search (BFS), and MCTA's A^* search with h^L and d^L. Our suboptimal configurations are UPPAAL's standard randomized depth-first search (rDFS), which is by far UPPAAL's most efficient standard method across many examples, including ours. The results, reported in this thesis, are averaged over ten runs. Additionally we use MCTA's greedy search with any of h^L, h^U, d^L and d^U.

The results for the suboptimal configurations are in Table 5.1 (rDFS, h^L, and h^U) and in Table 5.2 (d^L and d^U). The results clearly demonstrate the potential of our heuristic functions. Consider Table 5.1 first. Except in the F^B examples, where h^L behaves very badly, the heuristic searches consistently find the error paths much faster. Due to the reduced search space size, they can solve more of the larger C examples. At the same time, they find *much*, by orders of magnitude, shorter error paths in *all* cases. In the F^B examples, h^L does worse than h^U because its heuristic value does not improve if only one of the two target automata moves closer to its destination: the ATG becomes shorter only if both get closer.

Table 5.2 shows the results for the suboptimal configurations obtained with the d^L and d^U heuristics. Except in the Fischer variants, using greedy search with the d^L and d^U heuristics, behaves, much like rDFS. In most cases even more states are explored. The error paths are up to two orders of magnitude

Table 5.1. Experimental results for the suboptimal configurations rDFS, greedy search with h^L and h^U. Dashes indicate out of memory ($>$ 4 GByte).

	runtime in s			explored states			trace length		
Exp.	rDFS	h^L	h^U	rDFS	h^L	h^U	rDFS	h^L	h^U
C_1	0.1	0.0	0.0	24404	1989	373	880	89	62
C_2	0.3	0.1	0.1	64042	3559	663	770	123	74
C_3	0.4	0.1	0.0	86142	4242	928	618	119	68
C_4	4.5	0.6	0.5	921415	20081	10406	1569	116	103
C_5	46.1	5.1	3.0	8388325	141174	55676	3745	345	118
C_6	–	57.4	9.0	–	1253431	185293	–	641	130
C_7	–	419.8	54.8	–	9475793	878345	–	1175	200
C_8	–	102.5	106.8	–	2601885	1878328	–	556	385
C_9	–	179.8	516.2	–	4641449	7890458	–	995	280
M_1	0.4	0.0	0.1	39838	2431	5125	1246	294	71
M_2	1.3	0.1	0.2	127973	7436	13153	2809	640	93
M_3	0.9	0.2	0.2	97987	12924	12602	2716	591	100
M_4	4.1	0.2	0.6	408400	13497	30276	11940	750	150
N_1	1.7	0.1	0.2	55690	4834	7691	1100	329	80
N_2	5.9	131.3	0.8	188784	185770	23392	3350	40652	122
N_3	4.3	0.1	1.5	146601	7533	37381	3028	499	113
N_4	27.7	1.2	10.4	917774	46948	141354	14713	1661	230
F_5^A	0.0	0.0	0.0	419	9	9	111	8	8
F_{10}^A	0.1	0.0	0.0	10435	9	9	1406	8	8
F_{15}^A	0.7	0.0	0.0	43273	9	9	4641	8	8
F_5^B	0.0	0.0	0.0	293	167	7	78	12	6
F_{10}^B	0.0	2.3	0.0	7933	86462	7	1207	22	6
F_{15}^B	1.5	–	0.0	93632	–	7	9973	–	6
A_2	0.0	0.0	0.0	95	33	28	31	18	18
A_3	0.0	0.0	0.0	6030	202	76	105	24	18
A_4	0.2	9.7	0.0	46642	75106	39	752	36	28
A_5	–	65.7	1.7	–	257208	4027	–	74	47
A_6	–	–	–	–	–	–	–	–	–

longer than those found by rDFS, except in Fischer variant A. Note that the heuristics cannot handle variant C of the Fischer protocol as the target formula for these examples is not expressed in terms of location predicates. For this reason, they are left out in the table. In variant B, similarly to h^L, the d^L heuristic quickly fails. In variant A, due to the construction both d^L and d^U are constantly 1, and the search spaces are identical to those of a non-randomized depth-first search. In the arbiter examples, the d^L heuristic performs slightly better than h^L, in terms of runtime and explored states. However, the error traces found by h^L and h^U are much shorter than the ones found by d^L and d^U. Note that the

h^U heuristic dominates all other heuristics on the A examples and finds nearly optimal error traces.

Table 5.2. Experimental results for greedy search with d^L and d^U. The rDFS results are the same as in Table 5.1.

	runtime in s			explored states			trace length		
Exp.	rDFS	d^L	d^U	rDFS	d^L	d^U	rDFS	d^L	d^U
C_1	0.1	0.0	0.1	24404	10251	11336	880	747	460
C_2	0.3	0.1	0.2	64042	31886	32921	770	1279	846
C_3	0.4	0.2	0.3	86142	53025	48948	618	1190	817
C_4	4.5	2.1	2.6	921415	378723	373492	1569	4401	1461
C_5	46.1	21.1	26.9	8388325	2957129	2851025	3745	14518	2326
C_6	–	218.3	208.3	–	24247904	23461978	–	93504	17209
C_7	–	–	–	–	–	–	–	–	–
C_8	–	–	–	–	–	–	–	–	–
C_9	–	–	–	–	–	–	–	–	–
M_1	0.4	0.1	0.2	39838	9885	10991	1246	1306	1348
M_2	1.3	1.0	210.2	127973	47783	437394	2809	6878	157110
M_3	0.9	0.7	348.4	97987	35502	400854	2716	4797	149101
M_4	4.1	3.0	6045.7	408400	118621	2346575	11940	25555	408092
N_1	1.7	0.3	0.5	55690	12052	13764	1100	1413	2324
N_2	5.9	4.8	3.9	188784	82488	73660	3350	12557	10245
N_3	4.3	2.6	4173.2	146601	46305	881199	3028	12598	234661
N_4	27.7	59.6	25.9	917774	504586	277981	14713	51651	28635
F_5^A	0.0	0.0	0.0	419	48	48	111	40	40
F_{10}^A	0.1	0.0	0.0	10435	48	48	1406	40	40
F_{15}^A	0.7	0.0	0.0	43273	48	48	4641	40	40
F_5^B	0.0	0.0	0.0	293	449	9	78	65	6
F_{10}^B	0.0	169.1	0.0	7933	5502590	9	1207	1860	6
F_{15}^B	1.5	–	0.0	93632	–	9	9973	–	6
A_2	0.0	0.0	0.0	95	27	23	31	22	13
A_3	0.0	0.0	0.0	6030	151	277	105	91	39
A_4	0.2	0.3	0.2	46642	28742	16942	752	501	129
A_5	–	0.1	–	–	2883	–	–	2606	–
A_6	–	–	–	–	–	–	–	–	–

The results for the optimal configurations in Table 5.3 demonstrate that h^L also has some potential to improve finding optimal error paths, but to a lesser extent than in the suboptimal setting. A^* with h^L has the smallest search spaces in all cases, and the best runtimes in all cases except the large C_i examples, *of which it can solve more than the other configurations due to the lower memory requirements*. The d^L heuristic, on the other hand, most of the time yields per-

formance very similar to that of BFS. None of the configurations could solve C_7, C_8, or C_9.

Table 5.3. Experimental results for our optimal configurations, i. e., BFS, A^* search with h^L, and A^* search with d^L

	runtime in s			explored states			trace
Exp.	BFS	d^L	h^L	BFS	d^L	h^L	length
C_1	0.2	0.1	0.1	35325	22673	10470	54
C_2	0.6	0.3	0.4	109583	60588	24658	54
C_3	0.8	0.4	0.5	143013	80900	28694	54
C_4	8.8	4.1	3.8	1400895	581942	184413	55
C_5	72.8	38.1	29.1	12484178	4246042	1140376	56
C_6	–	–	270.1	–	–	9250356	56
M_1	0.6	0.3	0.2	50001	17810	14128	47
M_2	2.4	1.4	0.9	223662	62887	47543	50
M_3	2.5	1.4	1.1	234587	64690	54156	50
M_4	10.6	6.4	4.9	990513	226368	180353	53
N_1	4.7	3.5	2.8	100183	50894	40482	49
N_2	22.8	20.5	16.4	442556	215012	177131	52
N_3	23.2	22.5	19.8	476622	226509	196083	52
N_4	105.5	130.5	119.9	2001222	939069	830062	55
F_5^A	0.0	0.0	0.0	1467	597	71	8
F_{10}^A	0.4	0.2	0.0	37942	13102	511	8
F_{15}^A	5.2	4.5	0.1	348827	108017	1701	8
F_5^B	0.0	0.0	0.0	362	78	54	6
F_{10}^B	0.1	0.0	0.0	5422	523	429	6
F_{15}^B	0.4	0.1	0.1	34307	1718	1504	6
A_2	0.0	0.0	0.0	630	209	62	12
A_3	0.3	0.2	0.1	103935	18792	2006	17
A_4	–	–	100.4	–	–	813303	22
A_5	–	–	–	–	–	–	–
A_6	–	–	–	–	–	–	–

To conclude this section, we remark that it is not easy to construct heuristics for the used examples by hand. It took Dierks, who is an expert in the used examples, as well as in PLC automata and their translation into timed automata, two weeks of trials and intensive experimentation before getting to a performance better than what we report above for our fully automatic technology [33].

5.5 Exploiting Automata Locations

Although the performance of the presented heuristics in this chapter is quite impressive, the heuristics so far do not include clocks in the computation of heuristic values. In Sec. 5.2 we mentioned that it does not make sense to treat clocks like integer variables when using the monotonicity abstraction to compute heuristic values. The reason for this is that clocks trivialize very fast.

In this section we discuss a possible way how to overcome this limitation. First we sketch the general idea, afterwards we present a refined version of the build-atg algorithm and prove that the resulting heuristic is admissible.

5.5.1 The General Idea

One possibility to avoid that clocks trivialize too fast is to distinguish between the clock value subsets that can be reached *at the individual automaton locations*. Due to location invariants restricting the passage of time, the intervals possible at individual locations are more restricted than the *globally* reachable interval. Particularly, constraints on how one clock value can change due to a transition often transfer to all other clocks as well since for them time elapses in the same way.

Consider the system depicted in Fig. 5.12 as an example. The system consists of two timed automata B and Q that share a common clock variable x. Say the initial state is $\langle B = b_1, Q = q_1, x = [0, \infty) \rangle$ and an error state is reached if Q is in its double circled location q_3.

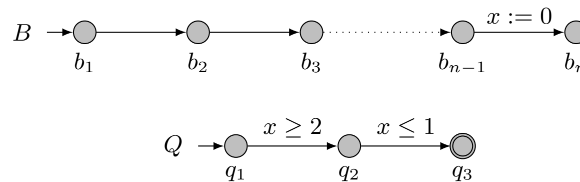


Fig. 5.12. A simple example how to split clocks over locations

The idea of *splitting* clock value subsets over individual locations when computing the ATG is as follows. First, the clock values of x that can be reached in each initial location of each automaton are $[0, \infty)$. As all other locations are not reached yet, the current clock values for these locations are empty. In the first iteration of the construction of the ATG, automaton B takes the edge to b_2. As an effect, the clock values that can be achieved in b_2 are set to $[0, \infty)$. Automaton Q takes the edge from q_1 to q_2. Because of the edge's guard, the clock values that can be achieved in q_2 are $[2, \infty)$. As long as x is not reset, the clock values

for x in location q_2 cannot be smaller than 2. Therefore, the edge from q_2 to q_3 is not enabled until the edge from b_{n-1} to b_n is taken. This edge is applied in iteration $n-1$, setting the achieved clock values for q_2 to $[0,\infty)$. After the next iteration an abstract error state is reached.

It is not difficult to see that the length of a shortest possible error trace is $n+1$. First Q takes the edge to q_2. Since x has to be reset in order to enable the edge from q_2 to q_3, B has to take all its edges. Afterward the edge from q_2 to q_3 is enabled, which yields an error trace of length $n+1$. The heuristic value of h^L and h^U for the initial state is 2. If we used the extended version of the ATG construction to compute h^L or h^U, then the heuristic value would be n.

5.5.2 The h^L_X Heuristic

In this section we present a refined version of the build-atg algorithm that treats clocks like in the example from the last section. Pseudocode for this algorithm is given in Fig. 5.13.

The build-atg$_X$ algorithm takes as input a reachability problem, consisting of a timed automata system $\mathcal{S}$ and a target formula φ, a state s for which we want to compute a heuristic value and the zone Z of that state. Lines 2–6 of Fig. 5.13 initialize the data structures. The locations and integer variables are treated as in the build-atg function (see Fig. 5.3). Clocks are represented by zones and are treated as follows. $X_k(l)$ is a zone that represents a set of overapproximated clock values that can be achieved in location l in at most k steps. Note that a step in this context corresponds to a discrete transition followed by a delay transition. In the initial iteration of the algorithm, $X_0(l)$ equals the input zone Z if l is currently reached. For all other locations l' of all automata $X_0(l')$ represents the empty zone.

Lines 9–12 of the algorithm initialize the L_{k+1}, V_{k+1} and X_{k+1} data structures by just copying the corresponding structures from the previous iteration. In line 8, the controlling expression of the while loop checks if an abstract state is reached that satisfies the target formula. The target formula is satisfied if there is a state s in $\{\langle \bar{l}, \bar{v}, \bigwedge_{q\in\bar{l}} X_k(q)\rangle \mid \bar{l} \in \Pi_i L_k(A_i), \bar{v} \in \Pi_j V_k(v_j)\}$ that satisfies the target formula. Here, Π denotes the Cartesian product operator.

The body of the for loop that starts in line 13 does the following for every edge $e = l \xrightarrow[f,r]{g_I \wedge g_X} l'$ of every automaton A_i. Here, g_I denotes the integer part of e's guard and g_X denotes the clock part. First it checks if e's start location l is in L_k and if e's integer guard g_I is satisfied by V_k (same as for the build-atg algorithm, see Fig. 5.3). If this is the case, the algorithm computes the successor zone Z'. Therefore the zone of e's start location $X_k(l)$ is first intersected with g_X. Afterwards e's clocks resets r are applied. Then, the *up* function sets the upper bounds of all clocks to infinity. The resulting zone is cut with the location

invariant of e's end location l'. Note that the computation of Z' corresponds exactly to the semantics of timed successors (see Sec. 3.4.2). If afterwards Z' is not empty, i. e., the clock guard and the location invariant are satisfied by Z', then L_{k+1} and V_{k+1} are updated as in the build-atg algorithm (see Fig. 5.3). The zone of e's end location $X_{k+1}(l')$ is set to $\|X_{k+1}(l) + Z'\|$. The plus operator computes the convex hull of both zones and $\|\cdot\|$ normalizes the resulting zone with respect to the clock ceilings of the system $\mathcal{S}$.

The for loop in line 21 applies the resets of e to all zones $X_{k+1}(l)$, where l is not a location of the current automaton A_i. This is done by setting $X_{k+1}(l)$ to $\|X_{k+1}(l) + \mathit{up}(\mathit{free}(X_{k+1}(l), r)) \wedge I(l)\|$. Again the plus operator computes the zone representing the convex hull of $X_{k+1}(l)$ and $\mathit{free}(X_{k+1}(l), r)$. The function *down* sets the lower bounds in $X_{k+1}(l)$ of all clocks in r to zero and leaves the upper bounds untouched. Lines 23–31 check if a fixpoint is reached.

Note that, for the sake of presentation, the depicted algorithm does not deal with synchronization. However the implementation thereof is straight forward. Figure 5.14 illustrates the functioning of the build-atg$_X$. The figure shows a timed automata system consisting of two automata. Let A_1 denote the automaton at the top and A_2 denote the automaton at the bottom. The automata share a common clock variable x. Location l_1 is labeled with the invariant $x \leq 3$ and location l_2' is labeled with $x \leq 1$. Suppose that the initial state of the system is given by $\langle A_1 = l_1, A_2 = l_2, x = [1,3]\rangle$ and a target formula φ is $(A_1 = l_1') \wedge (A_2 = l_2')$. The exact distance to a target state is two: first the system has to take the transition from l_1 to l_1' and afterwards the transition from l_2 to l_2'. If the system first takes the transition from l_2 to l_2', then an error state is not reachable.

The first line of the table from Fig. 5.14 gives the abstract initial state, i. e., the content of the data structures L_k and X_k at layer 0. The zones for locations l_1 and l_2 equal the zone of the initial state, the zones for all other locations are empty. In the next iteration all applicable transitions are applied. Let us begin with the transition from l_1 to l_1'. Since the guard of that transition is satisfied by $X_0(l_1)$, $X_1(l_1')$ is set to $[2, \infty)$. As an additional effect, $L_1(A_1)$ is updated to $\{l_1, l_1'\}$. Since this edge has no resets, the only thing which is done is that the upper bounds of $X_1(l_2)$ are set to infinity. Next the algorithm processes the edge from l_2 to l_2'. The zone $X_1(l_2')$ is set to $[0, 1]$. Note that actually also the upper bounds of $X_1(l_1)$ had to be increased, but because of l_1's location invariant this has no effect. The target formula is not satisfied by L_1, X_1, i. e., there is no state in $\{\langle \bar{l}, \bigwedge_{q \in \bar{l}} X_1(q)\rangle \mid \bar{l} \in L_1(A_1) \times L_1(A_2)\}$ that satisfies φ. The state $\langle A_1 = l_1', A_2 = l_2', X_1(l_1') \wedge X_1(l_2')\}$ does not satisfy φ since its zone is empty. In the next iteration, all changes are due to the reset of $l_2 \to l_2'$. It causes $X_2(l_1')$ to be $[0, \infty)$. Now the state $\langle A_1 = l_1', A_2 = l_2', X_2(l_1') \wedge X_2(l_2')\rangle$ satisfies φ, since its zone $x = [0, 1]$ is not empty.

```
1 function build-atg_X(S, φ, s, Z):
2     for each v do: V_0(v) := {s(v)}
3     for each A_i do:
4         L_0(A_i) := {s(A_i)}
5         for each l of A_i do: X_0(l) := ∅
6         X_0(s(A_i)) := Z
7     k := 0
8     while L_k, V_k, X_k ⊭_A φ do:
9         for each A_i do:
10            L_{k+1}(A_i) := L_k(A_i)
11            for each l of A_i do: X_{k+1}(l) := X_k(l)
12        for each v do: V_{k+1}(v) := V_k(v)
13        for each A_i do:
14            for each edge l --(g_I ∧ g_X) / (f,r)--> l' of A_i with l ∈ L_k(A_i) and V_k ⊨_A g_I do:
15                Z' := up(r(X_k(l) ∧ g_X)) ∧ I(l')
16                if Z' ≠ ∅ then:
17                    L_{k+1}(A_i) := L_{k+1}(A_i) ∪ {l'}
18                    for each assignment v := c_0 + Σ_{j=1}^{n} c_j · v_j of f do:
19                        V_{k+1}(v) := V_{k+1}(v) ∪ {c_0 + Σ_{j=1}^{n} c_j · e_{v_j} | e_{v_j} ∈ V_k(v_j)}
20                    X_{k+1}(l') := ‖X_{k+1}(l') + Z'‖
21                    for each l ∈ ⋃_{A≠A_i} L_k(A) do:
22                        X_{k+1}(l) := ‖X_{k+1}(l) + up(free(X_{k+1}(l), r)) ∧ I(l)‖
23        fixpoint-reached := true
24        if exists A_i such that L_{k+1}(A_i) ≠ L_k(A_i) then: fixpoint-reached := false
25        if fixpoint-reached then:
26            for each A_i do:
27                for each l of A_i do:
28                    if X_{k+1}(l) ≠ X_k(l) then: fixpoint-reached := false
29        if fixpoint-reached and V_{k+1}(v) = V_k(v) for all v then:
30            minlayer := ∞
31            return
32        k := k + 1
33    minlayer := k
```

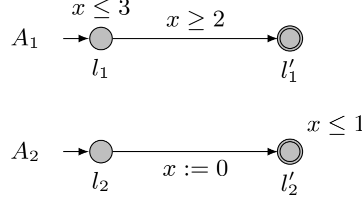
Fig. 5.13. The build-atg$_X$ algorithm is an extension of the build-atg algorithm with clocks split over locations.

If we replace the build-atg function with the build-atg$_X$ function in the implementation of the h^L heuristic we get the h^L_X heuristic. Pseudocode thereof is given in Fig. 5.15.

5.5.3 Admissibility of the h^L_X Heuristic

In this subsection we will prove that the h^L_X heuristic from Fig. 5.15 is an admissible heuristic. Admissibility is a consequence of the next theorem.

Theorem 5.7. *In the kth iteration of the build-atg$_X$ algorithm the set* $S_k = \{\langle \bar{l}, \bar{v}, \bigwedge_{q \in \bar{l}} X_k(q) \rangle \mid \bar{l} \in \prod_i L_k(A_i), \bar{v} \in \prod_j V_k(v_j)\}$ *is a superset of the states*



Layer k	$L_k(A_1)$	$L_k(A_2)$	$X_k(l_1)$	$X_k(l'_1)$	$X_k(l_2)$	$X_k(l'_2)$
0	$\{l_1\}$	$\{l_2\}$	$[1,3]$	$\emptyset$	$[1,3]$	$\emptyset$
1	$\{l_1, l'_1\}$	$\{l_2, l'_2\}$	$[0,3]$	$[2,\infty)$	$[1,\infty)$	$[0,1]$
2	$\{l_1, l'_1\}$	$\{l_2, l'_2\}$	$[0,3]$	$[0,\infty)$	$[1,\infty)$	$[0,1]$

Fig. 5.14. The functioning of the build-atg$_X$ algorithm

```
1 function h^L_X(S, φ, s, Z):
2     build-atg_X(S, φ, s, Z)
3     return minlayer
```

Fig. 5.15. The h^L_X heuristic

that are reachable within at most k steps. Here, $\prod$ denotes the Cartesian product operator.

Proof (Theorem 5.7). We will prove the proposition by induction over k. For $k = 0$, S_k just contains the initial state for which we want to compute a heuristic value. For the induction steps, we have to show that S_{k+1} overapproximates the state that are reachable within at most $k+1$ steps, given that S_k contains at least the states that are reachable within at most k steps. This holds if we show that for all applicable transitions t with $t = s \rightarrow s'$, where $s \in S_k$, then $s' \in S_{k+1}$.

Let t be such a transition that is induced by the edge $e = l_i \xrightarrow[f,r]{g_I \wedge g_X} l'_i$, $s = \langle \bar{l}, \overline{v}, Z \rangle$ and $s' = \langle \bar{l}', \overline{v}', Z' \rangle$. Since $s(A_j) \in L_k(A_j)$ for all A_j and $s(v) \in V_k(v)$ for all v, the edge e is processed by the algorithm, i. e., the body of the for loop starting in line 14 is executed. Next we have to show that $Z' \subseteq X_{k+1}(s'(A_j))$ for all A_j. If this is the case we know that $Z' \subseteq \bigwedge_j X_{k+1}(s'(A_j))$ and hence $s' \in S_{k+1}$. Let A_i denote the automaton to which e belongs. For the zone D' of the successor state s',

$$\begin{aligned} D' &= up(r(Z \wedge g_X)) \wedge I(s') \\ &\subseteq up(r(X_k(s(A_i)) \wedge g_X)) \wedge I(s') \end{aligned}$$

holds by induction hypothesis. As $I(s'(A_i))$ is a conjunct of $I(s')$, $I(s') \subseteq I(s'(A_i))$ holds for all automata A_i. From this it follows that $D' \subseteq Z'$, where Z' is the zone from the algorithm in line 15.

As D' is not empty, Z' is not empty either and hence the integer and clock updates are applied. We do not include integer variables and locations in this proof as they are handled in exactly the same way as in the build-atg algorithm. For $X_{k+1}(s'(A_i))$ it holds that $D' \subseteq X_{k+1}(s'(A_i))$, since $X_{k+1}(s'(A_i))$ contains Z' (line 20 of the algorithm).

It remains to prove that $D' \subseteq X_{k+1}(s'(A_j))$ for all automata $A_j \neq A_i$. For the zone D' of the successor state it holds that

$$\begin{aligned} D' &= \mathit{up}(r(Z \wedge g_X)) \wedge I(s') \\ &\subseteq \mathit{up}(r(X_k(s(A_j)) \wedge g_X)) \wedge I(s') \\ &\subseteq \mathit{up}(r(X_{k+1}(s(A_j)) \wedge g_X)) \wedge I(s') \end{aligned}$$

since by induction hypothesis it holds that $D \subseteq X_k(s(A_j))$ and furthermore $X_k(s(A_j)) \subseteq X_{k+1}(s(A_j))$ for all A_j by construction.

Since $s'(A_j) = s(A_j)$ for all automata $A_j \neq A_i$ and $Z \wedge g_X \subseteq Z$ for all zones Z, the following holds:

$$\begin{aligned} D' &\subseteq \mathit{up}(r(X_{k+1}(s(A_j)) \wedge g_X)) \wedge I(s') \\ &\subseteq \mathit{up}(r(X_{k+1}(s'(A_j)))) \wedge I(s') \\ &\subseteq \mathit{up}(r(X_{k+1}(s'(A_j)))) \wedge I(s'(A_j)). \end{aligned}$$

Since a reset operation r affects the lower *and* the upper bound of a clock, only setting the lower bound to zero and leaving the upper bound unchanged yields a zone that subsumes the original zone. Therefore we have

$$\begin{aligned} D' &\subseteq \mathit{up}(r(X_{k+1}(s'(A_j)))) \wedge I(s'(A_j)) \\ &\subseteq \mathit{up}(\mathit{free}(X_{k+1}(s'(A_j)), r) \wedge I(s'(A_j)). \end{aligned}$$

By this equation it follows that $D' \subseteq X_{k+1}(s'(A_j))$ for all automata A_j. □

5.5.4 Evaluation

We implemented this algorithm and evaluated it on the benchmarks from Sec. 5.4. It turned out that compared to the h^L heuristic, the h^L_X heuristic found only slightly shorter error traces and the number of explored states only drops

down very marginally. Due to the computational overhead, the runtime of the h^L_X heuristic was up to several orders higher than the time needed for the h^L heuristic. Of course it might be possible to come up with a more efficient implementation, but it remains puzzling why the guidance of the h^L_X heuristic on our benchmarks is only comparable to that of the h^L heuristic, although the latter heuristic does not take clocks into account.

Thinking a little more about the values a clock can achieve in a location, it is obvious that in each location the possible values for each clock are bounded. Probably the easiest way to derive such bounds is as follows. An upper bound for a clock in a location is given by the invariant of that location. A lower bound for each clock is zero since clocks only evaluate to the non-negative reals.

Typically, if the values of all clocks in a location l are between their respective lower and upper bounds, the clock guard of each outgoing edge is satisfied. This is because of the following reasons. As the lower bound of all clocks is zero, every clock constraint of the form $x \leq c$ or $x < c$, where x is a clock and c a positive integer, is always satisfied. Clock constraints of the form $x \geq c$ or $x > c$ that occur in the edge guard are satisfied if the upper bound of x, induced by l's location invariant, is greater (or equal) than c. But this is normally the case, since otherwise the edge is never applicable at all. Probably only for synchronized transitions this does not always hold to exclude certain synchronization combinations.

So let us come back to the question why the h^L_X heuristics behaves only similarly to the h^L heuristic on our benchmarks. It turned out that after one or two iterations of the build-atg$_X$ algorithm, almost all clocks trivialized, i. e., their lower bounds are zero and their upper bounds are equal to the corresponding location invariant. This means that after a few iterations of the build-atg$_X$ algorithm, X_k has reached a fixpoint. From then on all clock guards are satisfied and thus including the clocks does not provide useful information.

5.6 Conclusion

We have introduced methods for automatically generating two heuristic guidance functions in MCTA. We have shown the functions' potential for yielding more efficient finding of error states, by reducing the number of search states that need to be considered, as well as guiding the search to short error paths.

We also tried to include clocks in the computation of heuristic values in a nontrivial way, but it seems that, at least on our benchmarks, this only brings a slight improvement in terms of explored states and reported error trace length. Since the runtime of h^L_X is up to orders of magnitude higher compared to h^L, we therefore did not investigate this direction further.

6

Heuristic Functions Based on Predicate Abstraction

Predicate abstraction [50] is an abstraction method which is broadly established in model checking. In this chapter, we show how to use predicate abstraction to generate heuristic functions. The overall methodology follows the *pattern database* approach from AI [29]. In this approach, the abstract state space is exhaustively built in a preprocessing step and used as a lookup table during search. While typically pattern databases use rather primitive abstractions ignoring some of the relevant symbols, we use predicate abstraction, dividing the state space into equivalence classes with respect to a list of logical expressions, the predicates. We empirically explore the behavior of the resulting family of heuristics. In particular, while several challenges remain open, we show how to obtain heuristic functions that are competitive with other state-of-the-art heuristics for directed model checking. This chapter is based on joint work [63] which was mainly done by Jörg Hoffmann and Jan-Georg Smaus.

6.1 Predicate Abstraction

Predicate abstraction is an abstraction method that is typically applied to a system before it is analyzed with a model checker. As a consequence of this abstraction, the state space of the abstract system is usually much smaller than the state space of the original system, and thus model checking is less expensive. Depending on certain properties of the abstraction and the outcome of the analysis of the abstract system, one can draw conclusions about the real system. For instance, if the property φ under consideration holds for the abstract system and the abstraction is property preserving with respect to φ, then φ also holds for the real system. Methods based on predicate abstraction have been extremely successful in the verification of temporal safety properties [5, 19, 58].

6.1.1 The General Idea

The general idea of predicate abstraction [50] is to partition the state space of the system under consideration into *equivalence classes*. These equivalence classes are induced by a set of predicates $\mathcal{P}$, i. e., a set of Boolean expressions defined over the variables of the system. Thus a system state s can be represented by a vector $\overline{b}$ of truth values, stating which of the predicates are true or false in s. Two system states s_1 and s_2 are equivalent with respect to $\mathcal{P}$ iff they satisfy the same predicates from $\mathcal{P}$.

For every transition $s \to s'$ of the original system, there is an abstract transition $\overline{b} \to \overline{b}'$. The abstract system thus overapproximates the real system, which enables us to analyze the abstract system in order to draw certain kinds of conclusions about the real system. If there is no reachable abstract error state in the abstract system, then there is no reachable error state in the real system.

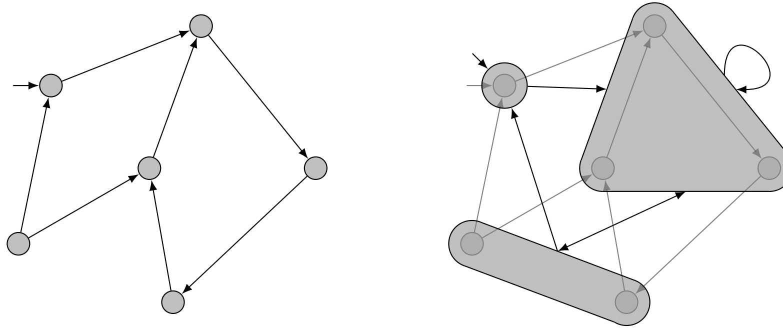


Fig. 6.1. The general idea of predicate abstraction

Figure 6.1 illustrates the relation of the original system's and the abstract system's state space obtained via predicate abstraction. The state space of the original system is depicted on the left. The right side of the picture shows the abstract state space induced by some predicate abstraction and a transparent overlay of the original state space to ease observing the correspondence of concrete and abstract states. The abstract states, i. e., the equivalence classes, are the grayed regions. The circle nodes in these grayed regions are the states of the original state space that are equivalent with respect to the chosen predicates. There is an abstract transition (a black arrow) from the abstract state $\overline{b}$ to the abstract state $\overline{b}'$, if there is a concrete transition (a grayed arrow) connecting the concrete states s and s' that are subsumed by $\overline{b}$ and $\overline{b}'$, respectively.

6.1.2 Predicate Abstraction for Timed Automata

In this subsection we explain how to generate a predicate abstraction for a system of timed automata. Let $\mathcal{S} = A_1 \parallel \ldots \parallel A_n$ be a system of timed automata. Further, let X and V denote the set of clocks and integer variables, respectively, that appear in at least one automaton A_i of the system. In this section we will refer to the set $\mathrm{var}(\mathcal{S}) = X \cup V \cup \{A_i\}_{i=1}^{n}$ as the variables of the system $\mathcal{S}$.

A predicate abstraction of a timed automata system $\mathcal{S}$ is defined by a finite set $\mathcal{P}$ of predicates over $\mathrm{var}(\mathcal{S})$. There are three different types of predicates, namely predicates over integer variables, predicate over clocks and predicates over automata. A predicate over integer variables is a linear expression of the form $c_0 \bowtie \sum_{i=1}^{n} c_i \cdot v_i$, where $c_0, c_i \in \mathbb{Z}$, $v_i \in V$ for $i = 1, \ldots, n$ and $\bowtie$ is a comparator. A predicate over clocks is either of the form $x \bowtie c$ or $x - y \bowtie c$, where $x, y \in X$ and $c \in \mathbb{Z}$. A location predicate is of the form $A = l$, where l is a location of automaton A. Such a predicate evaluates to true in a state s iff the current location of A is l, i. e., $s \models (A = l)$ iff $s(A) = l$.

We denote by a *bitvector* for $\mathcal{P}$ any conjunction that contains exactly each $p \in \mathcal{P}$, possibly negated. Let $\mathcal{T}(\mathcal{S}) = (S, s_0, T)$ be the state space of $\mathcal{S}$. For $s \in S$, we will henceforth use $s \in \mathcal{T}(\mathcal{S})$ as an abbreviation. For a bitvector $\bar{b}$, the set $[\bar{b}] = \{s \in \mathcal{T}(\mathcal{S}) \mid s \models \bar{b}\}$ is called the *extension* of $\bar{b}$. The extensions of the bitvectors are equivalence classes in the state space $\mathcal{T}(\mathcal{S})$ of $\mathcal{S}$. If $s \in [\bar{b}]$, then we will also write $[s]$ instead of $[\bar{b}]$. The *abstract state space* for $\mathcal{P}$, denoted $\mathcal{T}^{\mathcal{P}}(\mathcal{S})$, is the directed graph where the nodes are all bitvectors for $\mathcal{P}$, and there is a transition from $\bar{b}$ to $\bar{b}'$ iff there exist $s \in [\bar{b}]$ and $s' \in [\bar{b}']$ such that there is a transition from s to s' in $\mathcal{S}$. Obviously, $\mathcal{T}^{\mathcal{P}}(\mathcal{S})$ is an overapproximation of $\mathcal{T}(\mathcal{S})$, i. e., if s' is reachable from $s \in \mathcal{T}(\mathcal{S})$, then $[s']$ is reachable from $[s] \in \mathcal{T}^{\mathcal{P}}(\mathcal{S})$.

6.1.3 Building the Abstract State Space

When building the abstract state space, one has to frequently decide if there is a transition from a bitvector $\bar{b}$ to a bitvector $\bar{b}'$. Obviously, enumerating $[\bar{b}]$ and $[\bar{b}']$ is not feasible, because the number of bitstrings is exponential in the number of predicates. Instead, one formulates the transitions of the real system as conjunctions over constraints on variable values before and after the transition. We will explain how to obtain these formulas in the next subsection. With these formulas, the test if there is a transition between $[\bar{b}]$ and $[\bar{b}']$ comes down to the satisfiability of a conjunction of constraints. We use this method to generate abstract state spaces, regressing from the target formula in a breadth-first manner to build the fraction of the abstract state space that is reachable from that condition. Concretely, we repeatedly consider a formula φ that formulates the

properties of the set of states that should be inserted next into the state space, i. e., φ is initially the target formula, and later the conjunction of the constraints given by the regressed abstract state and the transition.

The precise method to find the corresponding abstract states would be to enumerate all bitvectors and check if they are satisfiable in conjunction with φ. As this is not feasible, we additionally use a Cartesian abstraction, and set the resulting state to $\bigwedge_{p \in \mathcal{P}} \{p \mid \varphi \models p\} \wedge \bigwedge_{p \in \mathcal{P}} \{\neg p \mid \varphi \models \neg p\}$. That is, one just checks which "bits" are definitely implied, and leaves the others unspecified. We denote such partial bitvectors with $\bar{c}$, keeping the notation $[\bar{c}] = \{s \in \mathcal{T}(\mathcal{S}) \mid s \models \bar{c}\}$. By $\mathcal{T}_{\mathcal{P}}^{C}(\mathcal{S})$ we denote the abstract state space, i. e., the graph of partial bitvectors built in this way.

6.1.4 Encoding Transitions

For the generation of the abstract state space, we have to translate every transition $t = s \rightarrow s'$, where $s, s' \in \mathcal{T}(\mathcal{S})$ are states of a timed automata system $\mathcal{S}$ into a corresponding conjunction over Boolean expressions. The variables that occur in these expressions refer to the automata, the integer variables and the clock variables of the systems. In these encodings, primed variables refer to the successor state s', unprimed variables refer to the source state s of t.

A timed transition with duration $d \geq 0$ can be encoded by the conjunction over the following expressions. The conjunction $\bigwedge_i (A_i = A_i') \wedge \bigwedge_j (v_j = v_j')$ encodes that a timed transition only affects clock variables, i. e., automata locations and integer variables remain unchanged. The values of each clock x_k is incremented by d, this is stated by the next conjunction $\bigwedge_k (x_k + d = x_k') \wedge d \geq 0$. Finally, the successor state s' has to satisfy the invariant $I(s') = \bigwedge_i s(A_i)$. Therefore the conjunction $\bigwedge_m \bigwedge_n (A_m \neq l_n \vee I(l_n))$ is introduced. This means that if automaton A_m is in location l_k, then the successor state has to satisfy $I(l_k)$. Note that the variables that occur in the encoding of the location invariants are primed, i. e., these variables refer to the successor state.

A discrete transition induced by some τ edge $l \xrightarrow[f,r]{g,\tau} l'$ of automaton A_i can be encoded by the conjunction over the following expressions. The conjunct $(A_i = l) \wedge (A_i' = l') \wedge \bigwedge_{j \neq i} (A_j = A_j')$ encodes that only the location of automaton A_i is changed by the transition. All other automata remain in their current locations. If $v := c_o + \sum_{i=1}^{n} c_i \cdot v_i$ is an effect of the edge, then the conjunct $v' = c_o + \sum_{i=1}^{n} c_i \cdot v_i$ is introduced. For all other variables v that are not affected by the edge's effect, a conjunct of the form $v' = v$ is introduced. Clock resets are treated similarly: for each clock x that is reset by the edge, a conjunct of the form $x' = 0$ is introduced. For all other clocks x, there are conjuncts of the form $x' = x$. Synchronized transitions are translated in a similar fashion.

6.2 Pattern Database Heuristics

As already said, this chapter is based on joint work with Hoffmann, Smaus, Rybalchenko and Podelski. In our paper [63], predicate abstraction is for the first time, as far as we are aware, used to define heuristic functions. In a manner reminiscent of the *pattern database* approach [29], we construct the entire abstract state space prior to the actual search. During search, the abstract state space is used as a lookup table, i. e., states are mapped onto their abstract counterparts, and the error distance of the counterpart is taken as the heuristic estimate. In difference to our approach, pattern databases traditionally use simple abstractions, for example Qian and Nymeyer obtain their abstractions by mostly ignoring some of the relevant symbols [86]. In Chap. 7 we describe their approach as well as an improvement in more detail.

6.2.1 Predicate Abstraction Pattern Databases

To turn a predicate abstraction into a heuristic function, we simply map the state for which we want a heuristic estimate into the abstract state space and read the error distance from there. Precisely, if φ is the target formula, $\mathcal{P}$ is the predicate set, and $s \in \mathcal{T}(\mathcal{S})$ is a state of the system $\mathcal{S}$, we get

$$h^{\mathcal{P}}(s) = \min_{\bar{c}_1, \bar{c}_2 \in \mathcal{T}_{\mathcal{P}}^{C}(\mathcal{S})} \{d^{\mathcal{P}}(\bar{c}_1, \bar{c}_2) \mid s \in [\bar{c}_1], \exists s' \in [\bar{c}_2] : s' \models \varphi\}.$$

Here, $d^{\mathcal{P}}(\cdot,\cdot)$ denotes the graph distance in $\mathcal{T}_{\mathcal{P}}^{C}(\mathcal{S})$, which is ∞ if there is no path from the first to the second argument. Since the bitvectors $\bar{c}$ in $\mathcal{T}_{\mathcal{P}}^{C}(\mathcal{S})$ are partial, there may be several $\bar{c}_1$ with $s \in [\bar{c}_1]$. The heuristic value $h^{\mathcal{P}}(s)$ of s is ∞ if no error state is reachable in $\mathcal{T}_{\mathcal{P}}^{C}(\mathcal{S})$ from any such $\bar{c}_1$. This implies that no error state is reachable in $\mathcal{T}(\mathcal{S})$ from s. Since we minimize over all $\bar{c}_1$ and $\bar{c}_2$, it follows that $h^{\mathcal{P}}$ is admissible. This is stated by the next proposition.

Proposition 6.1. *Let $\langle \mathcal{S}, \varphi \rangle$ be a reachability problem, $\mathcal{P}$ be a set of predicate and s be a state of $\mathcal{T}(\mathcal{S})$. Then*

$$h^{\mathcal{P}}(s) \leq \min_{s' \in \mathcal{T}(\mathcal{S})} \{d(s, s') \mid s' \models \varphi\},$$

where $d(\cdot,\cdot)$ denotes the graph distance in $\mathcal{T}(\mathcal{S})$.

The proposition holds for the following reasons. As the abstract state space is an overapproximation of the real state space, there exists an abstract transition for every transition of the real system. As a consequence, every solution for the system is also a solution for the abstract system. Another interesting property of this kind of heuristics is that they are monotone in the predicate set. This is stated in the next proposition.

Proposition 6.2. *Let $\langle \mathcal{S}, \varphi \rangle$ be a reachability problem and let $\mathcal{P}_1$ and $\mathcal{P}_2$ be predicate sets such that $\mathcal{P}_1 \subseteq \mathcal{P}_2$. Further, let s be a state of $\mathcal{T}(\mathcal{S})$. Then it holds that*

$$h^{\mathcal{P}_1}(s) \leq h^{\mathcal{P}_2}(s).$$

This is simply because, $\mathcal{T}_{\mathcal{P}_2}(\mathcal{S})$ makes all distinctions that $\mathcal{T}_{\mathcal{P}_1}(\mathcal{S})$ makes. What it tells us is that, if we refine a predicate set $\mathcal{P}$ by inserting new predicates into it, we obtain a heuristic function that *dominates* the previous one, in that it provides a potentially better lower bound. In fact, in such cases, one can expect that A^* with $h^{\mathcal{P}_2}$ expands fewer states than A^* with $h^{\mathcal{P}_1}$ in practice. More precisely, Pearl proved that, except for some borderline cases, every state that is expanded by $h^{\mathcal{P}_2}$ is also expanded by $h^{\mathcal{P}_1}$ [82].

6.2.2 Implementation Details

We implemented the proposed heuristics from this chapter in the UPPAAL/DMC model checker. Therefore, we had to modify the definition of $h^{\mathcal{P}}$ to work on the UPPAAL/DMC search space, i. e., the zone automaton $\mathcal{Z}(\mathcal{S})$ of the system $\mathcal{S}$. Let $s \in \mathcal{Z}(\mathcal{S})$, φ be the target formula and $[s] \subseteq \mathcal{T}(\mathcal{S})$ be the corresponding set of system states. We define

$$h^{\mathcal{P}}(s) = \min_{\overline{c}_1, \overline{c}_2 \in \mathcal{T}^C_{\mathcal{P}}(\mathcal{S})} \{ d^{\mathcal{P}}(\overline{c}_1, \overline{c}_2) \mid [s] \cap [\overline{c}_1] \neq \emptyset, \exists s' \in [\overline{c}_2] : s' \models \varphi \}.$$

Obviously, again this leads to an admissible heuristic function, and the monotonicity in the predicate set is preserved. While the definition looks fairly complicated, we will see in the next section that, once the abstract state space $\mathcal{T}^C_{\mathcal{P}}(\mathcal{S})$ is built, the function can be implemented quite efficiently.

For every search state UPPAAL/DMC encounters, the heuristic function must be computed. This makes it indispensable to implement that function efficiently. To overcome this problem, we do the following:

1. We formulate the mapping of search states into $\mathcal{T}^C_{\mathcal{P}}(\mathcal{S})$ as a *bitset* inclusion problem, and
2. we use a tree-like data structure to address this inclusion problem efficiently.

Remember that the abstract states in $\mathcal{T}^C_{\mathcal{P}}(\mathcal{S})$ are partial bitvectors $\overline{c}$, i. e., sets of possibly negated predicates $p \in \mathcal{P}$. Given a UPPAAL/DMC search state $s \in \mathcal{Z}(\mathcal{S})$, by $\psi(s)$ we denote the constraint conjunction corresponding to s, i. e., location and integer valuations plus the zone of s. We define the following bitset

$$\begin{aligned}\overline{c}(s) = &\{p \mid \psi(s) \models p\} \\ &\cup \{\neg p \mid \psi(s) \models \neg p\} \\ &\cup \{p, \neg p \mid \psi(s) \not\models p, \psi(s) \not\models \neg p\}.\end{aligned}$$

In words, $\overline{c}(s)$ contains all bits that may possibly be true in s, i. e., that are satisfied by at least one system state in $[s]$. While the definition of $\overline{c}(s)$ involves entailment checks, due to our particular circumstances $\overline{c}(s)$ can be computed efficiently. First, the UPPAAL/DMC search state contains precise valuations for all locations and integer variables; the uncertainty is exclusively about the clocks. So predicates not involving clocks can simply be evaluated in s. Second, whether a clock predicate is implied by s or not can be read off from a single pass over the zone of s, which is given in form of a difference bound matrix. We observe:

$$[s] \cap [\overline{c}_1] \neq \emptyset \Leftrightarrow \overline{c}(s) \supseteq \overline{c}_1.$$

This is because $[s] \cap [\overline{c}_1] \neq \emptyset$ iff $\overline{c}_1$ contains no bit that is known to be false in s. In other words, if all bits contained in $\overline{c}_1$ may be true in s. We obtain:

$$h^{\mathcal{P}}(s) = \min_{\overline{c}_1, \overline{c}_2 \in \mathcal{T}^C_{\mathcal{P}}(\mathcal{S})} \{d^{\mathcal{P}}(\overline{c}_1, \overline{c}_2) \mid \overline{c}(s) \supseteq \overline{c}_1, \exists s' \in [\overline{c}_2] : s' \models \varphi\}.$$

This is a syntactic characterization except for $\exists s' \in [\overline{c}_2] : s' \models \varphi$; but that we have dealt with already when building $\mathcal{T}^C_{\mathcal{P}}(\mathcal{S})$; we simply mark, during that preprocess, the respective $\overline{c}_2$, i. e., the start state in our backward search as error states. Of course, we also annotate each state $\overline{c}$ with its distance to the nearest error state. Since we build $\mathcal{T}^C_{\mathcal{P}}(\mathcal{S})$ backward breadth-first, we get that distance for free. We are left with the following problems:

1. In the preprocess, store $\mathcal{T}^C_{\mathcal{P}}(\mathcal{S})$ as a set of bitsets annotated with their error distance,
2. during search, quickly find all bitsets that are contained in $\overline{c}(s)$.

Both can be accomplished using a data structure called *Unlimited Branching Tree* [60]. In a nutshell, this is a tree structure that stores sets of sets, exploiting shared elements between the sets to optimize space usage and access time for answering subset queries of the precise kind we need here. The details are not essential for understanding our approach, so we omit them. A node in the tree may have as many branches as there are distinct elements in the sets, hence the name.

6.2.3 Selecting Predicates

An important characteristic of our method, which it shares with traditional pattern databases, is that it yields a very large family of heuristics, rather than just a single one. Every different set of predicates yields a different abstract state space, which gives a different heuristic function. The main question is: *How should we choose the predicates?* This is the same main question as in the standard use of predicate abstraction. However, in our approach the abstraction does

not have to be precise enough to verify the property of interest, in order to be useful. So we have much more freedom of design.

In this subsection, we introduce two approaches to choosing a set of predicates. The first one simply collects the predicates from the syntax, e. g. transition guards of the timed automata system, which is not likely to be property-preserving. The second one uses *counterexample guided abstraction refinement*.

Selecting Predicates Syntactically

The first approach to building a set of predicates is a syntax-based approach. As indicated before, the abstraction predicates are simply read off the description of the timed automata system. Given a timed automata system $\mathcal{S} = A_1 \parallel \ldots \parallel A_n$, the created set of predicates consists of all expressions that appear as an edge guard, or as a location invariant of some automaton A_i. Further, the abstraction distinguishes between the locations, which is equivalent to including the location predicate $A_i = l$ for each automaton A_i of the system and each location l of A_i.

Selecting Predicates via Abstraction Refinement

The second approach to building a set of predicates is based on counterexample guided abstraction refinement (CEGAR) [25]. CEGAR is an iterative abstraction refinement methodology to analyze if a system satisfies a given property. Therefore, the system under consideration is first abstracted. Afterwards, the abstract system is examined using a model checker. If there exists an abstract counterexample, then it is analyzed whether it is spurious or not. If the abstract counterexample is spurious, then the initial abstraction is refined in order to exclude the spurious counterexample. In Chap. 9, we give a more detailed introduction to CEGAR, as well as an application to timed automata.

Here we apply CEGAR to generate a set of predicates. We implemented this method via an interface to the ARMC tool [85]. ARMC is a recent model checker based on CEGAR. Predicates are generated from spurious error paths by an analysis using a constraint based interpolation [90] to find a concise reason for the failure (the spuriousness) of the path. We modified ARMC to feature a maximal number of iterations as an input parameter. If ARMC finds a correct abstraction, i. e., one with no reachable error states, ARMC stops with no output, causing our overall program to terminate. Recall that the absence of abstract error traces implies the absence of real error traces. If ARMC finds a real error trace, it stops and outputs the abstract state space. The same happens otherwise, i. e., if the maximum iteration is reached. The abstract state space is read in, and stored for later lookup.

For our purposes, we can stop the process at any time — we do not have to wait until a real error path is found. In our experiments, we simply fix a number of refinement iterations which becomes an input parameter. We will see that, by using abstraction refinement for a heuristic — combining abstraction refinement and state space search — we can solve examples that cannot be solved by either method (abstraction refinement or blind state space search) alone. Generating the predicates using abstraction refinement is, we think, particularly promising: this technique has the power to adapt the heuristic function quite flexibly and intelligently to the individual problem instance at hand. Surprisingly, we were not yet able to obtain entirely convincing results with the technique; we believe there is hope for the future. This will be discussed in detail later.

6.2.4 Combining Multiple Pattern Databases

Apart from the parametrization given by the choice of predicates, we explore another parameter defining how the timed automata system is *split* into several parts. It turns out that predicate abstraction is much too time-consuming when done on the entire automata system. So we apply another abstraction method beforehand. We define a partitioning of the set of automata, i. e., we *split* the system and hand each part to the predicate abstraction engine *in separate*. The splits are made so that few potential interactions are violated. The transition guards responsible for the violated interactions are removed. During search, a heuristic value is looked up in each part, i. e., in each corresponding abstract state space. These values are aggregated by taking their sum as the overall heuristic value. Since we removed the guards responsible for violated interactions, this aggregated heuristic value is still admissible. This kind of heuristics is commonly referred to as additive pattern databases. There are many possible strategies for making the split. We use a simple greedy strategy parametrized by the *split bound* b, i. e., the maximal number of automata in a single part of the partitioning.

A very simple abstraction method is to partition the set of automata contained in a system. As said, we use this abstraction prior to predicate abstraction, in order to make the latter feasible. One simply considers each part of the partitioning — a subset of the automata — in separate. The only problem with this approach is that, of course, the automata typically interact with each other in various ways, and cannot be split without violating such interactions. We identify a possible definition of what interaction means. We approximate that definition to obtain an admissible splitting strategy.

Let e_1 be an edge of automaton A_1, and e_2 be an edge of automaton $A_2 \neq A_1$. Let f be an effect of e_1, i. e., a variable assignment, or a synchronization label, and let γ be the guard of e_2, i. e., a constraint over variables, or a

synchronization label. Let π be a computation trace of the system on which e_1 occurs before e_2. We say that f *affects* γ if the following holds. When removing f from e_1 and simulating the execution of π while ignoring the guards of all edges between e_1 and e_2, γ is no longer satisfied at the point when e_2 should be executed. Similarly, this definition is made also for location invariants. We say that automaton A_1 affects automaton A_2 if there is an effect of an edge of A_1 that affects a guard of an edge, or a location invariant of A_2. We say that A_1 and A_2 *interact* if A_1 affects A_2, or vice versa.

Proposition 6.3. *Let $\langle \mathcal{S} = A_1 \parallel \ldots \parallel A_n, \varphi\rangle$ be a reachability problem, where $\mathcal{S}$ is a timed automata system and φ a target formula. Further, let $\Pi \subseteq \{A_1, \ldots, A_n\}$ such that no automaton which is not in Π affects any automaton in Π. Then, for every state s of $\mathcal{T}(\mathcal{S})$, the following holds:*

$$\min_{s' \in \mathcal{T}(\mathcal{S}|_\Pi)} \{d_\Pi(s|_\Pi, s') \mid s' \models \varphi|_\Pi\} \leq \min_{s' \in \mathcal{T}(\mathcal{S})} \{d(s, s') \mid s' \models \varphi\},$$

where $s|_\Pi$ is s restricted to the locations and variables mentioned in Π, $\mathcal{T}(\mathcal{S}|_\Pi)$ is the state space of Π, $\varphi|_\Pi$ is φ restricted to conjunctions over variables and locations mentioned in Π and $d(\cdot,\cdot)$ or $d_\Pi(\cdot,\cdot)$ is the graph distance in $\mathcal{T}(\mathcal{S})$ or $\mathcal{T}(\mathcal{S}|_\Pi)$, respectively.

In words, the isolated automata Π provide an admissible distance estimate. The reason for this is that any solution path for $\langle \mathcal{S}, \varphi\rangle$ can be restricted onto the edges present in Π, to obtain a solution for $\langle \Pi, \varphi|_\Pi\rangle$. Otherwise, if that restricted path was not a solution for Π, a constraint in Π would be unsatisfied and we could construct a contradiction since an edge on the sub-path for $\{A_1, \ldots, A_n\} \setminus \Pi$ would have to affect that constraint. Note that, in particular, Prop. 6.3 says that, if $\langle \mathcal{S}, \varphi\rangle$ is solvable, then $\langle \Pi, \varphi|_\Pi\rangle$ in isolation is also solvable. We further have:

Proposition 6.4. *Let $\langle \mathcal{S}, \varphi\rangle$ be a reachability problem, and $\Pi_1, \ldots, \Pi_m$ be a partitioning of $\mathcal{S}$, with $\Pi_i \cap \Pi_j = \emptyset$ for $i \neq j$, such that for all automata $A_i \in \Pi_i$ and $A_j \in \Pi_j$ with $i \neq j$ holds that A_i and A_j do not interact. Then, for any state s of $\mathcal{T}(\mathcal{S})$, the following holds:*

$$\sum_{i=1}^{m} \min_{s' \in \mathcal{T}(\mathcal{S}|_{\Pi_i})} \{d_{\Pi_i}(s|_{\Pi_i}, s') \mid s' \models \varphi|_{\Pi_i}\} \leq \min_{s' \in \mathcal{T}(\mathcal{S})} \{d(s, s') \mid s' \models \varphi\},$$

where the notations are as in Prop. 6.3.

This tells us that we can safely add the individual heuristic values. The reason is that we can partition any solution path for $\langle \mathcal{S}, \varphi\rangle$ into independent solution paths for each reachability problem $\langle \Pi_1, \varphi|_{\Pi_1}\rangle, \ldots, \langle \Pi_m, \varphi|_{\Pi_m}\rangle$.

What we have just seen is not yet practical since there normally is no split that does not violate *any* interaction. Otherwise there would be no point in posing both parts of the system within the same problem. We become practical by finding *potential* interactions, and simply removing guards that constitute violated potential interactions. Concretely, we use the simplistic notion saying that effect f cannot affect condition γ if the variable x affected by f and any variable that can transitively be affected by the value of x, does not appear in γ. For example, $x := 1$ can affect $x + y > 2$. On the other hand, $x := 1$ can affect $y > 2$ if there also is an effect $y := x$ somewhere, but not if there is no chain of variables from x to y. In our preprocess, we simply consider all pairs of occurring f and γ, and check if they satisfy this criterion. If they do not, we say that they have a potential interaction. We then greedily put automata together into one part of the partitioning, such that few potential interactions to automata in other parts remain. For those interactions that do remain, we remove the responsible γ. Note that the latter will in particular remove synchronization actions that also occur in other parts. Figure 6.2 shows our partitioning algorithm. The input is a reachability problem $\langle \mathcal{S}, \varphi \rangle$ and the split bound b, i. e., the maximum number of automata in one partition.

```
1  function partition-system(S = A_1 || ... || A_n, φ, b):
2      rem := {A_1, ..., A_n}
3      i := 1
4      while {A ∈ rem | A affects φ} ≠ ∅ do:
5          select A ∈ {A ∈ rem | A affects φ}
6          Π_i := {A}
7          while |Π_i| < b and (rem \ Π_i) ≠ ∅ do:
8              for each A' ∈ (rem \ Π_i) do:
9                  h_1(A') := |{A'' ∈ Π_i | A' can synchronize with A''}|+
                       |{A'' ∈ Π_i | A' interacts with A''}|
10                 h_2(A') := number of target constraints that A' affects
11             Π_i := Π_i ∪ arg max{(h_1(A'), h_2(A')) | A' ∈ (rem \ Π_i)}
12         rem := rem \ Π_i
13         i := i + 1
```

Fig. 6.2. A greedy algorithm to compute a partitioning of a timed automata system

With Prop. 6.4, our resulting heuristic function is still admissible, *except* for the effects of synchronization. In a solution path to $\langle \mathcal{S}, \varphi \rangle$, a set of synchronized edges will be counted as a single transition, while in the partitioned system every edge will be counted separately. In our opinion, this potential non-admissibility is benign. For example, if only binary synchronization is allowed, then the real error state distance is over-estimated by at most a factor of 2, in the *worst case*, which one can realistically expect to be far away from the typical case. We ran a

number of tests using our heuristics with A^*, and never obtained a sub-optimal solution.

6.3 Evaluation

As already mentioned, we implemented all heuristics proposed in this chapter in the UPPAAL/DMC model checker [69]. The results that we are going to present in this section are obtained on an AMD Opteron system, running at 2.3 GHz. We set the memory limit to 4 GByte.

6.3.1 Experimental Setup

We evaluated our pattern database heuristics by comparing them to other heuristics search methods using A^* and greedy search. The heuristic $h^{\mathcal{P}}_{syn}$ uses the syntax-based method of Sec. 6.2.3 to obtain a set of predicates with splitting bound $b = 2$. We set b to 2 for the following reasons. With increasing splitting bound b, the preprocessing time for $h^{\mathcal{P}}_{syn}$ quickly becomes larger than the actual time spent for the search. The smallest number of explored states are obtained with $b = 4$, the shortest overall runtimes are obtained with $b = 1$. Therefore, we set $b = 2$, which is a good trade-off between runtime and the number of explored states. The $h^{\mathcal{P}}_{AR}$ heuristic uses the abstraction refinement approach for generating PDBs. For this heuristic, the preprocessing time constantly grows with increasing splitting bound b. The higher the number r of refinement steps is, the faster grows the preprocessing time with increasing b. With increasing splitting bound, the number of explored states decreases. This effect becomes stronger for higher values of r. Note that the decrease of the number of explored states never pays off in terms of runtime. The runtime heavily depends on the number of refinement steps r. The results for $h^{\mathcal{P}}_{AR}$ are obtained with splitting bound $b = 2$ and $r = 4$ refinement iterations. For both heuristics, $h^{\mathcal{P}}_{syn}$ and $h^{\mathcal{P}}_{AR}$, we report results that we obtained by combining multiple pattern databases. The selected parameters b and r for $h^{\mathcal{P}}_{syn}$ and $h^{\mathcal{P}}_{AR}$ are the most successful configurations, of the many possible configurations of our code, that we found in our experiments. For a detailed experimental evaluation of the parameters, the interested reader is referred to Chap. B. The h^{aa} results are obtained with a heuristic proposed by Dräger et al. [40]. A brief description of this heuristic can be found in Sec. 4.3.2. The results reported in the table are obtained with $N = 50$. This is the best value for N, i. e., when increasing N, the number of explored states does not decrease much, but the overhead for merging the automata increases a lot.

As benchmarks we took the Single-tracked Line Segment case study (the C examples). The M and N examples come from the Mutual Exclusion case study.

The F^A and F^B examples are two flawed versions of the Fischer protocol. The A examples model arbiter trees. A more detailed description of our benchmarks can be found in the appendix of this thesis.

6.3.2 Experimental Results

From a quick glance at Table 6.1 and Table 6.2, one sees that our new heuristics are indeed competitive with the other heuristics. The heuristics based on the monotonicity abstraction explore very few states in the M and N examples. The h^U heuristic yields the best results the C examples. Note that UPPAAL's rDFS cannot solve C_6–C_9 within the given memory bounds of 4 GByte. It is also worth mentioning that the h^L and h^U heuristics take more time to compute than the other heuristics. This is because the heuristics based on the monotonicity abstraction are the only ones of the tested heuristics that are *not* organized as a lookup table, i. e., the abstract problem has to be solved in every search state. Note that Table 6.1 was already presented in Sec. 5.4.3. Here it is just repeated to ease comparing the results.

The results obtained with A^* are shown in Table 6.3. Here it turns out that no configuration can solve the larger C and A examples. They are left out from the table. Except in the A examples, the best results are obtained with the $h^{\mathcal{P}}_{syn}$ and $h^{\mathcal{P}}_{AR}$ heuristics. They explore fewer states than the corresponding configurations with h^L and h^{aa}. The overall runtime of the two heuristics based on predicate abstraction is comparable with the runtime for h^{aa}.

6.4 Discussion

When we empirically determined the best parameters b and r for our heuristics, i. e., the splitting bound and the number of refinement iterations, we made the following curious observation. When using greedy search, the number of explored states often *increases* when we make the abstraction more refined. In particular, this is a surprise since every time we refine the abstraction we obtain a heuristic that dominates the previous one (see Sec. 6.2.1). At first glance, it seems impossible that a heuristic that dominates another one yields a larger number of explored states. But this only holds in the A^* context [82]. A closer look reveals that this is possible quite naturally for greedy search. Imagine a state s that has two successors s'_1 and s'_2. Suppose that s'_1 leads to an error state on a narrow path with only little branching of length 10. Further suppose that s'_2 is the start of a huge part of the state space containing no error states at all. Let h be an admissible heuristic function that is dominated by another admissible heuristic h'. Say $h(s'_1) = 5$, $h(s'_2) = 8$ and $h'(s'_1) = 9$, $h'(s'_2) = 8$. Although, h' is more precise than h, it will yield a much larger number of explored states.

Table 6.1. First part of the greedy search results. The rDFS results were obtained with UPPAAL's randomized depth-first search, the h^L and h^U results were obtained with MCTA's greedy search and the particular heuristic. Dashes indicate out of memory ($>$ 4 GByte).

	runtime in s			explored states			trace length		
Exp.	rDFS	h^L	h^U	rDFS	h^L	h^U	rDFS	h^L	h^U
C_1	0.1	0.0	0.0	24404	1989	373	880	89	62
C_2	0.3	0.1	0.1	64042	3559	663	770	123	74
C_3	0.4	0.1	0.0	86142	4242	928	618	119	68
C_4	4.5	0.6	0.5	921415	20081	10406	1569	116	103
C_5	46.1	5.1	3.0	8388325	141174	55676	3745	345	118
C_6	–	57.4	9.0	–	1253431	185293	–	641	130
C_7	–	419.8	54.8	–	9475793	878345	–	1175	200
C_8	–	102.5	106.8	–	2601885	1878328	–	556	385
C_9	–	179.8	516.2	–	4641449	7890458	–	995	280
M_1	0.4	0.0	0.1	39838	2431	5125	1246	294	71
M_2	1.3	0.1	0.2	127973	7436	13153	2809	640	93
M_3	0.9	0.2	0.2	97987	12924	12602	2716	591	100
M_4	4.1	0.2	0.6	408400	13497	30276	11940	750	150
N_1	1.7	0.1	0.2	55690	4834	7691	1100	329	80
N_2	5.9	131.3	0.8	188784	185770	23392	3350	40652	122
N_3	4.3	0.1	1.5	146601	7533	37381	3028	499	113
N_4	27.7	1.2	10.4	917774	46948	141354	14713	1661	230
F_5^A	0.0	0.0	0.0	419	9	9	111	8	8
F_{10}^A	0.1	0.0	0.0	10435	9	9	1406	8	8
F_{15}^A	0.7	0.0	0.0	43273	9	9	4641	8	8
F_5^B	0.0	0.0	0.0	293	167	7	78	12	6
F_{10}^B	0.0	2.3	0.0	7933	86462	7	1207	22	6
F_{15}^B	1.5	–	0.0	93632	–	7	9973	–	6
A_2	0.0	0.0	0.0	95	33	28	31	18	18
A_3	0.0	0.0	0.0	6030	202	76	105	24	18
A_4	0.2	9.7	0.0	46642	75106	39	752	36	28
A_5	–	65.7	1.7	–	257208	4027	–	74	47
A_6	–	–	–	–	–	–	–	–	–

This is due to the fact that h' is exclusively more precise for s'_1 and *not* for s'_2. Similar situations may also occur when we do a refinement step with ARMC. In fact, the defining characteristic of the refinement step is that it *excludes the detected spurious error path*, i. e., exactly one shortest spurious error path. All other spurious error paths of the same length may remain. The heuristic values in the environment of the removed spurious error path will increase. This is the region of the state space where the heuristic is refined. The heuristic values in the environment of the other spurious error paths will remain the same. Note

Table 6.2. Second part of the greedy results. All results were obtained with UPPAAL/DMC's greedy search and the particular heuristic. Dashes indicate out of memory (> 4 GByte).

Exp.	runtime in s			explored states			trace length		
	h^{aa}	$h^{\mathcal{P}}_{syn}$	$h^{\mathcal{P}}_{AR}$	h^{aa}	$h^{\mathcal{P}}_{syn}$	$h^{\mathcal{P}}_{AR}$	h^{aa}	$h^{\mathcal{P}}_{syn}$	$h^{\mathcal{P}}_{AR}$
C_1	0.1	1.2	3.3	2025	1588	1508	148	158	112
C_2	0.1	1.3	3.6	4740	3786	4098	197	180	128
C_3	0.2	1.4	3.8	6970	3846	5583	197	186	136
C_4	0.4	1.9	4.5	31628	30741	41831	172	240	279
C_5	1.9	3.5	7.2	260088	185730	425264	267	422	506
C_6	17.2	16.6	22.7	2948925	1942277	2850769	376	756	770
C_7	224.5	158.9	174.2	29948574	18455933	20632762	854	1063	2216
C_8	216.6	120.2	204.0	28413627	14793386	25598687	706	975	2878
C_9	–	–	–	–	–	–	–	–	–
M_1	0.2	0.7	1.4	21260	23257	36808	89	100	115
M_2	0.8	1.5	2.4	78117	84475	145178	101	127	136
M_3	0.9	1.5	2.6	85301	92548	143275	104	97	134
M_4	3.0	3.8	6.3	287122	311049	552341	123	135	191
N_1	1.2	1.9	3.6	30970	31593	51434	109	127	82
N_2	5.3	6.6	9.2	149013	172531	246878	126	157	136
N_3	5.4	6.2	11.5	158585	167350	296450	107	129	167
N_4	28.4	33.6	42.9	785921	975816	1411536	146	212	270
F^A_5	0.0	0.0	0.0	80	80	29	20	20	8
F^A_{10}	0.0	0.1	0.0	130	130	44	20	20	8
F^A_{15}	0.1	0.3	0.0	180	180	59	20	20	8
F^B_5	0.0	0.1	0.0	21	21	23	6	6	6
F^B_{10}	0.0	0.3	0.0	36	36	38	6	6	6
F^B_{15}	0.1	0.4	0.0	51	51	53	6	6	6
A_2	0.0	0.0	0.1	31	46	46	12	12	12
A_3	0.1	0.0	0.1	863	187	187	20	21	21
A_4	0.4	0.1	0.1	2283	10633	10633	33	78	78
A_5	14.1	–	–	1220430	–	–	212	–	–
A_6	–	–	–	–	–	–	–	–	–

that, due to our splitting algorithm (see Sec.6.2), ARMC only gets a part of the whole system. Therefore it is possible that ARMC finds an abstract error trace that is also a real one, but only for the specific part of the system not for the whole system. If the removed spurious error path happens to be the (only) one that actually corresponds to a real solution, then the refinement will increase our search space in pretty much the way as illustrated with s_1 and s'_2 in the example above.

Our intuition was confirmed quite clearly when we ran the following test on example C_4 with splitting bound $b = 3$, i. e., for C_4, splitting bound $b = 3$ means

Table 6.3. Summary of A^* results. The h^L results were obtained with MCTA, all other results were obtained with UPPAAL/DMC. Dashes indicate out of memory (> 4 GByte).

Exp.	runtime in s h^L	h^{aa}	$h^{\mathcal{P}}_{syn}$	$h^{\mathcal{P}}_{AR}$	explored states h^L	h^{aa}	$h^{\mathcal{P}}_{syn}$	$h^{\mathcal{P}}_{AR}$	trace length
C_1	0.1	0.1	1.3	3.4	10470	9784	7088	9402	54
C_2	0.4	0.3	1.3	3.7	24658	34644	15742	25931	54
C_3	0.5	0.3	1.5	3.9	28694	40078	15586	30559	54
C_4	3.8	2.1	2.6	5.6	184413	324080	108603	233052	55
C_5	29.1	15.8	8.3	15.1	1140376	2449667	733761	1708529	56
C_6	270.1	211.8	66.0	134.0	9250356	24332209	7360078	16868109	56
M_1	0.2	0.2	0.8	1.4	14128	19422	22634	26909	47
M_2	0.9	0.9	1.7	2.4	47543	77523	94602	112739	50
M_3	1.1	1.1	2.1	2.9	54156	94882	121559	148474	50
M_4	4.9	5.5	6.5	8.0	180353	436953	466967	569469	53
N_1	2.8	2.4	3.2	4.2	40482	46920	46966	56985	49
N_2	16.4	11.5	10.9	11.2	177131	211132	211935	219579	52
N_3	19.8	12.4	11.8	17.0	196083	238161	233609	318317	52
N_4	119.9	61.8	50.7	63.7	830062	1111400	1036002	1258229	55
F_5^A	0.0	0.0	0.0	0.0	71	1457	1457	278	8
F_{10}^A	0.0	0.5	0.5	0.1	511	37922	37922	1448	8
F_{15}^A	0.1	6.0	6.2	0.1	1701	348797	348797	4218	8
F_5^B	0.0	0.0	0.1	0.0	54	21	21	23	6
F_{10}^B	0.0	0.0	0.1	0.0	429	36	36	38	6
F_{15}^B	0.1	0.1	0.4	0.0	1504	51	51	53	6
A_2	0.0	0.0	0.0	0.1	62	120	155	107	12
A_3	0.1	0.1	0.1	0.1	2006	5763	20658	14574	17
A_4	100.4	102.4	–	–	813303	13163834	–	–	22

that we have two pattern databases. We incrementally increased the number of refinement iterations and measured the number of explored states as well as the length of the shortest abstract error path found in both abstractions at the maximum level. The data we obtained are given in Table 6.4.

When increasing r from 2 to 3, we first notice the hypothesized effect. In difference to before, the length of $trace_1$, found in the first abstraction did not increase and promptly the number of explored states went slightly up. At this level, the abstract error trace of the second abstraction became a real error trace for the given part of the system. This is the reason why it stays fixed from now on. With $r = 10$, the length of $trace_1$ finally increases again and promptly the number of explored states goes sharply down. In this refinement step, $trace_1$ became also a real error trace, so here stops our experiment.

One can try to overcome this phenomenon by making the refinement mechanism less focused on a single error path, trying to exclude the spurious error

Table 6.4. Results for the C_4 example, obtained with greedy search using $h^{\mathcal{P}}_{AR}$ and splitting bound $b = 3$, i. e., two pattern databases. Abbreviations: r: number of refinements, *states*: number of explored states, *trace*$_i$: length of shortest abstract error trace in ith abstraction.

r	*states*	*trace*$_1$	*trace*$_2$
0	516282	6	5
1	511180	9	7
2	49384	13	10
3	56081	13	12
4	46837	13	12
5	63606	13	12
6	279374	13	12
7	279374	13	12
8	279374	13	12
9	361555	13	12
10	50060	16	12

paths more *broadly*. The straightforward idea is to introduce predicates so that *all* shortest spurious error paths are removed and not just a single one. This may require too much computational overhead. It remains to be seen if one can define successful selection heuristics and greedy strategies that remove the most *relevant* error paths, or that introduce only the most *relevant* predicates. An idea for the latter may be to define relevance of a predicate based on how many error paths it serves to remove. Another idea might be to use a sort of perimeter search within the abstraction, where the error condition would be *broadened* to the final layer of a depth-bounded backwards breadth-first search. Alternatively, of course, one can use our above observations simply to design an automatic selection of the number of refinement steps: refine until, in an iteration k, the length of the shortest spurious error path does, for the first time, not increase; take the heuristic function defined by the abstract state space from iteration $k-1$.

6.5 Conclusion

There clearly is promise in defining heuristic functions for model checking based on predicate abstraction. Apart from the idea, we have contributed a method to efficiently store and query the heuristic information, a method to split a system of timed automata without losing admissibility and a first empirical exploration. Our empirical results are not yet at a level that would be thoroughly satisfying, but we are competitive with the other techniques that have been proposed so far. The main surprise was the good performance of syntax-based predicate abstractions. We did not expect that one can do so well with

such a simple form of abstraction predicates. In particular, we expected abstraction refinement to yield much better heuristics. The reason why this is not (yet) the case appears to lie in the following oddity. One would expect that a more refined heuristic yields a smaller search space. However, in disquietingly many cases, refining the abstraction yielded a *larger* search space in our experiments. We believe there is hope to overcome this with modified refinement strategies.

7

Fast Directed Model Checking via Russian Doll Abstraction

In the previous chapters we have presented several powerful heuristics for directed model checking, but many problems still prove to be notoriously hard. For instance, so far we are only able to detect some unnecessarily long error paths in the harder instances of our benchmark set. With the heuristic we are going to introduce in this chapter, we can find provably *shortest* error paths for these benchmarks in a matter of seconds. The heuristic is based on a kind of *Russian Doll* principle, where the heuristic for a given problem arises through using UPPAAL/DMC for the complete exploration of a simplified problem. The simplification consists in removing parts from the problem that are distant from the error property. As our empirical results confirm, this simplification often preserves the characteristic structure leading to the error.

7.1 Russian Doll Abstraction

The proposed heuristics from this chapter belong, like the heuristics from the previous chapter, to the family of pattern database heuristics (PDB). PDB heuristics were first proposed in AI [29] for solving hard search problems in single agent games such as the famous Rubik's Cube. A PDB heuristic is obtained by abstracting the problem at hand by ignoring some of the relevant symbols, e. g., some of the variables. Prior to search, the usually much smaller state space of the abstracted problem is built completely. During search, the abstract state space is used as a lookup table to obtain the heuristic values. When dealing with PDBs, the main question is, which symbols should be ignored, i. e., how should we abstract the problem to obtain a PDB? In AI, most strategies are aimed at exploiting parts of the problem that are largely independent. For instance, the strategies proposed by Culberson and Schaeffer [29] and Haslum et al. [55] are of that type. The idea behind them is to generate a separate PDB for several parts of the system and accumulate the heuristic values. Indeed, also the heuris-

tics from Chap. 6 and Edelkamp et al.'s heuristic based on the graph distance [42, 43] can be seen as an instance of this type.

7.1.1 The General Idea

The main idea of our abstraction selection strategy is actually an extension and an improvement of Qian et al.'s work [87]. Our selection strategy is based on what we call a *Russian Doll* principle: Rather than trying to split the entire system into, more or less, independent parts, one homes in on the part of the system that is most relevant to the target formula and leaves that part *entirely* intact. We chose the name Russian Doll based on the intuition that the part left intact resembles the child Russian Doll, which is smaller but still characteristically similar to the parent. Intuitively, this is more suitable for model checking than traditional AI techniques, because a particular combined behavior of the automata nearest to the target formula is often essential in how the error arises. The child Russian Doll preserves such combined behaviors and should hence provide useful search guidance. The excellent results we obtained in our benchmarks indicate that this is indeed the case, even with rather small child dolls, i. e., with rather small abstractions.

Given the key idea of the Russian Doll strategy, i. e., keep all and only symbols that are of *immediate relevance* to the target formula to be checked, the question remains what is *relevant*. Answering this question precisely involves solving the problem in the first place. However, one can design computationally easy strategies that are intuitively very adequate for model checking. The basic idea is to do some form of abstract cone of influence (COI) computation [27] and ignore those symbols that do not appear in the COI. Qian et al. [87] use a simple syntactic backward chaining process that iteratively collects variable names and requires the user to specify a threshold on the maximal considered distance, i. e., the number of iterations of the kept variables from the target formula. In our work, we use a more sophisticated procedure based on the monotonicity abstraction (see Chap. 5). Our procedure selects a subset of the relevant symbols (automata, synchronization labels, clock variables and integer variables) based on an abstract error path. No user input is required. Once it is decided which parts to keep, our implementation outputs those parts in the UPPAAL input language. In Russian Doll style, UPPAAL/DMC itself is then used to compute the entire state space of the abstracted problem which is stored in a lookup table. During search, heuristic values are just read off that table.

7.1.2 Obtaining Abstractions

This section presents the technicalities of generating the simplified problem in UPPAAL's input language and using UPPAAL/DMC to compute the heuristic

function. We start this section by introducing the notion of *variable abstractions*. Given a system $\mathcal{S}$ and an abstraction set $\mathcal{A}$, i. e., a subset of the system's variables, a variable abstraction is an abstraction of $\mathcal{S}$ where the variables from $\mathcal{A}$ are ignored. This is stated formally in the next definition.

Definition 7.1 (Variable Abstraction). *Let $\mathcal{S} = A_1 \parallel \ldots \parallel A_n$ be a system of timed automata and let $\mathcal{A} \subseteq \{A_1, \ldots, A_n\} \cup X \cup V \cup \Sigma$ be an abstraction set, where X denotes the set of clocks of the system, V is the set of integer variables of $\mathcal{S}$ and Σ is the set of synchronization labels. We define the abstraction α with respect to $\mathcal{A}$ as follows.*

$$\alpha(\text{true}) = \text{true}$$
$$\alpha(\text{false}) = \text{false}$$

If φ and ψ are clock constraints, location constraints and/or integer constraints, then

$$\alpha(\varphi) = \begin{cases} \text{true} & \text{var}(\varphi) \cap \mathcal{A} \neq \emptyset \\ \varphi & \text{otherwise} \end{cases}$$
$$\alpha(\varphi \wedge \psi) = \alpha(\varphi) \wedge \alpha(\psi)$$

If a is a synchronization label, then

$$\alpha(a) = \begin{cases} \tau & a \in \mathcal{A} \\ a & \text{otherwise} \end{cases}$$

If f is a set of integer assignments and/or clock resets, then

$$\alpha(f) = \{e \mid e \in f \wedge \text{var}(e) \cap \mathcal{A} = \emptyset\}$$

If $e = l \xrightarrow[f]{g,a} l'$ is an edge, where g is the guard of the edge, a is a synchronization label, f is a set of integer assignments and/or clock resets, then the abstract edge $\alpha(e)$ is defined as

$$\alpha(l \xrightarrow[f]{g,a} l') = l \xrightarrow[\alpha(f)]{\alpha(g),\alpha(a)} l'$$

If $A = \langle L, l^0, E, \Sigma, X, V, I\rangle$ is a timed automaton, then the abstract automaton is defined as

$$\alpha(A) = \langle L, l^0, \{\alpha(e) \mid e \in E\}, \Sigma \setminus \mathcal{A}, X \setminus \mathcal{A}, V \setminus \mathcal{A}, \alpha(I)\rangle,$$

where $\alpha(I)(l) = \alpha(I(l))$, for all locations l. Let $\prod$ denote the parallel composition. For a system $\mathcal{S}$ of timed automata, the abstract system is defined as

$$\alpha(\mathcal{S}) = \prod_{A \in \mathcal{S} \setminus \mathcal{A}} \alpha(A).$$

For states $s \in \mathcal{T}(\mathcal{S})$, where $\mathcal{T}(\mathcal{S})$ is the state space of $\mathcal{S}$, the abstraction of s is defined as $s|_{\overline{\mathcal{A}}}$, i. e., the projection of s on the variables not contained in $\mathcal{A}$.

In words, given an abstraction set $\mathcal{A}$, the corresponding variable abstraction α applied to a system of timed automata $\mathcal{S}$ simply ignores any automaton that appears in $\mathcal{A}$, as well as any constraints or effects that involve variables or synchronization labels from $\mathcal{A}$. Note that $\alpha(\mathcal{S})$ is still a system of timed automata. We call this kind of abstractions variable abstractions, since they are based on ignoring a subset of the system's variables. With the notion of α, we can now define what we call an abstract reachability problem. Let $\langle \mathcal{S}, \varphi \rangle$ be a reachability problem, defined as usual, then the corresponding abstract reachability problem $\alpha(\langle \mathcal{S}, \varphi \rangle)$ is defined as $\langle \alpha(\mathcal{S}), \alpha(\varphi) \rangle$.

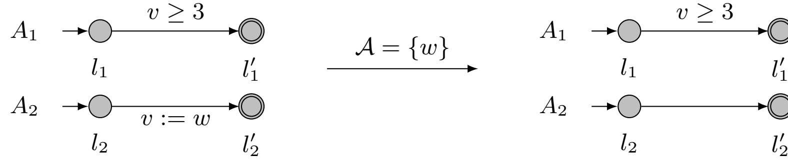


Fig. 7.1. Not every abstraction set yields an overapproximation.

It is important to note that the abstraction does not always induce an overapproximation of the original system. For instance, consider the system $\mathcal{S}$ of two timed automata A_1 and A_2, that share the integer variables v and w, on the left of Fig. 7.1. If the system's initial state is given by $\langle A_1 = l_1, A_2 = l_2, v = 0, w = 7 \rangle$ then it is possible to reach a state where both automata are in their double circled locations. If we abstract the system with respect to the abstraction set $\mathcal{A} = \{w\}$, then we get the system depicted on the right of Fig. 7.1. In the abstract system, such a state is no longer reachable. The following is a sufficient condition on $\mathcal{A}$ to ensure that $\mathcal{A}$ induces an overapproximation.

Definition 7.2 (Closed Abstraction Set). *Let $\mathcal{S} = A_1 \parallel \ldots \parallel A_n$ be a system of timed automata and let $\mathcal{A}$ be an abstraction set. In the following, let $A_i = \langle L_i, l_i^0, E_i, \Sigma_i, X_i, V_i, I_i \rangle$ denote an automaton of $\mathcal{S}$. We define $\mathcal{A}$ is* closed *iff all of the following conditions hold:*

1. For each automaton $A_i \in \mathcal{S} \cap \mathcal{A}$:
   a) $\Sigma_i \subseteq \mathcal{A}$ and
   b) if there is an edge $e \in E_i$ such that $x := 0$ is a clock reset of e, then $x \in \mathcal{A}$ and

c) if there is an edge $e \in E_i$ such that $v := \mathrm{exp}$ is an effect of e, then $v \in \mathcal{A}$.

2. *For each automaton $A_i \in \mathcal{S} \setminus \mathcal{A}$:*
   if there is an edge $e \in E_i$ such that $v := \mathrm{exp}$ is an effect of e, where $\mathrm{var}(\mathrm{exp}) \cap \mathcal{A} \neq \emptyset$, then $v \in \mathcal{A}$.

It is easy to verify that a value of a variable, i. e., a clock or an integer variable, never depends on variables that are in the abstraction set. We will see in the next section that closed abstraction sets yield admissible heuristic functions. It is obvious that any abstraction set can be closed by extending it according to Definition 7.2.

An abstract system that is induced by a closed abstraction set simulates the original system, i. e., there is a simulation relation [78] between the original system and the abstract system.

Proposition 7.3. *Let $\mathcal{S}$ be a system of timed automata and let $\mathcal{A}$ be a closed abstraction set. The relation $H \subseteq \mathcal{T}(\mathcal{S}) \times \mathcal{T}(\alpha(\mathcal{S}))$ with $H(s, \hat{s})$ iff $\hat{s} = \alpha(s)$, where $\mathcal{T}(\mathcal{S})$ is the state space of $\mathcal{S}$ and $\mathcal{T}(\alpha(\mathcal{S}))$ is the state space of $\alpha(\mathcal{S})$, is a simulation relation.*

Intuitively this means that the abstract system can behave like the original system, i. e., if the original system can take a transition, then the abstract system can also take a transition leading to an abstract state that corresponds to the original successor state in some sense.

Proof (Proposition 7.3). According to the definition of simulation relation, we have to prove that for all $s \in \mathcal{T}(\mathcal{S})$ and all $\hat{s} \in \mathcal{T}(\alpha(\mathcal{S}))$, if $H(s, \hat{s})$, then the following conditions hold:

1. $s|_{\overline{\mathcal{A}}} = \hat{s}$ and
2. for every transition $s \rightarrow s'$ of $\mathcal{T}(\mathcal{S})$ there is a transition $\hat{s} \rightarrow \hat{s}'$ of $\mathcal{T}(\alpha(\mathcal{S}))$ with $H(s', \hat{s}')$.

Proving the first point is trivial, since this is exactly the definition of α. For proving the second point, we have to distinguish between discrete transitions and timed transitions. Let $t = s \rightarrow s'$ be a discrete transition induced by some τ edge of an automaton $A \in \mathcal{A}$. As A is not present in the abstraction, the corresponding abstract transition $\hat{t}$ is a delay transition with duration time $\delta = 0$. From Definition 7.2 it follows that t only affects automata and variables from $\mathcal{A}$, hence $\hat{t} = \alpha(s) \rightarrow \alpha(s')$. The same also holds for synchronized edges. For every successor state s' of s that is reached via a timed transition t with duration $\delta \in \mathbb{R}_{\geq 0}$, it holds that

$$s' \models \underbrace{I(s') \wedge \bigwedge_{A \in \mathcal{S}} (A = s(A)) \wedge \bigwedge_{v \in V} (v = s(v)) \wedge \bigwedge_{x \in X} (x = s(x) + \delta)}_{=\psi} .$$

For the corresponding abstract time successor $\hat{s}'$ of $\hat{s}$ reached via a delay transition $\hat{t}$ with the same duration, the following holds.

$$\hat{s}' \models \underbrace{\alpha(I)(\hat{s}') \wedge \bigwedge_{A \in \mathcal{S} \setminus \mathcal{A}} (A = \hat{s}(A)) \wedge \bigwedge_{v \in V \setminus \mathcal{A}} (v = \hat{s}(v)) \wedge \bigwedge_{x \in X \setminus \mathcal{A}} (x = \hat{s}(x) + \delta)}_{=\psi'}$$

What we have to show is that $\hat{s}' \models \alpha(\psi)$. This can be proven by just applying the definition of α several times to ψ', bearing in mind that the value of a variable $v \notin \mathcal{A}$ only depends on variables that are also not in $\mathcal{A}$. The proof for discrete transitions that are induced by an edge of an automaton $A \notin \mathcal{A}$ is analogous to the proof for timed transitions. □

7.1.3 Pattern Databases

As already mentioned, in our approach, a pattern database for a reachability problem $\langle \mathcal{S}, \varphi \rangle$ is obtained as the result of a complete state space exploration using UPPAAL/DMC. Recall again that UPPAAL/DMC's search space coincides with the zone automaton $\mathcal{Z}(\mathcal{S})$ of the system under consideration $\mathcal{S}$. This means that each state $s \in \mathcal{Z}(\mathcal{S})$ that UPPAAL/DMC considers corresponds to a set of system states where all automata locations and integer variables are fixed but the clock valuation can be any of a particular zone. We will use $[s] \subseteq \mathcal{T}(\mathcal{S})$ to denote the set of system states corresponding to $s \in \mathcal{Z}(\mathcal{S})$.

Note that all the presented techniques from this chapter can be easily applied to discrete state spaces, in a manner that should become obvious in the following. What we do in order to obtain a pattern database heuristic for the reachability problem $\langle \mathcal{S}, \varphi \rangle$ is the following. First, an abstraction set $\mathcal{A}$ is chosen with the techniques detailed below in Sec. 7.2. Given $\mathcal{A}$, we use UPPAAL/DMC to solve $\langle \alpha(\mathcal{S}), \mathit{false} \rangle$, i. e., UPPAAL/DMC is used to generate the entire reachable zone automaton $\mathcal{Z}(\alpha(\mathcal{S}))$. During the traversal thereof, the abstract state space is written on the fly into a file, in a simple format. Once UPPAAL/DMC has stopped, an external program is used to find all error states in the file and to compute $d^{\alpha(\mathcal{S},\varphi)}(s')$ for all states $s' \in \mathcal{Z}(\alpha(\mathcal{S}))$, where $d^{\alpha(\mathcal{S},\varphi)}(s')$ denotes the distance from s' to a state that satisfies $\alpha(\varphi)$. The program is a version of Dijkstra's algorithm with multiple sources [36]. In other words, UPPAAL/DMC

computes the zone automaton of the abstracted problem and an external program finds the distances to the abstracted error states.

It remains to specify how the zone automaton of the abstract problem and $d^{\alpha(\mathcal{S},\varphi)}$ are used to implement a heuristic function for solving $\langle \mathcal{S}, \varphi \rangle$. The core operation is to map a state from $\mathcal{Z}(\mathcal{S})$ to a set of corresponding states from $\mathcal{Z}(\alpha(\mathcal{S}))$. For a state $s \in \mathcal{Z}(\mathcal{S})$, by $s|_{\overline{\mathcal{A}}}$ we denote the projection of the system states in s onto the variables not contained in $\mathcal{A}$.

Definition 7.4. *Let $\langle \mathcal{S}, \varphi \rangle$ be a reachability problem, where $\mathcal{S}$ is a system of timed automata and φ is a target formula. Further, let $\mathcal{A}$ be an abstraction set. The abstraction of $s \in \mathcal{Z}(\mathcal{S})$ under $\mathcal{A}$ is defined as*

$$\alpha(s) = \{s' \in \mathcal{Z}(\alpha(\mathcal{S})) \mid s' \cap s|_{\overline{\mathcal{A}}} \neq \emptyset\}.$$

The heuristic value of $s \in \mathcal{Z}(\mathcal{S})$ under $\mathcal{A}$ is defined as

$$h^{\mathcal{A}}(s) = \min_{s' \in \alpha(s)} \{d^{\alpha(\mathcal{S},\varphi)}(s')\}.$$

Note that $s' \cap s|_{\overline{\mathcal{A}}} \neq \emptyset$ may be the case for more than one $s' \in \mathcal{Z}(\alpha(\mathcal{S}))$, i. e., we have $s' \cap s|_{\overline{\mathcal{A}}} \neq \emptyset$ iff s' and s agree completely on the automata locations of $A \setminus \mathcal{A}$ and on the values of $V \setminus \mathcal{A}$, and the zone of s' is consistent with the zone of s.[1] Testing consistency of two zones is a standard operation for which UPPAAL/DMC provides a highly efficient implementation. Consequently, in our implementation, we store $\mathcal{Z}(\alpha(\mathcal{S}))$ in a hash table indexed on the automata and integer variables of $\alpha(\mathcal{S})$ not contained in $\mathcal{A}$, where each table entry contains a list of zones, one for each corresponding abstract state s'. Of course, $d^{\alpha(\mathcal{S},\varphi)}(s')$ is also stored in each list entry. Obtaining heuristic values is then realized via a hash table lookup plus zone consistency checks in the list, selecting the smallest $d^{\alpha(\mathcal{S},\varphi)}(s')$ of those s' for which the check succeeded.

Lemma 7.5. *Let $\langle \mathcal{S}, \varphi \rangle$ be a reachability problem, where $\mathcal{S}$ is a system of timed automata and φ is a target formula. Further, let $\mathcal{A}$ be a closed abstraction set for $\mathcal{S}$ and $s \in \mathcal{Z}(\mathcal{S})$ be a state. Let $d^{\mathcal{S},\varphi}(s)$ denote the distance of s to the nearest state $s_e \in \mathcal{Z}(\mathcal{S})$ satisfying φ, then the following holds.*

$$h^{\mathcal{A}}(s) \leq d^{\mathcal{S},\varphi}(s)$$

Proof (Lemma 7.5). As $\mathcal{Z}(\mathcal{S})$ and $\mathcal{T}(\mathcal{S})$ are bisimulation equivalent, it suffices to prove the lemma for $\hat{s} \in [s] \subseteq \mathcal{T}(\mathcal{S})$. Since there is a simulation relation between $\alpha(\mathcal{S})$ and $\mathcal{S}$ (see Prop. 7.3) it follows that every trace $\pi = t_1, \ldots, t_n$ in $\mathcal{T}(\mathcal{S})$ is also trace in $\mathcal{T}(\alpha(\mathcal{S}))$. It remains to show that $\hat{s} \models \varphi$ implies $\alpha(\hat{s}) \models \alpha(\varphi)$. This is proven via structural induction over the definitions of $\models$ and α. □

[1] In a discrete state space, s' and s agree completely on all non-abstracted variables.

Note that, Lemma 7.5 does *not* hold if $\mathcal{A}$ is not closed. For instance, this can be seen in the example from Fig. 7.1. The figure illustrates that, a symbol that is abstracted away can contribute to changing the status of a symbol that is not abstracted away. The importance of Lemma 7.5 is that, plugging our heuristic function into A^*, we can guarantee to find a shortest possible, i. e., an optimal, error path.

7.2 Choosing Abstraction Sets

Having specified how to proceed once an abstraction set $\mathcal{A}$ is chosen, it remains to clarify how that choice is made. The traditional AI design principle for PDBs is to look for different parts of the problem that are largely independent and to construct a separate PDB for each of them, accumulating the heuristic values. For instance Korf and Felner [66] and Haslum et al. [55] demonstrated that this principle is powerful. Now, consider this design principle in model checking. An error typically arises due to some complex interaction between several automata. If one tears those automata apart, the information about this interaction is lost. A different approach, first mentioned by Qian et al. [87], is to keep only one PDB that includes as much as possible of those parts of the system that are of immediate relevance to the target formula. The intuition is that the particular combined behavior responsible for the error should be preserved. To realize this idea, one needs a definition of what is *close* to the target formula, and what is *distant*. The notion of cone of influence [27] computation lends itself naturally to obtain such a definition. Qian et al. [87] use a simple method based on syntactic backward chaining over variable names. In this section, we first present Qian et al.'s approach, afterwards we introduce our more sophisticated method based on the monotonicity abstraction. As we shall see, this method leads to much better empirical behavior on our benchmark set.

7.2.1 A Cone-of-influence-based Method

Qian et al.'s [87] method adapted to timed automata starts with the symbols, i. e., the automata, the integer variables and the clock variables, occurring in the target formula. This set of symbols forms layer 0. Iteratively, new layers are added, where layer $n + 1$ arises from layer n by including any symbol y that does not occur in a layer $n' \leq n$, and that may be involved in modifying the status of a symbol x in layer n, e. g., x and y may be variables and there may exist an assignment $x := \mathit{exp}$ where $y \in \mathrm{var}(\mathit{exp})$. The abstraction set is then chosen based on a user-supplied cut-off value d. The resulting abstraction set $\mathcal{A}$ will contain exactly all the symbols in layers $n > d$.

Let us give an example for this method. Say $\langle \mathcal{S}, A_2 = l_2 \rangle$ is a reachability problem, where $\mathcal{S}$ consists of the three automata A_1, A_2 and A_3, depicted in Fig. 7.2. The automata share two integer variables u and a. Let the initial state of the system be $\langle A_1 = l_1, A_2 = l_1, A_3 = l_1, u = 0, a = 0 \rangle$. Layer 0 of Qian et al.'s method only contains the automaton A_2, mentioned in the target formula. The next layer also contains u, since this variable influences A_2, i. e., it appears in the guard of A_2's edge. Layer 2 additionally contains A_1 and a, since u can be influenced by A_1 via a. All subsequent layers contains all variables of the system. Say we fixed the cut-off value d to 2, then the abstraction set $\mathcal{A}$ would only contain A_3. Note that $\mathcal{A}$ is not closed, the corresponding closed abstraction set is $\mathcal{A} = \{A_3, a, u\}$.

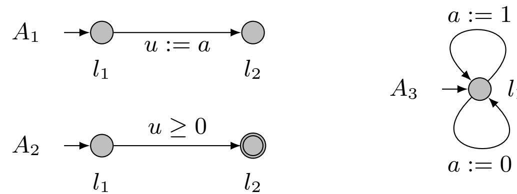


Fig. 7.2. A system with three automata

Intuitively, the problem with this syntactic backward chaining is that it is not discriminative enough between transitions that are actually relevant for violating the error property, and transitions that are not. In our experiments, we observed that, typically, the layers n converge to the entire set of symbols very quickly. For instance for our largest benchmark example, this is already the case for $n = 5$. When cutting off very early, e. g. at $n = 2$, one misses some symbols that are important, and at the same time one includes many symbols that are not important.

7.2.2 A Method Based on the Monotonicity Abstraction

Our key idea for improving on the difficulties of Qian et al.'s method is to do a more informed relevance analysis. For obtaining an abstraction set for the reachability problem $\langle \mathcal{S}, \varphi \rangle$, we first abstract the problem according to the monotonicity abstraction (see Chap. 5) and compute an abstract error path. Afterwards we set $\mathcal{A}$ to those symbols that are not affected by any of the transitions contained in the abstract error path. This way, we get a fairly targeted notion of what is relevant for reaching an error and what is not. Note that this approach does not require any parameters and hence as a side effect we also get rid of the need to request an input parameter from the user. This means that our method for choosing abstraction sets is *fully* automatic.

As already mentioned, the monotonicity abstraction is detailed in Chap. 5. Here we will only briefly revisit the method by constructing an abstraction set for the system depicted in Fig. 7.2. Recall that the main idea of the monotonicity abstraction is based on the simplifying assumption that state variables accumulate, rather than change, their values. The value of a variable in a state is now a subset, rather than an element, of the variable's domain. If a variable is assigned a new value, then this value is included into the variable's value set, without removing any old values. Hence the value range of each state variable grows monotonically over transitions.

Figure 7.3 gives our method for choosing the abstraction set $\mathcal{A}$. The algorithm takes as input a reachability problem and returns a closed abstraction set for it. The algorithm first obtains an abstract solution π, by first building an ATG and extracting π afterwards.

1 **function** select-abstraction-set($\mathcal{S}$, φ):
2 build-atg($\mathcal{S}$, φ, s_0)
3 π = extract-solution-modified($\mathcal{S}$, φ, s_0)
4 $\mathcal{A}_0 :=$ { all symbols from $\mathcal{S}$ not affected by π}
5 **return** close($\mathcal{A}$)

Fig. 7.3. Selecting an abstraction set based on the monotonicity abstraction

The method consists of two parts, a forward chaining and a backward chaining step. The forward chaining step, build-atg (cf. Fig. 5.3), simulates the simultaneous execution of all transitions in parallel, starting from the start state. In a layer-wise fashion, this computes for every state variable what the subset of reachable values is. The forward step stops when it reaches a layer where the target formula is satisfied. The backward step, extract-solution-modified, then starts at this layer, selecting transitions that can be responsible for achieving variable values that satisfy the target formula. The guards of these transitions yield new state variable values that must be achieved at an earlier layer. The process is iterated, selecting new transitions to support the new values and so on. The outcome of the process is a sequence $\pi = t_1, \ldots, t_n$ of transitions that leads from the start state to a state satisfying the target formula, when executed under the monotonicity abstraction. We then collect all symbols not affected by π, i. e., we set $\mathcal{A}_0$ to

$$\begin{aligned}\mathcal{A}_0 := \{A_i \in \mathcal{S} \;&|\; E_i \cap \pi = \emptyset\} \\ \cup\ \{a \in \Sigma \;&|\; \text{not ex. } e \in \pi \text{ s.t. } a_e = a\} \\ \cup\ \{v \in X \cup V \;&|\; \text{not ex. } e \in \pi \text{ s.t. } v \in \text{lhs}(f_e) \text{ and} \\ &\quad \text{ex. } A_i \in \mathcal{S} \setminus \mathcal{A}_0 \text{ and ex. } e \in E_i \text{ s.t. } v \in \text{var}(g_e)\}.\end{aligned}$$

In this notation, $e \in \pi$ is a shorthand for asking whether any of the transitions t from π involves the edge $e = l \xrightarrow[f_e, r_e]{g_e, a_e} l'$. The expression $E \cap \pi$ is a shorthand for $E \cap \{e \mid e \in \pi\}$. In words, we keep all automata, actions, clock variables and integer variables that are modified on the path, and we keep all clock and integer variables that are relevant to a guard in an automaton that we keep. We obtain our final abstraction set $\mathcal{A}$ by closing $\mathcal{A}_0$ according to Definition 7.2.

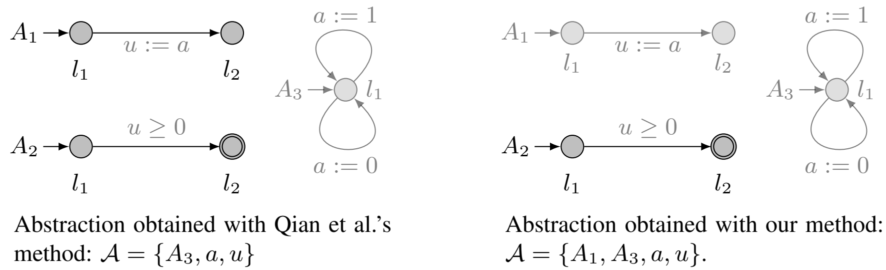
1 **function** extract-solution-modified($\mathcal{S}$, φ, s):
2   **for** $k := 0, \ldots,$ *minlayer* **do**:
3   **for each** A_i **do**:
4             $TL_k(A_i) := \emptyset$
5   **for each** v **do**:
6             $TV_k(v) := \emptyset$
7   make-target(*minlayer*, φ)
8   **for** $k :=$ *minlayer*, $\ldots, 1$ **do**:
9   **for each** A_i **do**:
10   **for each** $l' \in TL_k(A_i)$ **do**:
11   select t enabled at $k-1$ that ends in l'
12   make-target($k-1$, t's start locations, t's guard formulas)
13   **for each** v **do**:
14   **for each** $c \in TV_k(v)$ **do**:
15   select t enabled at $k-1$ with effect $v := c_0 + \sum_{i=0}^{n} c_i v_i$ such that $c \in \{c_0 + \sum_{i=0}^{n} c_i e_{v_i} \mid e_{v_i} \in V_{k-1}(v_i)\}$
                for each v_i that occurs in the effect **do**:
                    select $e_{v_i} \in V_{k-1}(v_i)$ such that $c = c_0 + \sum_{i=0}^{n} c_i e_{v_i}$
                    $TV_{k-1}(v_i) := TV_{k-1}(v_i) \cup \{e_{v_i}\}$
16   make-target($k-1$, t's start locations, t's guard formulas)
17   **return** selected transitions

Fig. 7.4. The modified algorithm for constructing abstraction sets

Let us remark two things. First, recall that, when dealing with linear arithmetic, the complexity of constructing an abstract solution is exponential. Hence, the complexity of our abstraction selection algorithm is also exponential. It is only exponential in the maximum number of variables of any linear expression over integer variables that occur on the description of the system. Without linear arithmetic, the complexity is polynomial (see Chap. 5 for details). Second, we use a slightly modified version of the extract-solution algorithm described in Sec. 5.3.3. The modified algorithm is depicted in Fig. 7.4. The modified algorithm is obtained by removing the grayed lines from the original one. The modified algorithm does not consider indirect variable dependencies. We found this method to yield better performance, by selecting more relevant variable subsets.

7.2.3 Comparison of the Methods

Reconsider the system from Fig. 7.2. If we apply our monotonicity abstraction based selection algorithm to this system, then the abstract solution only consists of the transition induced by the edge of automaton A_2. Therefore, our abstraction set is $\{A_1, A_3, a, u\}$. Figure 7.5 illustrates the different effects of our method and the one of Qian et al. While our abstract system only consists of one automaton, their abstract system consists of two, which unnecessarily blows up the computation of the PDB. Also note that, on the one hand, the abstraction set obtained with our approach depends on the given system, its initial state and the given target formula. On the other hand, Qian et al.'s method does *not* depend on the system's initial state. In this example, the induced PDB heuristics obtained with either methods are the same.



Abstraction obtained with Qian et al.'s method: $\mathcal{A} = \{A_3, a, u\}$

Abstraction obtained with our method: $\mathcal{A} = \{A_1, A_3, a, u\}$.

Fig. 7.5. A comparison of the two methods

The monotonicity abstraction forms an appropriate basis for choosing abstraction sets because it is computationally efficient and provides useful information about relevance in the problem. Let us consider an example to illustrate this. Figure 7.6 illustrates one of our industrial case studies, called Single-tracked Line Segment. It concerns the design of a real-time controller for a segment of tracks where trams share a piece of track. Each end of the shared piece of track is connected to two other tracks. The safety property to be checked requires that never both directions are given permission to enter the shared segment simultaneously. That property can be violated because some of the temporal conditions in the control automaton are not strict enough.

Let us consider Fig. 7.6 in some more detail. As one would expect, *Actuator 1* and *Actuator 2* are the two automata in direct control of the signals allowing (signal up) or disallowing (signal down) a tram to enter the shared track. In particular, the invariant expresses that always at most one of those signals is up. The *main controller* automaton contains the faulty control logic that governs how the signals are set. The four *counter* automata count how many trains have passed on each of the four tracks that connect to the shared segment. The *error*

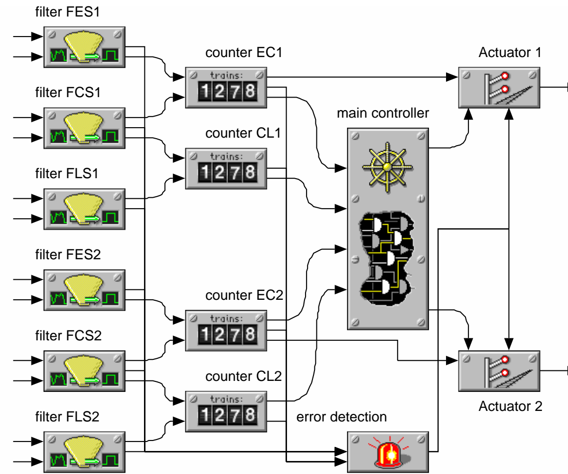


Fig. 7.6. The Single-tracked Line Segment case study

detection detects inconsistencies between the counts, meaning that a train that should have left the shared segment is actually still inside it. Finally, each *filter* automaton receives an input variable from a sensor, and removes the noise from the signal by turning it into a step function based on a simple threshold test. This avoids, for instance, mistaking a passing truck for a tram.

In this example, the advantage of our method is that the found abstract error path touches only *Actuator 1*, *Actuator 2*, and the *control unit*. That is, the abstract error path involves exactly those automata that are immediately responsible for the error. Further, the abstract error path involves exactly the variables that are crucial in obtaining the error. The other, irrelevant, variables and automata have only an indirect influence on the error path and need not be touched to obtain an error under the monotonicity abstraction. Now, consider what happens if we apply Qian et al.'s [87] syntactic backward chaining instead. In the start layer, indexed 0, of the chaining, we have only *Actuator 1* and *Actuator 2*. In the next layer, indexed 1, we correctly get the *control unit*, but we also get *error detection* and two of the *counter* automata. In just one more step, at layer 2, we get every automaton in the whole system. Hence, based on this information, there is no way of separating the relevant symbols from the irrelevant ones.

7.3 Evaluation

In this section, we empirically evaluate our heuristics by comparing them with other heuristics as well as with two uninformed search methods in two different settings. In the optimal setting, we are interested in finding shortest possible

error traces. Therefore we use A^* search with all the admissible heuristics from the next subsection. As a base line, we use UPPAAL's breadth-first search (BFS). In the suboptimal setting, we are interested in finding any solution for the given reachability problems. Here we use greedy search with the heuristics described in the next subsection. As a base line, we use UPPAAL's randomized depth-first search (rDFS) in this setting. The rDFS results are averaged over ten runs.

7.3.1 Experimental Setup

We compare our heuristic with the graph distance-based heuristic d^L proposed by Edelkamp et al. [43] as implemented in MCTA. We also compare our heuristic with the pattern database heuristic proposed by Qian et al.'s [87] which we re-implemented in UPPAAL/DMC. In the tables, we call this heuristic h^{coi}. The reported results for h^{coi} where obtained by first using their abstraction set selection strategy with the cut-off value $d = 3$. This is the best cut-off value that we detected by an exhaustive search in the range of possible values. For a detailed discussion about possible cut-off values, the interested reader is referred to Chap. C in the appendix of this thesis. The pattern databases were built according to this set. We also compare our heuristic with the h^{aa} heuristic proposed by Dräger et al. [39, 40], for which we set the input parameter N to 100 (see Sec. 4.3.2 for more information). Of course we also compare our heuristic with our own, previously introduced heuristics: The h^L heuristic, which is based on the monotonicity abstraction (see Chap. 5), as well as the $h^{\mathcal{P}}_{syn}$ heuristic for which we set the splitting bound b to 2 and the $h^{\mathcal{P}}_{AR}$ heuristic, for which we fixed the number of refinement steps to $r = 4$ and the splitting bound b to 2 (see Chap. 6).

Note that, if a technique requires a parameter setting, then we choose the setting that performs best in terms of total runtime. It is important to note that this does not compromise the other performance parameters: The number of explored states correlate positively with the runtime, the length of the detected error path does not vary significantly over parameter settings.

The examples C_i, for $i = 1, \ldots, 9$, in Tables 7.1 and 7.2 model the Single-tracked Line Segment case study that was explained above. The examples M_i and N_i for $i = 1, \ldots, 4$, come from the Mutual Exclusion case study. A detailed description of all used benchmarks can be found in the appendix of this thesis.

7.3.2 Experimental Results

All reported results are obtained on an AMD Opteron 2.3 GHz system with 4 GByte of memory. Note that the reported runtime results include any preprocessing. For the runtime needed to construct the PDBs, the reader is referred

to Chap. C in the appendix of this thesis. Let us first consider the results for the optimal setting, depicted in Table 7.1. The results for the C examples are striking. While all other techniques suffer from severe scalability issues, we can detect error states in even the largest example in basically no time at all. This is due to the quality of the heuristic, which is clearly indicated in the number of explored search states. Only for $A^*$with the h^{coi} heuristic, the number of explores states is similar to that of our Russian Doll approach, but on the expense of a significantly longer preprocessing.

Concerning the runtime results of the M and N examples, it turns out that the performance of our technique is comparable to those of the other pattern database heuristics. It is worse in the smaller examples where the overhead for computing the Russian Doll pattern database does not pay off in terms of total runtime. Note that this is benign, i. e., what matters are the hard cases. It is remarkable that, consistently, our method explores at least one order of magnitude fewer search states than any of the others, except the h^{coi}-based method. This clearly indicates again that our approach yields the best search guidance.

The results for the suboptimal settings are depicted in Table 7.2. The results for the h^L, h^{aa}, $h^{\mathcal{P}}_{syn}$ and $h^{\mathcal{P}}_{AR}$ heuristics are all much better, compared to the optimal search: They explore fewer states in less time. This improvement is bought at the cost of significantly longer error paths. In most cases, the returned error paths are more than an order of magnitude longer than the shortest possible error path. For rDFS and the heuristic functions by Edelkamp et al. [43], the trace length increase is even more drastic, by another order of magnitude, and with only a moderate gain in runtime.

Concerning the C examples, Qian et al.'s heuristic yields the smallest number of explored states and the shortest error traces. However, the runtime behavior is not convincing. In fact, the runtime of all approaches is clearly dominated by the runtime of our Russian Doll approach. Note that with the $h^{\mathcal{A}}$ heuristic, the number of explored states as well as the runtime increases a bit compared to the A^* configuration. It is not clear to us why this is the case, the loss in error path quality is relatively minor.

In summary, the empirical results clearly demonstrate how superior our Russian Doll heuristic function is, on these examples, in comparison to previous techniques.

7.3.3 Discussion

By far the closest relative to our work is the work by Qian et al.'s [87] which uses an intuitively similar strategy for generating pattern database heuristics. As we have shown, our improved strategy yields much better heuristic functions, at least in our suite of benchmarks. It remains to be seen whether that is also the

Table 7.1. Results for the optimal search. The runtime is the total runtime including any preprocessing. Dashes indicate out of memory ($>$ 4 GByte).

	runtime in s				search space				trace
Exp.	BFS	d^L	h^L	h^{aa}	BFS	d^L	h^L	h^{aa}	length
C_1	0.2	0.1	0.1	0.2	35325	22673	10470	8649	54
C_2	0.6	0.3	0.4	0.4	109583	60588	24658	21719	54
C_3	0.8	0.4	0.5	0.5	143013	80900	28694	28753	54
C_4	8.8	4.1	3.8	2.5	1400895	581942	184413	328415	55
C_5	72.8	38.1	29.1	16.6	12484178	4246042	1140376	2466025	56
C_6	–	–	270.1	216.9	–	–	9250356	24754930	56
C_7	–	–	–	–	–	–	–	–	–
C_8	–	–	–	–	–	–	–	–	–
C_9	–	–	–	–	–	–	–	–	–
M_1	0.6	0.3	0.2	0.3	50001	17810	14128	21945	47
M_2	2.4	1.4	0.9	0.8	223662	62887	47543	70361	50
M_3	2.5	1.4	1.1	1.4	234587	64690	54156	108968	50
M_4	10.6	6.4	4.9	4.8	990513	226368	180353	388475	53
N_1	4.7	3.5	2.8	2.4	100183	50894	40482	46996	49
N_2	22.8	20.5	16.4	9.4	442556	215012	177131	183215	52
N_3	23.2	22.5	19.8	12.6	476622	226509	196083	250928	52
N_4	105.5	130.5	119.9	54.6	2001222	939069	830062	1014036	55

pattern database heuristics

	runtime in s				search space				trace
Exp.	$h^{\mathcal{P}}_{syn}$	$h^{\mathcal{P}}_{AR}$	h^{coi}	$h^{\mathcal{A}}$	$h^{\mathcal{P}}_{syn}$	$h^{\mathcal{P}}_{AR}$	h^{coi}	$h^{\mathcal{A}}$	length
C_1	1.3	3.4	0.8	0.8	7088	9402	130	130	54
C_2	1.3	3.7	2.5	1.6	15742	25931	187	89813	54
C_3	1.5	3.9	3.3	0.8	15586	30559	197	197	54
C_4	2.6	5.6	29.8	0.9	108603	233052	466	1140	55
C_5	8.3	15.1	275.1	1.0	733761	1708529	2147	7530	56
C_6	66.0	134.0	267.8	1.2	7360078	16868109	6229	39436	56
C_7	–	–	267.0	1.8	–	–	16357	149993	56
C_8	–	–	237.6	1.8	–	–	16353	158361	56
C_9	–	–	–	1.9	–	–	–	127895	57
M_1	0.8	1.4	2.4	3.3	22634	26909	15008	190	47
M_2	1.7	2.4	9.4	3.9	94602	112739	63356	4417	50
M_3	2.1	2.9	9.1	3.8	121559	148474	77462	11006	50
M_4	6.5	8.0	38.7	4.5	466967	569469	300262	41359	53
N_1	3.2	4.2	19.2	19.8	46966	56985	345	345	49
N_2	10.9	11.2	107.0	13.2	211935	219579	534	3811	52
N_3	11.8	17.0	105.8	16.2	233609	318317	570	59062	52
N_4	50.7	63.7	529.3	40.6	1036002	1258229	810	341928	55

Table 7.2. Results for the greedy search. The runtime is the total runtime including any preprocessing. Dashes indicate out of memory (> 4 GByte).

	runtime in s				search space				trace length			
Exp.	rDFS	d^L	h^L	h^{aa}	rDFS	d^L	h^L	h^{aa}	rDFS	d^L	h^L	h^{aa}
C_1	0.1	0.0	0.0	0.1	24404	10251	1989	1456	880	747	89	88
C_2	0.3	0.1	0.1	0.3	64042	31886	3559	4139	770	1279	123	151
C_3	0.4	0.2	0.1	0.4	86142	53025	4242	4899	618	1190	119	182
C_4	4.5	2.1	0.6	0.8	921415	378723	20081	46543	1569	4401	116	275
C_5	46.1	21.1	5.1	3.1	8.4e+6	3.0e+6	141174	386523	3745	14518	345	391
C_6	–	218.3	57.4	22.7	–	2.4e+7	1.3e+6	3.7e+6	1096	93504	641	536
C_7	–	–	419.8	300.6	–	–	9.5e+6	3.5e+7	–	–	1175	923
C_8	–	–	102.5	196.2	–	–	2.6e+6	2.6e+7	–	–	556	888
C_9	–	–	179.8	–	–	–	4.6e+6	–	–	–	995	–
M_1	0.4	0.1	0.0	0.2	39838	9885	2431	17007	1246	1306	294	68
M_2	1.3	1.0	0.1	0.7	127973	47783	7436	59944	2809	6878	640	100
M_3	0.9	0.7	0.2	0.7	97987	35502	12924	59594	2716	4797	591	68
M_4	4.1	3.0	0.2	2.1	408400	118621	13497	204396	12k	25555	750	78
N_1	1.7	0.3	0.1	0.8	55690	12052	4834	23700	1100	1413	329	88
N_2	5.9	4.8	131.3	3.8	188784	82488	185770	110482	3350	12557	40652	105
N_3	4.3	2.6	0.1	3.1	146601	46305	7533	96710	3028	12598	499	98
N_4	27.7	59.6	1.2	16.5	917774	504586	46948	511146	15k	51651	1661	193

pattern database heuristics

	runtime in s				search space				trace length			
Exp.	$h^{\mathcal{P}}_{syn}$	$h^{\mathcal{P}}_{AR}$	h^{coi}	$h^{\mathcal{A}}$	$h^{\mathcal{P}}_{syn}$	$h^{\mathcal{P}}_{AR}$	h^{coi}	$h^{\mathcal{A}}$	$h^{\mathcal{P}}_{syn}$	$h^{\mathcal{P}}_{AR}$	h^{coi}	$h^{\mathcal{A}}$
C_1	1.2	3.3	0.9	0.7	1588	1508	130	130	158	112	54	54
C_2	1.3	3.6	2.4	0.8	3786	4098	187	56894	180	128	54	127
C_3	1.4	3.8	3.3	0.8	3846	5583	197	290	186	136	54	56
C_4	1.9	4.5	29.8	0.8	30741	41831	474	1163	240	279	56	57
C_5	3.5	7.2	268.5	1.1	185730	425264	2673	39837	422	506	57	75
C_6	16.6	22.7	270.5	1.3	1.9e+6	2.9e+6	9027	80878	756	770	57	64
C_7	158.9	174.2	269.6	5.0	1.8e+7	2.1e+7	178513	697104	1063	2216	70	64
C_8	120.2	204.0	238.1	8.2	1.5e+7	2.6e+7	28678	1.2e+6	975	2878	57	97
C_9	–	–	–	15.2	–	–	–	2.3e+6	–	–	–	108
M_1	0.7	1.4	2.2	3.1	23257	36808	4461	249	100	115	77	55
M_2	1.5	2.4	8.2	3.5	84475	145178	22172	495	127	136	101	76
M_3	1.5	2.6	8.6	3.4	92548	143275	47705	993	97	134	92	53
M_4	3.8	6.3	33.2	3.5	311049	552341	138147	3577	135	191	81	105
N_1	1.9	3.6	19.5	19.4	31593	51434	242	242	127	82	56	56
N_2	6.6	9.2	106.3	13.2	172531	246878	503	470	157	136	59	63
N_3	6.2	11.5	103.5	11.6	167350	296450	534	1787	129	167	64	70
N_4	33.6	42.9	557.5	7.7	975816	1.4e+6	1217	10394	212	270	65	80

case for other problems. It should also be noted that Qian et al.'s [87] use their heuristic function in a rather unusual BDD-based iterative deepening A^* procedure, and compare that to a BDD-based breadth-first search. As the authors state themselves, it is not clear in this configuration how much of their empirically observed improvements is due to the heuristic guidance, and how much of it is due to all the other differences between the two search procedures. In our work, we use standard heuristic search algorithms. We finally note that Qian et al.'s [87] state as the foremost topic for future work to find better techniques choosing the abstraction. This is exactly what we have done in this chapter. We remark on the side that we developed our technique independently from Qian et al. [87], and only became aware of their work later.

7.4 Conclusion

We have explored a novel strategy for generating pattern database heuristics for directed model checking. As it turns out, this strategy results in an unprecedented efficiency of detecting shortest possible error paths, solving within a few seconds several benchmarks that were previously hardly solvable at all.

Our empirical results must of course be related to the benchmarks on which they were obtained, and it is a priori not clear to what extent they will carry over to other model checking problems. However, there certainly is a non-zero chance that they *will* carry over. This makes the further exploration of this kind of strategy an exciting direction, which we hope will inspire other researchers as well.

8

Useless Transitions are Useful

In the previous chapters of this thesis, we mainly focused on the development of heuristic functions for directed model checking to accelerate the detection of reachable error states. In this chapter, we propose a general enhancement to directed model checking based on the evaluation of *state transitions*. We present a schema, parametrized by a heuristic function, to evaluate transitions and propose a new method for the state space traversal. Our framework can be applied automatically to a wide range of heuristic functions. The empirical evaluation impressively shows its practical potential. Apparently, the new method identifies a sweet spot in the trade-off between scalability (memory consumption) and short error traces.

8.1 Evaluating State Transitions

Before we start to present our new method, let us briefly recall how the state space traversal in directed model checking is guided in order to detect error states. Generally, these guidance criteria are automatically extracted from the model under consideration by taking an abstraction thereof and computing an abstract distance function h, a heuristic function. For a state s, the heuristic value $h(s)$ approximates the distance of s to a nearest error state. These values are used during the state space traversal in order to determine which state is explored next.

Each different version of directed model checking thus arises through the choice of the abstraction that is used to compute the heuristic function, and by the choice of the algorithm for traversing the state space. In the previous chapters, we mainly focused on the first point, i. e., we defined abstractions that lead to heuristic functions to guide the state space traversal efficiently towards an error state. Considering the second point, there are two predominantly used algorithms of directed model checking, namely A^* and *greedy search* (cf. [82]).

A^* is guaranteed to find shortest possible error traces for admissible heuristics, but is often too memory consuming for large systems. Greedy search does not necessarily find shortest possible error traces, but mostly scales much better than A^* in practice.

In this chapter, we present a new version of directed model checking that seems to identify a sweet spot in the trade-off between scalability, i. e., memory consumption, and short computed error traces. It is based on the concept of *useless transitions* which is an adaptation of the *useless actions* approach that has been introduced in the context of AI Planning [95]. As indicated by its name, the concept of useless transitions extends directed model checking by additionally evaluating transitions, not just states. We will see that this is a general concept in the sense that useless transitions can be computed fully automatically with the given heuristic function. That is, whatever the choice of the underlying abstraction for computing the heuristic function has been, we can use the already computed heuristic function in order to effectively recognize useless transitions. We will characterize a class of heuristics for which our method is suited best. We define a new (non-deterministic) strategy for the state space traversal that takes these useless transitions into account. The new strategy is an amalgam of the two strategies A^* and greedy search. For the two extreme cases of abstraction, it becomes the former or the latter, respectively.

We have implemented our method and have applied it to a number of benchmarks coming from the AVACS benchmark suite. This allowed us to experimentally compare the new directed model checking method with the two existing predominant methods A^* and greedy search. The empirical results impressively show the benefit of our approach: We obtain almost shortest error traces, whereas the number of expanded states reduces significantly compared to A^* and also to greedy search in most cases.

8.1.1 An Illustrating Example

In this subsection we provide an example that greatly oversimplifies the issues at hand but gives an intuition about useless transitions and their potential usefulness. Figure 8.1 depicts a system consisting of n parallel components A_1 to A_n. In the initial state of the system, every automaton A_i is in location l_0^i, and an error state is reached if all automata are in their double circled locations. Suppose that we apply directed model checking to check if this error state is reachable. Further suppose that we therefore use the d^L heuristic, the *maximum graph distance* heuristics proposed by Edelkamp et al. [42, 43].

It turns out that, for this problem, the maximum graph distance is a rather uninformed heuristic function. It cannot distinguish states that are nearer to the error state from others. If there is at least one automaton A_i for which $s(A_i) =$

l_0^i, then the heuristic value is 1. We may characterize the state space topology induced by this heuristic as follows. There is one single plateau, i. e., for all of the 2^n reachable system states but for the error state the abstract distance is 1. This means that the guidance based on the d^L heuristic is very poor. In fact, for every heuristic function, similar situations arise.

In the example, it is trivial to see that each state transition that is induced by an edge of an automaton that leaves the automaton's error location (depicted by a double circle) should be avoided as much as possible during the state space traversal. It is a useless transition! Without steps corresponding to such transitions, the state space traversal stops after n steps and returns a shortest possible error path.

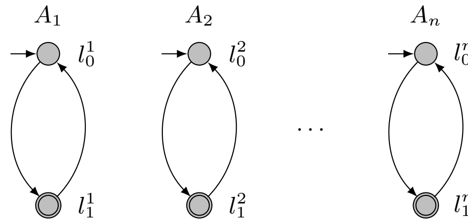


Fig. 8.1. An automata system with n parallel components

8.1.2 Existing Methods

The problem of evaluating state transitions has been studied mostly in the area of AI Planning. In this context, we have proposed an approach to avoid *useless actions* which has led to a significantly improved search behavior on a wide range of planning instances [95]. In Sec. 8.2, we adapt this technique to the context of directed model checking of timed automata systems. Complementary to useless actions, Hoffmann and Nebel [61] and Helmert [56] proposed what they call *helpful actions* and *preferred operators*, respectively. These methods are used to select a set of promising successors to a search state. The helpfulness of a transition is determined during the computation of the heuristic values, which are obtained by solving an abstract problem. Roughly speaking, a transition is considered helpful if it is contained in that abstract solution. However, this approach is specific to the applied distance function.

8.2 Transition-based Directed Model Checking

Until now, directed model checking algorithms have roughly followed the scheme depicted in Fig. 8.2. The figure shows a directed model checking al-

gorithm, that was already discussed in Sec. 4.2.1. Recall that the evaluate function (line 15 of the algorithm) depends on the applied version of directed model checking, i. e., if applied with A^* or greedy search. For A^*, evaluate(s, h) returns $h(s) + c(s)$, where $c(s)$ is the length of the path on which s was reached for the first time. For greedy search, it simply evaluates to $h(s)$.

```
1  function dmc(𝒮, φ, h):
2      open = empty priority queue
3      closed = ∅
4      priority = evaluate(s₀, h)
5      open.insert(s₀, priority)
6      while open ≠ ∅ do:
7          s = open.getMinimum()
8          if s ⊨ φ then:
9              return True
10         if s ∉ closed then:
11             closed = closed ∪ {s}
12             for each outgoing transition t of s do:
13                 s′ = successor(s, t)
14                 if s′ ∉ closed then:
15                     priority = evaluate(s′, h)
16                     open.insert(s′, priority)
17     return False
```

Fig. 8.2. A basic directed model checking algorithm

Directed model checking algorithms that follow this scheme suffer from the fact that A^* is often, due to the high memory consumption, not practical and the error traces of greedy search are often of poor quality. In this section, we propose an extension based on transition evaluation. We will first define the theoretical concept of useless transitions and then its practical counterpart, the relatively useless transitions. This notion can be directly used to combine A^* and greedy search to a new *transition-based* directed model checking algorithm.

8.2.1 Useless and Relatively Useless Transitions

In this section, we give the definition of useless transitions. We will first give an exact notion that captures precisely our intuition on the one hand, but is computationally hard on the other hand. To overcome this problem, we will investigate ways to approximate this definition, leading to the concept of relatively useless transitions. As already mentioned, our technique is a very general enhancement for heuristic search. We therefore first introduce our approach for general transition systems and adapt it to the context of timed automata afterwards. The systems that we consider in this section are concurrent systems with interleaving and binary synchronization. These systems differ from timed automata in

that they do not feature any kind of variables, i. e., no clocks and no integer variables. We will use the symbol $\mathcal{S}$ to denote such systems and $\mathcal{T}(\mathcal{S})$ to denote the corresponding state space.

Intuitively, a transition is useless if it is not needed to reach the nearest error state on a shortest path. This is formally stated in the next definition.

Definition 8.1 (Useless Transition). *Let $\langle \mathcal{S}, \varphi \rangle$ be a reachability problem. A transition t of $\mathcal{T}(\mathcal{S})$ leading from a state s to a state s' is* useless *in s iff no shortest trace from s to the nearest error state starts with this transition.*

We use $d(s)$ to denote the distance of a state s to a nearest error state. More precisely, $d(s) = n$ if there is a trace π from s to an error state with $|\pi| = n$ and there is no trace π' from s to an error state with $|\pi'| < n$. If we want to stress that d is a function also on the system $\mathcal{S}$, we will write $d^{\mathcal{S}}(s)$.

By Definition 8.1, a transition t is useless in a state s if and only if the real error distance d does not decrease by one, i. e., a transition from s to s' is useless iff $d(s) \leq d(s')$. To see this, recall that $d(s) \leq d(s') + 1$ for every transition. If a shortest error trace starts from s with t, then $d(s) = d(s') + 1$. Otherwise the error distance does not decrease, i. e., $d(s) < d(s') + 1$. Since the distance values are all integers, this is equivalent to $d(s) \leq d(s')$. We will use the inequality $d(s) \leq d(s')$ in connection with the idea of *removing* a useless transition. Therefore, we will define the notion of *reduced systems*. To do this, we first need some more terminology. For a concurrent system $\mathcal{S} = A_1 \parallel \ldots \parallel A_n$, we define a function $\mu_{\mathcal{S}}$ that maps transitions from the system's state space $\mathcal{T}(\mathcal{S}) = (S, s_0, T)$ to the edges of the automata A_i, that induce this transition. Note that this function is implicitly given by the implementation of the successor generator (see line 13 of Fig. 8.2).

Figure 8.3 illustrates the mapping $\mu_{\mathcal{S}}$. On the left of the picture, there is a system consisting of two automata A_1 and A_2. On the right, the picture shows the state space of the system. Given the transition $\langle l'_1, l'_2 \rangle \rightarrow \langle l_1, l_2 \rangle$, $\mu_{\mathcal{S}}$ evaluates to $\{l'_1 \xrightarrow{a!} l_1, l'_2 \xrightarrow{a?} l_2\}$. These are the edges that induce the transition.

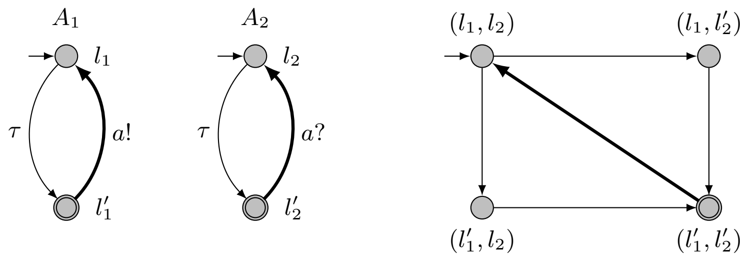


Fig. 8.3. A system and its state space

Based on the definition of $\mu_{\mathcal{S}}$, we now define *reduced systems*.

Definition 8.2 (Reduced system). *Let* $\mathcal{S} = A_1 \parallel \ldots \parallel A_n$ *be a system and* t *a transition of* $\mathcal{S}$*'s state space* $\mathcal{T}(\mathcal{S})$*. The* reduced system *with respect to* t *is defined as*

$$\mathcal{S}_t = A'_1 \parallel \ldots \parallel A'_n,$$

where $A'_i = \langle L_i, l_i^0, E_i \setminus \mu_{\mathcal{S}}(t), \Sigma_i \rangle$*.*

Note that, according to the definition of $\mu_{\mathcal{S}}$, at most two automata A_i are affected by reducing the system (one in the case of interleaving, two in the case of binary synchronization). Roughly speaking, a transition t of the system's state space $\mathcal{T}(\mathcal{S})$ corresponds to one or two edges of one or two automata of $\mathcal{S}$. The reduced system $\mathcal{S}_t$ is obtained by removing these edges from the corresponding automata. Note that removing *one* edge from an automaton A removes *several* transitions from the system's state space $\mathcal{T}(\mathcal{S})$. Figure 8.4 illustrates this. The figure shows the reduced system of the system from Fig. 8.3. The system is reduced with respect to the transitions $t = \langle l_1, l_2 \rangle \rightarrow \langle l'_1, l_2 \rangle$. Removing the corresponding edge $l_1 \rightarrow l'_1$ from automaton A_1 yields the depicted system $\mathcal{S}_t$ and its state space $\mathcal{T}(\mathcal{S}_t)$.

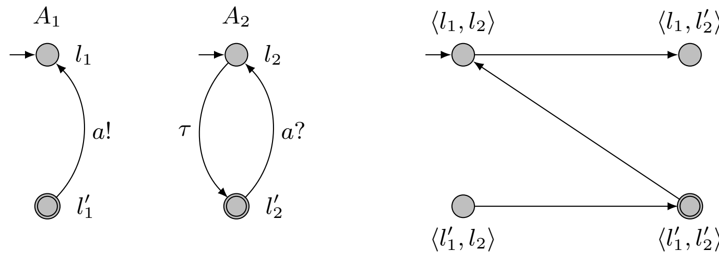


Fig. 8.4. Removing an edge induces several transitions to be removed.

Based on the definition of reduced systems, we now give a proposition that leads to a testing criterion for useless transitions.

Proposition 8.3. *Let* $\langle \mathcal{S}, \varphi \rangle$ *be a reachability problem and let* $\mathcal{T}(\mathcal{S}) = (S, s_0, T)$ *be the state space of* $\mathcal{S}$*. Further, let* $t = s \rightarrow s' \in T$ *be a transition.*

$$\textit{If } d^{\mathcal{S}_t}(s) \leq d^{\mathcal{S}}(s')\textit{, then } t \textit{ is useless in } s.$$

Proof (Proposition 8.3). If $d^{\mathcal{S}_t}(s) \leq d^{\mathcal{S}}(s')$, then $d^{\mathcal{S}}(s) \leq d^{\mathcal{S}}(s')$ because $d^{\mathcal{S}}(s) \leq d^{\mathcal{S}_t}(s)$, i. e., the error distance cannot decrease in reduced systems. As $d^{\mathcal{S}}(s) \leq d^{\mathcal{S}}(s')$ iff t is useless in s, the claim follows directly. □

This property can be interpreted as follows. A transition t is useless in s if the error state is still reachable from s on the same shortest trace when the corresponding edges that induce t are *removed* from the system. However, this characterization is not practical as computing exact distances is **PSPACE**-complete [67].

The above characterization has its own intricacies. To illustrate this, we use the two systems depicted in Fig. 8.5. An error state in each system is reached if its double circled location is reached. Observe that, for a transition to be useless, it is *not* enough to require that $d^{\mathcal{S}_t}(s) \leq d^{\mathcal{S}}(s)$. To see this, consider the left system. Here, every transition that is induced by the outgoing edges of l_0 would then wrongly be recognized as useless. Furthermore, it does *not* suffice to require $d^{\mathcal{S}}(s) = d^{\mathcal{S}}(s')$. The transition induced by the edge from l_0 to l_1 leaving the initial state of the right system would not be recognized as useless although it is, i. e. it is not part of a shortest error trace.

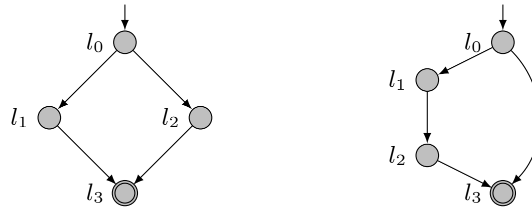


Fig. 8.5. Two example systems

A direct way to approximate the test for uselessness, provided by Proposition 8.3, is to use the given heuristic function h instead of d. This is reasonable because h is designed for exactly the purpose of approximating d. When we want to stress that h is a function also on the system S, we will write $h^{\mathcal{S}}$.

Definition 8.4 (Relatively Useless Transition). *Let $\langle \mathcal{S}, \varphi \rangle$ be a reachability problem and let $\mathcal{T}(\mathcal{S}) = (S, s_0, T)$ be the state space of $\mathcal{S}$. Further, let $t \in T$ be a transition with $t = s \rightarrow s'$ and let $h^{\mathcal{S}}$ be a heuristic function. Then*

$$t \text{ is relatively useless for } h^{\mathcal{S}} \text{ in } s \text{ if } h^{\mathcal{S}_t}(s) \leq h^{\mathcal{S}}(s').$$

Note that this is exactly the testing criterion from Proposition 8.3 where the exact distance function d has been replaced by the heuristic function h. Obviously, the quality of this approximation strongly depends on h's precision. A very uninformed function, e. g. a function that constantly returns zero, recognizes every transition as relatively useless. However, the more sophisticated the distance estimation, the more precise is the approximation. We will come back to this point in the next section. Intuitively, taking a relatively useless transition

t does not seem to guide the state space traversal towards an error state as the *stricter* distance estimate in $\mathcal{S}_t$ does not increase.

One would expect that transitions should not be relatively useless if they lead to states nearer to an error state. Indeed, under the reasonable assumption that heuristic functions h never decrease their estimate in reduced systems, i. e., $h^{\mathcal{S}}(s) \leq h^{\mathcal{S}_t}(s)$ for all systems $\mathcal{S}$, transitions t and states s, transitions leading to better estimates are *never* relatively useless in any system $\mathcal{S}$.

Proposition 8.5. *Let $\langle \mathcal{S}, \varphi \rangle$ be a reachability problem and $\mathcal{T}(\mathcal{S}) = (S, s^0, T)$ be the state space of $\mathcal{S}$. Further, let h be a heuristic function such that for all $s \in S$ and $t \in T$, $h^{\mathcal{S}}(s) \leq h^{\mathcal{S}_t}(s)$. Let $t \in T$ be a transition with $t = s \rightarrow s'$.*

If $h(s') < h(s)$, then t is not *relatively useless for h in s.*

Proof (Proposition 8.5). Assume that t is relatively useless, i. e., $h^{\mathcal{S}_t}(s) \leq h^{\mathcal{S}}(s')$. As $h^{\mathcal{S}}(s) \leq h^{\mathcal{S}_t}(s)$, we have $h^{\mathcal{S}}(s) \leq h^{\mathcal{S}}(s')$, showing that the distance estimate does not decrease when the relatively useless transition t is applied. □

8.2.2 Directed Model Checking with Relatively Useless Transitions

In this subsection, we put the pieces together. So far, we have presented a notion of useless transitions to identify transitions that should be less preferred during the state space traversal. A direct way to integrate this information is to "penalize" states that result from applying such a transition. This is reasonable because avoiding transitions that are not likely to appear in shortest error traces is likely to improve the detection of short error traces. States that are reached by applying such a useless transition should be less preferred when traversing the state space.

As argued in Sec. 8.1 and Sec. 8.2, there are two choices to be made when the directed model checking approach is applied, namely choosing the underlying abstraction for the heuristic functions, and choosing the algorithm that is essentially determined by the *evaluate* function that computes the priority values for the states. Here, we assume that a heuristic function h is already given, and h is additionally used to determine relatively useless transitions. For the second point, we give a simple extension of the *evaluate* function in Fig. 8.6. Recall that s and t (lines 2 and 3) are stored in the successor state and can be accessed easily. As outlined above, states that result from applying a relatively useless transition are "penalized". As penalty value for s, we chose $c(s)$, the length of the trace on which s was reached for the first time. This leads to a combination of A^* and greedy search as discussed in more detail below.

Overall, this algorithm is an amalgam of the algorithms A^* and greedy search based on transition evaluation. Its behavior depends on the accuracy of

```
1 function evaluate(s', h):
2     s = predecessor of s'
3     t = transition from s to s'
4     if t is relatively useless for h in s then:
5         priority = h(s') + c(s')
6     else:
7         priority = h(s')
8     return priority
```

Fig. 8.6. Evaluation function based on relatively useless transitions

the underlying heuristic function h. As mentioned earlier, the more accurate h is, the more transitions are classified correctly, and therefore, the more it tends towards greedy search. At the extreme ends of the spectrum, it becomes greedy search, i. e., for the perfect distance function that classifies every transition correctly, and breadth-first search, respectively, which is a degenerated version of A^* for the distance function that constantly returns zero. From this perspective, our algorithm can be considered as a combination of greedy search and A^*.

8.2.3 Discussion

Although it is technically possible to apply our algorithm to all kinds of heuristic functions, there are heuristic functions that are probably best suited for this concept. Let us have a look at this class of functions. Roughly speaking, heuristic functions can be divided into two classes, namely those that compute the values on the fly by solving an abstract problem in every search state, and those that do it in a preprocessing step, typically by computing a lookup table (e. g., a pattern database). The concept of useless transitions seems to be best suited for heuristic functions that are computed on the fly because the time overhead to compute this information is comparatively low. Contrarily, distance functions from the second class are less suited because for every reduced system, an additional pattern database has to be computed (recall that for the computation of the useless values, the system is reduced and the heuristic value is recomputed on this modified system). However, as we will see, for heuristic functions computed on the fly, the overall performance can often be significantly improved.

The performance of our approach strongly depends on the quality of h that is used to guide the search and to determine useless transitions. The higher the precision of h, the more transitions are evaluated correctly, and hence, the better the overall performance as many unnecessary states need not be considered. In small examples, heuristic functions like the graph distance heuristic d^L proposed by Edelkamp et al. [42, 43] could already lead to improvements. Pointing to our motivating example in Sec. 8.1.1, we recognize that all transitions corresponding to edges from down to up are relatively useless for the graph dis-

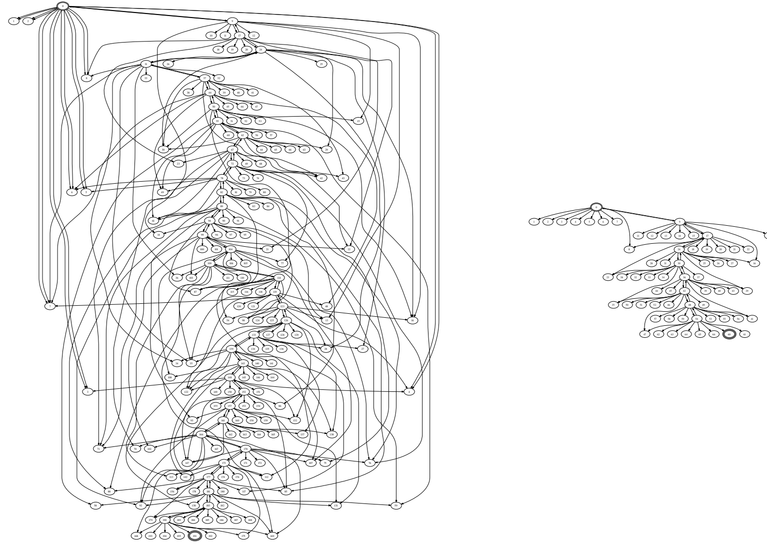


Fig. 8.7. Avoiding useless transitions applied to the example system with 9 parallel automata

tance heuristic, whereas all other transitions are not. In this example, applying our algorithm leads to a shortest possible error trace with dramatically smaller explored state space than with greedy search or A^*. Figure 8.7 is to provide a visual impression. It shows the explored state spaces when the state space traversal does not respectively does take into account the relatively useless transitions recognized by our tool. For more complex examples, more sophisticated heuristic functions are needed to benefit from our approach, as we will empirically show in the next section.

8.3 Evaluation

We have implemented the algorithm from the previous section in our model checker MCTA [72]. We compare our search method with A^* and greedy search. As the examples from our benchmark set are actually timed automata, they differ from the concurrent systems for which we have introduced the concept of useless transitions, in the following points. Timed automata additionally feature integer and clock variables and transitions can additionally be labeled with integer variable and clock guards and integer variable assignments and clock resets. Recall that the concept of useless transitions is general and can be adapted to that class of automata in a straightforward way: Let $\mathcal{S}$ be a system of timed automata. To check if a transition $t = s \rightarrow s'$ of the zone automaton $\mathcal{Z}(\mathcal{S})$ is useless, determine the set of edges that induce this transitions. Then, remove

these edges from the system and compute the heuristic value for s in the reduced system.

To get a conservative approximation of useless transitions, we have implemented our concept in a stronger way than described in the last section. When the reduced system $\mathcal{S}_t$ is computed for a system $\mathcal{S} = A_1 \parallel \ldots \parallel A_n$ and a transition t of $\mathcal{Z}(\mathcal{S})$, we *additionally* remove all edges in the automata A_i that read variables that are set by some edge $e \in \mu_{\mathcal{S}}(t)$, and edges that lead to the same location as some edge $e \in \mu_{\mathcal{S}}(t)$.

8.3.1 Experimental Setup

We evaluated our algorithm for a number of heuristic functions. To ease comparing the results obtained with our approach, we include the results for the two heuristics, h^L and h^U, based on the monotonicity abstraction introduced in Chap. 5. Recall that computing heuristic values with the h^U heuristic is more expensive than using h^L. However, as we will see in Sec. 8.3.2, this pays off in better search behavior. We also include results for the maximum graph distance heuristic d^L introduced by Edelkamp et al. [42, 43].

As usual, we evaluated our approach on benchmarks coming from the AVACS benchmark suite. The C examples model the Single-tracked Line Segment case study, the M and N examples stem from the Mutual Exclusion case study. The F^A and F^B examples are flawed versions of the Fischer protocol. The benchmarks A_2–A_6 contain arbiter trees of height 2–6. The appendix of this thesis provides more information about all our benchmarks.

8.3.2 Experimental Results

The reported experimental results were obtained on a 2.3 GHz AMD Opteron system with 4 GByte of memory. We compare our new state space traversal technique, denoted *UT*, with A^* and greedy search (*G*) in three different configurations. In the first configuration, h^L is used as the heuristic function, the second uses h^U and the third configuration uses the d^L heuristic.

Table 8.1 shows the results of the first configuration. Here, the number of explored states significantly decreases compared to A^* and we are able to solve much larger problems. Compared to greedy search, the length of the found error traces are significantly shorter. Moreover, due to better search guidance, we additionally often get significant improvements in terms of the number of explored states and traversal time.

The results for the second configuration are depicted in Table 8.2. Note that h^U is more informative than h^L, and search behavior therefore is mostly better (in particular, the Fischer protocol examples are trivial for h^U). This fact directly

Table 8.1. Experimental results for the h^L heuristic with A^*, greedy search (*G*), and our combined approach (*UT*). Dashes indicate out of memory (> 4 GByte).

	runtime in s			explored states			trace length		
Exp.	A^*	G	UT	A^*	G	UT	A^*	G	UT
C_1	0.1	0.0	0.1	10470	1989	1822	54	89	81
C_2	0.4	0.1	0.1	24658	3559	1471	54	123	81
C_3	0.5	0.1	0.1	28694	4242	1266	54	119	81
C_4	3.8	0.6	0.1	184413	20081	843	55	116	96
C_5	29.1	5.1	0.1	1140376	141174	398	56	345	67
C_6	270.1	57.4	0.1	9250356	1253431	398	56	641	67
C_7	–	419.8	0.1	–	9475793	398	–	1175	67
C_8	–	102.5	0.2	–	2601885	770	–	556	74
C_9	–	179.8	0.4	–	4641449	1543	–	995	122
M_1	0.2	0.0	0.1	14128	2431	3087	47	294	83
M_2	0.9	0.1	0.1	47543	7436	4775	50	640	95
M_3	1.1	0.2	0.2	54156	12924	8971	50	591	80
M_4	4.9	0.2	0.3	180353	13497	12416	53	750	89
N_1	2.8	0.1	0.2	40482	4834	4922	49	329	87
N_2	16.4	131.3	0.6	177131	185770	17449	52	40652	180
N_3	19.8	0.1	1.3	196083	7533	29661	52	499	125
N_4	119.9	1.2	7.9	830062	46948	125802	55	1661	390
F_5^A	0.0	0.0	0.0	71	9	9	8	8	8
F_{10}^A	0.0	0.0	0.0	511	9	9	8	8	8
F_{15}^A	0.1	0.0	0.0	1701	9	9	8	8	8
F_5^B	0.0	0.0	0.0	54	167	7	6	12	6
F_{10}^B	0.0	2.3	0.0	429	86462	7	6	22	6
F_{15}^B	0.1	–	0.0	1504	–	7	6	–	6
A_2	0.0	0.0	0.0	62	33	15	12	18	12
A_3	0.1	0.0	0.0	2006	202	32	17	24	17
A_4	100.4	9.7	0.1	813303	75106	95	22	36	22
A_5	–	65.7	0.1	–	257208	34	–	74	27
A_6	–	–	0.7	–	–	39	–	–	32

influences the performance when applied with our algorithm: With *UT* and h^U, we obtain even better results than with *UT* and h^L. Note that h^U is not admissible, which means that there is no guarantee to obtain a shortest possible error trace in this setting in theory. However, in practice, we obtained shortest possible traces in our examples with A^*, except in C_4–C_6. The trace length of *UT* is always shorter or of the same length than with greedy search. In particular, note that for both h^L and h^U configurations without *UT*, the large C examples C_7–C_9 could only be solved with an error trace of very poor quality compared to

UT. Moreover, the largest arbiter example A_6 could not be solved at all without *UT* within 4 GByte of memory.

Table 8.2. Experimental results for the h^U heuristic with A^*, greedy search (*G*), and our combined approach (*UT*). Dashes indicate out of memory ($>$ 4 GByte).

	runtime in s			explored states			trace length		
Exp.	A^*	G	UT	A^*	G	UT	A^*	G	UT
C_1	0.2	0.0	0.0	6000	373	237	54	62	54
C_2	0.4	0.1	0.0	13920	663	206	54	74	54
C_3	0.5	0.0	0.0	16146	928	193	54	68	54
C_4	4.3	0.5	0.1	106689	10406	170	59	103	55
C_5	31.1	3.0	0.1	635203	55676	143	58	118	61
C_6	274.2	9.0	0.1	5161691	185293	143	58	130	61
C_7	–	54.8	0.1	–	878345	143	–	200	61
C_8	–	106.8	0.1	–	1878328	378	–	385	94
C_9	–	516.2	0.2	–	7890458	533	–	280	115
M_1	0.2	0.1	0.1	14035	5125	4555	47	71	71
M_2	1.0	0.2	0.1	46681	13153	2816	50	93	78
M_3	1.1	0.2	0.3	52556	12602	10541	50	100	71
M_4	5.6	0.6	0.3	179903	30276	8825	53	150	81
N_1	2.7	0.2	0.2	39500	7691	5410	49	80	77
N_2	16.0	0.8	0.1	158323	23392	5198	52	122	120
N_3	17.3	1.5	0.6	180071	37381	16268	52	113	112
N_4	116.1	10.4	0.4	759644	141354	11166	55	230	228
F_5^A	0.0	0.0	0.0	9	9	9	8	8	8
F_{10}^A	0.0	0.0	0.0	9	9	9	8	8	8
F_{15}^A	0.0	0.0	0.0	9	9	9	8	8	8
F_5^B	0.0	0.0	0.0	7	7	7	6	6	6
F_{10}^B	0.0	0.0	0.0	7	7	7	6	6	6
F_{15}^B	0.0	0.0	0.0	7	7	7	6	6	6
A_2	0.0	0.0	0.0	20	28	20	12	18	18
A_3	0.0	0.0	0.0	23	76	27	17	18	17
A_4	0.1	0.0	0.0	336	39	34	22	28	22
A_5	9.4	1.7	0.3	6644	4027	42	27	47	27
A_6	–	–	1.6	–	–	50	–	–	32

Table 8.3 gives the results for the third configuration (d^L). Here, we observe that the results with *UT* are less significant than with the first two configurations. This is because, having a closer look at the heuristic values in many of the instances, the heuristic values are often constant. This is due to the very coarse abstraction (i. e., the graph distance) used by d^L. Therefore, too many transitions are relatively useless for this heuristic, causing the search process to degenerate

Table 8.3. Experimental results for the d^L heuristic with A^*, greedy search (G), and our combined approach (UT). Dashes indicate out of memory ($>$ 4 GByte)

	runtime in s			explored states			trace length		
Exp.	A^*	G	UT	A^*	G	UT	A^*	G	UT
C_1	0.1	0.0	0.1	22673	10251	13194	54	747	61
C_2	0.3	0.1	0.3	60588	31886	34353	54	1279	61
C_3	0.4	0.2	0.6	80900	53025	47049	54	1190	61
C_4	4.1	2.1	3.7	581942	378723	331062	55	4401	62
C_5	38.1	21.1	31.0	4246042	2957129	2369452	56	14518	63
C_6	–	218.3	301.0	–	24247904	19240827	–	93504	59
C_7	–	–	–	–	–	–	–	–	–
C_8	–	–	–	–	–	–	–	–	–
C_9	–	–	–	–	–	–	–	–	–
M_1	0.3	0.1	0.3	17810	9885	12773	47	1306	100
M_2	1.4	1.0	0.7	62887	47783	34263	50	6878	97
M_3	1.4	0.7	0.8	64690	35502	40105	50	4797	101
M_4	6.4	3.0	3.2	226368	118621	124604	53	25555	91
N_1	3.5	0.3	0.8	50894	12052	21336	49	1413	88
N_2	20.5	4.8	6.2	215012	82488	101174	52	12557	95
N_3	22.5	2.6	5.8	226509	46305	102765	52	12598	118
N_4	130.5	59.6	40.0	939069	504586	461638	55	51651	132
F_5^A	0.0	0.0	0.0	597	48	597	8	40	8
F_{10}^A	0.2	0.0	0.2	13102	48	13102	8	40	8
F_{15}^A	4.5	0.0	4.6	108017	48	108017	8	40	8
F_5^B	0.0	0.0	0.0	78	449	9	6	65	6
F_{10}^B	0.0	169.1	0.0	523	5502590	9	6	1860	6
F_{15}^B	0.1	–	0.0	1718	–	9	6	–	6
A_2	0.0	0.0	0.0	209	27	212	12	22	12
A_3	0.2	0.0	0.3	18792	151	16089	17	91	17
A_4	–	0.3	–	–	28742	–	–	501	–
A_5	–	0.1	–	–	2883	–	–	2606	–
A_6	–	–	–	–	–	–	–	–	–

towards A^*. However, *UT* mostly still explores fewer states than A^*, thereby producing *significantly* shorter error traces than greedy search.

Overall, the concept of useless transitions has shown its potential in an impressive way. The results show a significant improvement of the error traces in comparison to greedy search as well as a significant reduction of the explored state space compared to A^*. On many problems, the size of the explored state space is even smaller than with greedy search. Our experimental evaluation has shown this effect on a large number of benchmarks, ranging from academic to industrial examples with instances of different difficulties, ranging

from very easy to very hard. We have seen that the overall performance of *UT* depends on the precision of the underlying heuristic function. With *UT*, a sophisticated heuristic like h^L already often leads to significant better guidance of the state space traversal than with greedy search and A^*. More informative heuristic functions, like h^U, also lead to better search guidance when applied with *UT*, and hence, the number of explored states further decreases. With less informative heuristics, like d^L, the impact of *UT* decreases and the whole search process degenerates towards A^*.

8.4 Conclusion

We have introduced the concept of useless transitions to directed model checking as an adaptation of the useless actions approach that has been successfully proposed in the area of AI Planning. Based on useless transitions, we have proposed a hybrid algorithm between A^* and greedy search that seems to identify the sweet spot of the trade-off between scalability and short computed error traces. We have implemented this algorithm and evaluated it empirically on a number of benchmarks for a number of heuristic functions. Our empirical evaluation shows a substantial performance gain in terms of explored states when compared with A^*, and a significant solution quality improvement when compared with greedy search. Due to better guidance abilities, we often even explore less states than greedy search.

As outlined in the discussion section, our approach seems to be currently best suited for heuristics that are computed on the fly, and less suited for heuristics based on pattern databases. This is because the time overhead seems to be too large when adapting it for such functions in a straight forward way. To investigate how to adapt our concept *efficiently* to pattern databases heuristics will be an important topic for future research. Furthermore, it will be interesting to refine our concept to more than two degrees of uselessness. We expect that algorithms exploiting that knowledge further improve the state space traversal.

9

Automatic Abstraction Refinement for Timed Automata

While the previous chapters deal with the development and improvement of directed model checking methods, this chapter deals with counterexample-guided abstraction refinement (CEGAR) for real-time systems. At a first glance directed model checking and CEGAR seem to be two completely different methods. But both approaches have the capability to analyze huge systems whose time and space complexity lies beyond the scope of dump brute force methods. Typically, both approaches achieve this by using overapproximations of the system under consideration. In directed model checking overapproximations are used to compute heuristic functions. CEGAR approaches iteratively search for reachable error states in abstractions of increasing complexity. Since these abstractions are overapproximations, absence of abstract counterexamples implies a valid result for the original system. If an abstract counterexample is found it is used to construct either a counterexample for the original system or to identify a slightly refined abstraction in which the found spurious counterexample cannot occur anymore.

The main part of this chapter deals with the development of an efficient CEGAR approach for real-time systems modeled in a subset of timed automata, namely PLC automata [31]. Afterwards we will come back to directed model checking by comparing our CEGAR approach with our previously discussed directed model checking methods.

9.1 Counterexample-guided Abstraction Refinement

Counterexample-guided abstraction refinement is an abstraction-based approach to tackle the state explosion problem. The main idea of this approach is to find an appropriate abstraction of the system under consideration that can be analyzed within the given memory and time limit. If the used abstraction is an overapproximation, then the absence of an abstract counterexample implies the

absence of concrete counterexamples. Such an abstraction is called a *safe* abstraction. However, if a model checker analyzes a safe abstraction and discovers a counterexample, the question arises whether the counterexample is spurious or not, i. e., whether there exists a corresponding counterpart in the full model.

Without automation, model checking by abstractions consists of several, error-prone and time-consuming tasks. First, the user has to select a proper initial abstraction which usually requires a deep knowledge of the system under consideration. If the model checker finds a counterexample in this abstraction, then one has to analyze whether it is spurious or not. If the counterexample is real, then we are done, otherwise the selected abstraction was too coarse. In this case the initial abstraction must be refined so that the spurious counterexample is eliminated. These steps are repeated until there is no abstract counterexample, or the abstract counterexample is also a counterexample for the full system.

The approach presented in this chapter automates *all* steps of the abstraction refinement loop in the setting of real-time specifications given as PLC automata [33]. This leads to a fully automated abstraction refinement loop as depicted in Fig. 9.1. The loop starts with the coarsest possible abstraction and iterates as long as spurious counterexamples are found. If the model checker finds an abstract counterexample (CE), then a counterexample analyzer is used to check whether it is spurious. Our analyzer first constructs a test automaton from the abstract counterexample. The composition of the full model with the newly generated automaton is then fed into the model checker. We extended the model checker in such a way that if the counterexample is spurious, the model checker also reports hints why it is spurious. These hints are then used in the refinement step in order to eliminate the abstract counterexample. The termination of this abstraction refinement loop is guaranteed by the fact that each iteration removes at least one of finitely many abstracted variables.

To demonstrate the potential of our approach we carried out several nontrivial case studies, i. e., the respective full models cannot be handled within the given memory resources by blind search. Our approach is able to handle them with only a fraction of the resources, starting from the initial abstraction towards a refined abstraction for which a definite answer could be found.

9.1.1 Existing Methods for Timed Automata

Abstraction refinement was pioneered by Clarke et al. [26] in the early 90s. Since then many researchers have automated this process starting with the work of Balarin and Sangiovanni-Vincentelli [4]. Also in the area of timed automata there are approaches for model checking by iterative refinement of approximations. One of these approaches was implemented in Laroussinie and Larsen's compositional model checker CMC [74]. This tool starts with a small subset of

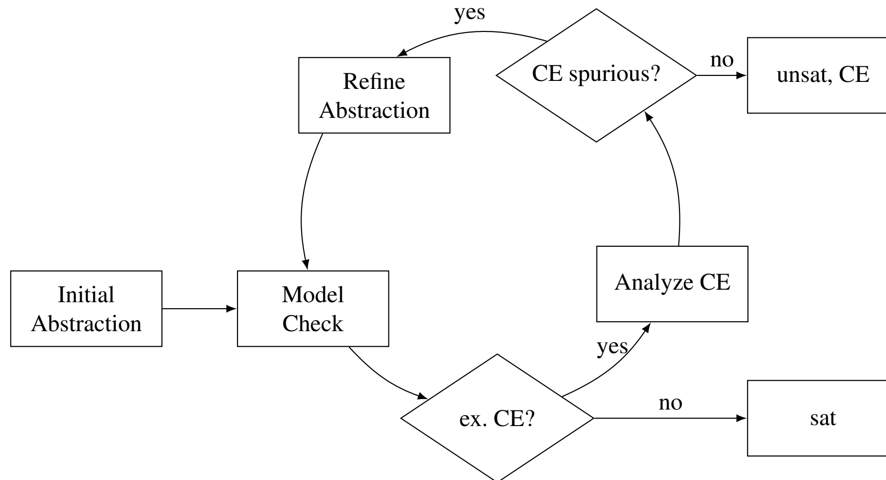


Fig. 9.1. A Counterexample-guided abstraction refinement loop

the automata of the system. It subsequently adds automata to this set and minimizes the intermediate result.

Another abstraction refinement approach was developed by Möller et al. [79] and Sorea [92] in which predicate abstraction was used at the level of the regions which are defined by predicates over clocks. The approach uses symbolic counterexamples from failed model checking attempts. Such a symbolic counterexample represents a sequence of sets of states, and can be seen as generalization of a linear counterexample. To exclude a spurious symbolic counterexample from further iterations, new abstraction predicates are chosen randomly from a set of predefined predicates. Except for the fact that new abstraction predicates are chosen randomly, this approach is, in some respect, quite similar to what we are proposing here. The main differences are the nature of the counterexamples and that new abstraction variables are selected more carefully. Unfortunately the authors do not give any runtime results.

9.2 From PLC Automata to Timed Automata

In our setting, a reachability problem $\langle \mathcal{S}, \varphi \rangle$ consists of a real-time system $\mathcal{S}$, which is given in terms of PLC automata, together with a target formula φ. To solve such a reachability problem with our framework, we use MOBY/RT to compute an abstraction of the given PLC automata. MOBY/RT is a tool for the development and analysis of PLC automata [80]. From the verification perspective the most important feature of MOBY/RT is the generation of safe abstractions of an arbitrary set of PLC automata together with a temporal property into

the input syntax of UPPAAL 3.4. These abstractions are generated according to the entities (e. g., variables and delays) of the PLC automata the user has chosen for abstraction.

In the following we will briefly describe PLC automata and how they are related to timed automata.

9.2.1 PLC Automata

The formal specification language called PLC automata has been developed to enable formal verification of real-time properties of PLC programs. A Programmable Logic Controller (PLC) is a standardized hardware platform which is especially equipped to simplify the design of real-time controllers in practice. It can be seen as a simple computer with a special real-time operating system. A PLC communicates with the environment via unbuffered asynchronous input and output channels. The environment may change the values of the inputs arbitrarily whereas the outputs are controlled by the PLC. PLCs behave in a cyclic manner where every cycle consists of the following three phases: first the inputs are polled, then the new output values are computed and finally the outputs are updated. The repeated execution of this cycle is managed by the operating system. The only part the programmer has to adapt is the computing phase. Depending on the program and on the number of inputs and outputs there is an upper time bound for such a cycle.

In the definition of PLC automata we consider the upper time bound for a complete cycle and the possibility to delay the system's reactions depending on state and input. Figure 9.2 gives an example of a PLC automaton. The automaton has three locations l_0, l_1, l_2 and an output variable *output* that ranges over *ok*, *test* and *alarm*. It reacts to a Boolean input variable *signal*. Every location has two labels shown below its name in the picture. They define a delay time d and a constraint ψ on the input. The value of d defines the *minimal* amount of time that the system should stay in the corresponding location provided that, meanwhile, only input values satisfying ψ are polled.

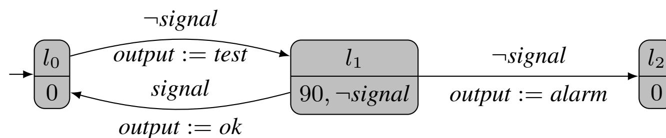


Fig. 9.2. An example of a PLC automaton

A PLC automaton describes the behavior of the system in the computation phase. The operational behavior is similar to a finite state machine, i. e., depend-

ing on the polled input value, the system changes both its state and its output. The behavior is modified in only one case: if the annotations of the current state are d and ψ, then a transition is only executed when the polled input does not satisfy ψ *or* the current state holds longer than d time units. That means a transition is disabled if the polled input satisfies ψ and the current state has not exceeded the delay time d.

Thus, the PLC automaton in Fig. 9.2 behaves as follows: it starts in location l_0 and remains there as long as it reads only the input *signal*, provided by the environment. The first time it reads $\neg$*signal* it changes to location l_1. In l_1 the automaton reacts to the input value *signal* by changing the state back to l_0 *independently* of the time it stayed in state l_1. If it reads $\neg$*signal* in l_1 the behavior of the system depends on the duration it already stayed in l_1. If 90 time units have elapsed it can take the transition to l_2. Otherwise, no transition is fired. In l_2 the automaton remains forever. If the cycle time of the automaton is smaller than 2 time units, we know that the automaton changes its output to *alarm* when $\neg$*signal* holds longer than 90 time units because the cycle time of 2 time units has to be considered.

Note that PLC automata are implementable. Dierks proposed a translation of PLC automata into source code for PLCs [30]. Olderog and Dierks [80] also developed a translation into C++ code, which is tailored to Lego Mindstorms. The intended logical relationship between the execution of the code by the real-world hardware and the semantics given in the rest of this paper is *refinement*. In other words: the real-world implementation cannot show a behavior that is not covered by the formal semantics.

9.2.2 Semantics of PLC Automata

The semantics of PLC automata is defined in terms of timed automata [30, 33]. In this thesis we do not want to give a formal definition thereof, since only an intuitive understanding of their functioning is needed to understand our approach. Here, we just present a sketch of the semantics using the example of Fig. 9.2. The semantics of this automaton, depicted in Fig. 9.3, consists of two timed automata and the following shared variables and clocks: z is a clock that represents the duration of the PLC cycle, y is a clock that measures how long the system stays in l_1, *Out* and *sig* represent the variables *Output* and *signal* respectively, used in the PLC automaton. The variable *Psig* represents the *polled* value of the input variable *signal*. The locations of the PLC automaton appear as a set of locations in the timed automaton. For example, the location l_0 has two representatives in the timed automaton in order to represent the internal state of the PLC within the cycle, i. e., l_0/p stands for polling, l_0/cu stands for computing and updating. The transitions between l_0/p and l_0/cu implement the

polling behavior of the PLC automaton in location l_0. The polling step copies the current value of *sig* to the variable *Psig* which is used for the subsequent computations. The outgoing transitions from l_0/cu represent the reactions of the PLC automaton in location l_0. Depending on the polled value of *sig* the system switches to l_1/p or remains in l_0/cu. Moreover, these transitions reset the z clock because the cycle has been finished. If the automaton switches to l_1/p the output variable is changed appropriately and the clock y is reset to be able to check whether 90 time units have elapsed while staying in location l_1. Since l_1 is equipped with a delay the semantics needs more locations to represent the behavior. After polling (transition from l_1/p to l_1/c) the timed automaton checks whether the delay time has passed or the delay condition is not satisfied. If this is the case, then a transition to l_1/u is enabled. Otherwise the system can switch to l_1/d ("delayed") where the cycle is finished. Note that the semantics is non-deterministic with respect to the timing in order to model the physical reality. To ensure progress each location has the invariant $z \leq 2$. Note that, for the sake of readability, these invariants are not shown in Fig. 9.3. Therefore, the timed automaton has to execute a cycle within the upper bound because only transitions to l_i/p reset the z clock. To model the environment that can change the input variable *sig* arbitrarily a so-called driver automaton, depicted on the right of Fig. 9.3, is added that can always toggle the input.

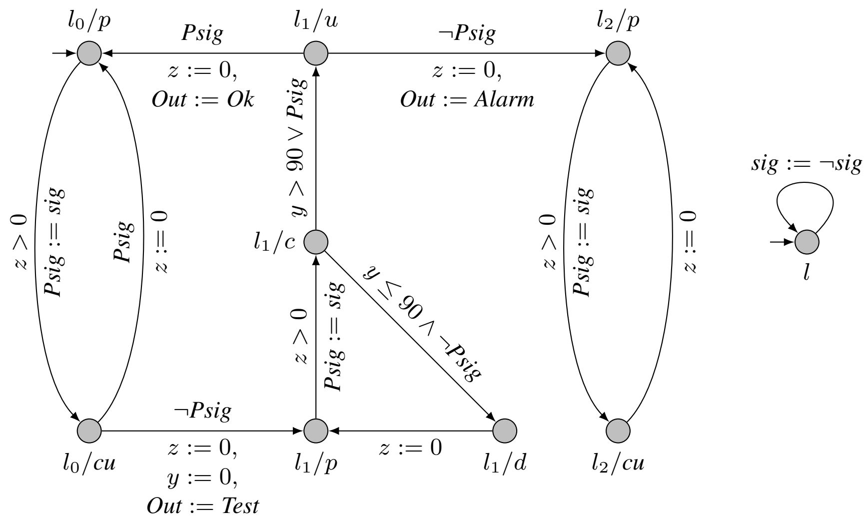


Fig. 9.3. The semantics of the PLC automaton from Fig. 9.2 in terms of timed automata

9.2.3 Generating Abstractions

Abstractions of this semantics are generated by selecting a set of bounded integer variables and clocks. Then, in a sense reminiscent of the definition of closed abstraction sets (see Sec. 7.1.2), the abstract semantics is derived from the full semantics by removing all assignments to the abstracted variables and clocks and replacing all constraints by the strongest constraint that is weaker than the original and that does not contain abstracted entities. For example, the guard $y \leq 90 \wedge Psig$ is replaced by $Psig$ if the clock y is abstracted and sig is not.

9.3 Abstraction Refinement for Timed Automata

In this section we describe how we automate the single components of the abstraction refinement loop. In the next subsection we present how an abstract counterexample is checked whether it is spurious or not. In the following subsection, we close the loop by explaining how we get a refined abstraction, provided that the encountered abstract counterexample is spurious.

9.3.1 Counterexample Analysis

In the abstraction refinement loop, when model checking a safety property of an abstract system returns a counterexample, one has to investigate whether it is spurious or not. One possibility of doing this is to build a *linear* test automaton T from the abstract counterexample, in which every location has at most one outgoing edge. This test automaton is then composed with the full system $\mathcal{S}$ to be verified. If the last location of the test automaton is reachable in the composed system, then we know that the abstract counterexample is also a counterexample for the full system.

It is desirable that the test automaton T is able to find a concretization if there is any and, moreover, it is mandatory that T restricts the behavior of the full system as much as possible. If this is not the case, the test automaton is useless because it would make no difference to model check the full system only. We therefore construct the test automaton so that it synchronizes with the full system as often as possible. By doing this, only a small fraction of the full system's state space is explored. This makes perfect sense, because we are only interested in a concretization of the abstract trace.

To additionally restrict the behavior of timed automata, UPPAAL provides so-called *committed locations*. A system state is a committed state if at least one of the current automata locations is committed. Committed locations are a mechanism to restrict the behavior of the overall system. A committed state cannot delay and the next transition applied to this state must involve an outgoing

edge of a committed location. Figure 9.4 shows a system of three parallel timed automata P, Q and R. The system has three clocks x, y and z, three integer variables k, m and n, binary synchronization labels a and b and a committed location q_2 (indicated by the "c"). For example, if the system is in a state where P is at p_1, Q is at the committed location q_2 and R is at r_1, then this system state is committed. The system then has to take the edge from q_2 to q_3 which requires Q to synchronize with R.

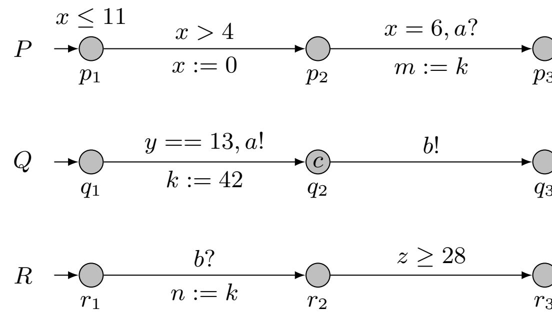


Fig. 9.4. A simple UPPAAL model

A counterexample, as provided by UPPAAL, is a finite sequence of states that are either connected via transitions or via delays with a certain duration. Suppose the initial state of the system depicted in Fig. 9.4 is given by the three initial locations p_1, q_1 and r_1 together with the initial values of the integer variables and the initial values for the clocks. All clocks and integer values are 0 in this state. If we use UPPAAL to verify whether the temporal formula $\exists\Diamond(P = p_3 \wedge Q = q_3 \wedge R = r_3)$ holds, the tool reports the error trace given on the left of Fig. 9.5. The trace starts with the initial state of the system and ends with a state satisfying the given property. Each state of the trace assigns each automaton of the system a location, each variable a value of the variable's domain and each clock a rational value. A test automaton built from such a trace proceeds if the full model of the system executes a transition that enables the guard of the next transition of the test automaton. The problem is to decide whether a transition in the full model matches. Checking the values of the integer variables is straight-forward. To check the clock values is a bit tricky because a trace may also have clock valuations with rational numbers. Below we show how this problem can be solved. Another problem is to match the locations. The locations are not subject to abstractions. Hence, they appear in both the full and the abstract model. However, UPPAAL does not provide any syntactic means to refer to locations in guards, i. e., location constraints may only occur in the target formula. Therefore, we introduce an auxiliary integer variable for each automaton

of the system. The current value of this variable identifies uniquely the current location of the automaton. Thus, the test automaton is able to match locations by checking the corresponding auxiliary variable. In order to preserve the same behavior with respect to global time we add a clock T to the test automaton. This clock is never reset and therefore it represents the current duration of the trace at any time. The next three paragraphs give a detailed description on how to build a test automaton.

Translating the Initial State

The first element of an abstract trace is the initial state of the abstract system. For reasons of completeness, we add a corresponding check for reasons of completeness. We add a transition leading from the initial location t_0 of the test automaton to a new location t_1. The guard of this transition checks if the system's variables have the right values at time point *time* $= 0$. To restrict the full model's behavior t_0 is a committed location.

Translating Delay Transitions

A delay transition with duration $d \in \mathbb{Q}$, where $\mathbb{Q}$ denotes the rational numbers, of the abstract counterexample is translated as follows: suppose that, after the transition is made, the system is in a state which is described by the valuation *val* and that $T \in \mathbb{Q}$ is the sum of all durations that occur before the transition. Let the most recently added location of the test automaton be t_n. In this case we add two locations t_{n+1} and t_{n+2} to the test automaton and the two transitions $t_n \xrightarrow{\text{await}(T+d)} t_{n+1}$ and $t_{n+1} \xrightarrow{\text{await}(T+d) \wedge \text{check}(\mathit{val})} t_{n+2}$. Here $\text{await}(q)$ for $q \in \mathbb{N}$ is $T = q$ and for $q \in \mathbb{Q} \setminus \mathbb{N}$ is $\lfloor q \rfloor > T \wedge T < \lceil q \rceil$. Note that by the definition of $\text{await}(q)$ the test automaton searches for an overapproximation of the given abstract trace. The expression $\text{check}(val)$ is the conjunction of tests whether all discrete variables of *val* are equal to the valuation of the state. Note that this approach is correct because if there is no concretization found using this overapproximation then there is no concretization at all. If a concretization is found then the concrete trace may differ from the abstract trace with respect to the non-integer delays but it is a trace of the full model that satisfies the reachability property. An alternative approach would be to multiply all timing constants with the common denominator and check for equality only.

Translating τ Transitions

A τ transition in the counterexample also introduces two locations in the test automaton. Suppose the last added location of the test automaton is t_n

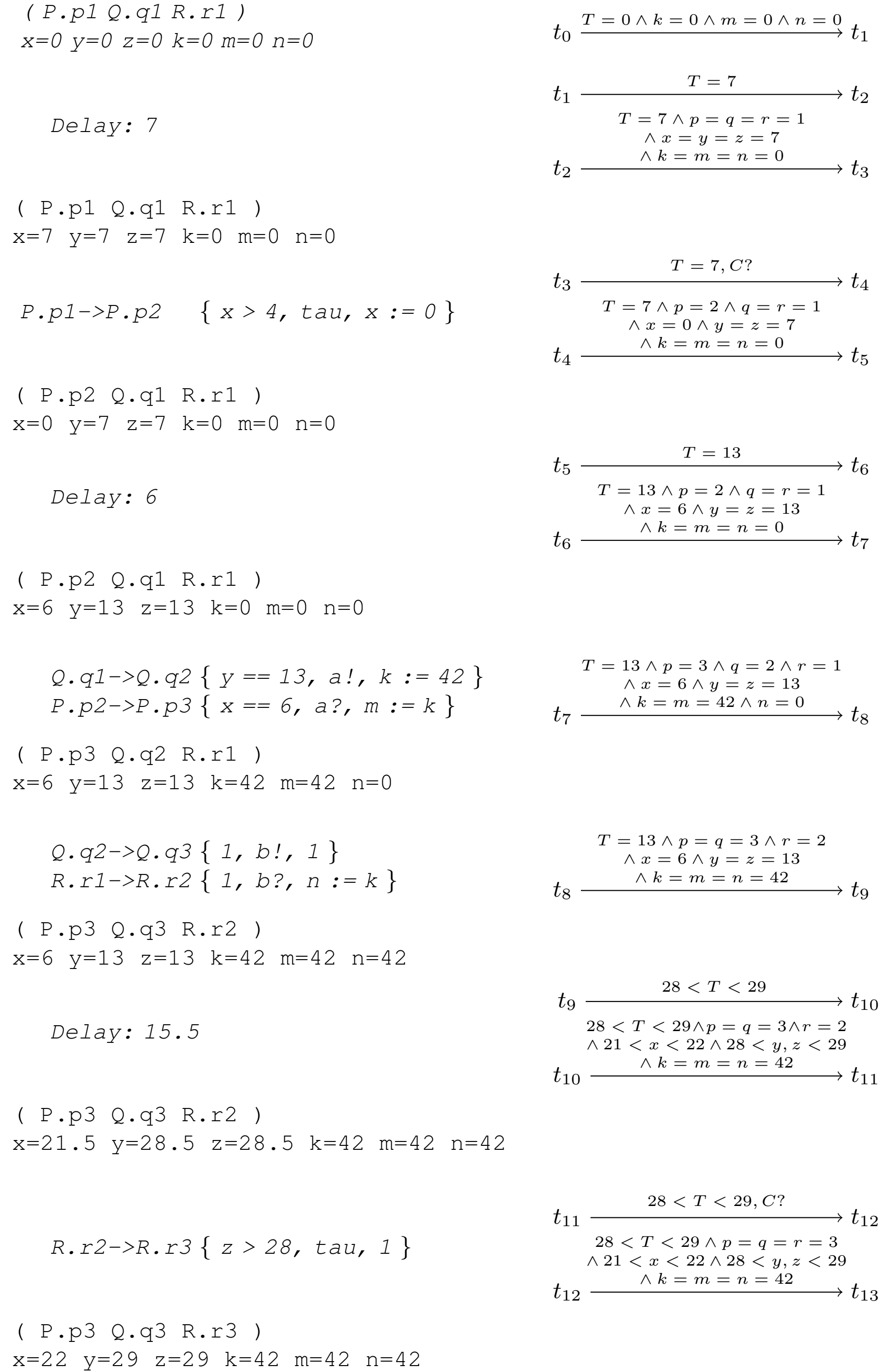


Fig. 9.5. A trace reported by UPPAAL for the system from Fig. 9.4 when analyzing the formula $\exists\Diamond(P = p_3 \wedge Q = q_3 \wedge R = r_3)$ and the corresponding test automaton

and after the τ transition the system is in a state described by the valuation *val*. Let T and t_n be defined as above. Here, we add the locations t_{n+1} and t_{n+2} to the test automaton and the two transitions $t_n \xrightarrow{\text{await}(T)\wedge C?} t_{n+1}$ and $t_{n+1} \xrightarrow{\text{await}(T)\wedge\text{check}(val)} t_{n+2}$. Note that we exploit the fact that the abstract model made a τ transition to introduce a synchronization of the test automaton with the full model via the channel C. By this, the test automaton is notified as soon as the full model changed the values for the variables and the search space is reduced. Since time is supposed to pass, the location t_n cannot be committed. However, to restrict the behavior of the system t_{n+1} can be committed.

Translating Synchronized Transitions

A synchronized transition of the counterexample is translated as follows: suppose again, that the valuation *val*, T and t_n are defined as above. In contrast to how we handle a τ transition, we cannot force the test automaton to participate. However, we know that whenever MOBY/RT generates a synchronized transition this only happens directly after a τ transition and that all synchronized transitions are leaving a committed location. Note that this is a MOBY/RT specific property. Therefore we add the location t_{n+1} and the transition $t_n \xrightarrow{\text{check}(val)\wedge\text{await}(T)} t_{n+1}$. As synchronized transitions can happen sequentially when more than one automaton is in the network and the transition's origin is a committed location, we have to make location t_n committed, too.

Consider the trace generated by UPPAAL on the left of Fig. 9.5 again. If we apply the above construction rules we get the test automaton described on the right of that figure. For the sake of readability, locations in the test automaton are not marked as committed.

9.3.2 Refining Abstractions

When an abstract counterexample reveals to be spurious, a model checker normally does not give any hint why this is the case. Usually, the next steps one has to do in order to get a refined version of the current abstraction is to analyze the counterexample. Here, one has to identify variables or clocks respectively that hinder the progress of the test automaton. Normally, this has to be done *manually* and, on the one hand, is a tedious and time consuming procedure and on the other hand requires a deep understanding of the model to verify. We will henceforth refer to both integer variables and clocks simply as variables.

Our approach automates the analysis of the counterexample. We did this by extending UPPAAL so that if an abstract counterexample is spurious, UPPAAL reports this *and* at the same time provides a set of variables that should not be abstracted in the next iteration of the abstraction refinement loop.

To determine a refined abstraction, we exploit the fact that our test automata are linear. If the abstract counterexample turns out to be spurious, then there is a unique edge in the test automaton whose starting location is reached, but not its target location. We call the starting location of this edge the *dead end location* and the transition induced by this edge we call the *dead end transition*. The dead end transition can either be blocked because there is no enabled transition in the full system that can synchronize with the test automaton, or because there is no reachable state with a valuation of the variables that satisfies the guard of the dead end transition.

Automatic Analysis of Counterexamples

Our approach for automatically analyzing abstract counterexamples determines a minimal set of variables $\mathcal{V}_{hint}$, so that if these variables had different values, the test automaton T could take at least the dead end transition. This is done on the fly, while UPPAAL checks if the error trace is spurious. Figure 9.6 sketches a typical verification algorithm for safety properties. The arguments of the *verify* function are the system under consideration, composed with the test automaton $\mathcal{S} \parallel T$, and the property φ to verify. The state s_0 is the initial state of the system. We extended this algorithm by including the lines 13–15.

```
1  function verify(S || T, φ):
2      open = {s0}
3      closed = ∅
4      while open ≠ ∅ do:
5          s = open.pop()
6          if s ⊨ φ then:
7              return True
8          if s ∉ closed then:
9              closed.push(s)
10             for each outgoing transition t of s do:
11                 if t is enabled then:
12                     s′ = succ(s, t)
13                     open.push(s′)
14                     progress(s′)
15                 else:
16                     analyze(s, t)
17     return False
```

Fig. 9.6. Reachability analysis

During the analysis, UPPAAL checks each outgoing edge from a location in the current state s if it is enabled. If this is the case, the successor state s' is computed and added to the open list. In addition to the normal verification function,

we now call *progress* with the successor state s' (line 14). Pseudo-code for the *progress* function is given in Fig. 9.7. It determines the reached location of the test automaton that has currently the smallest distance to the test automaton's last location. Remember, if the last location of the test automaton is reachable, then the counterexample is a real counterexample. If the counterexample is spurious, then after the execution of the *verify* function, the location *dead_end* from Fig. 9.7 is the dead end location.

```
1 function progress(s):
2     if dist(s(test)) < dist(dead_end):
3         dead_end = s(test)
4         V_hint = ∅
```

Fig. 9.7. On the fly detection of the dead end location

If, during the generation of successor states, a transition t is not enabled it is passed together with its starting state s to the *analyze* function (line 16). Pseudo code for this function is shown in Figure 9.8. The *analyze* function checks if the test automaton in s is in the current *dead_end* location. If this is the case, then it checks if applying t would enable the current dead end transition t_{dead_end}. If this is the case, then *analyze* collects all the variables and clocks respectively that appear in unsatisfied constraints of t's guard. If the set of these variables u is smaller than $\mathcal{V}_{hint}$, then $\mathcal{V}_{hint}$ is updated. After the execution of the *verify* function $\mathcal{V}_{hint}$ contains variables that hinder the execution of the dead end transition.

```
1 function analyze(s, t):
2     if s(test) ≠ dead_end then:
3         return
4     if t_dead_end is synchronized then:
5         if t can synchronize with t_dead_end then:
6             V = {c ∈ inv(succ(s, t)) | c unsat constraint}∪
7                 {c ∈ guard(t) | c unsat constraint}
8             if |V| ≤ |V_hint| ∨ V_hint = ∅ then:
9                 V_hint = V
10    else if assignment of t makes guard(t_dead_end) true then:
11        V = {c ∈ inv(succ(s, t)) | c unsat constraint}∪
12            {c ∈ guard(t) | c unsat constraint}
13        if |V| ≤ |V_hint| ∨ V_hint = ∅ then:
14            V_hint = V
```

Fig. 9.8. On the fly extraction of least blocking variables. Used expressions: $\text{inv}(s)$: conjunction of s's location invariants, $s(test)$: the location of the test automaton in s, succ(s, t): the successor state of s reached through t, guard(t): t's guard, dist(l): distance from a location l of the test automaton to the last location of the test automaton in terms of transitions.

Explanations

After UPPAAL has checked that the counterexample is spurious, all variables that occur in the set of unsatisfied constraints $\mathcal{V}_{hint}$ are reported. These variables should not be abstracted in the next iteration, as they hinder the progress of the test automaton. This ensures that the revealed spurious counterexample will not be found in the next iteration. In the following, we explain that the reported variables are likely to be helpful.

From the construction of the test automaton we know that there are at most two types of transitions in the test automaton: synchronized transitions and τ transitions. The guard of such a synchronized transition is always satisfiable because it was already satisfied when the starting location of the transition was reached. It only checks that no time elapses since the last transition. Depending on which part the transitions represent from the abstract counterexample, we can distinguish three different cases.

If the dead end transition belongs to one of the transitions introduced for a delay in the abstract counterexample, then the progress of the test automaton is blocked because the full system cannot idle due to an unsatisfied location invariant. This is only possible if this location invariant talks about a clock that was abstracted away because in the abstraction it is possible to take this transition. Therefore this clock should not be abstracted in the next iteration of the abstraction refinement loop.

If the dead end transition belongs to one of the two transitions introduced for a τ transition in the abstract counterexample, then we know that the progress of the test automaton stops because of a τ transition in the full system. The reason for this is that the clock guards of the two introduced transitions are satisfied. So the only reason why this may block is that the guard that checks the valuation of the variables is not satisfied. But this is only possible if there is no enabled τ transition in the full system whose assignments would make the guard true. As there is such a transition in the abstraction, we again know that this must be because of an unsatisfied transition guard. So the variables that occur in the unsatisfied constraints of this guard should not be abstracted in the next iteration.

The last possible reason why the dead end transition is blocked is that there is no enabled transition with an assignment that would make the guard of the dead end transition true. From the construction of the test automaton this can only be a synchronized transition in the full system because a τ transition in the full system always has to synchronize with the test automaton which is not possible. As we know that there is a synchronized transition in the abstract system that makes the guard of the dead end transition true, this transition would do it also in the full system. The reason why the progress of the test automaton is stopped

is that either the guard of this synchronized transition or a location invariant of the successor state is not satisfied.

9.4 Evaluation

To demonstrate the potential of our approach we chose the Single-tracked Line Segment (SLS). It is the specification of a control system for a single-tracked line segment for tramways, and is implemented by distributed PLC automata [31]. We took three different models of the SLS case study as examples. As the safety property to verify, we chose the mutual exclusion of drive permissions, i. e., the control system never gives permission to both directions simultaneously. All the reported results were obtained on an AMD Opteron system. We set the memory limit to 4 GByte.

The first model (S_1) we checked is a manipulated system that we obtained by changing a delay time but with the assumption that everything is implemented on only one hardware device. The full timed automata model we got from MOBY/RT had 9 processes, 2 clocks and 24 integer variables. Table 9.1 shows in the first row the resources needed to check the full model of S_1 using the standard UPPAAL verification engine. It took 305 seconds to verify that the manipulated delay time does not lead to an error if the system is implemented on one device. In the following rows the steps of the abstraction refinement loop are given. Each step consists of a verification run to find an abstract counterexample (left columns) and the check for spuriousness (right columns). For these runs we use UPPAAL/DMC's greedy search with the h^L heuristic (see Chap. 5 for details). The use of directed model checking makes sense here, because it can be expected that the current abstraction contains (abstract) counterexamples and directed model checking detects counterexamples faster. Note that whenever a spurious counterexample is found a refined abstraction is derived. This refined abstraction considers more entities (at least one clock or one integer variable more than before).

For S_1 it turns out that with our counterexample guided abstraction refinement we can prove correctness of the model using an abstraction with 1 clock, 4 processes and 14 integer variables less than the full model. The accumulated number of explored states, i. e., the sum over both state columns of the S_1 example, is 986878. This is only about 3% of the states that are explored by UPPAAL to produce the result for the full model. Also the summarized time consumption of our CEGAR approach is of the same magnitude, compared with UPPAAL's runtime for the full model. Note that in abstraction 3, the number of processes has changed. The reason for this is that whenever a variable is added to the next abstraction that is triggered by the environment, then an additional automaton

is added to the system that drives this variable. These driver automata are automatically generated by MOBY/RT.

Table 9.1. Abstraction refinement results for the experiments. Abbreviations: *#c*: number of clocks, *#p*: number of processes, *#v*: number of integer variables, *time*: runtime in seconds, *states*: number of explored states, *trace*: length of found error trace, *CE*: counterexample, dashes indicate out of memory (more than 4 GByte) or, in the trace column, absence of error states

	abstract search						spuriousness check					
Exp.	#c	#p	#v	states	time	trace	#c	#p	#v	states	time	result
S_1: full							2	9	24	35918002	305.2	verified
Abstr. #1	0	4	3	47	0.0	19	3	10	24	3403	0.0	spurious CE
Abstr. #2	1	4	3	47	0.0	20	3	10	24	460243	4.3	spurious CE
Abstr. #3	1	5	5	58	0.0	21	3	10	24	51841	0.8	spurious CE
Abstr. #4	1	5	6	44	0.0	21	3	10	24	166725	1.8	spurious CE
Abstr. #5	1	5	8	160	0.0	31	3	10	24	287819	3.0	spurious CE
Abstr. #6	1	5	10	16491	0.7	–						verified
S_2: full							3	10	25	–	–	–
Abstr. #1	0	5	3	53	0.0	27	4	11	25	7615	0.0	spurious CE
Abstr. #2	1	5	3	53	0.0	27	4	11	25	7615	0.0	spurious CE
Abstr. #3	2	5	3	53	0.0	27	4	11	25	10424455	124.8	spurious CE
Abstr. #4	2	5	6	2477	0.1	76	4	11	25	4346623	50.8	spurious CE
Abstr. #5	2	6	8	11091	0.4	59	4	11	25	786511	9.3	spurious CE
Abstr. #6	2	6	9	992	0.0	40	4	11	25	1540343	17.0	spurious CE
Abstr. #7	2	6	11	4310	0.2	56	4	11	25	1360127	20.4	spurious CE
Abstr. #8	3	6	11	5016	0.2	68	4	11	25	2946704	31.9	disproved
S_3: full							3	10	25	–	–	–
Abstr. #1	0	5	3	53	0.0	27	4	11	25	7615	0.0	spurious CE
Abstr. #2	1	5	3	53	0.0	27	4	11	25	7615	0.0	spurious CE
Abstr. #3	2	5	3	53	0.0	27	4	11	25	10424455	124.8	spurious CE
Abstr. #4	2	5	6	2477	0.1	76	4	11	25	4346623	50.8	spurious CE
Abstr. #5	2	6	8	11091	0.4	59	4	11	25	786511	9.3	spurious CE
Abstr. #6	2	6	9	992	0.0	40	4	11	25	1540343	17.0	spurious CE
Abstr. #7	2	6	11	4310	0.2	56	4	11	25	1360127	20.4	spurious CE
Abstr. #8	3	6	11	5016	0.2	–						verified

For the next verification problem we removed the assumption about the partitioning of the PLC automata onto hardware devices. The second experiment (S_2) represents a distributed system. Now, the manipulated delay time leads to an incorrect system. However, it was not possible to find a counterexample with UPPAAL's randomized depth-first search in the full model within the given memory limit of 4 GByte. This time the abstraction refinement loop had to iterate 8 times to generate an abstraction for which a definite answer was found,

i. e., a counterexample in the full model. The computed abstraction saved 4 processes and 14 integer variables.

In our final experiment (S_3) we reverted to the original delay time. Again, it was not possible to check the full model within the memory limits. The abstraction refinement loop generated the same sequence of refined abstractions and terminated after 8 iterations again. But this time the final abstraction has no counterexample.

9.5 Discussion

All the experiments from the last section show that the abstraction refinement approach presented in this chapter is able to generate abstractions effectively for which definite verification results can be found. The main benefits are that there is no need for human interaction at all, an abstraction of the model is computed automatically for which a reliable verification result can be computed and that this approach reduces significantly the resources (runtime and number of explored states). However, there is no guarantee that this approach computes a *minimal* abstraction but it is obvious that it will terminate since each iteration adds at least one of the finitely many entities of the model.

One interesting question that pops up when reading the result Table 9.1 is: How does our abstraction refinement loop perform compared to directed model checking methods? Table 9.2 is to answer this question.

The first column of the table gives the used heuristic with the search method given in the row above. The heuristics for which we report results are the following. The $h^{\mathcal{A}}$ heuristic is the PDB heuristic based on the Russian Doll approach from Chap. 7. The heuristics $h^{\mathcal{P}}_{syn}$ and $h^{\mathcal{P}}_{AR}$ are two PDB heuristics, based on predicate abstraction (see Chap. 6). The splitting bound b for the $h^{\mathcal{P}}_{syn}$ heuristic was set to 2. For the $h^{\mathcal{P}}_{AR}$ heuristic, we set the splitting bound b to 1 and the number of refinement steps r to 4. The h^{aa} is a heuristic proposed by Dräger et al. [40]. A brief description of their method was presented in Sec. 4.3.2. Recall that their heuristic is parametrized, we set the parameter N to 100, which works good on these benchmarks. Further, we report results for the two heuristics h^L and h^U based on the monotonicity abstraction (see Chap. 5 for details), and the two graph distance-based heuristics d^L and d^U proposed by Edelkamp et al. [42].

The results for all the pattern database heuristics as well as the results for the h^{aa} heuristic were obtained with UPPAAL/DMC. All other results were obtained with MCTA.

It turns out that all direct model checking methods can prove the S_1 example error-free, with respect to the given target formula. Note that this is not

Table 9.2. Abstraction refinement compared to directed model checking. Abbreviations: *AR*: our abstraction refinement approach, *states*: number of explored states, *time*: runtime in seconds, *trace*: length of the detected error path. Dashes either indicate out of memory, i. e., more than 4 GByte or that there is no error path.

	S_1			S_2			S_3		
	states	time	trace	states	time	trace	states	time	trace
AR	986878	10.6	–	21444038	255.1	153	18497334	223.2	–
UPPAAL/DMC's greedy search									
$h^{\mathcal{A}}$	0	0.1	–	2283214	15.0	108	0	0.8	–
$h^{\mathcal{P}}_{syn}$	11320444	155.1	–	43712781	428.5	895	–	–	–
$h^{\mathcal{P}}_{AR}$	18593678	121.8	–	26422889	211.5	3225	–	–	–
h^{aa}	15061671	329.3	–	–	–	–	–	–	–
UPPAAL/DMC's A^* search									
$h^{\mathcal{A}}$	0	0.1	–	135259	1.6	57	0	0.8	–
$h^{\mathcal{P}}_{syn}$	12791886	163.8	–	–	–	–	–	–	–
$h^{\mathcal{P}}_{AR}$	26690320	201.3	–	–	–	–	–	–	–
h^{aa}	21822058	401.2	–	–	–	–	–	–	–
MCTA's greedy search									
h^L	4519322	191.4	–	4641286	170.6	995	–	–	–
h^U	4639011	298.2	–	7889320	523.6	280	–	–	–
d^L	7003688	53.2	–	–	–	–	–	–	–
d^U	6703513	51.1	–	–	–	–	–	–	–
MCTA's A^* search									
h^L	6437232	209.6	–	–	–	–	–	–	–
h^U	6448250	323.9	–	–	–	–	–	–	–
d^L	9431992	77.2	–	–	–	–	–	–	–
d^U	9424486	78.8	–	–	–	–	–	–	–
MCTA's useless transitions									
h^L	4559825	267.8	–	1543	0.3	122	–	–	–
h^U	4540415	405.3	–	399	0.2	105	–	–	–
d^L	6784648	102.3	–	–	–	–	–	–	–
d^U	6784981	101.4	–	–	–	–	–	–	–

surprising, because even UPPAAL's breadth-first search can solve this problem. This means that the state space of this example is not too big, given a memory limit of 4 GByte. However, all methods, except the Russian Doll heuristic $h^{\mathcal{A}}$, need more time to solve this problem than the abstraction refinement approach. Again, this is not surprising, since the only benefit of heuristic search in the absence of reachable error states is that the heuristic can be used for pruning (recall that all heuristics, except the d^L and d^U heuristics, can detect dead ends).

More precisely, if a heuristic estimate for a state is infinity, this state can safely be excluded from the search. The $h^{\mathcal{A}}$ heuristic does not expand any states. The reason for this is that in the underlying abstraction, which is used as the pattern database, no abstract error state is reachable. Hence, after building the PDB, the search is over.

It is interesting to see that, in the absence of reachable error states, all the heuristics explore a different number of states, depending on whether A^* or greedy search is used. At a first glance this seems to be wrong, and in fact it would be wrong for discrete state systems. However, since we are dealing with the *symbolic* state space of timed automata, this is possible for the following reasons. Recall that a symbolic state consists of a discrete part and a continuous part, i. e., the zone of the state. Suppose there are several states $s_1, \ldots, s_n$ with the same discrete part but different zones. Say Z_{s_i} is the zone of s_i and $Z_{s_i} \subseteq Z_{s_j}$ iff $i \leq j$. If the states are explored in increasing order of their indices, then all the states $s_1, \ldots, s_n$ are added to the open queue, and thus explored. The reason for this is that when the algorithm checks if s_i is in the open or the closed queue, then there is no such state, because the zone of s_i is not subsumed by any previously visited state. If the states are expanded in decreasing order, then only s_n is explored, because every state s_i, with $i < n$ is subsumed by s_n and thus not inserted into the open queue.

Let us have a closer look at the results for the S_2 example. Here, due to the size of the state space, A^* was only able to solve this problem with the $h^{\mathcal{A}}$ heuristic. But again, the runtime is orders of magnitude shorter than the runtime of our CEGAR loop. The runtime results for greedy search obtained with $h^{\mathcal{A}}$, h^L and $h^{\mathcal{P}}_{AR}$ are shorter or at least of the same order of magnitude. The runtime of $h^{\mathcal{A}}$ is one order of magnitude shorter than the runtime of our CEGAR approach. The results obtained with the useless transitions approach with h^L and h^U are even 2 orders of magnitude shorter. In our opinion it is not surprising that the directed model checking methods perform well on S_2. The reason for this is that in this example there is a reachable error state, and directed model checking is especially tailored to the fast detection of such states.

In the last example, only our abstraction refinement loop and the methods that use the $h^{\mathcal{A}}$ heuristic can prove this example error free. Here, for the same reason as for the S_1 example, the search with our Russian doll approach terminates after the construction of the PDB, because there is no reachable abstract error state. All the other heuristics are not able to solve this problem. The reason for this is that the state space of this example is too big, and too few states can be pruned.

9.6 Conclusion

In this chapter, we first presented an approach for counterexample-guided abstraction refinement for a subclass of timed automata. Afterwards, we empirically compared this approach with some directed model checking techniques. For the CEGAR approach, we defined how to construct test automata that can be used to check whether a full model is able to behave as the abstract trace, i. e., to check whether an abstract trace is spurious or not. Moreover, we extended the model checker UPPAAL in such a way that it executes an analysis of *why* a full model cannot execute a spurious trace. The result of this analysis is used to refine the given abstraction in a way that the spurious counterexample cannot occur anymore. This approach allowed us to construct a closed abstraction refinement loop. We want to add that our abstraction refinement approach was developed for PLC automata, but the methodology is generally applicable to the full range of timed automata based models expressible within UPPAAL.

The empirical comparison of the CEGAR approach and the directed model checking techniques revealed the following. Especially in the presence of reachable error states, directed model checking is often superior to our CEGAR approach. Our best-performing directed model checking method, namely the Russian doll approach from Chap. 7, even outperformed the CEGAR approach in the absence of reachable error states.

10

Discussion

The general lesson that can be learned from this thesis is that directed model checking is a powerful means to detect subtle errors in large systems of timed automata. In this thesis, we proposed several directed model checking techniques with which the state explosion problem of real-time systems can be efficiently alleviated. With the techniques from this thesis it is now possible to detect reachable error states in huge practically relevant systems that are beyond the scope of previously proposed directed model checking methods and blind search.

10.1 Conclusion

At the point when we started to work on directed model checking for timed automata, detecting reachable error states in large real-time systems was hardly possible. The prevailing methods that existed at that time could only cope with systems of moderate size. The only way to analyze larger systems was to apply abstractions or hand-tailored heuristics as implemented in UPPAAL CORA (see Sec. 4.3.2). However, both approaches require a deep understanding of the system under consideration, maybe even an intuition where possible errors lurk. When model checking is used in an abstraction refinement loop or for debugging purposes, it is desirable to obtain short error traces. Short error traces are easier to understand and provide useful information about which particular behavior of the system is responsible for reaching an error state. However detecting provable shortest error traces with hand-crafted heuristics or abstractions is a time-consuming and error-prone process.

With the directed model checking techniques presented in the thesis at hand, the situation has changed a lot. We proposed several powerful heuristics and search enhancements for model checking timed automata. All of them are fully automatically generated, based on the declarative description of the system un-

der consideration. As a consequence, directed model checking, as proposed in this thesis, is a push-button technology that needs no additional user input. The developed techniques allow to cope with systems of non-trivial size that previously could hardly be analyzed automatically. In the following, we will briefly highlight the main contributions of this thesis.

The two heuristics h^L and h^U based on the monotonicity abstraction, were the first heuristics that we have developed that allowed us to detect reachable error states in the largest instances of our benchmark set. They clearly outperformed the previously proposed heuristics by Edelkamp et al. [42] and blind search as implemented in UPPAAL. However, the detection of provable shortest error traces with A^* and h^L was only possible for two more instances of our benchmark set.

We used the monotonicity abstraction not only to generate the both heuristics h^L and h^U, but it is also part of a novel strategy for the generation of a highly efficient pattern database heuristic. By using the monotonicity abstraction to obtain abstraction sets on which a pattern database heuristic can be built, we get a fairly targeted notion of what is relevant to the target formula under consideration. The resulting pattern database heuristic allows us to detect shortest possible error traces in the largest of our benchmarks in a matter of seconds.

Our approach to avoid useless transitions is a powerful means to further alleviate the state explosion problem. It can be seen as a special form of heuristic search, where not only just states are heuristically evaluated, but also state transitions. The framework can be applied to a broad range of heuristics, namely those heuristics that are computed on-the-fly. A nice property of our approach is that the same heuristic that is used to evaluate states can be used to evaluate transitions, without changing its implementation. Compared to A^* and greedy search, this approach scales much better, and, at the same time, yields rather short error traces.

Last but not least, we also want to add that all techniques developed in this thesis are implemented in at least one of our model checkers UPPAAL/DMC and MCTA. So far these are the only available tools for directed model checking of timed automata. Especially MCTA has proved to be a very powerful directed model checking tool, i. e., MCTA outperforms standard UPPAAL not only in terms of runtime but also in terms of explored states and error trace length. We implemented MCTA from scratch and released it under the terms of the GPL and in the hope that it also serves as a platform to develop and implement additional directed model checking techniques for timed automata.

10.2 Future Work

This is the last section in this chapter, and thus also in this thesis. As a final remark, we want to point our vision on how to deal with larger and larger systems. We believe that the state explosion problem of huge systems has to be tackled in two ways. First, one has to exploit the underlying structure of the systems under consideration *more* explicitly. For example, the heuristics and search methods so far do not explicitly take into account structural properties like communication or causal dependencies of variables or transitions. Second, we need a tight integration of directed model checking and other model checking techniques like compositional model checking or abstraction refinement. For instance, when doing abstraction refinement, it seems promising to us to exploit the fact that one is repeatedly confronted with several abstractions of the same system. We believe that addressing these points will not only lead to new potential heuristics, but also to novel strategies for the state space traversal.

A

Case Studies and Benchmarks

A.1 The Fischer Protocol

The well-known Fischer Protocol is a mutual-exclusion protocol, first proposed by Fischer [46]. For a more detailed description of the protocol we refer the interested reader to Lamport's article [73]. The system consists of n timed automata (identical up to renaming), plus a shared integer variable id ranging from 0 to n. Each automaton A_i behaves as follows: after remaining idle for some time, it checks whether the common resource is free (test $id = 0$) and if so, before k time units sets id to i. Then it waits for some time and, making sure that id is still equal to i, enters the critical section. If id is not equal to i (meaning that some other automaton has requested access) then automaton A_i has to retry later.

A.1.1 The Models

We use three variants of the Fischer protocol with 5, 10 and 15 automata. The test examples are denoted F_i^A and F_i^B, respectively, and i is the number of parallel automata. The error condition is that at least two of the automata are simultaneously in the location labeled with *cs* (see Figure A.1). We injected this error by weakening the temporal conditions in the automata (from $<$ to $\leq$). The variants differ in the way they encode the error condition.

- Variant A adds an additional automaton with synchronization.
- Variant B selects and specifies two of the automata for the error condition.

Table A.1 provides the number of components of the model.

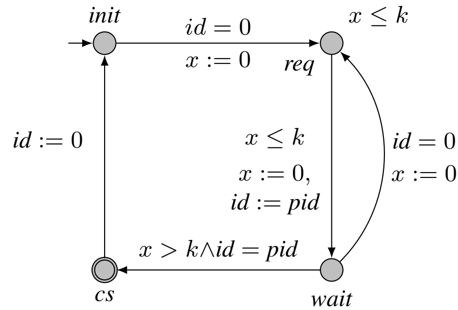


Fig. A.1. Timed automata model for the Fischer protocol

Table A.1. Number of system components. Abbreviations: *Exp.*: the model's name, *aut.*: the number of automata in the model, *clock*: the number of clocks in the model, *vars*: the number of integer variables, *sync*: number of synchronization labels

Exp.	aut.	clocks	vars	sync
F_5^A	6	5	1	2
F_{10}^A	11	10	1	2
F_{15}^A	16	15	1	2
F_5^B	5	5	1	0
F_{10}^B	10	10	1	0
F_{15}^B	15	15	1	0
F_5^C	5	5	2	0
F_{10}^C	10	10	2	0
F_{15}^C	15	15	2	0

A.2 The Single-tracked Line Segment Case Study

The Single-tracked Line Segment case study (SLS) stems from an industrial project partner of the UniForM-project [68]. The problem is to design a distributed real-time controller for a segment of tracks where trams share a piece of track. Figure A.2 sketches the system architecture.

The railway lines are marked by thick lines and share a short part. The short arrows describe the directions of trains. The movements of trains are detected by six sensors, three for each direction. The arrival of trains is recognized by entrance sensors ES1 and ES2. The sensors CS1 and CS2 check whether a train enters the critical section, while LS1 and LS2 recognize leaving trains. The information delivered by the sensors are processed by two computing devices called PLC 1 and PLC 2. They notice the movements of trains and compute which direction is allowed to use the critical section.

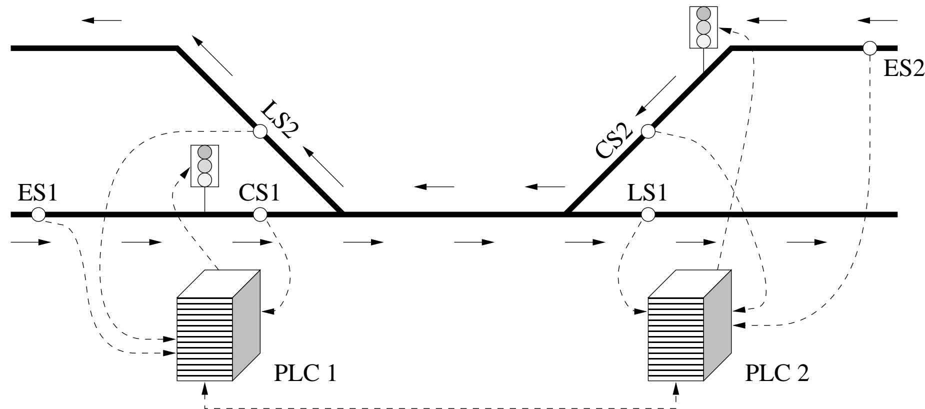


Fig. A.2. The SLS architecture

A.2.1 The Models

A distributed controller was modeled in terms of PLC automata [33], which is an automata-like notation for real-time programs. According to their tasks, we can distinguish five different automata in the system:

- The signals of the sensors are filtered by six components. The purpose of these filters is to compensate some inherent unreliabilities of the sensor hardware.
- Four counters accumulate the information about passing trains produced by the filters. For each zone of interest we need a counter to determine the number of trains in this zone. For each direction there is a *waiting zone*, i. e., a zone between the entrance sensor and the critical sensor, and there is a *critical zone*, i. e., a zone between the critical sensor and the leaving sensor. If a counter recognizes that the number of trains in its corresponding zone leaves the plausible range, which is $[0, 2]$ for the SLS, an error signal is raised.
- One component summarizes the error signals of all filters and counters.
- The permissions which direction is allowed to enter the critical section are computed in the main controller. The decision depends on the current values of the counter and the current state of the main controller.
- Two automata produce the signal for the traffic lights for each direction. To this end they need the information whether there is a train in its waiting zone, whether its direction has got the permission to enter the critical section, and whether there is an error in the system or not.

The system diagram of the controller for the SLS is given in Fig. A.3. The figure shows 14 system nodes, which are represented by icons. For example,

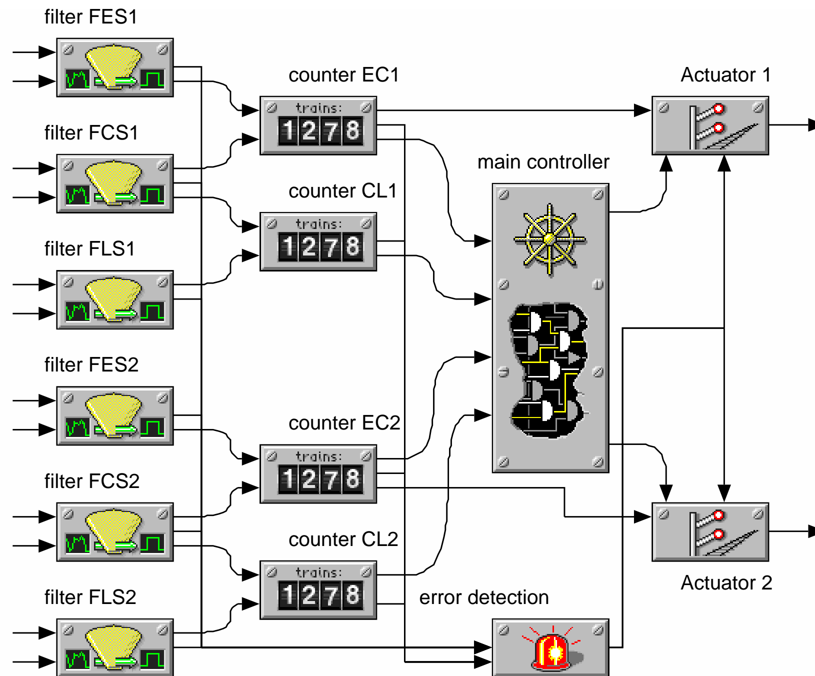


Fig. A.3. System diagram of the SLS controller

on the left side of the diagram, there are six filter components, which are responsible for the transformation of the sensor outputs into reliable values. The resulting signals, which are delivered to the traffic lights on the railway, are generated by the two actuator components on the right of the diagram.

This system of PLC automata can be transformed into (abstractions of) their semantics in terms of timed automata with the tool MOBY/RT [80]. For the evaluation of our approach we choose the property that never both directions are given permission to enter the shared segment simultaneously. This property is ensured by 3 PLC automata (main controller, Actuator 1, Actuator 2) of the whole controller, and we injected an error by manipulating a delay so that the asynchronous communication between these automata is faulty. In MOBY/RT abstractions are offered for the translation into the timed automata. The given set of PLC automata has eight input variables. We constructed nine models with increasing size by decreasing the number of abstracted inputs. Table A.2.1 provides the number of components of the model.

Table A.2. Number of system components. Abbreviations: *Exp.*: the model's name, *aut.*: the number of automata in the model, *clock*: the number of clocks in the model, *vars*: the number of integer variables, *sync*: number of synchronization labels

Exp.	aut.	clocks	vars	sync
C_1	5	3	12	3
C_2	6	3	14	3
C_3	6	3	15	3
C_4	7	3	17	3
C_5	8	3	19	3
C_6	9	3	21	3
C_7	10	3	23	3
C_8	10	3	24	3
C_9	10	3	25	3

A.3 The Mutual Exclusion Case Study

The Mutual Exclusion case study models a real-time protocol to ensure mutual exclusion in a distributed system via asynchronous communication. The protocol is described in full detail in Dierk's article [32]. The case study consists of three automata. Two of them, S_1 and S_2, have exactly two locations, one of those is considered to be safe and the other one is considered to be unsafe in some sense. They are given in Fig. A.4.

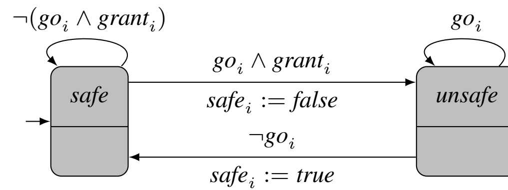


Fig. A.4. The automaton S_i, the loop transitions are not necessary but added for clarity.

These automata are driven by two Boolean input variables go_i and $grant_i$. The first one represents the desire to enter the unsafe state. Since go_i it is an input, it is not known beforehand when the transition to the unsafe state is required by the environment. Moreover, in case of the unsafe state the go_i input controls how long the unsafe state is kept. The second input, $grant_i$, is assumed to be triggered by an output of a controller. Hence, the automaton S_i can only enter the unsafe state if the grant is given. The Boolean output variable $safe_i$ can be read by both the controller and the environment. The controller needs this information for changing grants and the environment needs it to do, for example, something that requires exclusive access to some resource.

A.3.1 The Models

A first naive variant of a controller modeled as PLC automata [31] is given in Fig. A.5. The idea to achieve mutual exclusion of the unsafe locations in S_1 and S_2 is to give grant to S_i not before S_{3-i} has set $safe_{3-i}$ to *true*. This is done in the locations "wait for S_i". As soon as the controller detects that S_i has entered the unsafe locations it retracts the grant for S_i by the edge from "grant for S_i" to "wait for S_i". This naive controller does not try to be fair since it may happen that S_2 waits forever for a grant while the controller is in location "grant for S_1" but S_1 never enters the unsafe location. Even worse, if all three automata are implemented on different devices it is not very difficult to construct a trace that violates the main requirement of mutual exclusion.

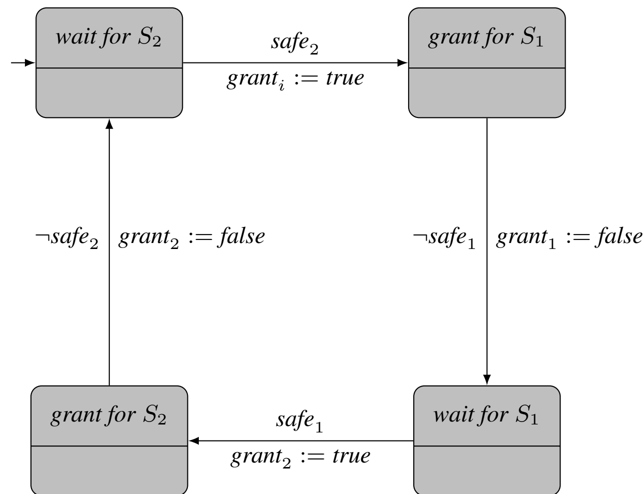


Fig. A.5. A model of faulty controller

The failure is due to the fact that the cyclic behavior of the automata involved is not synchronized. Hence, S_i may get a grant, and become unsafe afterwards. This is *polled* by the controller and it will retract the grant at the end of its cycle. But before this happens S_i can become safe again and poll its grant value which is still set to true. Now the controller finishes its cycle, retracts the grant and poll the status of S_i which safe. Therefore, it permits S_{3-i} to enter the unsafe location. Since S_i has polled a given grant it will become unsafe, too.

The fault is not that S_i can switch to the safe location and poll a grant twice. Since the S_i are given it is the controller's task to cope with this behavior. The core of the fault is that the controller believes that polling $safe_i$ to be *true* is sufficient to give grant to the other automaton. It has to consider that S_i has polled a given grant and will become unsafe. To avoid this problem the controller should

wait a period of time in order to be sure that S_i cannot change to the unsafe location anymore. The time distance between polling a grant and switching to the unsafe state is less than ε_i. The time distance between polling S_i to be safe and giving grant to S_{3-i} is less than ε_c. Since it is necessary for the counterexample that these periods overlap it will be sufficient to wait in state "wait for S_i" for $\varepsilon_i + \varepsilon_c$ time units. With this modification the mutual exclusion property is valid. Figure A.6 contains a correct controller.

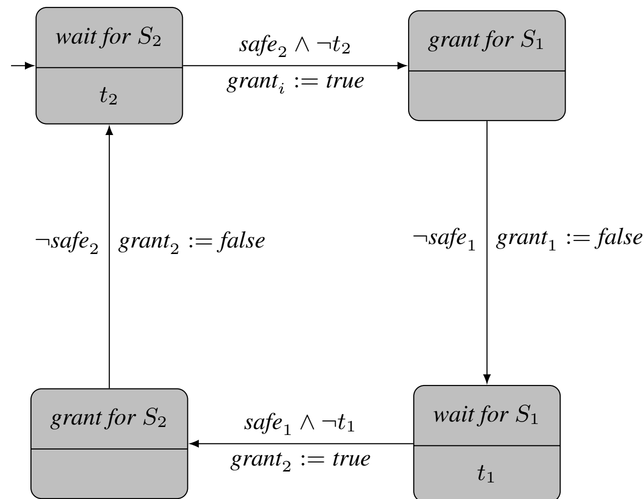


Fig. A.6. A model of a correct controller

The models which serve as benchmarks are flawed versions of this controller. We introduced an error by increasing an upper time bound in the model. The flawed specification was then transformed into its timed automata semantics by applying various abstractions techniques. The resulting models do not have many automata but a non-trivial number of clocks and variables. Table A.3 provides the number of components of the models.

A.4 The Arbiter Tree Case Study

This case study models a mutual exclusion protocol based on a tree of binary arbiter processes. Figure A.7 shows an instance with four clients. Client processes are situated at the leaves of the tree. In order to gain access to the shared resource, they may send a request to their respective parent, which in turn passes the request on to its parent, and so on. When the root process of the tree receives a request, it generates a grant which is then propagated back down. When the client is done with the resource, it sends a release signal.

Table A.3. Number of system components. Abbreviations: *Exp.*: the model's name, *aut.*: the number of automata in the model, *clock*: the number of clocks in the model, *vars*: the number of integer variables, *sync*: number of synchronization labels

Exp.	aut.	clocks	vars	sync
M_1	3	4	11	0
M_2	4	4	13	0
M_3	4	4	13	0
M_4	5	4	15	0
N_1	3	7	11	0
N_2	4	7	13	0
N_3	4	7	13	0
N_4	5	7	15	0

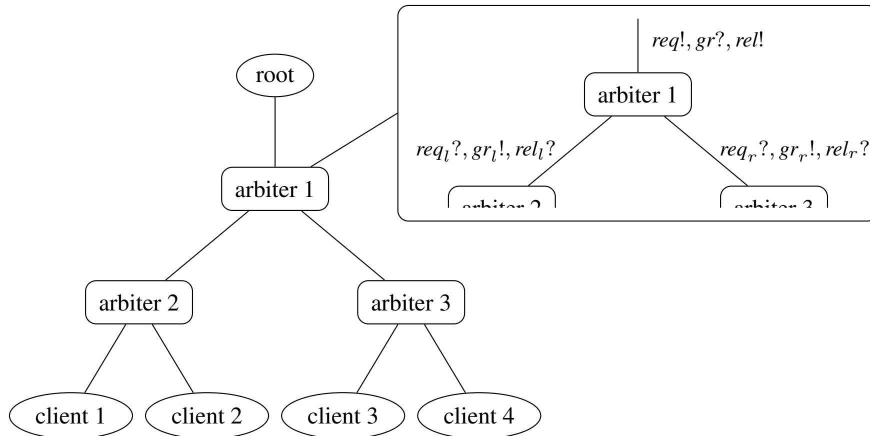


Fig. A.7. Arbiter tree: Access to a shared resource is controlled by binary arbiters arranged in a tree, with a central root process.

In order to avoid blocking client processes, arbiters need to be ready to receive requests from one of their children even when they are already processing one for the other child. In this case it makes sense to send a grant to the second child as soon as a release is received from the first, instead of first forwarding the release upwards and sending a new request. This can be applied asymmetrically (giving a grant to the left child first, and always forwarding releases from the right child to the parent) in order to not monopolize the resource.

A.4.1 The Models

We model the arbiter tree using finite automata for the root, arbiter, and client processes. The instances are parametrized by the height H of the tree, and contain $2^H - 1$ arbiters and 2^H client processes. The smallest example ($H = 2$)

has $1.02 \cdot 10^6$ product states; this increases to $1.88 \cdot 10^{104}$ states in the largest instance ($H = 6$).

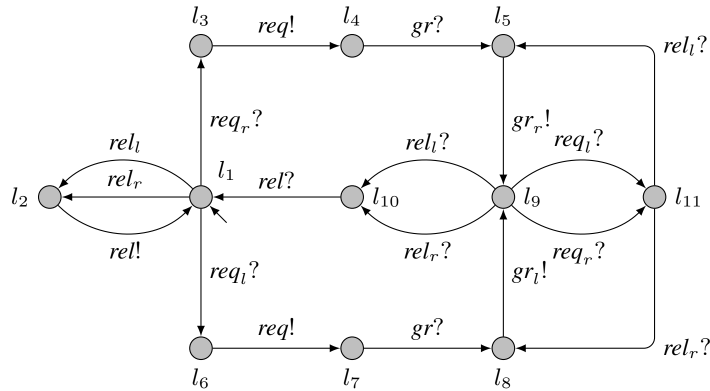


Fig. A.8. A faulty arbiter automaton

The case study uses an incorrect implementation which eventually allows several client processes to access the resource simultaneously. This situation results from a faulty client process sending spurious release signals, and a flaw in the arbiters which makes them not check for this possibility and discard such a signal. One such faulty arbiter is shown in Fig. A.8. Table A.4 provides the number of components of the model.

Table A.4. Number of system components. Abbreviations: *Exp.*: the model's name, *aut.*: the number of automata in the model, *clock*: the number of clocks in the model, *vars*: the number of integer variables, *sync*: number of synchronization labels

Exp.	aut.	clocks	vars	sync
A_2	8	0	0	21
A_3	16	0	0	45
A_4	32	0	0	93
A_5	64	0	0	189
A_6	128	0	0	381

B

Determining Good Parameters for $h^{\mathcal{P}}_{syn}$ and $h^{\mathcal{P}}_{AR}$

In this chapter we show some experimental results which we used to determine good parameters for the two heuristics $h^{\mathcal{P}}_{syn}$ and $h^{\mathcal{P}}_{AR}$, based on predicate abstraction from Chap. 6. All results are obtained on an AMD Opteron system with 2.3 GHz and 4 GByte of memory.

B.1 Experimental Results for the Syntax-Based Abstractions

Table B.1 shows our results that we obtained when using the syntactical approach to select a set of predicates. In the table, the split bound $b \in \{1, 2, 3, 4\}$ increases from left to right. The split bound is given as the top entry of each column. Note that we do not provide the results where the entire automata system is handed to the abstraction engine in total. The reason for this is that when using the whole system, the abstract state space could only be built for the Fischer examples and the smaller M and N examples. Consider the M and N examples and what happens as b increases from 1 to 4. The preprocessing time increases sharply, quickly becoming larger than the time spent for the actual search. Strangely, the number of explored states also grows, from $b = 1$ to $b = 2$, before decreasing again from $b = 2$ to $b = 4$. It is unclear to us what causes this behavior. The smallest search spaces are obtained with $b = 4$, the smallest overall runtimes are obtained with $b = 1$. Note that all M and N examples can be solved quite quickly even with a blind search (cf. Table 6.1). The C examples are more interesting in that respect, because blind search scales badly there. They exhibit very similar behavior in terms of the overhead. The number of explored states decreases sharply from $b = 1$ to $b = 2$ and increases sharply from $b = 2$ to $b = 3$. Again, the reason for this behavior is unclear.

All in all, the syntax-based abstractions give surprisingly good performance, e. g. $h^{\mathcal{P}}_{syn}$ with $b = 2$ is very competitive in the C examples, but they do not seem to have much potential for further improvements. One could try to allow

Table B.1. Results for greedy search with $h^{\mathcal{P}}_{syn}$. The first row gives the split bound, i. e., the maximum number of automata per PDB. Dashes indicate out of memory ($>$ 4 GByte).

	explored states				runtime in s				trace length			
	1	2	3	4	1	2	3	4	1	2	3	4
F^A_5	80	80	80	80	0.0	0.0	0.0	0.0	20	20	20	20
F^A_{10}	130	130	130	130	0.0	0.1	0.2	0.0	20	20	20	20
F^A_{15}	180	180	180	180	0.0	0.3	0.5	0.0	20	20	20	20
F^B_5	21	21	21	21	0.0	0.1	0.1	0.0	6	6	6	6
F^B_{10}	36	36	36	36	0.2	0.3	0.2	0.0	6	6	6	6
F^B_{15}	51	51	51	51	0.4	0.4	0.5	0.0	6	6	6	6
C_1	1455	1588	4698	6207	0.3	1.2	8.6	46.3	119	158	95	78
C_2	3273	3786	10843	10507	0.5	1.3	9.4	1.6	137	180	99	95
C_3	5879	3846	10375	10195	0.4	1.4	9.3	1.6	135	186	99	95
C_4	44837	30741	66336	66761	0.9	1.9	20.2	4.3	163	240	73	68
C_5	301065	185730	436678	435309	3.2	3.5	26.5	9.0	241	422	121	101
C_6	2.8e+6	1.9e+6	4.2e+6	3.9e+6	21.8	16.6	78.5	54.4	409	756	134	166
C_7	3.5e+7	1.8e+7	4.1e+7	4.0e+7	386.1	158.9	567.0	575.9	844	1063	249	214
C_8	3.2e+7	1.5e+7	2.5e+7	2.5e+7	329.4	120.2	336.2	353.7	1006	975	1203	581
C_9	–	–	–	–	–	–	–	–	–	–	–	–
M_1	16446	23257	12780	12780	0.3	0.7	3.1	0.4	219	100	73	73
M_2	68956	84475	37780	34947	0.8	1.5	4.5	1.1	137	127	108	82
M_3	62371	92548	55726	55098	0.7	1.5	4.7	1.2	147	97	110	100
M_4	275433	311049	198407	139875	2.5	3.8	7.5	2.3	215	135	118	115
N_1	22025	31593	12327	12327	0.8	1.9	5.8	0.6	197	127	81	81
N_2	122382	172531	80276	58766	3.7	6.6	10.2	3.0	206	157	154	122
N_3	140201	167350	76116	70677	4.1	6.2	9.3	2.5	213	129	122	120
N_4	739268	975816	657486	390297	19.7	33.6	28.9	12.9	280	212	263	277
A_2	44	46	32	32	0.0	0.0	0.0	0.0	13	12	12	12
A_3	710	187	863	116	0.0	0.0	0.0	0.1	39	21	20	20
A_4	73628	10633	2283	1.8e+6	0.4	0.1	0.0	16.5	129	78	33	32
A_5	–	–	1.2e+6	4.1e+6	–	–	14.5	52.2	–	–	212	53
A_6	–	–	–	–	–	–	–	–	–	–	–	–

more freedom in the selection of the predicates, but such an approach is likely to be wild guesswork, at least without a deeper analysis of the system. An idea worth trying is to integrate syntax-based predicate selection into abstraction refinement: as a start, one could select, amongst others, the guards that are not satisfied by the spurious error path.

Table B.2. Results for A^* search with $h^{\mathcal{P}}_{syn}$. The first row gives the split bound b, i. e., the maximum number of automata per PDB. Dashes indicate out of memory (> 4 GByte).

	explored states				runtime in s				trace
	1	2	3	4	1	2	3	4	length
F_5^A	1457	1457	1457	1457	0.0	0.0	0.0	0.1	8
F_{10}^A	37922	37922	37922	37922	0.4	0.5	0.7	0.8	8
F_{15}^A	348797	348797	348797	348797	5.6	6.2	6.6	7.2	8
F_5^B	21	21	21	21	0.1	0.1	0.1	0.1	6
F_{10}^B	36	36	36	36	0.1	0.1	0.3	0.3	6
F_{15}^B	51	51	51	51	0.3	0.4	0.5	0.6	6
M_1	30945	22634	21263	21263	0.5	0.8	3.3	3.3	53
M_2	122502	94602	71785	79976	1.6	1.7	4.8	7.2	54
M_3	128809	121559	95924	101201	1.6	2.1	5.2	7.4	56
M_4	563807	466967	342619	380434	6.9	6.5	9.6	13.4	57
N_1	57941	46966	42372	42372	3.1	3.2	7.4	7.4	60
N_2	253911	211935	178540	189536	12.6	10.9	15.9	20.0	61
N_3	287915	233609	240327	228807	14.6	11.8	18.5	21.3	63
N_4	1094412	1036002	954499	1058000	52.4	50.7	56.5	68.3	64
C_1	17127	7088	4956	7048	0.5	1.3	8.9	635.8	54
C_2	45739	15742	10875	10844	0.6	1.3	9.4	15.8	54
C_3	51905	15586	10639	10644	0.8	1.5	9.5	15.7	54
C_4	441464	108603	74767	74456	3.8	2.6	20.4	39.9	55
C_5	3383157	733761	494501	462102	27.4	8.3	28.7	51.7	56
C_6	33954943	7360078	4903968	4717608	370.1	66.0	95.6	120.7	56
C_7	–	–	–	–	–	–	–	–	–
C_8	–	–	28147792	29398550	–	–	450.1	722.4	57
C_9	–	–	–	–	–	–	–	–	–
A_2	454	155	68	68	0.0	0.0	0.0	0.0	12
A_3	58799	20658	5763	3354	0.2	0.1	0.1	0.1	17
A_4	–	–	13163834	1422336	–	–	129.6	16.1	22
A_5	–	–	–	–	–	–	–	–	–
A_6	–	–	–	–	–	–	–	–	–

B.2 Experimental Results for the Predicates Selected via ARMC

The $h^{\mathcal{P}}_{AR}$ heuristic has two parameters: the split bound b and the maximum number of refinement iterations r in ARMC. Table B.3 restricts to the C examples – which are the most relevant examples. The data are arranged in a slightly unusual way, grouped by example rather than by configuration parameters. We chose this form to ease observing how the behavior for an example changes as a function of the configuration parameters.

Table B.3. Results for greedy search with the $h^{\mathcal{P}}_{AR}$ heuristic. The columns headed with 0, 1, 4 and 7, give the number of refinement steps r. The first column gives the used splitting bound b, i. e., the maximum number of automata per PDB. Dashes indicate out of memory (> 4 GByte).

	explored states				runtime				trace length			
	0	1	4	7	0	1	4	7	0	1	4	7
C_1												
1	19778	17330	2806	2806	0.4	0.5	0.6	0.7	793	499	207	207
2	8769	8861	1508	1508	0.7	1.5	3.3	3.4	110	217	112	112
3	8769	8861	1172	6362	1.0	2.2	7.2	24.6	110	217	74	86
4	16291	12044	3630	21050	7.2	21.9	112.6	862.5	84	167	124	178
C_2												
1	62046	59031	8143	8143	0.6	0.7	0.8	0.8	961	781	336	336
2	39710	35245	4098	4098	1.0	1.8	3.6	3.6	94	267	128	128
3	39710	35245	3256	25601	1.2	2.5	7.7	26.2	94	267	84	156
4	39710	35245	3256	25601	1.3	3.0	9.8	34.2	94	267	84	156
C_3												
1	88015	89194	10191	10191	0.8	0.9	0.8	0.8	915	745	328	328
2	67166	53616	5583	5583	1.1	2.0	3.8	3.8	118	277	136	136
3	67166	53616	4278	30407	1.4	2.7	8.1	27.0	118	277	88	168
4	67166	53616	4278	30407	1.5	3.2	10.1	35.2	118	277	88	168
C_4												
1	897900	872580	79069	79069	5.0	5.2	1.4	1.4	2304	1585	672	672
2	516282	511180	41831	41831	3.6	4.7	4.5	4.5	138	380	279	279
3	516282	511180	46837	279374	4.0	5.7	9.9	32.0	138	380	258	125
4	516282	511180	46837	279374	4.2	6.2	11.8	40.7	138	380	258	125
C_5												
1	9031659	8411983	1126261	1126261	54.4	51.5	7.8	7.8	4124	4124	1438	1438
2	4904033	6873082	425264	425264	29.7	45.0	7.2	7.2	294	1652	506	506
3	4904033	6873082	473470	2371892	30.2	46.1	13.2	46.0	294	1652	356	453
4	4904033	6873082	288614	1920988	30.4	46.9	15.8	53.5	294	1652	252	205
C_6												
1	–	–	4477523	4477523	–	–	28.9	28.7	–	–	6619	6619
2	–	–	2850769	2850769	–	–	22.7	22.9	–	–	770	770
3	–	–	2653185	24601726	–	–	28.2	211.1	–	–	800	408
4	–	–	3812970	20493668	–	–	44.9	208.2	–	–	1130	357
C_7												
1	–	–	–	–	–	–	–	–	–	–	–	–
2	–	–	20632762	20632762	–	–	174.2	167.9	–	–	2216	2216
3	–	–	20652018	–	–	–	194.2	–	–	–	2266	–
4	–	–	25656694	–	–	–	257.9	–	–	–	717	–
C_8												
1	–	–	–	–	–	–	–	–	–	–	–	–
2	–	–	25598687	25598687	–	–	204.0	203.0	–	–	2878	2878
3	–	–	32200707	–	–	–	296.9	–	–	–	1887	–
4	–	–	27966148	–	–	–	263.9	–	–	–	1183	–

Let us start with some of the simpler observations to be made in Table B.3. For $b = 1$ and $b = 2$, the table entry for $r = 7$ is always identical to the entries with $r = 4$. This is because ARMC finds feasible error paths. More precisely, with $b = 1$ ARMC finds feasible error paths, in all examples and in all parts of the partitionings, i. e., in each single automaton in 4 refinement iterations. So increasing the maximum number of refinement iterations beyond 4 does not have any effect. With $b = 2$, ARMC finds feasible error paths in 3 refinement iterations already.

Now, consider what happens as we let the configuration parameters vary. Consider first the splitting bound b when moving up or down in the table. Compared to the $h^{\mathcal{P}}_{syn}$ results, the number of explored search states is more stable. In most cases, the number of explored states stays the same, or decreases, with increasing splitting bound. Particularly with many refinement iterations, there is a relatively sharp monotonic decrease over increasing splitting bound. Notable exceptions to this rule are a few configurations for C_1, and when moving from $b = 3$ to $b = 4$ in the C_6 example. We observe that the decreased search space size never pays off in terms of runtime, i. e., when moving downwards in a column within one example, the runtime almost always increases monotonically.

B.3 Runtime Results for the Generation of PDBs

In this section we will have a closer look at the runtime needed for the preprocessing, i. e., the time that is needed in order to build a PDB. Table B.4 shows the total runtime, i. e., preprocessing time plus search time, and the preprocessing time for different configurations of our heuristics. The first row of the table gives the number of refinement steps r used for the $h^{\mathcal{P}}_{AR}$ heuristic. The first column of the table gives the splitting bound b, i. e., the maximum number of automata per PDB.

For both approaches, i. e., the syntax-based and the abstraction refinement-based approach, example C_1 is exceptionally hard for the preprocess. The reason for this is that this example only consists of 4 automata, which have a large abstract state space together. In the examples $C_2, \ldots, C_8$, one of these automata is split away. The preprocessing time for both heuristics consistently grows with growing splitting bound b. The C_6 example is the first example where the larger overhead for the $h^{\mathcal{P}}_{syn}$ heuristic pays off in runtime, i. e., the runtime results for $b = 2$ are faster than that for $b = 1$, which is due to the reduced search space (see Table B.1). In particular for the $h^{\mathcal{P}}_{AR}$ heuristic, the higher the number of refinement iterations is, the faster grows the preprocessing time with growing values of b. In terms of runtime, the number of refinement iterations definitely is the better parameter to invest extra computation time. Most notably, C_6 is

not solved with less than 4 refinement iterations. Then again, refining too much apparently is not a good idea either. First, observe that with increasing number of refinement iterations we get a consistent increase of the preprocessing time. With only a few exceptions, the number of explored search states consistently decreases a little when stepping from $r = 0$ to $r = 1$ and decreases sharply when stepping from $r = 1$ to $r = 4$. It increases sharply when stepping from $r = 4$ to $r = 7$. In terms of runtime, the decrease in search space size does not pay off due to the longer preprocessing time in C_1–C_4. It does pay off in C_5–C_8 when the search spaces explode.

Table B.4. Preprocessing runtime results for greedy search with the $h^{\mathcal{P}}_{AR}$ and $h^{\mathcal{P}}_{syn}$ heuristics. The columns headed with 0, 1, 4 and 7, give the number of refinement steps r used for $h^{\mathcal{P}}_{AR}$. The first column gives the used splitting bound b, i. e., the maximum number of automata per PDB. Dashes indicate out of memory (> 4 GByte).

	total runtime in s					preprocessing in s				
	$h^{\mathcal{P}}_{AR}$				$h^{\mathcal{P}}_{syn}$	$h^{\mathcal{P}}_{AR}$				$h^{\mathcal{P}}_{syn}$
	0	1	4	7		0	1	4	7	
					C_1					
1	0.4	0.5	0.6	0.7	0.3	0.2	0.3	0.6	0.6	0.2
2	0.7	1.5	3.3	3.4	1.2	0.6	1.4	3.3	3.3	1.0
3	1.0	2.2	7.2	24.6	8.6	0.9	2.1	7.1	24.5	7.8
4	7.2	21.9	112.6	862.5	617.9	7.0	21.8	112.5	862.2	571.5
					C_2					
1	0.6	0.7	0.8	0.8	0.5	0.2	0.3	0.6	0.7	0.3
2	1.0	1.8	3.6	3.6	1.3	0.7	1.6	3.5	3.5	1.1
3	1.2	2.5	7.7	26.2	9.4	0.9	2.2	7.6	26.0	8.3
4	1.3	3.0	9.8	34.2	15.3	1.1	2.7	9.7	34.0	13.6
					C_3					
1	0.8	0.9	0.8	0.8	0.4	0.2	0.4	0.7	0.7	0.3
2	1.1	2.0	3.8	3.8	1.4	0.7	1.6	3.7	3.7	1.2
3	1.4	2.7	8.1	27.0	9.3	1.0	2.4	8.0	26.7	8.3
4	1.5	3.2	10.1	35.2	15.3	1.1	2.8	10.0	35.0	13.6
					C_4					
1	5.0	5.2	1.4	1.4	0.9	0.3	0.5	0.9	0.9	0.4
2	3.6	4.7	4.5	4.5	1.9	0.8	1.9	4.1	4.2	1.5
3	4.0	5.7	9.9	32.0	20.2	1.1	2.9	9.5	30.4	17.7
4	4.2	6.2	11.8	40.7	39.1	1.3	3.3	11.4	39.1	34.7
					C_5					
1	54.4	51.5	7.8	7.8	3.2	0.4	0.6	1.1	1.1	0.6
2	29.7	45.0	7.2	7.2	3.5	0.9	2.1	4.6	4.6	1.8
3	30.2	46.1	13.2	46.0	26.5	1.3	3.1	10.3	32.4	19.9
4	30.4	46.9	15.8	53.5	49.2	1.5	3.7	14.0	42.4	40.1
					C_6					
1	–	–	28.9	28.7	21.8	0.4	0.0	1.2	1.2	0.5
2	–	–	22.7	22.9	16.6	0.0	2.2	5.0	5.0	1.8
3	–	–	28.2	211.1	78.5	0.0	3.3	11.0	34.6	29.1
4	–	–	44.9	208.2	98.3	2.0	5.1	19.9	64.3	43.9
					C_7					
1	–	–	–	–	386.1	0.4	0.0	0.0	0.0	0.6
2	–	–	174.2	167.9	158.9	1.0	2.4	5.4	5.3	1.8
3	–	–	194.2	–	567.0	1.5	0.0	11.8	0.0	37.4
4	–	–	257.9	–	637.0	2.1	5.5	21.3	0.0	60.9
					C_8					
1	–	–	–	–	329.4	0.4	0.8	1.4	1.3	0.6
2	–	–	204.0	203.0	120.2	0.0	2.6	5.8	5.8	1.8
3	–	–	296.9	–	336.2	1.5	3.8	12.5	37.7	45.7
4	–	–	263.9	–	555.4	2.2	5.7	22.0	69.7	201.6

C

Preprocessing for the h^{coi} and $h^{\mathcal{A}}$ Heuristics

C.1 Preprocessing for the h^{coi} Heuristic

The results in Table C.1 are obtained with UPPAAL/DMC's greedy search in conjunction with the h^{coi} heuristic. Recall that Qian et al.'s COI-based method to construct abstraction sets is parametrized [87]. Their method starts with the symbols that are mentioned in the target formula. This set of symbols forms layer 0. Then, iteratively, new layers are constructed, where layer $n + 1$ arises from layer n by including any symbol y that does not occur in a layer $n' \leq n$, and that may be involved in modifying the status of a symbol x in layer n. The abstraction set is then chosen based on a user defined cut-off value d. The resulting abstraction set contains exactly all the symbols in layers $n > d$.

If d increases then the preprocessing time also increases, with a few exceptions. This is not surprising, since for larger values of d, fewer symbols are abstracted and thus the abstract state space becomes larger. In our examples, for values greater than 4, the abstract state space equals the original state space. For values of d greater than 3, it is not possible to construct a PDB for the larger C examples, because the entire reachable abstract state space of these examples does not fit in 4 GByte of memory. Regarding the number of explored states and length of the detected error trace, it can be observed that the larger d is, the fewer states are explored and the shorter is the detected error trace. Again this is not surprising, since with increasing d, the heuristic becomes more informed. Obviously, for our benchmarks, the best value for d is 3. For smaller values, the resulting heuristic is not well informed. For larger values, preprocessing for the examples C_6–C_9 runs out of memory.

Table C.1. Results for greedy search with h^{coi}. The numbers in the first row gives the cut-off value d. Dashes indicate out of memory ($>$ 4 GByte).

	total runtime in s				preprocessing time in s			
Exp.	1	2	3	4	1	2	3	4
M_1	0.2	0.1	2.2	3.5	0.0	0.0	2.1	3.4
M_2	0.7	0.1	8.2	15.1	0.0	0.0	7.8	14.6
M_3	0.9	1.1	8.6	14.5	0.0	0.0	7.6	14.0
M_4	2.2	2.4	33.2	59.3	0.0	0.1	30.3	57.5
N_1	0.5	17.1	19.5	19.8	0.0	15.3	19.1	19.4
N_2	2.9	89.3	106.3	106.9	0.0	75.9	104.8	105.5
N_3	3.7	90.4	103.5	104.4	0.0	81.2	102.1	103.0
N_4	20.0	454.7	557.5	541.2	0.0	393.3	552.1	535.9
C_1	0.2	2.3	0.9	0.8	0.1	2.0	0.8	0.8
C_2	0.5	3.3	2.4	2.4	0.1	2.7	2.4	2.4
C_3	0.7	4.3	3.3	3.3	0.0	3.6	3.3	3.2
C_4	6.3	15.9	29.8	29.5	0.1	9.5	29.3	29.0
C_5	61.2	83.9	268.5	268.4	0.1	24.7	265.5	265.3
C_6	–	–	270.5	–	–	–	267.5	–
C_7	–	–	269.6	–	–	–	265.6	–
C_8	–	–	238.1	–	–	–	235.3	–
C_9	–	–	–	–	–	–	–	–

	explored states				trace length			
Exp.	1	2	3	4	1	2	3	4
M_1	11284	3632	4461	249	168	118	77	55
M_2	59667	11196	22172	517	475	117	101	62
M_3	85629	80269	47705	476	588	113	92	67
M_4	216938	200585	138147	1052	418	321	81	74
N_1	13902	27446	242	242	158	135	56	56
N_2	93467	174640	503	503	623	119	59	59
N_3	104104	126301	534	534	492	127	64	64
N_4	422499	607512	1217	1217	241	146	65	65
C_1	23173	21316	130	130	64	71	54	54
C_2	75111	72810	187	187	76	85	54	54
C_3	101049	97089	197	197	74	149	54	54
C_4	1023058	982003	474	474	85	248	56	56
C_5	9156066	8524179	2673	2673	123	373	57	57
C_6	–	–	9027	–	–	–	57	–
C_7	–	–	178513	–	–	–	70	–
C_8	–	–	28678	–	–	–	57	–
C_9	–	–	–	–	–	–	–	–

C.2 Preprocessing for the $h^{\mathcal{A}}$ Heuristic

In this section we provide a brief overview of the preprocessing time needed for our Russian Doll approach. Table C.2 gives the results for A^* and greedy search obtained with the $h^{\mathcal{A}}$ heuristic. The number of explored states and the length of the detected error trace only presented for convenience, but they are not discussed here. For a discussion see Sec. 7.3.

Note that for both search methods, the same PDB is constructed, i. e., the preprocessing time for both search methods is the same. It turns out that the preprocessing time for each C example is done in less than one second. For A^* search, also the actual search takes at most one second. This also holds for greedy search, except for the bigger C examples. Here, the runtime of the actual search clearly dominates the preprocessing time. From the results of the M and N examples it can be observed that most of the total time is spend during the preprocessing. This holds for A^* as well as greedy search, except for the A^* results of N_4.

Table C.2. Results for A^* and greedy search with the $h^{\mathcal{A}}$ heuristic. Abbreviations: *runtime:* total time including any preprocessing, *prep.*: preprocessing time. The memory limit was 4 GByte.

	explored states		runtime in s		prep. in s	trace length	
Exp.	A^*	greedy	A^*	greedy		A^*	greedy
M_1	190	249	3.4	3.2	3.0	47	55
M_2	4417	495	3.7	3.5	3.3	50	76
M_3	11006	993	3.9	3.4	3.2	50	53
M_4	41359	3577	4.6	3.5	3.4	53	105
N_1	345	242	19.8	19.4	19.0	49	56
N_2	3811	470	13.2	13.1	12.8	52	63
N_3	59062	1787	16.3	11.7	11.3	56	70
N_4	341928	10394	40.6	7.7	7.2	59	80
C_1	130	130	0.9	0.8	0.8	54	54
C_2	89813	56894	1.3	0.9	0.5	54	127
C_3	197	290	0.8	0.9	0.8	54	56
C_4	1140	1163	0.9	0.8	0.8	55	57
C_5	7530	39837	1.1	1.1	0.8	56	75
C_6	39436	80878	1.2	1.4	0.9	56	64
C_7	149993	697104	1.7	5.1	0.9	56	64
C_8	158361	1180067	1.9	8.2	0.9	56	97
C_9	127895	2250288	1.7	15.2	0.9	57	108

References

[1] Rajeev Alur, Costas Courcoubetis, and David L. Dill. Model-checking in dense real-time. *Information and Computation*, 104(1):2–34, 1993.

[2] Rajeev Alur and David L. Dill. Automata for modeling real-time systems. In Mike Paterson, editor, *Proceedings of the 17th International Colloquium on Automata, Languages and Programming (ICALP 1990)*, volume 443 of *Lecture Notes in Computer Science*, pages 322–335. Springer-Verlag, 1990.

[3] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[4] Felice Balarin and Alberto L. Sangiovanni-Vincentelli. An iterative approach to language containment. In Costas Courcoubetis, editor, *Proceedings of the 5th International Conference on Computer Aided Verification (CAV 1993)*, volume 697 of *Lecture Notes in Computer Science*, pages 29–40. Springer-Verlag, 1993.

[5] Thomas Ball, Rupak Majumdar, Todd D. Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of C programs. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2001)*, pages 203–213. ACM Press, 2001.

[6] Gerd Behrmann, Johan Bengtsson, Alexandre David, Kim G. Larsen, Paul Pettersson, and Wang Yi. Uppaal implementation secrets. In Werner Damm and Ernst-Rüdiger Olderog, editors, *Proceedings of the 7th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT 2002)*, volume 2469 of *Lecture Notes in Computer Science*, pages 3–22. Springer-Verlag, 2002.

[7] Gerd Behrmann, Alexandre David, and Kim G. Larsen. A tutorial on Uppaal. In Marco Bernardo and Flavio Corradini, editors, *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Sys-*

tems (SFM-RT 2004), volume 3185 of *Lecture Notes in Computer Science*, pages 200–236. Springer-Verlag, 2004.

[8] Gerd Behrmann and Ansgar Fehnker. Efficient guiding towards cost-optimality in Uppaal. In Tiziana Margaria and Wang Yi, editors, *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2001)*, volume 2031 of *Lecture Notes in Computer Science*, pages 174–188. Springer-Verlag, 2001.

[9] Gerd Behrmann, Ansgar Fehnker, Thomas Hune, Kim Guldstrand Larsen, Paul Pettersson, Judi Romijn, and Frits W. Vaandrager. Minimum-cost reachability for priced timed automata. In Maria Domenica Di Benedetto and Alberto L. Sangiovanni-Vincentelli, editors, *Proceedings of the 4th International Workshop on Hybrid Systems: Computation and Control (HSCC 2001)*, volume 2034 of *Lecture Notes in Computer Science*, pages 147–161. Springer-Verlag, 2001.

[10] Johan Bengtsson and Wang Yi. Timed automata: Semantics, algorithms and tools. In Jörg Desel, Wolfgang Reisig, and Grzegorz Rozenberg, editors, *Lectures on Concurrency and Petri Nets*, volume 3098 of *Lecture Notes in Computer Science*, pages 87–124. Springer-Verlag, 2004.

[11] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking. *Advances in Computers*, 58:118–149, 2003.

[12] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In Rance Cleaveland, editor, *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS 1999)*, volume 1579 of *Lecture Notes in Computer Science*, pages 193–207. Springer-Verlag, 1999.

[13] Stefan Blom, Bert Lisser, Jaco van de Pol, and Michael Weber. A database approach to distributed state space generation. *Electronic Notes in Theoretical Computer Science*, 198(1):17–32, 2008.

[14] Blai Bonet and Héctor Geffner. Planning as heuristic search. *Artificial Intelligence*, 129(1–2):5–33, 2001.

[15] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8), 1986.

[16] Janusz A. Brzozowski and Carl-Johan H. Seger. Advances in asynchronous circuit theory – part II: Bounded inertial delay models, MOS circuits, design techniques. In *Bulletin of the European Association for Theoretical Computer Science*, number 43, pages 199–263. 1991.

[17] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. In

Proceedings of the 5th Annual IEEE Symposium on Logic in Computer Science (LICS 1990), pages 428–439. IEEE Computer Society, 1990.

[18] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, 1992.

[19] Sagar Chaki, Edmund M. Clarke, Alex Groce, Somesh Jha, and Helmut Veith. Modular verification of software components in C. *IEEE Transactions on Software Engineering (TSE)*, 30(6):388–402, 2004.

[20] Yixin Chen, Benjamin W. Wah, and Chih-Wei Hsu. Temporal planning using subgoal partitioning and resolution in SGPlan. *Journal of Artificial Intelligence Research*, 26:323–369, 2006.

[21] Gianfranco Ciardo, Joshua Gluckman, and David Nicol. Distributed state space generation of discrete-state stochastic models. *Journal on Computing*, 10(1):82–93, 1998.

[22] Alessandro Cimatti, Edmund M. Clarke, Fausto Giunchiglia, and Marco Roveri. NuSMV: A new symbolic model verifier. In Nicolas Halbwachs and Doron Peled, editors, *Proceedings of the 11th International Conference on Computer Aided Verification (CAV 1999)*, volume 1633 of *Lecture Notes in Computer Science*, pages 495–499. Springer-Verlag, 1999.

[23] Edmund M. Clarke. The birth of model checking. In Orna Grumberg and Helmut Veith, editors, *25 Years of Model Checking*, volume 5000 of *Lecture Notes in Computer Science*, pages 1–26. Springer-Verlag, 2008.

[24] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In Dexter Kozen, editor, *Logics of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer-Verlag, 1982.

[25] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In E. Allen Emerson and A. Prasad Sistla, editors, *Proceedings of the 12th International Conference on Computer Aided Verification (CAV 2000)*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169. Springer-Verlag, 2000.

[26] 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.

[27] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, 1999.

[28] Olivier Coudert, Jean Christophe Madre, and Christian Berthet. Verifying temporal properties of sequential machines without building their state diagrams. In Edmund M. Clarke and Robert P. Kurshan, editors, *Proceedings of the 2nd International Workshop on Computer Aided Verification (CAV*

1990), volume 531 of *Lecture Notes in Computer Science*, pages 23–32. Springer-Verlag, 1991.

[29] Joseph C. Culberson and Jonathan Schaeffer. Pattern databases. *Computational Intelligence*, 14(3):318–334, 1998.

[30] Henning Dierks. *Specification and Verification of Polling Real-Time Systems*. PhD thesis, University of Oldenburg, Germany, 1999.

[31] Henning Dierks. PLC automata: A new class of implementable real-time automata. *Theoretical Computer Science*, 253(1):61–93, 2001.

[32] Henning Dierks. Comparing model-checking and logical reasoning for real-time systems. *Formal Aspects of Computing*, 16(2):104–120, 2004.

[33] Henning Dierks. *Time, Abstraction and Heuristics – Automatic Verification and Planning of Timed Systems using Abstraction and Heuristics*. Habilitation thesis, University of Oldenburg, Germany, 2005.

[34] Henning Dierks, Gerd Behrmann, and Kim G. Larsen. Solving planning problems using real-time model-checking (translating PDDL3 into timed automata. In Froduald Kabanza and Sylvie Thiébaux, editors, *Proceedings of the AIPS-Workshop on Planning via Model-Checking*, pages 30–39, 2002.

[35] Henning Dierks, Sebastian Kupferschmid, and Kim G. Larsen. Automatic abstraction refinement for timed automata. In Jean-François Raskin and P. S. Thiagarajan, editors, *Proceedings of the 5th International Conference on Formal Modelling and Analysis of Timed Systems (FORMATS 2007)*, volume 4763 of *Lecture Notes in Computer Science*, pages 114–129. Springer-Verlag, 2007.

[36] Edsger W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[37] David L. Dill. Timing assumptions and verification of finite-state concurrent systems. In Joseph Sifakis, editor, *Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems (AVMFSS 1989)*, volume 407 of *Lecture Notes in Computer Science*, pages 197–212. Springer-Verlag, 1990.

[38] David L. Dill. The Murφ verification system. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the 8th International Conference on Computer Aided Verification (CAV 1996)*, volume 1102 of *Lecture Notes in Computer Science*, pages 390–393. Springer-Verlag, 1996.

[39] Klaus Dräger, Bernd Finkbeiner, and Andreas Podelski. Directed model checking with distance-preserving abstractions. In Antti Valmari, editor, *Proceedings of the 13th International SPIN Workshop on Model Checking Software (SPIN 2006)*, volume 3925 of *Lecture Notes in Computer Science*, pages 19–34. Springer-Verlag, 2006.

[40] Klaus Dräger, Bernd Finkbeiner, and Andreas Podelski. Directed model checking with distance-preserving abstractions. *International Journal on Software Tools for Technology Transfer*, 11(1):27–37, 2009.

[41] Stefan Edelkamp. Generalizing the relaxed planning heuristic to non-linear tasks. In Susanne Biundo, Thom W. Frühwirth, and Günther Palm, editors, *Proceedings of the 27th Annual German Conference on AI (KI 2004)*, volume 3238 of *Lecture Notes in Computer Science*, pages 198–212. Springer-Verlag, 2004.

[42] Stefan Edelkamp, Stefan Leue, and Alberto Lluch-Lafuente. Directed explicit-state model checking in the validation of communication protocols. *International Journal on Software Tools for Technology Transfer*, 5(2):247–267, 2004.

[43] Stefan Edelkamp, Alberto Lluch-Lafuente, and Stefan Leue. Directed explicit model checking with HSF-SPIN. In Matthew B. Dwyer, editor, *Proceedings of the 8th International SPIN Workshop on Model Checking Software (SPIN 2001)*, volume 2057 of *Lecture Notes in Computer Science*, pages 57–79. Springer-Verlag, 2001.

[44] Stefan Edelkamp, Peter Sanders, and Pavel Simecek. Semi-external LTL model checking. In Aarti Gupta and Sharad Malik, editors, *Proceedings of the 20th International Conference on Computer Aided Verification (CAV 2008)*, volume 5123 of *Lecture Notes in Computer Science*, pages 530–542. Springer-Verlag, 2008.

[45] E. Allen Emerson and Edmund M. Clarke. Using branching time temporal logic to synthesize synchronization skeletons. *Science of Computer Programming*, 2(3):241–266, 1982.

[46] Michael J. Fischer. Re: Where are you? E-mail message to Leslie Lamport. Arpanet message sent on June 25, 1985 18:56:29 EDT, number 8506252257.AAO7636@YALE-BULLDOG.YALE.ARPA, 1985.

[47] Maria Fox and Derek Long. PDDL2.1: An extension to PDDL for expressing temporal planning domains. *Journal of Artificial Intelligence Research*, 20:61–124, 2003.

[48] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[49] Alfonso Gerevini, Alessandro Saetti, and Ivan Serina. Planning through stochastic local search and temporal action graphs. *Journal of Artificial Intelligence Research*, 20:239–290, 2003.

[50] Susanne Graf and Hassen Saïdi. Construction of abstract state graphs with PVS. In Orna Grumberg, editor, *Proceedings of the 9th International Conference on Computer Aided Verification (CAV 1997)*, volume 1254 of *Lecture Notes in Computer Science*, pages 72–83. Springer-Verlag, 1997.

[51] Alex Groce and Willem Visser. Heuristics for model checking Java programs. *International Journal on Software Tools for Technology Transfer*, 6(4):260–276, 2004.

[52] Richard Wesley Hamming. Error detecting and error correcting codes. *The Bell System Technical Journal*, 26(2):147–160, 1950.

[53] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[54] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. Correction to a formal basis for the heuristic determination of minimum cost paths. *SIGART Newsletter*, 37:28–29, 1972.

[55] Patrik Haslum, Adi Botea, Malte Helmert, Blai Bonet, and Sven Koenig. Domain-independent construction of pattern database heuristics for cost-optimal planning. In *Proceedings of the 22nd AAAI Conference on Artificial Intelligence (AAAI 2007)*, pages 1007–1012. AAAI Press, 2007.

[56] Malte Helmert. The Fast Downward planning system. *Journal of Artificial Intelligence Research*, 26:191–246, 2006.

[57] Malte Helmert, Patrik Haslum, and Jörg Hoffmann. Explicit-state abstraction: A new method for generating heuristic functions. In Dieter Fox and Carla P. Gomes, editors, *Proceedings of the 23rd AAAI Conference on Artificial Intelligence (AAAI 2008)*, pages 1547–1550. AAAI Press, 2008.

[58] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Kenneth L. McMillan. Abstractions from proofs. In Neil D. Jones and Xavier Leroy, editors, *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2004)*, pages 232–244. ACM Press, 2004.

[59] Jörg Hoffmann. The Metric-FF planning system: Translating "ignoring delete lists" to numeric state variables. *Journal of Artificial Intelligence Research*, 20:291–341, 2003.

[60] Jörg Hoffmann and Jana Koehler. A new method to index and query sets. In Thomas Dean, editor, *Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI 1999)*, pages 462–467. Morgan Kaufmann, 1999.

[61] Jörg Hoffmann and Bernhard Nebel. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14:253–302, 2001.

[62] Jörg Hoffmann, Julie Porteous, and Laura Sebastia. Ordered landmarks in planning. *Journal of Artificial Intelligence Research*, 22:215–278, 2004.

[63] Jörg Hoffmann, Jan-Georg Smaus, Andrey Rybalchenko, Sebastian Kupferschmid, and Andreas Podelski. Using predicate abstraction to generate heuristic functions in Uppaal. In Stefan Edelkamp and Alessio Lo-

muscio, editors, *Proceedings of the 4th Workshop on Model Checking and Artificial Intelligence (MOCHART 2006)*, volume 4428 of *Lecture Notes in Computer Science*, pages 51–66. Springer-Verlag, 2007.

[64] Gerard J. Holzmann. *The SPIN Model Checker – Primer and Reference Manual*. Addison-Wesley, 2003.

[65] Shahid Jabbar and Stefan Edelkamp. I/O efficient directed model checking. In Radhia Cousot, editor, *Proceedings of the 6th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2005)*, volume 3385 of *Lecture Notes in Computer Science*, pages 313–329. Springer-Verlag, 2005.

[66] Richard E. Korf and Ariel Felner. Disjoint pattern database heuristics. *Artificial Intelligence*, 134(1-2):9–22, 2002.

[67] Dexter Kozen. Lower bounds for natural proof systems. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science (FOCS 1977)*, pages 254–266. IEEE Computer Society, 1977.

[68] Bernd Krieg-Brückner, Jan Peleska, Ernst-Rüdiger Olderog, and Alexander Baer. The UniForM workbench, a universal development environment for formal methods. In Jeannette M. Wing, Jim Woodcock, and Jim Davies, editors, *Proceedings of the World Congress on Formal Methods in the Development of Computing Systems (FM 1999)*, volume 1709 of *Lecture Notes in Computer Science*, pages 1186–1205. Springer-Verlag, 1999.

[69] Sebastian Kupferschmid, Klaus Dräger, Jörg Hoffmann, Bernd Finkbeiner, Henning Dierks, Andreas Podelski, and Gerd Behrmann. Uppaal/DMC – abstraction-based heuristics for directed model checking. In Orna Grumberg and Michael Huth, editors, *Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2007)*, volume 4424 of *Lecture Notes in Computer Science*, pages 679–682. Springer-Verlag, 2007.

[70] Sebastian Kupferschmid, Jörg Hoffmann, Henning Dierks, and Gerd Behrmann. Adapting an AI planning heuristic for directed model checking. In Antti Valmari, editor, *Proceedings of the 13th International SPIN Workshop on Model Checking Software (SPIN 2006)*, volume 3925 of *Lecture Notes in Computer Science*, pages 35–52. Springer-Verlag, 2006.

[71] Sebastian Kupferschmid, Jörg Hoffmann, and Kim G. Larsen. Fast directed model checking via russian doll abstraction. In C. R. Ramakrishnan and Jakob Rehof, editors, *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2008)*, volume 4963 of *Lecture Notes in Computer Science*, pages 203–217. Springer-Verlag, 2008.

[72] Sebastian Kupferschmid, Martin Wehrle, Bernhard Nebel, and Andreas Podelski. Faster than Uppaal? In Aarti Gupta and Sharad Malik, editors,

Proceedings of the 20th International Conference on Computer Aided Verification (CAV 2008), volume 5123 of *Lecture Notes in Computer Science*, pages 552–555. Springer-Verlag, 2008.

[73] Leslie Lamport. A fast mutual exclusion algorithm. *ACM Transactions on Computer Systems*, 5(1):1–11, 1987.

[74] François Laroussinie and Kim G. Larsen. CMC: A tool for compositional model-checking of real-time systems. In Stanislaw Budkowski, Ana R. Cavalli, and Elie Najm, editors, *Proceedings of the International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols (FORTE 1998)*, volume 135 of *IFIP Conference Proceedings*, pages 439–456. Kluwer, 1998.

[75] Kim G. Larsen, Gerd Behrmann, Ed Brinksma, Ansgar Fehnker, Thomas Hune, Paul Pettersson, and Judi Romijn. As cheap as possible: Efficient cost-optimal reachability for priced timed automata. In Gérard Berry, Hubert Comon, and Alain Finkel, editors, *Proceedings of the 13th International Conference on Computer Aided Verification (CAV 2001)*, volume 2102 of *Lecture Notes in Computer Science*, pages 493–505. Springer-Verlag, 2001.

[76] Flavio Lerda and Riccardo Sisto. Distributed-memory model checking with SPIN. In Dennis Dams, Rob Gerth, Stefan Leue, and Mieke Massink, editors, *Proceedings of the 5th and 6th International SPIN Workshops on Model Checking Software (SPIN 1999)*, volume 1680 of *Lecture Notes in Computer Science*, pages 22–39. Springer-Verlag, 1999.

[77] Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[78] Robin Milner. An algebraic definition of simulation between programs. In David C. Cooper, editor, *Proceedings of the 2nd International Joint Conference on Artificial Intelligence (IJCAI 1971)*, pages 481–489. Morgan Kaufmann, 1971.

[79] M. Oliver Möller, Harald Rueß, and Maria Sorea. Predicate abstraction for dense real-time system. *Electronic Notes in Theoretical Computer Science*, 65(6), 2002.

[80] Ernst-Rüdiger Olderog and Henning Dierks. Moby/RT: A tool for specification and verification of real-time systems. *Journal of Universal Computer Science*, 9(2):88–105, 2003.

[81] Susan S. Owicki and Leslie Lamport. Proving liveness properties of concurrent programs. *ACM Transactions on Programming Languages and Systems*, 4(3):455–495, 1982.

[82] Judea Pearl. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley, 1984.

[83] Carl Pixley. Introduction to a computational theory and implementation of sequential hardware equivalence. In Edmund M. Clarke and Robert P. Kurshan, editors, *Proceedings of the 2nd International Workshop on Computer Aided Verification (CAV 1990)*, volume 531 of *Lecture Notes in Computer Science*, pages 54–64. Springer-Verlag, 1991.

[84] Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science (FOCS 1977)*, pages 46–57. IEEE Computer Society, 1977.

[85] Andreas Podelski and Andrey Rybalchenko. ARMC: The logical choice for software model checking with abstraction refinement. In Michael Hanus, editor, *Proceedings of the 9th International Symposium on Practical Aspects of Declarative Languages (PADL 2007)*, volume 4354 of *Lecture Notes in Computer Science*, pages 245–259. Springer-Verlag, 2007.

[86] Kairong Qian and Albert Nymeyer. Guided invariant model checking based on abstraction and symbolic pattern databases. In Kurt Jensen and Andreas Podelski, editors, *Proceedings of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, volume 2988 of *Lecture Notes in Computer Science*, pages 497–511. Springer-Verlag, 2004.

[87] Kairong Qian, Albert Nymeyer, and Steven Susanto. Abstraction-guided model checking using symbolic IDA* and heuristic synthesis. In Farn Wang, editor, *Proceedings of the 25th International Conference on Formal Techniques for Networked and Distributed Systems (FORTE 2005)*, volume 3731 of *Lecture Notes in Computer Science*, pages 275–289. Springer-Verlag, 2005.

[88] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in CESAR. In Mariangiola Dezani-Ciancaglini and Ugo Montanari, editors, *Proceedings of the 5th International Symposium on Programming on Programming*, volume 137 of *Lecture Notes in Computer Science*, pages 337–351. Springer-Verlag, 1982.

[89] Silvia Richter, Malte Helmert, and Matthias Westphal. Landmarks revisited. In Dieter Fox and Carla P. Gomes, editors, *Proceedings of the 23rd AAAI Conference on Artificial Intelligence (AAAI 2008)*, pages 975–982. AAAI Press, 2008.

[90] Andrey Rybalchenko and Viorica Sofronie-Stokkermans. Constraint solving for interpolation. In Byron Cook and Andreas Podelski, editors, *Proceedings of the 8th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2007)*, volume 4349 of *Lecture Notes in Computer Science*, pages 346–362. Springer-Verlag, 2007.

[91] Jan-Georg Smaus and Jörg Hoffmann. Relaxation refinement: A new method to generate heuristic functions. In Doron Peled and Michael

Wooldridge, editors, *Proceedings of the 5th International Workshop on Model Checking and Artificial Intelligence (MOCHART 2008)*, volume 5348 of *Lecture Notes in Artificial Intelligence*, pages 146–164. Springer-Verlag, 2009.

[92] Maria Sorea. Lazy approximation for dense real-time systems. In Yassine Lakhnech and Sergio Yovine, editors, *Proceedings of the 2nd Joint International Conferences on Formal Modelling and Analysis of Timed Systems (FORMATS 2004)*, volume 3253 of *Lecture Notes in Computer Science*, pages 363–378. Springer-Verlag, 2004.

[93] Ofer Strichman. Tuning SAT checkers for bounded model checking. In E. Allen Emerson and A. Prasad Sistla, editors, *Proceedings of the 12th International Conference on Computer Aided Verification (CAV 2000)*, volume 1855 of *Lecture Notes in Computer Science*, pages 480–494. Springer-Verlag, 2000.

[94] Willem Visser, Klaus Havelund, Guillaume P. Brat, Seungjoon Park, and Flavio Lerda. Model checking programs. *Automated Software Engeneering*, 10(2):203–232, 2003.

[95] Martin Wehrle, Sebastian Kupferschmid, and Andreas Podelski. Useless actions are useful. In Jussi Rintanen, Bernhard Nebel, J. Christopher Beck, and Eric Hansen, editors, *Proceedings of the 18th International Conference on Automated Planning and Scheduling (ICAPS 2008)*, pages 388–395. AAAI Press, 2008.

[96] Martin Wehrle, Sebastian Kupferschmid, and Andreas Podelski. Transition-based directed model checking. In Stefan Kowalewski and Anna Philippou, editors, *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2009)*, volume 5505 of *Lecture Notes in Computer Science*, pages 186–200. Springer-Verlag, 2009.

[97] C. Han Yang and David L. Dill. Validation with guided search of the state space. In *Proceedings of the 35th Conference on Design Automation (DAC 1998)*, pages 599–604. ACM Press, 1998.