Martin Wehrle

Transition-Based Directed Model Checking

Dissertation

zur Erlangung des Doktorgrades
der Technischen Fakultät
der Albert-Ludwigs-Universität Freiburg

Tag der Disputation:
23. Dezember 2011

Dekan:
Prof. Dr. Bernd Becker, *Albert-Ludwigs-Universität Freiburg*

Referenten:
Prof. Dr. Andreas Podelski, *Albert-Ludwigs-Universität Freiburg*
Prof. Bernd Finkbeiner, Ph.D., *Universität des Saarlandes*
Prof. Dr. Bernhard Nebel, *Albert-Ludwigs-Universität Freiburg*
Prof. Dr. Hannah Bast, *Albert-Ludwigs-Universität Freiburg*

Abstract

Software and hardware systems are rapidly increasing in size and complexity. However, with increasing system complexity, the system design process becomes more error-prone. In particular, this is the case for concurrent systems, where subtle bugs may occur because of unexpected thread interleavings. Therefore, approaches to effectively find bugs are required. Currently, the most common approach to bug finding is *testing*. Testing turned out to be often a very effective approach for this purpose. However, subtle bugs may still remain undiscovered.

In contrast to testing, *model checking* is an automated approach to check a model of the system for the absence of bugs. In this approach, an exploration of the entire state space of the system is performed. Although originally introduced for the purpose of proving correctness, model checking has also been applied to bug finding.

Directed model checking is a version of model checking that is specialized towards finding paths to error states in concurrent systems. Directed model checking attempts to avoid exploring the entire state space by directing the exploration. To direct the exploration, every encountered state is prioritized based on the expected distance to a nearest error state. States with higher priorities are preferably explored.

The contribution of this thesis is the concept of *transition-based directed model checking*, which extends directed model checking by additionally computing priorities for *transitions*, rather than only to compute priorities for *states*. Transitions with higher priorities are preferably applied. We demonstrate that transition-based directed model checking can significantly improve the classical directed model checking approach. Our demonstration proceeds in four steps. First, we present the notion of *useless transitions*. Based on this notion, we identify a criterion to estimate if a given transition is needed to find an error state. Second, we present an application of useless transitions to the area of AI planning, where the similar problem of automatically finding predefined goal states is addressed. Third, we present an alternative to useless transitions based on *interference contexts*. Compared to the Boolean property of uselessness for transitions, interference contexts provide a

more fine-grained concept to compute transition priorities. Fourth, we present an abstraction principle called *safe abstraction*. Based on safe abstraction, we identify transitions that can be ignored completely during the exploration of the state space without affecting completeness.

As part of the thesis, we have developed the directed model checking tool MCTA. In particular, we have implemented useless transitions, interference contexts, and safe abstraction. We demonstrate that these three instances of transition-based directed model checking outperform previous directed model checking techniques on large practical problems. Furthermore, we have integrated the above-mentioned application of useless transitions to the area of AI planning into the planning system FAST DOWNWARD. We demonstrate that useless transitions improve existing planning techniques on a large number of benchmarks from the international planning competitions.

Zusammenfassung

Soft- und Hardwaresysteme gewinnen heutzutage verstärkt an Größe und Komplexität, was zur Folge hat, dass die Entwicklung solcher Systeme anfälliger für Fehler wird. Insbesondere bei nebenläufigen Systemen können subtile Fehler auftreten, die durch unerwartete Interaktionen der parallel ausgeführten Programme verursacht werden. Aufgrund dieser Tatsachen werden effektive Verfahren benötigt, um solche Fehler zu finden. Das im Moment am weitesten verbreitete Verfahren zur Fehlersuche ist *Testen*. Testen hat sich in vielen Fällen als sehr effektiv herausgestellt, allerdings werden subtile Fehler auch mit Testen nicht immer gefunden.

Im Gegensatz zu Testen ist die *Modellprüfung* ein automatisiertes Verfahren, um die Abwesenheit von Fehlern zu zeigen. Hierzu wird der Zustandsraum eines Modells des gegebenen Systems exploriert. Obwohl die Modellprüfung ursprünglich eingeführt wurde, um die Korrektheit von Systemen zu zeigen, wird sie auch zur Fehlersuche eingesetzt.

Die *gerichtete Modellprüfung* ist eine Variante der Modellprüfung, die darauf spezialisiert ist, Fehlerzustände in nebenläufigen Systemen zu finden. Die gerichtete Modellprüfung nutzt die Tatsache aus, dass es oftmals ausreicht, nur einen Teil des Zustandsraums zu explorieren, um einen Fehlerzustand zu finden. Um dies zu erreichen, werden die Zustände während der Exploration mit Hilfe von Heuristiken bewertet. Zustände mit besserer Bewertung werden bevorzugt exploriert.

Der Beitrag dieser Arbeit besteht aus dem Konzept der *transitionsbasierten gerichteten Modellprüfung*, das die gerichtete Modellprüfung erweitert, indem nicht nur *Zustände*, sondern auch *Transitionen* bewertet werden. Transitionen mit besserer Bewertung werden bevorzugt ausgeführt. Wir zeigen, dass die transitionsbasierte gerichtete Modellprüfung die klassische gerichtete Modellprüfung signifikant verbessern kann. Hierzu präsentieren wir als erstes das Konzept der *nutzlosen Transitionen*, mit dem wir abschätzen können, welche Transitionen nicht benötigt werden, um einen Fehlerzustand zu finden. Im nächsten Schritt stellen wir eine Anpassung dieses Konzepts auf das Gebiet der Handlungsplanung der

Künstlichen Intelligenz vor. Die Handlungsplanung beschäftigt sich mit dem zur Fehlersuche konzeptionell ähnlichen Problem, einen Zielzustand innerhalb einer Planungsaufgabe zu erreichen. Weiterhin präsentieren wir eine Alternative zum Konzept der nutzlosen Transitionen, das eine feinere Abstufung zur Bewertung von Transitionen erlaubt und auf der Analyse von *Interferenzkontexten* basiert. Abschließend stellen wir ein Konzept vor, das uns die Berechnung von Transitionen erlaubt, die beweisbar nicht benötigt werden, um einen Fehlerzustand zu finden. Dieses Konzept nennen wir *sichere Abstraktion*.

Als Teil der Arbeit haben wir den gerichteten Modellprüfer MCTA entwickelt, in den wir auch die Konzepte der nutzlosen Transitionen, der Interferenzkontexte und der sicheren Abstraktion implementiert haben. Wir zeigen, dass diese drei Instanzen der transitionsbasierten gerichteten Modellprüfung frühere Verfahren in großen, praktischen Problemen verbessert. Darüber hinaus haben wir die Anpassung der nutzlosen Transitionen auf das Gebiet der Handlungsplanung in das Planungssystem FAST DOWNWARD integriert. Wir zeigen, dass das Konzept der nutzlosen Transitionen existierende Planungstechniken in vielen Benchmarks der internationalen Planungswettbewerbe verbessert.

Acknowledgments

First and foremost, I thank my supervisor Andreas Podelski for giving me the possibility to write my thesis at the chair of Software Engineering in an excellent research environment. I want to thank Andreas not only for giving me much freedom to develop my own research ideas, but also for valuable advice and many fruitful discussions. Moreover, I thank Bernd Finkbeiner for serving as the second reviewer of this thesis.

My special thanks go to Sebastian Kupferschmid for an intensive collaboration over the years, and for carefully proofreading preliminary versions of this thesis. I really enjoyed our endless discussions about research problems and beyond, the joint travelings to conferences, the chats during coffee breaks, and our joint programming sessions.

Furthermore, I want to thank my colleagues and friends for many fruitful discussions and a great atmosphere over the years. In addition to Andreas and Sebastian, I specifically want to thank Stephan Arlt, Jelena Barth, Sergiy Bogomolov, Berit Brauer, Jürgen Christ, Daniel Dietsch, Evren Ermis, Harald Fecher, Sergio Feo-Arenis, Slawomir Grzonka, Matthias Heizmann, Malte Helmert, Jochen Hoenicke, Marlis Jost, Alexander Malkis, Robert Mattmüller, Stefan Maus, Martin Mehlmann, Corina Mitrohin, Marco Muñiz, Amalinda Post, Martin Preen, Gabriele Röger, Martin Schäf, Nassim Seghir, Bernd Westphal, and Thomas Wies.

I also thank the German Research Foundation (DFG) for providing the financial support for our research center “Automatic Verification and Analysis of Complex Systems” (AVACS). Furthermore, I thank the members of the AVACS subproject R3 for an excellent research environment. Specifically, I want to thank Henning Dierks, Klaus Dräger, Bernd Finkbeiner, Michael Gerke, Sebastian Kupferschmid, Andrey Kupriyanov, Bernhard Nebel, Hans-Jörg Peter, Andreas Podelski, and Jan-Georg Smaus.

Finally, I want to thank my family, in particular my wife Andrea and my parents Karin and Bernhard Wehrle, for their support over the years.

Contents

1 Introduction 1
1.1 Model Checking 1
1.2 Directed Model Checking 2
1.3 AI Planning 3
1.4 Related Work 4
1.4.1 Directed Model Checking 4
1.4.2 AI Planning as Heuristic Search 6
1.4.3 Transition Prioritizing Techniques 7
1.5 Outline 8

2 Preliminaries 11
2.1 Notation 11
2.2 Directed Model Checking 15
2.3 Abstraction Based Distance Heuristics 16
2.4 Benchmark Set 18

3 Useless Transitions 21
3.1 Motivation 21
3.2 Useless Transitions 22
3.2.1 Useless Transitions 23
3.2.2 Relatively Useless Transitions: The Practical Counterpart 27
3.2.3 Directed Model Checking with Relatively Useless Transitions 28
3.3 Discussion: Where does it work, where not? 29
3.4 Evaluation 31
3.4.1 Experimental Setup 32
3.4.2 Experimental Results and Discussion 32
3.4.3 Useless Transitions in AI Planning 37
3.5 Related Work 41
3.6 Conclusions 42

4 Context-Enhanced Directed Model Checking 45
4.1 Motivation 45
4.2 Context-Enhanced Directed Model Checking 47
4.2.1 Interference Contexts 47
4.2.2 The Context-Enhanced Search Algorithm 49
4.3 Related Work 51
4.4 Evaluation 52
4.4.1 Experimental Setting 52
4.4.2 Experimental Results 53
4.4.3 Discussion 58
4.5 Conclusions 60

5 The Causal Graph Revisited for Directed Model Checking 63
5.1 Notation 63
5.2 The Causal Graph 68
5.3 Safe Abstraction 71
5.4 The Causal Graph Heuristic for Directed Model Checking 73
5.4.1 Independent Processes 74
5.4.2 Processes with Causal Predecessors 74
5.4.3 Causal Graphs with Cycles 76
5.4.4 Discussion: Sources of Imprecision 77
5.5 Evaluation 80
5.5.1 Implementation Details 80
5.5.2 Experimental Results and Discussion 81
5.6 Conclusions 84

6 The Model Checker MCTA 85
6.1 Timed Automata 85
6.2 Description of MCTA 87
6.2.1 Ingredients in a Nutshell 87
6.2.2 Results 88
6.2.3 Future Work 89

7 Conclusions and Future Research 91

References 95

1

Introduction

1.1 Model Checking

In these days, modern software and hardware systems are rapidly increasing in size and complexity, and already play important roles in many parts of our daily life. However, with increasing system complexity, the system design process becomes more error-prone because logical design flaws become more difficult to detect for the engineer. This is particularly the case for concurrent systems, where subtle bugs may occur due to unexpected thread interleavings. Therefore, approaches to effectively find such bugs are required. Currently, in software engineering, the most common approach to bug finding is testing [59]. However, although testing is able to detect many bugs in practice, subtle "corner case" bugs in concurrent systems may still remain undiscovered.

In contrast to testing, model checking [16] is an automated approach for the verification of concurrent finite state systems. For a given mathematical model $\mathcal{M}$ of the system and a given property φ, model checking determines whether the system satisfies the property, i. e., whether $\mathcal{M} \models \varphi$. For this, model checking performs an exploration of the entire state space of $\mathcal{M}$. Although originally introduced for the purpose of proving correctness, model checking has also been applied to bug finding. In case of erroneous systems, the model checking algorithm is supposed to return an error trace. The error trace is used as a diagnostic counterexample which makes it possible to effectively debug the system. Overall, bug detection is an important aspect of model checking.

The main obstacle of model checking is the problem that the overall number of system states grows exponentially in the size of the system. Therefore, to be able to handle large and practical relevant systems, it is crucial to come up with efficient approaches to tackle this problem. In recent years, significant progress could be achieved in this direction. State-of-the-art model checking approaches include approaches based on abstraction and abstraction refinement [15], bounded model

checking [8, 9], external model checking [26, 46], as well as directed model checking [27]. Directed model checking is optimized towards the detection of error states using distance heuristics and represents the central topic of this thesis. We will introduce this approach in more detail in the following sections.

1.2 Directed Model Checking

The goal of model checking is to prove a system error-free, which means that the system satisfies a given property. If this is not the case, the objective of model checking is to find an error trace. *Directed model checking* is a special version of model checking which is tailored to the fast detection of short error traces in concurrent finite state systems. To efficiently detect short error traces, the exploration of the state space is directed to those parts that show promise to contain reachable error states. In order to appropriately direct the exploration, a distance heuristic is applied. The distance heuristic assigns a heuristic value to each state that is encountered during the exploration. The heuristic value typically reflects an estimation of the state's distance to a nearest error state. States with low heuristic values are preferably explored. Consequently, error traces can often be found by only exploring a small fraction of the entire reachable state space. Informally speaking, directed model checking can be considered as the application of heuristic search to model checking.

Distance heuristics are mostly based on abstractions. The heuristic value for a concrete state s is typically determined by the length of an abstract error trace that starts in the corresponding abstract state of s. In other words, real error distances in the original system are approximated with the length of abstract error distances in the corresponding abstract system. We will give an overview of recent approaches in Section 1.4.

Obviously, there is a trade-off between the accuracy and the computation time for the distance heuristic. At the one extreme end of this spectrum, the constant zero function can be used, which is very cheap to compute. However, it is obvious that this function does not provide any further guidance information, and the directed exploration of the state space would degenerate to blind search. At the other end of the spectrum, the perfect distance function could be computed, which provides the exact error distance for every system state. Having such a function, the overall model checking problem would already be solved, and no more search would be needed at all (an error state is reached by following strictly n states with decreasing distance values, where n is the length of a shortest possible error trace). However, the problem to compute such a perfect distance function is as hard as the model checking problem itself. Overall, the challenge is to identify the sweet-spot of the trade-off to design distance heuristics that are efficiently computable on the one hand, and that are as accurate as possible on the other hand.

Directed model checking has proved to often perform much better than uninformed search methods like breadth-first or depth-first search [23, 51]. However, for large systems, even error detection with directed model checking becomes very challenging, and additional search techniques are required to handle the state spaces.

In particular, techniques that also evaluate the quality of *transitions*, rather than only states, are very promising [36, 42]. Such techniques have first been proposed in the area of AI planning and have proved to often further alleviate the state explosion problem. We provide an overview of such techniques in Section 1.4.

1.3 AI Planning

There is a strong relationship of detecting error states that violate a given property and *Artificial Intelligence (AI) Planning* [60]. The overall goal of AI planning is to develop algorithms to automatically achieve desired goal states from a given initial state with a given set of actions (formalized as *operators*). From a model checking point of view, operators resemble *guarded commands*, consisting of a precondition and an effect. An operator can be applied in a state s if s satisfies the operator's precondition. In this case, the successor state is determined from s according to the operator's effect.

Planning algorithms are applicable to arbitrary planning domains that are formalized in a common planning domain description language. For example, a robot might be supposed to autonomously navigate through a labyrinth and find the exit. Although motivated differently, this is a similar problem to finding error states in concurrent systems when considered from an abstract point of view. The easiest planning setting is classical STRIPS planning as described, e. g., in a paper by Fox and Long as a fragment of a richer planning language [29]. In the STRIPS formalism, effects of actions are deterministic and unconditional, and the world (i. e., the values of the state variables) is fully observable at every time point. STRIPS planning is known to be PSPACE-complete [12]. (There are various extensions, including non-deterministic planning and planning under partial observability; these are not relevant for this thesis). In the following, we focus on classical STRIPS planning because it is the most similar setting to our context of detecting error states. In the literature, various approaches to solve planning problems have been proposed, including SAT-based approaches [48, 69, 70] as well as approaches based on heuristic search [10, 13, 31, 36, 42]. Among these, planning as heuristic search has turned out to be one of the most successful approaches in recent years (for details and related work, see Section 1.4).

From an abstract point of view, directed model checking and planning as heuristic search are very similar approaches. Both of them can be considered as the application of heuristic search to a specific reachability problem, where the distance

heuristics are computed fully automatically based on a declarative problem description. In both areas, the objective is to find short sequences of transitions (or actions in the planning context) that lead from a given initial state to a state that satisfies some goal condition. In directed model checking, the goal condition describes an erroneous configuration of the system which is typically a negated invariant. In the area of AI planning, the objective is typically to reach a state where a desired situation is fulfilled.

1.4 Related Work

In this section, we provide an overview of recent and important approaches concerning directed model checking and AI planning as heuristic search. Furthermore, we present related work concerning the (heuristic) evaluation of transitions in Section 1.4.3.

1.4.1 Directed Model Checking

Directed model checking has been studied in different contexts and variants. In particular, many tools are based on this approach, including the tools HSF-SPIN [25], UPPAAL/DMC [51], and Java PATHFINDER [75].

Directed model checking has been pioneered by Yang and Dill [80] and by Edelkamp et al. [23, 25]. Yang and Dill called this approach *prioritized model checking* and proposed the *hamming distance heuristic*. The hamming distance heuristic represents states as strings of bits. The length of an error trace from a state s to an error state s' is estimated as the number of positions in the representations of s and s' such that the values of s and s' differ in that position. Furthermore, in the context of SPIN [45], Edelkamp et al. proposed two distance heuristics based on *graph distances* to guide the exploration of the state space. For a parallel system of processes, they use the local graph distance to estimate the actual error distance. On the one hand, this is a coarse abstraction because synchronization behavior and integer variables are ignored completely. On the other hand, the computation time is low. The distance heuristics differ in the way the local graph distances are used for the global distance estimation (i. e., by taking the maximum or the sum). Overall, all these distance heuristics can be computed automatically based on a declarative description of the system, but they are rather coarse approximations of the real error distance.

Kupferschmid et al. [52] introduce two sophisticated distance heuristics that are based on the *monotonicity abstraction*. This abstraction technique can be considered as an adaptation of the *ignoring delete lists* principle that was originally introduced in AI planning [10] (see Section 1.4.2). The idea of this abstraction is to assume that

every state variable, once it obtained a value, keeps this value forever. Hence, every state variable becomes set-valued and can have multiple values simultaneously in general. The first proposed distance heuristic performs a fixed-point iteration based on simultaneously applying transitions under the abstract semantics given by the monotonicity abstraction. The iteration starts with the values of the variables given by the current state, and ends if an abstract error state is found or a fixed-point is reached. The heuristic value is essentially determined by the number of iterations. Based on this fixed-point iteration and starting in the abstract error state, the second distance heuristic additionally extracts an abstract error trace. The heuristic value is defined as the number of abstract transitions in this abstract error trace. Both distance heuristics are more expensive to compute than those proposed by Edelkamp et al., but they often provide more accurate distance estimates.

The distance heuristics described so far are computed on-the-fly, which means that the heuristic values are obtained by computing abstract error traces in every global state *during* model checking. In contrast, many distance heuristics based on homomorphic abstractions are computed *prior* to model checking. Such distance heuristics are essentially organized as a lookup table. To obtain heuristic values during model checking, concrete states are mapped to corresponding abstract states in this table. Examples for such distance heuristics are *pattern database heuristics* [17]. A pattern database heuristic is usually computed based on the following steps. In a first step, a subset of the state variables of the system under consideration is selected. This subset is called the *pattern*. In a second step, the system under consideration is abstracted by ignoring the state variables that do not occur in the pattern. For this abstracted problem, the entire state space is computed, where the abstract states are stored in a lookup table (the pattern database) together with the abstract error distance. The distance value for a concrete state is defined as the abstract distance of the corresponding abstract state in the pattern database. Distance heuristics based on pattern databases differ in the selection of the pattern: Qian and Nymeyer [65] proposed to use a kind of cone-of-influence analysis, Kupferschmid et al. [54] proposed to use the monotonicity abstraction. Moreover, in a more general approach, Hoffmann et al. [44] and Smaus and Hoffmann [72] applied predicate abstraction to compute the pattern database.

Dräger et al. [21] proposed a distance heuristic that iteratively builds the product automaton of two automata of the system under consideration. More precisely, the abstract system is built by iteratively merging two automata and replacing them with the resulting product automaton. To avoid the state explosion problem, the product automaton is "shrunken" by merging states until it is smaller than a given threshold n. This process is repeated until only one automaton is left. The distance values are obtained by matching a concrete state to the corresponding abstract state in the product automaton, and then using the graph distance to an abstract error state as distance estimation. The parameters for this approach are the threshold n as

well as strategies which automata to select next for the product building and which states to merge.

1.4.2 AI Planning as Heuristic Search

Considering the results of the recent international planning competitions [1, 30], planning as heuristic search turned out to be the most successful planning technique. An important discipline is called *optimal* planning, which describes the task of finding shortest possible plans (i. e., plans with minimal number of actions in the STRIPS formalism). In this context, distance heuristics that never overestimate the real goal distance (so-called *admissible* distance heuristics) play an important role, because they produce optimal plans when applied with the A* algorithm [64]. However, if optimality of plans is not required, inadmissible distance heuristics applied with greedy best first search often perform better in practice. In the last decade, various (admissible and inadmissible) distance heuristics have been proposed in the literature. To be consistent with the terminology used in the AI planning community, we use the term “heuristic” synonymously with the term “distance heuristic” in this section.

An important class of planning heuristics is based on *delete relaxations*. This principle has been originally introduced in the planning community and works analogously to the monotonicity abstraction (in fact, the monotonicity abstraction has been introduced as an adaptation of the ignoring-delete-lists principle to the context of timed automata). Therefore, negative effects of operators are ignored, and variables have multiple values simultaneously in general. Heuristics based on delete relaxations aim at approximating the *optimal relaxed plan heuristic* h^+, which computes shortest plans under this relaxation. Unfortunately, the computation of h^+ is an NP-equivalent problem [12]. Bonet and Geffner [10] pioneered heuristic search planning and proposed the admissible approximation h^{max} and the inadmissible h^{add}. In a suboptimal setting, Hoffmann and Nebel [42] introduced the *FF* planning system with the h^{FF} heuristic in the same year which turned out to be a more accurate approximation of h^+ than h^{max} and h^{add}. This planning system pushed the frontier in planning, in particular winning the five years best paper award 2005 of JAIR [2]. As a side remark, the distance heuristics proposed by Kupferschmid et al. as described in the last section are adaptations of h^{max} and h^{FF} to directed model checking. Apart from approximating h^+ explicitly, Helmert [35] proposed the causal graph heuristic h^{CG}, which is implemented in the FAST DOWNWARD planning system [36]. Later on, Helmert and Geffner [38] showed that h^{CG} can also be viewed as a refined version of h^{add}. Based on this insight, they proposed the context enhanced additive heuristic h^{cea} which generalizes both h^{add} and h^{CG}. The overall performance of h^{cea} is empirically shown to be superior to that of h^{add} and h^{CG}.

Another class of planning heuristics is based on *landmarks*. Landmarks are predicates that are true in every plan at some point. In this context, Richter et al. [67] proposed an inadmissible landmark heuristic which estimates the goal distance in terms of the number of landmarks that have not yet been achieved. Similarly, *(disjunctive) action landmarks* [37, 47] describe (sets of) actions with the property that an action landmark (or at least one action in a disjunctive action landmark) has to be applied at least once in every valid plan. Based on action landmarks, Karpas and Domshlak [47] proposed an admissible landmark heuristic for optimal planning. Moreover, Helmert and Domshlak [37] presented a classification of different classes of heuristics, including delete relaxation and landmark heuristics. In particular, they proposed a compilation scheme from h^{max} to an admissible landmark heuristic that they called the landmark cut heuristic $h^{\text{LM-cut}}$. They empirically showed that $h^{\text{LM-cut}}$ approximates the optimal relaxed plan heuristic h^+ in an almost perfect way with a very low relative deviation.

A further class of planning heuristics is the class of *abstraction heuristics*. In AI planning, the term *abstraction* is defined as a homomorphic abstraction function that maps concrete states to abstract states. Planning heuristics that are based on that kind of abstraction include pattern database heuristics [22, 33] and *merge-and-shrink* abstractions [39, 40]. The latter is an adaptation of the approach proposed by Dräger et al. [20, 21] to planning.

Overall, we have particularly observed in this section that the close relationship between heuristic search planning and directed model checking is also reflected in the literature, where planning techniques based on heuristic search have been successfully applied to directed model checking, and vice versa.

1.4.3 Transition Prioritizing Techniques

Techniques to heuristically evaluate and prioritize transitions have mostly been studied in the area of AI planning. Such techniques estimate how promising it is to apply certain actions in a given state. In this context, Hoffmann and Nebel proposed the *helpful actions* technique within the FF planning system [42]. An action a is defined as helpful in a state s if a also occurs as a first action in an abstract solution computed by h^{FF} from s to a goal state. However, as helpful actions are identified during the computation of h^{FF} as a byproduct, helpful actions are also specific to the h^{FF} heuristic. Hoffmann and Nebel empirically showed the practical potential of giving preference to helpful actions in a local search setting. Based on this idea, Helmert [36] proposed *preferred operators* as an adaptation to the causal graph heuristic h^{CG} and generalized the approach to global search. Later on, preferred operators have been investigated more systematically by Richter and Helmert [66]. Moreover, Vidal [74] extended the idea of helpful actions to sequences of actions. Instead of successively preferring states that result from applying one helpful

action, a (generally longer) prefix of actions from the abstract solution is extracted if these actions are sequentially applicable in the current state. This idea has been further extended by Baier and Botea [5].

Other approaches explicitly remove actions that seem to be not needed to find a solution. In this context, Nebel et al. [61] identified irrelevant operators and variables by solving a relaxed problem. These operators and variables are removed from the planning problem in a preprocessing step. However, this approach is not solution preserving. Haslum and Jonsson [34] defined an operator as redundant if it can be simulated with a sequence of other operators. Based on this idea, they proposed an approach to compute redundant operators and reduced operator sets that is solution preserving. Bacchus and Ady [4] proposed to avoid action sequences that do not make sense. For example, a transporter should not load and unload an object without moving in between if the object should finally be located somewhere else. However, such action sequences have to be provided manually.

In the area of model checking, a straightforward approach for detecting the relevance of system components (i. e., automata or variables) and its transitions is *cone-of-influence* analysis [16]. In this approach, the dependencies of system components are analyzed starting from the components that occur in the specification of the property that is subject to model checking. Components that do not transitively depend on the property need not be considered. Moreover, *partial order reduction* can be considered as an approach to detect the relevance of transitions [16, 24, 32, 73]. Partial order reduction exploits that, without affecting completeness, independent transitions often need not be explored in all possible orderings. The different variants that have been proposed in the literature are based on similar ideas.

In the area of software model checking, Musuvathi and Qadeer [58] proposed an algorithm for detecting bugs in multithreaded programs which they called *iterative context bounding*. The idea is to avoid "jumping" during the state space exploration by giving preference to transitions that belong to the same thread as much as possible. We will describe this approach in more detail and compare it to our approaches in Chapter 4 where a similar idea is exploited.

1.5 Outline

The organization of this thesis is as follows. In Chapter 2, we provide the preliminaries that are needed for this work. In particular, we present our notation and formally introduce the distance heuristics from the directed model checking literature that will serve as a basis for our experimental evaluations. We finally introduce the various benchmark problems that are used in the experiments throughout this thesis.

The contribution of this thesis is the concept of *transition-based* directed model checking, which extends directed model checking by additionally computing pri-

orities for *transitions*, rather than only to compute priorities for *states*. We demonstrate that transition-based directed model checking can significantly improve the classical directed model checking approach on large and complex real-world benchmarks. The different techniques to compute priorities for transitions are presented in the subsequent chapters.

In Chapter 3, we introduce the concept of *useless transitions*, where we heuristically identify transitions that are not needed to find an error state of a system. To compute the property of uselessness, we use the distance heuristic that is also used for the exploration of the state space. We show that preferably applying non-useless transitions often improves directed model checking significantly [79]. Furthermore, we have adapted this concept to the area of AI planning. We demonstrate that useless transitions also improve existing planning techniques on a large number of benchmarks from the international planning competitions [78].

In Chapter 4, we introduce an alternative concept to useless transitions based on the notion of *interference* [77]. The motivation for this concept is to give preference to transitions that "profit" from previously applied transitions. Therefore, we introduce the notion of *interference contexts* and propose a multi-queue directed model checking algorithm. Compared to the Boolean property of useless transitions, interference contexts provide a more fine-grained concept to compute transition priorities. Moreover, the quality of the multi-queue directed model checking algorithm based on interference contexts is independent of the quality of the applied distance heuristic. In particular, it is successfully applicable to uninformed search.

With the concepts of useless transitions and interference contexts introduced in Chapter 3 and Chapter 4, we identify transitions that are expected to be not needed to find error states. However, although these concepts are very successful in practice, there is no theoretical guarantee that transitions identified with these concepts can *always* be pruned. Therefore, in order to not miss any error trace, we assign lower priorities to such transitions, but cannot prune them completely.

In Chapter 5, we present a technique for prioritizing transitions based on an abstraction principle that we call *safe abstraction* [76]. In contrast to the concepts of useless transitions and interference contexts, transitions identified with safe abstraction can be ignored completely during the exploration of the state space without affecting completeness. We prove that resulting spurious error traces can be efficiently extended to concrete error traces. As a further result, we present an adaptation of the causal graph heuristic [35, 36] from AI planning to directed model checking.

We finally present our model checker MCTA [56] in Chapter 6. MCTA is a directed model checking tool for timed automata systems in which we implemented the techniques presented in this thesis. We briefly introduce timed systems, describe MCTA's features, and provide an experimental evaluation of MCTA and UPPAAL, which is a state-of-the-art model checker for timed automata.

Most of the contributions of this thesis have been published. The relevant papers are presented below. They are listed in the order as they are described in the subsequent chapters.

- Martin Wehrle, Sebastian Kupferschmid, and Andreas 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)*, volume 5505 of *Lecture Notes in Computer Science*, pages 186–200. Springer-Verlag, 2009.
- Martin Wehrle, Sebastian Kupferschmid, and Andreas Podelski. Useless Actions are Useful, In *Proceedings of the 18th International Conference on Automated Planning and Scheduling (ICAPS 2008)*, pages 388–395. AAAI Press, 2008.
- Martin Wehrle and Sebastian Kupferschmid. Context-Enhanced Directed Model Checking, In *Proceedings of the 17th International SPIN Workshop on Model Checking Software (SPIN 2010)*, volume 6349 of *Lecture Notes in Computer Science*, pages 88–105. Springer-Verlag, 2010.
- Martin Wehrle and Malte Helmert. The Causal Graph Revisited for Directed Model Checking, In *Proceedings of the 16th International Symposium on Static Analysis (SAS 2009)*, volume 5673 of *Lecture Notes in Computer Science*, pages 86–101. Springer-Verlag, 2009.
- Sebastian Kupferschmid, Martin Wehrle, Bernhard Nebel, and Andreas Podelski. Faster than UPPAAL?, In *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.
- Sebastian Kupferschmid and Martin Wehrle. Abstractions and Pattern Databases: The Quest for Succinctness and Accuracy, In *Proceedings of the 17th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2011)*, volume 6605 of *Lecture Notes in Computer Science*, pages 276–290. Springer-Verlag, 2011.

2 Preliminaries

In this chapter, we give the preliminaries that are needed for this work. We introduce the notation, including our computational model, in Section 2.1. This is followed by a detailed introduction to directed model checking in Section 2.2. Moreover, we introduce distance heuristics from the literature in more detail in Section 2.3. Finally, we describe the benchmarks that are used in our experiments in Section 2.4.

2.1 Notation

Our computational model abstracts away from a concrete piece of software or hardware. In this work, we describe systems in terms of parallel processes. We focus on a model that is as simple as possible on the one hand, and sufficiently rich to capture the ideas of our contributions on the other hand. We define a process as a directed labeled graph consisting of a finite set of *locations* (or *local states*) and a finite set of *edges*. Edges are annotated with a *synchronization label*. Additionally, they can be guarded by integer constraints and can set integer variables to new values. In the following, let V be a finite set of bounded integer variables. The domain of $v \in V$ is denoted with $\mathrm{dom}(v)$. As we only consider bounded integer variables, we have $|\mathrm{dom}(v)| < \infty$ for all $v \in V$, and we will shortly speak of integer variables (instead of *bounded* integer variables) throughout this thesis. To formally define processes, we first need the notion of integer *guards* and *effects*.

Definition 2.1 (Integer Guard and Integer Effect).
Let V be a finite set of integer variables. Let $v \in V$ be an integer variable, $n \in \mathrm{dom}(v)$ be an integer value. An integer constraint *over V is a formula of the form $v \bowtie n$, where $\bowtie \in \{<, \leq, =, \geq, >\}$. An* integer guard *is a finite conjunction of integer constraints. An* integer assignment *over V is an expression of the form $v := n$. An* integer effect *is a finite set of integer assignments.*

With the notion of integer guards and integer effects, we define processes and systems in a compact way as follows. Let Σ be a finite set of synchronization labels containing a special (internal) void label $\tau \in \Sigma$.

Definition 2.2 (Process).
Let V be a finite set of integer variables. A process *p is a tuple (L, l, E), where L is a finite set of* locations *(or* local states*), $l \in L$ is the* initial *location, and $E \subseteq L \times Grd \times \Sigma \times Eff \times L$ is a finite set of* edges*, where Grd is the set of integer guards over V, and Eff is the set of integer effects over V. The synchronization label of an edge e is denoted with $label(e)$.*

We will write $l \xrightarrow{g,c,e} l'$ as a shorthand for an edge with source location l, destination location l', integer guard g, synchronization label c and integer effect e. When we want to stress that an edge belongs to process p, we will write $p.l \xrightarrow{g,c,e} l'$. A *system* is the parallel composition of processes that communicate via interleaved or synchronized edges and global integer variables.

Definition 2.3 (System).
Let $p_1, \ldots, p_n$ be processes with $p_i = (L_i, l_i, E_i)$ for $1 \leq i \leq n$, $L_i \cap L_j = \emptyset$ for $i \neq j$, $n \in \mathbb{N}$. A system *$\mathcal{M}$ is a tuple $(\mathcal{P}, V)$, where $\mathcal{P} = p_1 \parallel \ldots \parallel p_n$ is the parallel composition of $p_1, \ldots, p_n$, and V is the set of global integer variables with initial value $init(v_i) \in \mathrm{dom}(v_i)$ for all $v_i \in V$.*

As already outlined, the syntax of processes and systems as defined above is restricted, but sufficient to capture the ideas of the techniques from this thesis. We will see that all presented techniques can be extended to a richer class of systems as well (e. g., supporting linear arithmetic).

A *global state* of a system is formally defined as a valuation that indicates the current values of the system components. Therefore, a global state assigns a location to each process and value to each integer variable.

Definition 2.4 (Global State).
Let $\mathcal{M} = (\mathcal{P}, V)$ be a system with $\mathcal{P} = p_1 \parallel \ldots \parallel p_n$, $p_i = (L_i, l_i, E_i)$ for $1 \leq i \leq n$. A global state *of the system $\mathcal{M}$ is a valuation s that assigns each process p_i one of its locations of L_i, and each integer variable $v \in V$ a value of $\mathrm{dom}(v)$.*

The initial global state of a system is given by the valuation that maps the processes to their initial locations, and the integer variables to their initial values. We will shortly write $\langle p_1 = l_1, \ldots, p_n = l_n, v_1 = n_1, \ldots, v_m = n_m \rangle$ for a state to indicate that process p_i is in location l_i, and integer variable v_j has the value n_j. Analogously to the semantics of the propositional logic, a state s satisfies an integer constraint $v \bowtie n$, denoted with $s \models v \bowtie n$, if and only if $s(v) \bowtie n$ for

$\bowtie \in \{<, \leq, =, \geq, >\}$ and $n \in \mathbb{Z}$. A state s satisfies a *location constraint* $p = l$ for a process p and a location l, denoted with $s \models p = l$, if and only if $s(p) = l$. Furthermore, s satisfies a conjunction of constraints $\varphi_1 \wedge \cdots \wedge \varphi_n$ if and only if s satisfies every constraint φ_i for $1 \leq i \leq n$.

In the following, we formally define the operational semantics of systems in terms of transition systems. To simplify notation, we first explicitly introduce the notion of *transitions*.

Definition 2.5 (Transition).
Let $\mathcal{M} = (\mathcal{P}, V)$ be a system with $\mathcal{P} = p_1 \parallel \ldots \parallel p_n$, $p_i = (L_i, l_i, E_i)$ for $1 \leq i \leq n$. A transition *t of $\mathcal{M}$ is a set of edges, where either*

- *$t = \{e\}$, where $e \in E_i$ for some $i \in \{1, \ldots, n\}$, and $label(e) = \tau$, or*
- *$t = \{e_1, e_2\}$, where $e_1 \in E_i$, $e_2 \in E_j$ for some $i, j \in \{1, \ldots, n\}$, $i \neq j$, and $label(e_1) = label(e_2) \neq \tau$*

The size *$|t|$ of a transition t is defined as the number of edges in t. Note that $|t| \leq 2$ for each transition t: $|t| = 1$ for all internal τ transitions, $|t| = 2$ for binary synchronized transitions. The set of all transitions of $\mathcal{M}$ is denoted with $\mathcal{T}(\mathcal{M})$.*

Analogously to integer guards and integer effects, we define *transition guards* and *transition effects*. Informally speaking, the guard of a transition t is the conjunction of the integer guards and the location guards of t's edges, where the location guard of an edge e is a location constraint indicating that e must be in its source location. The effect of transitions is defined correspondingly as the union of all effects of the edges in t.

Definition 2.6 (Transition Guard and Transition Effect).
Let $\mathcal{M} = (\mathcal{P}, V)$ be a system with $\mathcal{P} = p_1 \parallel \ldots \parallel p_n$, $p_i = (L_i, l_i, E_i)$ for $1 \leq i \leq n$. Let t be a transition, and $e_j = (l_s^j, g_{int}^j, c, \mathit{eff}_{int}{}^j, l_d^j) \in t$ for $1 \leq j \leq |t|$. The transition guard *of t is defined as*

$$guard(t) = \bigwedge_{j=1}^{|t|} (g_{int}^j \wedge g_{loc}^j),$$

where the location guard g_{loc}^j for $e_j \in E_i$ is defined as $p_i = l_s^j$ (indicating that process p_i must be in the source location of e_j before applying t). The transition effect *of t is defined as*

$$effect(t) = \bigcup_{j=1}^{|t|} (\mathit{eff}_{int}^j \cup \mathit{eff}_{loc}^j),$$

where the location effect eff_{loc}^j for $e_j \in E_i$ is defined as $\{p_i := l_d^j\}$ (indicating that process p_i is in the destination location of e_j after applying t).

A transition t is applicable in a state s if the guard of t is satisfied by s. In this case, the successor state is obtained from s by changing the values of the variables in s according to t's effect.

Definition 2.7 (Transition Application).
Let $\mathcal{M} = (\mathcal{P}, V)$ be a system with $\mathcal{P} = p_1 \parallel \ldots \parallel p_n$, $p_i = (L_i, l_i, E_i)$ for $1 \leq i \leq n$. A transition $t \in \mathcal{T}(\mathcal{M})$ is applicable *in s iff $s \models guard(t)$. The* successor state *$t[s]$ of s under the application of t is obtained by setting the processes and integer variables to locations and values according to the effect of t, and retaining the others from s.*

The operational semantics of a system $\mathcal{M}$ is then defined as the state space induced by $\mathcal{M}$ consisting of the set of all global states. A global state s has a successor state s' if there is a transition t that is applicable in s and leads to s'.

Definition 2.8 (Operational Semantics).
Let $\mathcal{M}$ be a system. The operational semantics *$[\![\mathcal{M}]\!]$ of $\mathcal{M}$ is defined as the transition system (S, T), where*

- *S is the set of all global states of $\mathcal{M}$, and*
- *$T \subseteq S \times S$, where $(s, s') \in T$ iff there is a transition $t \in \mathcal{T}(\mathcal{M})$ such that t is applicable in s, and $s' = t[s]$.*

A trace *$\pi = s_0, t_0, s_1, \ldots, t_{n-1}, s_n$ is an alternating sequence of states and transitions such that t_{i-1} is applicable in s_{i-1}, and $t_{i-1}[s_{i-1}] = s_i$ for $1 \leq i \leq n$. The length $|\pi|$ of π is defined as the number of transitions in π, i. e., $|\pi| = n$ for the trace given above. A state s' is* reachable *from s if there is a trace from s to s'.*

As already outlined, in this thesis, we address the problem of detecting *error states* in a given system. Error states are defined as states that are reachable from the initial global state which have an undesirable property. In terms of CTL [28], finding error states corresponds to proving the formula $EF\varphi$, where φ represents an *error formula* that describes an undesirable property. This is equivalent to the falsification of invariants of a system, i. e., to disproving the CTL formula $AG\neg\varphi$. Traces that end in an error state are called *error traces*. The *error distance* $d(s)$ of a state s is defined as the length of a shortest error trace from s. In this thesis, we consider error formulas $\varphi = \varphi_1 \wedge \cdots \wedge \varphi_n$ that are conjunctions of constraints. Each constraint φ_i has the form $p = l$ for a process $p = (L, E)$ and a location $l \in L$ (indicating that p is in l) or $v = n$ for an integer variable v and $n \in \mathrm{dom}(v)$ (indicating that v has the value n). Overall, a model checking task consists of a system $\mathcal{M}$ together with an error formula φ. Without loss of generality, we assume that φ is consistent, i. e., $\varphi \not\models \bot$.

Definition 2.9 (Model Checking Task).
A model checking task *is a tuple* $\Theta = (\mathcal{M}, \varphi)$*, where* $\mathcal{M}$ *is a system, and* φ *is an error formula with* $\varphi \not\models \bot$*. The task is to find an error trace in* Θ*, i. e., a trace from the initial global state of* $\mathcal{M}$ *that ends in a global state satisfying* φ*.*

2.2 Directed Model Checking

In this section, we introduce directed model checking in more detail. Directed model checking is a variant of explicit state model checking that is optimized towards finding error states in concurrent systems efficiently. To efficiently find error states, the state space exploration is guided by a distance heuristic $d^{\#}$. The distance heuristic is usually computed automatically based on the declarative description of the system. In a nutshell, $d^{\#}$ is a function that maps global states s to natural numbers, which in turn provide an estimate of the error distance $d(s)$ of s. Typically, this estimate is the length of a corresponding abstract error trace that starts in a corresponding abstract state $s^{\#}$. More precisely, for a given abstraction principle, the distance estimates are computed by first abstracting the concrete model checking task $\Theta = (\mathcal{M}, \varphi)$ to the abstract model checking task $\Theta^{\#} = (\mathcal{M}^{\#}, \varphi^{\#})$. The distance estimate $d^{\#}(s)$ for a global state s is then defined as the length of an abstract error trace in $\Theta^{\#}$ starting in $s^{\#}$, where $s^{\#}$ is a corresponding abstract state to s. For each encountered global state during directed model checking, a priority value is computed based on $d^{\#}$, where global states with lower priority values are preferably explored.

The overall behavior of directed model checking is determined by two parameters. The first parameter is the abstraction that is used to compute abstract error distances. This parameter determines the precision of the distance heuristic $d^{\#}$. The second parameter is the search algorithm that determines how the priority values are computed based on $d^{\#}$. In other words, the search algorithm specifically determines how the distance estimates given by $d^{\#}$ are exploited during the state space exploration. Figure 2.1 shows a basic directed model checking algorithm which defines the priority values simply as the values given by $d^{\#}$. (Other search algorithms like A* compute the priority values differently – we omit a more detailed description because such algorithms are not relevant for the thesis.) The algorithm in Figure 2.1 is known as *greedy search*. Given a model checking task $(\mathcal{M}, \varphi)$ and a distance heuristic $d^{\#}$, the algorithm returns *False* if there is a state that satisfies the error condition represented by φ; otherwise it returns *True*. In the former case, an error trace is generated by back-tracing from the error state (therefore, in a practical implementation, every state additionally stores information about how it has been reached).

For the description of the algorithm, we use the following terminology. A state s is called *visited* if s has been encountered during the search, but the successor

```
1  function verify(M, φ, d#):
2      s0 = initial state of M
3      open = empty priority queue
4      closed = ∅
5      priority = d#(s0)
6      open.insert(s0, priority)
7      while open ≠ ∅ do:
8          s = open.pop-minimum()
9          if s ⊨ φ then:
10             return False
11         closed = closed ∪ {s}
12         for each transition t ∈ T(M) with s ⊨ guard(t) do:
13             s' = t[s]
14             if s' ∉ closed then:
15                 priority = d#(s')
16                 open.insert(s', priority)
17     return True
```

Fig. 2.1. A basic directed model checking algorithm.

states of s have not yet been computed. Moreover, a state s is called *explored* if the successor states of s have been computed. For the exploration of the state space, the algorithm maintains a priority queue *open* which contains states that are visited, but not yet explored. Through the method *open.pop-minimum*, a visited state s with minimum priority value is determined and removed from *open*. In the following, the algorithm first checks if s is an error state; if this is the case, an error trace is generated as described above. If s is not an error state, s is marked as explored by adding it to the *closed* set; this ensures that s is not considered again for exploration later. Finally, the successor states of s that are not already explored are added to *open*. Overall, the exploration process described above repeats with a new state from *open* with minimum priority value, until an error state is encountered or *open* becomes empty. In the latter case, we can conclude that no error state can be reached.

2.3 Abstraction Based Distance Heuristics

In this section, we introduce distance heuristics from the literature that are used as a basis for our experimental evaluations. These include the distance heuristics d^L and d^U based on the graph distance [23, 25] as well as the distance heuristics h^L and h^U based on the monotonicity abstraction [52].

Distance Heuristics Based on Graph Distances

In this section, we describe the distance heuristics d^L and d^U that have been conceptually proposed by Edelkamp et al. [23, 25]. For a model checking task $\Theta = (\mathcal{M}, \varphi)$

with system $\mathcal{M} = (\mathcal{P}, V)$ and $\mathcal{P} = p_1 \parallel \ldots \parallel p_n$, both d^L and d^U are based on the local graph distances to error locations in each process. More precisely, we define a location l of p_i to be an *error location* if the error formula φ contains the constraint $p_i = l$ as a conjunct. For a global state s, let $dist_i(s)$ denote the graph distance from p_i's current location to p_i's error location in s. For processes p_i that do not contain error locations, the graph distance is defined as zero, i. e., $dist_i(s) = 0$. The maximum graph distance heuristic d^L is defined as

$$d^L(s) = \max_{i=1,\ldots,n} dist_i(s).$$

The distance heuristic d^U sums these values, therefore

$$d^U(s) = \sum_{i=1}^{n} dist_i(s).$$

These distance heuristics are cheap to compute on the one hand, but are rather coarse approximations of the real error distance on the other hand. Synchronization behavior and integer variables of $\mathcal{M}$ are ignored completely.

Distance Heuristics Based on the Monotonicity Abstraction

In this section, we describe the distance heuristics h^L and h^U that have been proposed by Kupferschmid et al. [52]. As already outlined in Section 1.4, they are based on the *monotonicity abstraction* that abstracts the semantics of the system by assuming that no negative transition effects occur. Therefore, variables become set-valued, and once a variable obtained a value, it keeps this value forever. Roughly speaking, h^L and h^U compute abstract error traces under this abstraction. The distance estimate is obtained by the length of such abstract error traces. For the following considerations, let $\Theta = (\mathcal{M}, \varphi)$ be a model checking task with system $\mathcal{M} = (\mathcal{P}, V)$ and error formula φ. Let $\mathcal{P} = p_1 \parallel \ldots \parallel p_n$ and $V = \{v_1, \ldots, v_m\}$.

The semantics of the monotonicity abstraction is defined as follows. An *abstract state* $s^\#$ is a valuation of the processes and variables that assigns each process $p_i = (L_i, l_i, E_i)$ of $\mathcal{P}$ a subset $L'_i \subseteq L_i$ of its location set, and each integer variable $v_j \in V$ a subset $V'_j \subseteq \mathrm{dom}(v_j)$ of its domain. A transition $t \in \mathcal{T}(\mathcal{M})$ is *applicable* in the abstract state $s^\# = \langle p_1 = L'_1, \ldots, p_n = L'_n, v_1 = V'_1, \ldots, v_m = V'_m \rangle$ if there are values $l'_1 \in L'_1, \ldots, l'_n \in L'_n$ and $n'_1 \in V'_1, \ldots, n'_m \in V'_m$ such that t is applicable in the state $s = \langle p_1 = l'_1, \ldots, p_n = l'_n, v_1 = n'_1, \ldots, v_m = n'_m \rangle$, i. e., if $s \models guard(t)$. The successor state $t[s^\#]$ is defined as the abstract state obtained from $s^\#$ where the effects of t are unified with $s^\#$ accordingly: for location assignments $p_i := l$, we have $L'_i \cup \{l\}$ in $t[s^\#]$. For integer assignments $v_j := n$, we have $v_j \cup \{n\}$ in $t[s^\#]$.

Using this semantics, the h^L distance heuristic is computed as follows. Let $s = \langle p_1 = l_1, \ldots, p_n = l_n, v_1 = n_1, \ldots, v_m = n_m \rangle$ be a global state of $\mathcal{M}$. To compute $h^L(s)$, s is first mapped to the corresponding initial abstract state $s_0^{\#} = \langle p_1 = \{l_1\}, \ldots, p_n = \{l_n\}, v_1 = \{n_1\}, \ldots, v_n = \{n_m\} \rangle$ where the variables are set-valued and initially contain the values determined by s. The distance estimate $h^L(s)$ is defined as the minimum number of parallel transition applications until an abstract state is reached such that the error formula φ is satisfied, or a fixed-point is reached where φ is not satisfied. More precisely, in every iteration i, transitions that are applicable in the current abstract state $s_i^{\#}$ are first identified. The abstract successor state $s_{i+1}^{\#}$ is then obtained by applying all of these transitions in $s_i^{\#}$ (note that the order does not matter as transition application corresponds to unions of sets). The algorithm terminates either if an abstract state has been reached that satisfies the error formula, or if a fixed-point is reached. In the former case, the number i of iterations is returned as distance estimation. In the latter case, no abstract error state is reachable. As the monotonicity abstraction yields an over-approximation of the original semantics of the system, no concrete error state is reachable from s either, and therefore, infinity is returned. We finally remark that the algorithm is guaranteed to terminate as we are dealing with finite systems.

The h^U distance heuristic is also based on the monotonicity abstraction and performs the same fixed-point computation as h^L. In addition, it extracts an abstract error trace from the sequence of abstract states $s_0^{\#}, \ldots, s_n^{\#}$ obtained during the computation of h^L. Essentially, for each abstract state $s_i^{\#}$, a subset T' of the applicable transitions is computed such that applying the transitions from T' still yields an abstract error trace $\pi^{\#}$. The distance estimate $h^U(s)$ for a concrete state s is defined as the length of $\pi^{\#}$. For a more detailed description, we refer the reader to the literature [53]. We remark that h^U is not admissible, i. e., the abstract error traces that are found by h^U may be longer than the shortest ones. Therefore, we are not guaranteed to find shortest possible error traces with h^U and A*. However, although not admissible, we also remark that the additional computation compared to h^L often yields a more precise distance estimation than h^L. In a greedy search setting where admissibility is not required, this often leads to a better overall performance of directed model checking in terms of scalability and runtime.

2.4 Benchmark Set

In this section, we briefly introduce the benchmarks that are used in this thesis. Most of them stem from the AVACS[1] benchmark suite. The descriptions of the

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

benchmark problems mostly refer to the papers where they have been originally described [20, 52] or to the website of AVACS. In addition to the computational model as introduced in Section 2.1, some of them feature clock variables and represent timed automata [3]. Edges can be additionally guarded by clock constraints and reset clock variables as effects. Although the techniques introduced in this thesis are basically developed for untimed systems, they can be adapted to systems of timed automata as well. We will describe these adaptations in the corresponding chapters where our contributions are presented.

Single-Tracked-Line-Segment case study

The Single-Tracked-Line-Segment benchmarks come from a case study from an industrial project partner of the UniForM-project [49]. It models a distributed real-time controller for a segment of tracks where trams share a piece of track. The overall system is modeled as PLC automata [19, 49] and translated to timed automata with the tool Moby/RT [62]. The problem instances of this case study are translated abstractions of the overall system of increasing size and complexity. For the evaluation of the techniques presented in this thesis, we chose the property that never both directions are given permission to enter the shared segment simultaneously. However, a subtle error has been inserted by manipulating a delay so that the asynchronous communication between these automata is faulty.

Mutual-Exclusion case study

This case study models a real-time protocol that is supposed to ensure mutual exclusion in a parallel system of timed automata. The processes of the system communicate in an asynchronous way. The protocol is described in full detail by Dierks [19]. As in the case study on the Single-Tracked-Line-Segment, a subtle timing error has been inserted.

Arbiter-Tree case study

The *arbiters* case study [71] establishes mutual exclusion between several client processes. The benchmarks contain arbiter trees of increasing height, with an exponentially growing number of processes. Client processes are situated at the leaves of the tree. An error has been inserted that allows several client processes to access a resource simultaneously.

Towers-Of-Hanoi case study

These benchmarks model the Towers of Hanoi problem for a varying number of disks. The index of the examples gives the number of involved disks. Initially, all n disks are on the first peg. The goal is to move all disks to the second peg, where only one disk can be moved at a time point. Furthermore, we have the constraint that never a larger disk is on top of a smaller one.

Fischer Protocol

As a further set of benchmarks, we use *Fischer protocol* examples which again is a mutual exclusion protocol [57]. The error condition is that at least two of the processes of the system are simultaneously in a location that represents the critical section. An error has been inserted by weakening one of the conditions in the processes.

Planning benchmarks

We finally remark that one of the techniques presented in this thesis has also been successfully introduced in the area of AI planning, where we have evaluated it on planning problems from the international planning competitions. Such problems are often motivated by real world examples. As they are needed only once in this thesis, we will come back to these benchmarks in the corresponding chapter.

3

Useless Transitions

In this chapter, we present a concept for prioritizing transitions that we call "useless transitions". The chapter is based on joint work with Kupferschmid and Podelski [78, 79]. It is organized as follows. In the next section, we motivate the general problem in more detail and show the potential of prioritizing transitions. Afterwards, we formally introduce our concept and present an extended directed model checking algorithm in Section 3.2. After a discussion about strengths and weaknesses in Section 3.3, we empirically evaluate our concept in Section 3.4. We finally discuss related work in Section 3.5 and conclude the chapter in Section 3.6.

3.1 Motivation

As outlined in the introduction of this thesis, directed model checking has proved to be able to substantially outperform uninformed search methods such as depth-first search when the aim is to find global error states in concurrent systems. However, as we have also already discussed in the last chapter, the quality of directed model checking crucially depends on the applied distance heuristic. If the distance estimates are not informed enough, the overall search process may degenerate even to uninformed search. In the following, we provide a motivating example to show that additionally prioritizing transitions, rather than only prioritizing states, can significantly improve classical directed model checking.

Example 3.1. The example model checking task depicted in Figure 3.1 consists of a system $\mathcal{M}$ with $n+1$ parallel processes $p_0, \dots, p_n$. The initial global state of the system is given by the state $\langle p_0 = l_0, \dots, p_n = l_0 \rangle$ where every process is in its initial location. The global error state is given by the state $\langle p_0 = l_1, \dots, p_n = l_1 \rangle$ where every process is in its error location (indicated with a double circle). The transitions of $\mathcal{M}$ are interleaving transitions with no binary synchronization, i. e., $label(t) = \tau$ for all $t \in \mathcal{T}(\mathcal{M})$. For the sake of readability, we omit these labels in Figure 3.1.

Suppose that we apply directed model checking to check if this error state is reachable in $\mathcal{M}$. Further suppose that we therefore use the maximum graph distance heuristic as proposed by Edelkamp et al. [23, 25]. Recall that this function is based on the local distance of each single process p_i. For the global state s, let $dist_i(s)$ denote the graph distance from p_i's current location to p_i's error location in s. Then the maximum graph distance heuristic is defined as $d^L(s) = \max_{i=0,\ldots,n+1} dist_i(s)$.

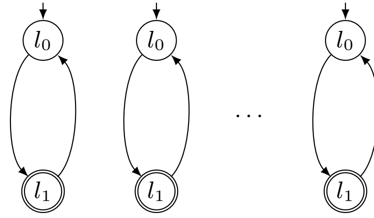

Fig. 3.1. An example system with $n + 1$ processes

For this problem, it turns out that the maximum graph distance is a rather uninformed distance heuristic. In most cases, it cannot distinguish global states that are nearer to the global error state from others. If at least one process is not yet in its error location l_1 in a global state s, then the estimated distance value of s is 1. We may characterize the state space topology induced by this distance heuristic as follows. There is one single plateau, because for all of the 2^{n+1} reachable global states but for the error state the abstract distance is 1. This effectively means that directed model checking with the maximum graph distance heuristic d^L completely degenerates to uninformed search in this case. Generally, as distance heuristics only *estimate* the distance to error states, there are systems where similar situations arise for every distance heuristic.

In this example, it is trivial to see that each transition where error locations are left should be avoided as much as possible during the exploration of the state space. Intuitively, in every global state, applying such transitions is useless to find the error state. Without steps corresponding to such transitions, the exploration of the state space stops after $n + 1$ steps and returns a shortest possible error trace. Although it is easy to see here which transitions should be less preferred in a given system state, it should also be obvious that this is no longer trivial to see when problems become larger and more complex. In this chapter, we address the question how to automatically identify such transitions in general.

3.2 Useless Transitions

In this section, we present the central concept of this chapter. We start by defining the notion of useless transitions in Section 3.2.1. This definition precisely captures

the intuition, but is computationally hard. Therefore, we present the notion of relatively useless transitions as an approximation in Section 3.2.2. Finally, we propose an extended directed model checking algorithm based on relatively useless transitions in Section 3.2.3.

3.2.1 Useless Transitions

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

Definition 3.2 (Useless Transition).
Let $(\mathcal{M}, \varphi)$ be a model checking task with system $\mathcal{M}$ and transition set $\mathcal{T}(\mathcal{M})$. A transition $t \in \mathcal{T}(\mathcal{M})$ is useless *in a global state s iff no shortest trace from s to a nearest error state starts with t.*

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 $|\pi'| < n$. When we want to stress that d is a function also on the system $\mathcal{M}$, we will write $d(\mathcal{M},\, s)$.

By Definition 3.2, a transition t is useless in a global state s if and only if the real error distance d does not decrease, i. e., a transition from s to s' is useless iff $d(s) \leq d(s')$. To see this, observe that $d(s) \leq d(s') + 1$ for every state s, every transition t that is applicable in s and successor state $s' = t[s]$. If a shortest error trace starts from s with t, then $d(s) = d(s')+1$. Otherwise, the error distance does not decrease and therefore $d(s) < d(s') + 1$. Since the distance values are all integers, this is equivalent to $d(s) \leq d(s')$. In the following, we will use the inequality $d(s) \leq d(s')$ in connection with the idea of *removing* transitions to derive a sufficient criterion for uselessness of transitions. Therefore, we will define the notion of *reduced systems*. Intuitively, the reduced system with respect to a transition t is defined as the system that is obtained by removing the edges contained in t (recall that a transition of a system either consists of *one* interleaving edge of a process, or of *two* binary synchronized edges of two different processes).

Definition 3.3 (Reduced System).
Let $\mathcal{M} = (\mathcal{P}, V)$ be a system with $\mathcal{P} = p_1 \parallel \ldots \parallel p_n$, $p_i = (L_i, l_i, E_i)$ for $1 \leq i \leq n$. Let $t \in \mathcal{T}(\mathcal{M})$ be a transition of $\mathcal{M}$. The reduced system *of $\mathcal{M}$ with respect to t is defined as the system*

$$\mathcal{M}_t = (p'_1 \parallel \ldots \parallel p'_n, V),$$

where $p'_i = (L_i, l_i, E_i \setminus t)$.

Note that, according to the definition of t, at most two processes are affected to obtain a reduced system with respect to t. However, in the operational semantics $[\![\mathcal{M}]\!] = (S, T)$ of $\mathcal{M}$, t generally corresponds to *several* $t_1, \ldots, t_n \in T$. This is illustrated in the following example.

Example 3.4. Consider the system $\mathcal{M}$ in Figure 3.2 that consists of the two processes p_1 and p_2 and the corresponding induced transition system $[\![\mathcal{M}]\!] = (S, T)$ that defines the semantics of $\mathcal{M}$. The dashed transition $t = \{p_1.l_0 \xrightarrow{\tau} l_0^*\}$ corresponds to $t_1, t_2 \in T$, where $t_1 = (\langle l_0, l_1\rangle, \langle l_0^*, l_1\rangle)$ and $t_2 = (\langle l_0, l_1^*\rangle, \langle l_0^*, l_1^*\rangle)$. The reduced system $\mathcal{M}_t$ is defined as the system where the edge $l_0 \xrightarrow{\tau} l_0^*$ is removed from p_1, which obviously corresponds to removing t_1 and t_2 from T.

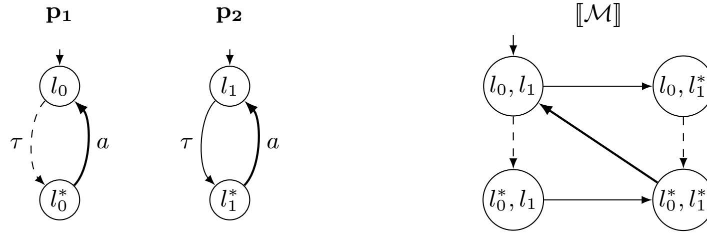

Fig. 3.2. Example system and corresponding operational semantics

Moreover, the (binary synchronized) transition $t' = \{p_1.l_0^* \xrightarrow{a} l_0, p_2.l_1^* \xrightarrow{a} l_1\}$ corresponds to $t_3 = (\langle l_0^*, l_1^*\rangle, \langle l_0, l_1\rangle)$ in T. The reduced system $\mathcal{M}_{t'}$ is obtained by removing the edges $l_0 \xrightarrow{a} l_0^*$ from p_1 and $l_1 \xrightarrow{a} l_1^*$ from p_2, which obviously corresponds to removing t_3 from T.

Based on the definition of reduced systems, we will provide a sufficient criterion to check transitions for uselessness. The criterion is based on the following idea. Assume there is a shortest error trace π from a state s such that a transition t does not occur in π. If π is not longer than the error traces from the successor state $t[s]$ obtained by applying t in s, then no shortest error trace from s starts with t. This is stated in the following proposition.

Proposition 3.5. *Let* $(\mathcal{M}, \varphi)$ *be a model checking task,* s *be a global state in* $\mathcal{M}$*, and* $t \in \mathcal{T}(\mathcal{M})$ *be a transition that is applicable in* s*. If* $d(\mathcal{M}_t, s) \leq d(\mathcal{M}, t[s])$*, then* t *is useless in* s*.*

Proof. If $d(\mathcal{M}_t, s) \leq d(\mathcal{M}, t[s])$, then $d(\mathcal{M}, s) \leq d(\mathcal{M}, t[s])$ because $d(\mathcal{M}, s) \leq d(\mathcal{M}_t, s)$ (the error distance cannot decrease in reduced systems because removing transitions induces underapproximations). As $d(\mathcal{M}, s) \leq d(\mathcal{M}, t[s])$ iff t is useless in s, the claim follows directly. □

The criterion given by Proposition 3.5 can be interpreted as follows. A transition t is useless in a state s if the error state is still reachable from s on the same shortest trace in the underapproximation that is obtained by *removing* t from the system. Note that the back direction of Proposition 3.5 does not hold. To see this, consider the example model checking task $(\mathcal{M}, \varphi)$ with $\mathcal{M} = (\mathcal{P}, V)$, $\mathcal{P} = p$, $V = \{a\}$ and error formula φ that is defined as $p = l_E$. The model checking task is depicted in Figure 3.3. The transitions of $\mathcal{M}$ are interleaving transitions, i. e., $label(t) = \tau$ for all $t \in \mathcal{T}(\mathcal{M})$. For the sake of readability, these labels are omitted in Figure 3.3.

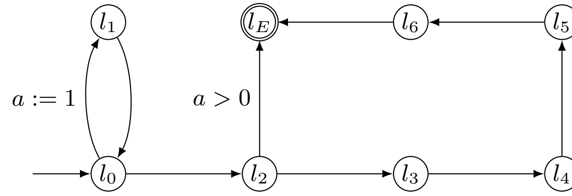

Fig. 3.3. Example system where the back direction of Proposition 3.5 does not hold

The model checking task consists of a system $\mathcal{M}$ with one process p with error location l_E and an integer variable a. The initial global state s_0 of $\mathcal{M}$ is defined as $\langle p = l_0, a = 0\rangle$. There are several traces that end in the error location l_E. The shortest error trace is obtained by first setting a to 1 by applying the transition that leads to l_1, then going back to l_0 and taking the “shortcut” from l_2 to l_E (note that this is possible because the guard $a > 0$ is satisfied in this case). The resulting error trace has length 4. A longer error trace leads from l_0 over locations $l_2, \ldots, l_6$. In particular, we observe that the transition $t = \{l_0 \to l_2\}$ is useless in the initial state because no shortest error trace in s_0 starts with t. Nevertheless, we have $d(\mathcal{M}_t, s_0) > d(\mathcal{M}, t[s_0])$. The error distance $d(\mathcal{M}_t, s_0)$ of the initial state of the reduced system with respect to t is infinity because t is still needed in the future. Moreover, the error distance of the successor state is finite, as it is still possible to reach the error location.

We remark that the condition for a transition to be useless in a state s given by Proposition 3.5 has some intricacies. To illustrate this, consider the two example model checking tasks in Figure 3.4. Both example tasks consist of a system $\mathcal{M}$ with one process p, the initial state $s_0 = \langle p = l_0\rangle$, and the error condition φ which is defined as $p = l_3$ (i. e., error states are those states where p is in its double circled location l_3). The transitions of these systems are interleaving transitions, i. e., $label(t) = \tau$ for all $t \in \mathcal{T}(\mathcal{M})$. For the sake of readability, these labels are omitted in Figure 3.4. We observe that, for a transition t to be useless in a state s, it is *not* enough to require that $d(\mathcal{M}_t,\, s) \leq d(\mathcal{M},\, s)$ holds instead of $d(\mathcal{M}_t,\, s) \leq d(\mathcal{M},\, t[s])$. To see this, consider the left example system in Figure 3.4. In this

process, the task is to reach the error location l_3 from l_0 via l_1 *or* l_2. The transitions $\{l_0 \rightarrow l_1\}$ and $\{l_0 \rightarrow l_2\}$ would both be wrongly classified as useless in the initial state if we required $d(\mathcal{M}_t, s_0) \leq d(\mathcal{M}, s_0)$ for a transition t to be useless in s_0.

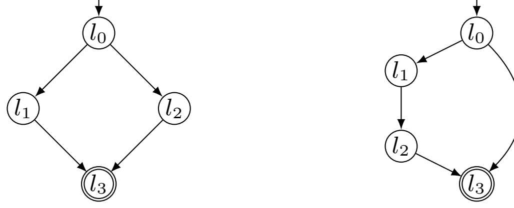

Fig. 3.4. Example model checking tasks

Furthermore, it does *not* suffice to only require that $d(\mathcal{M}_t, s) = d(\mathcal{M}, t[s])$ holds instead of $d(\mathcal{M}_t, s) \leq d(\mathcal{M}, t[s])$. To illustrate this, consider the right example system in Figure 3.4. In this system $\mathcal{M}$, the transition $t = \{l_0 \rightarrow l_1\}$ would not be classified as useless in the initial state $s_0 = \langle p = l_0 \rangle$ because $d(\mathcal{M}_t, s_0) = 1$, but $d(\mathcal{M}, t[s_0]) = 2$. However, t is useless in s_0 because it is not part of a shortest error trace from l_0 to l_3.

Finally, let us have a look at the complexity of the computation if a given transition t is useless in a global state s. We observe that deciding if t is useless in s is computationally hard.

Proposition 3.6. *Let $\mathcal{M}$ be a system with the property that an error trace with polynomially bounded length in the size of $\mathcal{M}$ exists. Let s be a global state in $\mathcal{M}$ and $t \in \mathcal{T}(\mathcal{M})$ be a transition in $\mathcal{M}$ that is applicable in s. If $P \neq NP$, then deciding if t is useless in s cannot be done in polynomial time in the size of $\mathcal{M}$.*

Proof. Assume the opposite, i. e., assume that deciding if t is useless in s can be done in polynomial time. The error trace existence problem in systems with polynomially bounded error trace length is NP-complete: NP hardness is shown by reduction from the satisfiability problem of classical propositional logic to the error trace existence problem; for the membership in NP, observe that the validity of polynomially bounded error traces can be checked in polynomial time (for more details, see, e. g., [68]). However, if we could decide in polynomial time if a transition is useless in a state, this would lead to a directed model checking algorithm that finds a shortest error trace in $\mathcal{M}$ in polynomial time: in every global state, decide in polynomial time which transition is not useless, and follow these transitions. This would contradict the assumption that $P \neq NP$. □

3.2.2 Relatively Useless Transitions: The Practical Counterpart

We have observed that the definition of useless transitions is computationally too hard to be efficiently applied in practice. A direct way to approximate the test for uselessness motivated by Proposition 3.5 is to use a given distance heuristic $d^{\#}$ instead of d. This is rational because $d^{\#}$ is designed for exactly the reason of approximating d. When we want to stress that $d^{\#}$ is a function also on the system $\mathcal{M}$, we will write $d^{\#}(\mathcal{M},\, s)$.

Definition 3.7 (Relatively Useless Transition).
Let $(\mathcal{M}, \varphi)$ be a model checking task, s be a global state in $\mathcal{M}$, and $t \in \mathcal{T}(\mathcal{M})$ be an applicable transition in s. Let $d^{\#}$ be a distance heuristic. Then t is relatively useless *for $d^{\#}$ in s if $d^{\#}(\mathcal{M}_t,\, s) \leq d^{\#}(\mathcal{M},\, t[s])$.*

Note that this is exactly the criterion from Proposition 3.5 where the real error distance d has been replaced by the estimated error distance $d^{\#}$. Obviously, the quality of this approximation strongly depends on $d^{\#}$'s precision. On the one hand, the very uninformed distance heuristic that constantly returns zero recognizes every transition as relatively useless in every global state, and no further information gain is obtained. On the other hand, the more sophisticated the distance estimations provided by $d^{\#}$, the more precise will be the approximation with relatively useless transitions. We will come back to this point in the discussion and experimental evaluation section. Intuitively, taking a relatively useless transition t does not seem to guide the exploration of the state space towards an error state as the *stricter* distance estimate in $\mathcal{M}_t$ does not increase.

One would expect that transitions should not be relatively useless if they lead to global states that are estimated to be nearer to an error state. Indeed, under the assumption that a distance heuristic $d^{\#}$ never decreases its distance estimate in reduced systems, i. e., $d^{\#}(\mathcal{M}, s) \leq d^{\#}(\mathcal{M}_t, s)$ for all systems $\mathcal{M}$, transitions $t \in \mathcal{T}(\mathcal{M})$ and global states s in $\mathcal{M}$, transitions leading to global states with better distance estimates are *never* relatively useless in any system $\mathcal{M}$. This is stated more formally in the following proposition.

Proposition 3.8. *Let $(\mathcal{M}, \varphi)$ be a model checking task with system $\mathcal{M}$ and transition set $\mathcal{T}(\mathcal{M})$. Let $d^{\#}$ be a distance heuristic such that $d^{\#}(\mathcal{M},\, s) \leq d^{\#}(\mathcal{M}_t,\, s)$ for all global states s and all transitions $t \in \mathcal{T}(\mathcal{M})$. Let s be a global state of $\mathcal{M}$ and $t \in \mathcal{T}(\mathcal{M})$ be a transition that is applicable in s. If $d^{\#}(t[s]) < d^{\#}(s)$, then t is* not *relatively useless for $d^{\#}$ in s.*

Proof. Assume that t is relatively useless for $d^{\#}$ in s, i. e., assume that $d^{\#}(\mathcal{M}_t,\, s) \leq d^{\#}(\mathcal{M},\, t[s])$. By assumption, the distance estimate with $d^{\#}$ does not decrease in the reduced system $\mathcal{M}_t$, i. e., $d^{\#}(\mathcal{M},\, s) \leq d^{\#}(\mathcal{M}_t,\, s)$. Therefore, we have

$d^{\#}(\mathcal{M}, s) \leq d^{\#}(\mathcal{M}, t[s])$, showing that the distance estimate does not decrease when the relatively useless transition t is applied in s. □

This result is rational and shows that our definition of relatively useless transitions makes sense. Roughly speaking, it shows that under the assumptions of Proposition 3.8, we do not need the notion of relatively useless transitions in a state s if there is a better successor state s' according to $d^{\#}$. However, if this is not the case, e. g., in situations during the search where every successor state has the same estimated distance value than the current state, the notion of relatively useless transitions can serve as an additional source of information and therefore improve the search behavior. We will show in the next section how to exploit this information during the exploration of the state space.

3.2.3 Directed Model Checking with Relatively Useless Transitions

In this section, we put the pieces together. So far, we have presented the notion of relatively useless transitions to identify transitions that should be less preferred during the exploration of the state space. A direct way to integrate this information to directed model checking is to additionally "penalize" states that result from applying such a transition. This is rational 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 a relatively useless transition should be less preferred when exploring the state space.

As argued in the beginning of this thesis, there are two choices to be made when the directed model checking approach is applied, namely choosing the underlying abstraction for the distance heuristics, and choosing the algorithm for the exploration of the state space. Here, we assume that a distance heuristic $d^{\#}$ is already given, and $d^{\#}$ is additionally used to determine relatively useless transitions. For the second point, we propose an extension of the basic directed model checking algorithm as given in Chapter 2 in Figure 2.1 on page 16.

Taking into account the considerations above, we extend the basic algorithm as follows. In addition to the standard open queue, we use a second open queue *deferred* which is used to maintain states that are reached by a relatively useless transition. In this sense, the deferred queue maintains relatively "useless states" that seem to be not needed according to our definition of relatively useless transitions. This deferred queue is only accessed if the standard priority queue is empty. By doing so, completeness of the algorithm is preserved, but there is a strong preference to explore states that are reached by a non-useless transition. The algorithm is given in Figure 3.5.

The most important extension compared to the basic directed model checking algorithm is shown in line 20, where every applied transition t is checked if it is

```
1  function UT-search(M, φ, d#):
2      s0 = initial state of M
3      open = empty priority queue
4      deferred = empty priority queue
5      closed = ∅
6      priority = d#(s0)
7      open.insert(s0, priority)
8      while open ∪ deferred ≠ ∅ do:
9          if open ≠ ∅ then:
10             s = open.pop-minimum()
11         else:
12             s = deferred.pop-minimum()
13         if s ⊨ φ then:
14             return False
15         closed = closed ∪ {s}
16         for each transition t ∈ T(M) with s ⊨ guard(t) do:
17             s' = t[s]
18             if s' ∉ closed then:
19                 priority = d#(s')
20                 if t is relatively useless for d# in s then:
21                     deferred.insert(s', priority)
22                 else:
23                     open.insert(s', priority)
24     return True
```

Fig. 3.5. The extended directed model checking algorithm based on useless transitions

relatively useless in the current state s for the distance heuristic that is also used for the search. If this is the case, the corresponding successor state s' is pushed into the deferred queue, which is only accessed if open is empty. If t is not relatively useless in s, then s' is pushed into the open queue as before. As already outlined above, this leads to a strong preference to explore states that have been reached by promising, i. e., non-useless transitions. Obviously, the number of explored deferred states indicates the precision of our approach: the higher the number of explored deferred states, the higher the number of misclassifications, i. e., the number of wrongly deferred states. We will come back to this point in the experimental section. Finally, we remark that if $d^{\#}$ can be computed automatically (which is the case for most of the recently proposed distance heuristics), our algorithm is also fully automatic, and no further user input is required.

3.3 Discussion: Where does it work, where not?

So far, we have motivated and introduced the notion of uselessness for transitions, and proposed a criterion to approximate this notion with a distance heuristic $d^{\#}$. However, the quality of the approximation is dependent on the accuracy of $d^{\#}$. In

this section, we discuss strengths and weaknesses of relatively useless transitions. In particular, an important question that arises in the context of relatively useless transitions is the question of the practical usability. To identify a class of distance heuristics for which our technique is suited best, recall that distance heuristics as considered in the literature are either computed on-the-fly (i. e., an abstracted model checking task is solved in every global state) or in a preprocessing step prior to directed model checking. Distance heuristics that belong to the latter class are mainly organized as a lookup table with the property that heuristic values are obtained with a simple table lookup. However, the computation of this table often needs considerable time.

Relatively useless transitions seem to be best suited for distance heuristics that are computed on-the-fly because the time overhead to compute this information is comparatively low. Contrarily, distance heuristics from the second class are less suited because for every reduced system, an additional lookup table has to be computed (recall that for the computation of the useless-values, the system is modified and the distance value is recomputed on this modified system). Although relatively useless transitions could be computed for distance heuristics belonging to that class as well, it is not obvious how to do this *efficiently*. This will be an interesting and important topic for future research. As we will see in our empirical evaluation, for distance heuristics that are computed on-the-fly, the overall model checking performance can often be significantly improved when relatively useless transitions are identified.

Moreover, the performance of our approach strongly depends on the quality of the distance heuristic $d^{\#}$ that is used to guide the search and to determine relatively useless transitions. The higher the precision of $d^{\#}$, the more transitions are evaluated correctly, and hence, the better the overall performance as many unnecessary states need not be considered. In small examples, rather coarse distance heuristics like the graph distance could already lead to improvements. Pointing back to our motivating example in Figure 3.1, we recognize that all transitions corresponding to edges from down to up are relatively useless for the maximum graph distance heuristic, whereas all other transitions are not. To see this, consider a small instance of the system with four parallel processes $p_0, \ldots, p_3$. Suppose the current global state under consideration is $s = \langle p_0 = l_1, p_1 = l_1, p_2 = l_0, p_3 = l_0 \rangle$. Then the transition $t = \{p_0.l_1 \rightarrow l_0\}$ that returns to the initial location in p_0 is relatively useless for d^L in s, because $d^L(\mathcal{M}_t, s) = 1$, and also $d^L(\mathcal{M}, t[s]) = 1$, and therefore $d^L(\mathcal{M}_t, s) \leq d^L(\mathcal{M}, t[s])$. This prevents our algorithm to apply such "up-going" transitions, and the global state $t[s]$ is not explored. Moreover, all other ("down-going") transitions t are not relatively useless in all global states s where they are applicable, as $d^L(\mathcal{M}_t, s) = \infty$ in this case (the local graph distance is infinity as no trace to the error location exists anymore when t is removed from the system). Overall, in this example, applying our extended algorithm leads to a shortest pos-

sible error trace with a dramatically smaller number of explored states than with plain greedy search. To get an impression of the state space reduction, Figure 3.6 shows the explored state spaces with the basic directed model checking algorithm compared to our extended algorithm based on relatively useless transitions for the example system with $n = 8$.

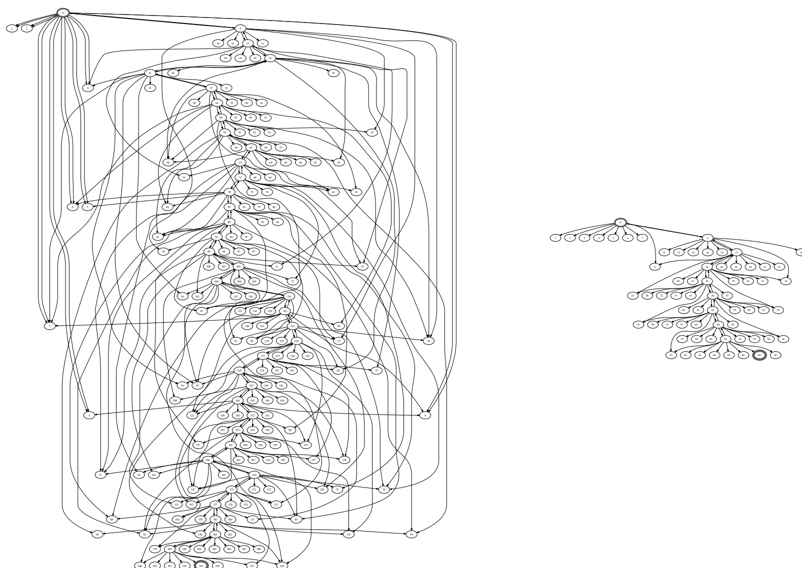

Fig. 3.6. State space reduction in the motivating example for $n = 8$

Overall, we observe that even a rather coarse distance heuristic based on the maximum graph distance can benefit significantly from our technique. In this simple example, useless transitions are perfectly identified with relatively useless transitions and the d^L distance heuristic. Hence, the overall search process turns from unguided search (recall that for every global state s except for the error state, $d^L(s) = 1$) to perfect search, and a shortest possible error trace is found. We will see that for larger and more complex systems, more sophisticated distance heuristics are needed to benefit from our technique. We will come back to this point in the next section.

3.4 Evaluation

In this section, we empirically evaluate our technique based on relatively useless transitions. First, we describe the experimental setup in Section 3.4.1. In Section 3.4.2, we evaluate our extended directed model checking algorithm on benchmarks

as described in Section 2.4. These include the industrial case studies “Single-tracked line segment” ($S_1, \ldots, S_9$) and “Mutual Exclusion” ($M_1, \ldots, M_4$ and $N_1, \ldots, N_4$), the Fischer protocols (F_5, F_{10}, F_{15}) as well as the arbiters case study ($A_2, \ldots, A_6$). Finally, our technique has also been investigated in the area of AI planning. We describe the adaptation of useless transitions to that area and present experimental results on domains of international planning competitions in Section 3.4.3.

3.4.1 Experimental Setup

We have implemented our extended directed model checking algorithm in our model checker MCTA [56] and evaluated it on a machine with AMD Opteron 2.3 GHz system with 4 GByte of memory. We set a timeout to 30 minutes. To get a conservative approximation of useless transitions, we have implemented our concept in a stronger way than described in the last section. When a reduced system with respect to some transition t is computed, we *additionally* remove edges that read an integer variable that is set by t, and edges that have the same destination location as some edge in t. In addition to the computational model as considered in this thesis, some of our benchmarks also feature clock variables and actually represent timed automata [3]. We adapted our concept to that class of systems in an straight forward way by treating the clock variables in the same way as we did with integer variables.

3.4.2 Experimental Results and Discussion

We have already discussed that the precision of relatively useless transitions depends on the precision of the underlying distance heuristic. For the evaluation, we apply our technique to the h^L distance heuristic [52] which is quite accurate. Furthermore, we apply relatively useless transitions to the more informed h^U distance heuristic [52], and also to the rather coarse d^U distance heuristic based on the plain graph distance [23, 25]. Recall that a more detailed description of these distance heuristics is given in Section 2.3 on page 16. We start by giving the results for the h^L distance heuristic in Table 3.1.

We report the number of explored states, the runtime in seconds, and the length of the found error traces. Table 3.1 additionally contains information about the benchmark models. The results are impressive. We observe that in almost all problem instances, the number of explored states could be decreased significantly, often by several orders of magnitude. This also impressively pays off in overall runtime in the S, F and A case studies, where the runtime could sometimes be decreased from hundreds of seconds to a fraction of a second. In particular, our extended directed model checking algorithm could also solve the large problems where classical directed model checking without search enhancements failed. Considering the

Table 3.1. Experimental results for h^L with greedy search (h^L) and for the new algorithm UT-search (h^L-*UT*). Abbreviations: #a: number of parallel automata, #v: number of variables, #d: number of explored deferred states over total number of explored states, dashes indicate out of memory (> 4 GByte) or out of time (> 30 min). Best results are shown in bold fonts.

			explored states		runtime in s		trace length		#d
Instance	#a	#v	h^L	h^L-*UT*	h^L	h^L-*UT*	h^L	h^L-*UT*	
S_1	5	15	1704	**1537**	**0.0**	0.1	84	**76**	0.000
S_2	6	17	3526	**1229**	**0.1**	**0.1**	172	**76**	0.000
S_3	6	18	4182	**1061**	**0.1**	**0.1**	162	**76**	0.000
S_4	7	20	29167	**879**	0.7	**0.2**	378	**77**	0.000
S_5	8	22	215525	**1116**	5.2	**0.3**	1424	**78**	0.000
S_6	9	24	1.7e+6	**1116**	37.4	**0.4**	5041	**78**	0.000
S_7	10	26	1.6e+7	**1114**	332.5	**0.6**	15085	**78**	0.000
S_8	10	27	7.1e+6	**595**	129.3	**0.3**	5435	**76**	0.000
S_9	10	28	9.6e+6	**2771**	201.1	**1.3**	5187	**94**	0.000
M_1	3	15	4581	**4256**	**0.1**	0.2	457	**97**	0.000
M_2	4	17	15832	**7497**	**0.3**	0.5	1124	**104**	0.000
M_3	4	17	**7655**	10733	**0.1**	0.7	748	**91**	0.000
M_4	5	19	71033	**16287**	**1.6**	1.7	3381	**98**	0.000
N_1	3	18	50869	**5689**	39.0	**0.3**	26053	**108**	0.000
N_2	4	20	30476	**22763**	**1.2**	1.5	1679	**259**	0.000
N_3	4	20	**11576**	35468	**0.4**	2.7	799	**204**	0.000
N_4	5	22	**100336**	142946	**5.3**	14.9	2455	**792**	0.000
F_5	5	6	179	**7**	**0.0**	**0.0**	12	**6**	0.000
F_{10}	10	11	86378	**7**	3.3	**0.0**	22	**6**	0.000
F_{15}	15	16	–	**7**	–	**0.0**	–	**6**	0.000
A_2	8	0	36	**15**	**0.0**	**0.0**	21	**12**	0.000
A_3	16	0	206	**48**	**0.0**	**0.0**	24	**17**	0.000
A_4	32	0	76811	**149**	8.4	**0.2**	42	**23**	0.000
A_5	64	0	263346	**34**	60.0	**0.2**	112	**27**	0.000
A_6	128	0	–	**39**	–	**1.2**	–	**32**	0.000

length of the encountered error traces, we observe that much shorter traces are found when relatively useless transitions are taken into account. Finally, we observe that the number of explored deferred states is zero in all the problems. This effectively means that useless transitions are approximated very precisely with h^L. Overall, we conclude that our concept is very powerful for an informed distance heuristic like h^L, often leading to better performance and shorter error traces compared to plain directed model checking.

Considering the results for the h^U distance heuristic with classical directed model checking in Table 3.2, we first observe that h^U is a more informed distance heuristic than h^L. In most of the problem instances, less states are explored than with h^L, and shorter error traces are found. In addition, one more problem could be solved. This is also reflected when our extended directed model checking algorithm is applied with h^U. Compared to our results for relatively useless transitions for

h^L, the number of explored states could mostly further be decreased, which also pays off in better overall runtime in almost all the problems. Still, no deferred states are explored at all, which shows that useless transitions are approximated very precisely with h^U. Overall, compared to classical directed model checking with h^U, this results in a strong improvement of the search guidance and very short error traces.

Table 3.2. Experimental results for h^U with greedy search (h^U) and with greedy search plus useless transitions (h^U-*UT*). Abbreviations: #a: number of parallel automata, #v: number of variables, #d: number of explored deferred states over total number of explored states, dashes indicate out of memory (> 4 GByte) or out of time (> 30 min). Best results are shown in bold fonts.

			explored states		runtime in s		trace length		#d
Instance	#a	#v	h^U	h^U-*UT*	h^U	h^U-*UT*	h^U	h^U-*UT*	
S_1	5	15	429	**243**	**0.0**	**0.0**	67	**54**	0.000
S_2	6	17	828	**212**	**0.0**	**0.0**	83	**54**	0.000
S_3	6	18	1033	**198**	**0.0**	**0.0**	79	**54**	0.000
S_4	7	20	12938	**174**	0.5	**0.1**	112	**55**	0.000
S_5	8	22	65506	**147**	2.5	**0.1**	176	**61**	0.000
S_6	9	24	453763	**147**	15.9	**0.1**	432	**61**	0.000
S_7	10	26	4.2e+6	**143**	131.6	**0.1**	924	**61**	0.000
S_8	10	27	3.4e+6	**1466**	105.7	**1.0**	2221	**56**	0.000
S_9	10	28	2.2e+7	**1575**	675.5	**1.1**	1952	**69**	0.000
M_1	3	15	7668	**4366**	**0.2**	**0.2**	**71**	73	0.000
M_2	4	17	18847	**2018**	0.5	**0.2**	119	**81**	0.000
M_3	4	17	19597	**17315**	**0.5**	1.9	**124**	163	0.000
M_4	5	19	46170	**15349**	**1.3**	2.7	160	**91**	0.000
N_1	3	18	9117	**5191**	**0.3**	**0.3**	99	**80**	0.000
N_2	4	20	23462	**3260**	1.1	**0.4**	154	**136**	0.000
N_3	4	20	43767	**19271**	2.1	**1.8**	**147**	149	0.000
N_4	5	22	152163	**15102**	13.9	**2.3**	**314**	377	0.000
F_5	5	6	**7**	**7**	**0.0**	**0.0**	**6**	**6**	0.000
F_{10}	10	11	**7**	**7**	**0.0**	**0.0**	**6**	**6**	0.000
F_{15}	15	16	**7**	**7**	**0.0**	**0.0**	**6**	**6**	0.000
A_2	8	0	25	**20**	**0.0**	**0.0**	21	**18**	0.000
A_3	16	0	82	**27**	**0.0**	**0.0**	18	**17**	0.000
A_4	32	0	39	**34**	**0.0**	0.1	28	**22**	0.000
A_5	64	0	4027	**42**	1.3	**0.4**	47	**27**	0.000
A_6	128	0	–	**50**	–	**2.4**	–	**32**	0.000

Finally, we investigate the performance of relatively useless transitions for the d^U distance heuristic which is based on the plain graph distance. In comparison to h^L and h^U, we observe that d^U is less informed because the graph distance is rather coarse compared to the monotonicity abstraction used by h^L and h^U. The results for d^U are given in Table 3.3.

Table 3.3. Experimental results for d^U with greedy search (d^U) and with greedy search plus useless transitions (d^U-*UT*). Abbreviations: #a: number of parallel automata, #v: number of variables, #d: number of explored deferred states over total number of explored states, dashes indicate out of memory ($>$ 4 GByte) or out of time ($>$ 30 min). Best results are shown in bold fonts.

Instance	#a	#v	explored states		runtime in s		trace length		#d
			d^U	d^U-*UT*	d^U	d^U-*UT*	d^U	d^U-*UT*	
S_1	5	15	11449	**9796**	**0.0**	0.1	**823**	842	0.713
S_2	6	17	33859	**31807**	**0.1**	0.5	1229	**1105**	0.873
S_3	6	18	**51526**	52107	**0.2**	0.8	**1032**	1144	0.882
S_4	7	20	**465542**	504749	**2.0**	7.5	**3132**	5364	0.970
S_5	8	22	**4.6e+6**	**4.6e+6**	**22.6**	70.6	14034	**14000**	0.992
S_6	9	24	–	–	–	–	–	–	–
M_1	3	15	185416	**7557**	156.4	**0.1**	106224	**923**	0.566
M_2	4	17	**56240**	294877	**1.1**	67.1	**13952**	51541	0.755
M_3	4	17	869159	**26308**	1729.5	**0.5**	337857	**1280**	0.769
M_4	5	19	726691	**100073**	428.0	**1.8**	290937	**4436**	0.882
N_1	3	18	**10215**	19698	**0.3**	0.7	2669	**1855**	0.567
N_2	4	20	–	**96307**	–	**5.1**	–	**8986**	0.747
N_3	4	20	–	**29254**	–	**1.2**	–	**784**	0.771
N_4	5	22	330753	**239877**	23.0	**17.8**	51642	**1969**	0.895
F_5	5	6	**9**	**9**	**0.0**	**0.0**	**6**	**6**	0.000
F_{10}	10	11	**9**	**9**	**0.0**	**0.0**	**6**	**6**	0.000
F_{15}	15	16	**9**	**9**	**0.0**	**0.0**	**6**	**6**	0.000
A_2	8	0	**23**	24	**0.0**	**0.0**	**13**	**13**	0.708
A_3	16	0	**296**	297	**0.0**	**0.0**	**39**	**39**	0.939
A_4	32	0	**19034**	19035	**0.2**	0.4	**129**	**129**	0.973
A_5	64	0	–	–	–	–	–	–	–

First, we observe that the results for d^U with relatively useless transitions are less significant than for h^L and h^U. Looking more closely at the distance estimates provided by d^U, we find out that often only a few different values occur. This is due to the relative coarse abstraction based on the graph distance, which ignores integer variables and synchronization behavior completely. As a consequence, many transitions are wrongly classified as useless with the d^U distance heuristic, and thus the number of explored deferred states is very high. This effectively means that no improvements may be obtained at all compared to classical directed model checking with d^U. This is the case in the S and A examples, where almost all states are deferred (more than eighty percent in most cases). However, we may still get benefits if the number of deferred states is not as high; this shows up for the M and N examples.

The overall conclusion of our experimental evaluation is that the concept of relatively useless transitions can significantly improve scalability and error trace quality of classical directed model checking. Furthermore, the experiments confirm that the precision of our technique strongly depends on the precision of the dis-

tance heuristic $d^{\#}$ that is applied. If $d^{\#}$ is informed, then useless transitions are captured very precisely with relatively useless transitions for $d^{\#}$, and impressive performance improvements can be achieved as we have seen for h^L and h^U. For less informed distance heuristics, our technique is less precise as well, as we have seen for d^U.

Precision of Relatively Useless Transitions Continued

In this section, we will analyze the precision of relatively useless transitions depending on the applied distance heuristic in more detail. Therefore, we investigate how precisely useless transitions are approximated by the h^L, h^U and d^U distance heuristics on a small model checking task *exactly*. During directed model checking, we compute for every encountered state s if s has been reached by a useless transition (i. e., we compare the real error distance of s to the real error distance of its successor). This result is then compared with the relatively-useless-classification of the applied distance heuristic. In Table 3.4, we exemplary report the relative number of true (and false) positives (and negatives) for the largest problem of the arbiter case study that could be handled in this way (A_3).

Table 3.4. Fraction of correct and incorrect classifications on the A_3 problem. Abbreviations: TP/FP: relative number of true/false positives, TN/FN: relative number of true/false negatives.

	visited transitions			
	TP	TN	FP	FN
h^U	0.669	0.217	0.078	0.036
h^L	0.587	0.233	0.102	0.078
d^U	0.235	0.037	0.728	0.000

We report the relative number of visited transitions encountered with our extended algorithm. A transition t is counted as visited if the corresponding successor state s has been created but not yet necessarily explored, i. e., s is not yet necessarily in the closed list. A transition is counted as a true positive if it is a useless transition, i. e., a transition that did not start a shortest error trace, that has also been classified as relatively useless by the corresponding distance heuristic. The other cases are handled accordingly.

Again, we observe that useless transitions are classified best with h^U and h^L. For these distance heuristics, the relative number of correctly classified transitions is high. In particular, it is important to note that the relative number of false positives (i. e., the number of transitions that have wrongly been classified as useless), is low. Obviously, such a wrongly classified transition can severely worsen the overall search process, as the resulting successor state is only explored if the regular open queue is empty. Considering the results for d^U, we observe that the number

of wrongly classified transitions is significantly higher, and there is a significant number of false positives in particular. This causes the lower performance when applying our framework with d^U.

3.4.3 Useless Transitions in AI Planning

In the introduction of this thesis, we have pointed out that there is a strong relationship between directed model checking and AI planning as heuristic search. In both settings, the task is to find a sequence of transitions such that a global state is reached which satisfies a given property. In the context of directed model checking, the property describes an undesired error condition, whereas in the context of AI planning, it describes some desired goal condition. In AI planning, transitions correspond to "actions", which have a guard and an effect. An action is applicable in a state s if s satisfies its guard, and the corresponding successor state is determined from s by setting the variables according to its effect. An error trace in model checking corresponds to a *plan* in AI planning. In our setting, a plan consists of a finite number of actions that are successively applicable from the initial state and lead to a state that satisfies the goal condition.

Because of the close relationship of these areas when considered from an abstract point of view, we have also investigated a variant of relatively useless transitions in the context of planning as heuristic search [78]. In that area, various benchmarks from the international planning competitions exist which are often motivated by real world problems. Therefore, the consideration of our technique in this context allows us to get a deeper insight of how relatively useless transitions perform in practice. For the evaluation, we have implemented our concept into the FAST DOWNWARD planning system [36].

Integration into the FAST DOWNWARD Planning System

FAST DOWNWARD is a recent state-of-the-art planning system based on heuristic search, in particular winning the planning competition 2004 in the track for classical planning. It supports various distance heuristics and additional search enhancement techniques such as *helpful actions* [42] and *preferred operators* [36]. In a nutshell, both of these search enhancements are based on the following idea. First, during the computation of a distance estimate for a state s in AI planning, an abstract plan is produced from the corresponding abstract state $s^{\#}$. Roughly speaking, an action a is defined as helpful (or preferred) in s if a corresponding abstract action $a^{\#}$ is contained as a possible first abstract action in such an abstract plan. In this way, helpful actions are identified during the computation of h^{FF} [42], preferred operators are identified for h^{CG} [36] and h^{cea} [38]. However, as we have just outlined, these actions are identified as a byproduct during the computation of the heuristic values,

and hence, these techniques are bound to the corresponding distance heuristics. If helpful actions or preferred operators are available, FAST DOWNWARD maintains two open queues *helpfulQueue* and *unknownQueue* during the search. As indicated by the name, states that are reached by helpful actions (or preferred operators, respectively) are maintained in *helpfulQueue*. Furthermore, the second queue contains all the states that have been encountered so far. The search engine selects states from *helpfulQueue* and *unknownQueue* in an alternating way.

As we have just discussed, in the planning context, an action exactly corresponds to a transition $t \in \mathcal{T}(\mathcal{M})$ for a given system $\mathcal{M}$. Therefore, we define *useless actions* and *relatively useless actions* as a straight forward adaptation of useless transitions and relatively useless transitions to the planning context. We have integrated relatively useless actions into FAST DOWNWARD by extending FAST DOWNWARD's queue selection scheme as follows. We additionally maintain a third open queue *uselessQueue*, which contains states that are reached by applying a relatively useless action. More precisely, for each global state s to which the action a is applied, we check if a is relatively useless in s for the distance heuristic that is also used for the search. If this is the case, the corresponding successor state is pushed into the *uselessQueue*. Actions that are classified as both helpful and relatively useless are maintained in the *uselessQueue*. This queue is only accessed if both other queues are empty. A conceptual description of the queue selection algorithm is shown in Figure 3.7.

```
1 function selectOpenQueue():
2     if unknownQueue = ∅ and helpfulQueue = ∅:
3         return uselessQueue
4     elif helpfulQueue ≠ ∅ and p(helpfulQueue) ≤ p(unknownQueue)
5         p(helpfulQueue) += 1
6         return helpfulQueue
7     elif unknownQueue ≠ ∅ and p(unknownQueue) ≤ p(helpfulQueue)
8         p(unknownQueue) += 1
9         return unknownQueue
10 return selectNonEmptyQueue()
```

Fig. 3.7. Conceptual description of the queue selection algorithm in FAST DOWNWARD

The queue containing states that are reached via a relatively useless action is only returned if both other queues are empty (lines 2 – 3). We maintain a priority function p for each open queue, which is increased by one if it was selected. To alternate between the *helpfulQueue* and the *unknownQueue*, the queue with lowest priority value is selected. The function selectNonEmptyQueue() returns a non empty last recently used queue if there exists one (line 10).

Experimental Setup and the Planning Benchmarks

We evaluated relatively useless actions for the h^{FF} [42], h^{cea} [38], h^{max} and h^{add} [10] distance heuristics as implemented in the FAST DOWNWARD planning system. We compare our technique against plain heuristic greedy best first search as well as helpful actions (for h^{FF}), preferred operators (for h^{cea}), and the combination thereof. The experimental evaluation has been performed on a machine with Intel Pentium 4 CPU with 3 GHz. We set a time bound of 10 minutes and a memory bound of 1 GByte for each run. As benchmark problems, we used domains from the recent international planning competitions. They are mostly motivated by real world problems and often very challenging for domain independent planners. A domain consists of several (usually 20 – 50) problem instances of different size and difficulty (see, e. g., [30]). For the evaluation, we have chosen domains from this suite that are hard for FAST DOWNWARD in terms of number of unsolved instances without search enhancements.

Results

As the overall number of planning benchmarks is high, we concisely present the results in terms of number of unsolved instances for different configurations. Table 3.5 and 3.6 provide an overview of the number of unsolved planning instances per domain. Table 3.5 shows the results for the h^{FF} and h^{cea} distance heuristics, for which helpful actions and preferred operators are available. Table 3.6 gives the results for the h^{add} and the h^{max} distance heuristics. We report the number of unsolved instances by domain for plain *greedy* best first search, as well as for greedy best first search with relatively *useless* actions, with *helpful* actions (or preferred operators, respectively), and for *both*, i. e., for the combination of relatively useless and helpful actions.

The results clearly indicate the potential of our technique in the context of AI planning. First, we observe that applying relatively useless actions to plain greedy best first search significantly reduces the number of unsolved instances (from 388 to 290 for h^{FF} and h^{cea}, from 467 to 374 for h^{add} and h^{max}). For example, the h^{FF} distance heuristic with relatively useless actions could not solve 2 out of 30 instances in the PATHWAYS domain, compared to 21 unsolved instances with plain greedy best first search. Moreover, the useless actions technique performs comparably to helpful actions and preferred operators. To see this, we observe in Table 3.5 that taking relatively useless actions into account leaves 290 problem instances unsolved, compared to 301 unsolved instances with helpful actions or preferred operators. Finally, it is very interesting that the combination of relatively useless actions with helpful actions or preferred operators often yields further significant performance gains. The number of unsolved instances could be further reduced from 301

Table 3.5. Number of unsolved instances by domain for the h^{FF} and h^{cea} distance heuristics where helpful actions (or preferred operators, respectively) are available. The best configuration is shown in bold fonts. The first column shows the problem domain and the total number of problem instances in parentheses. Abbreviations: greedy: plain heuristic (greedy best first) search, + useless: heuristic search with useless actions, + helpful: heuristic search with helpful actions or preferred operators, + both: heuristic search with the combination of useless and helpful actions.

	greedy	+ useless	+ helpful	+ both
h^{FF}				
PATHWAYS (30)	21	2	11	**1**
TPP (30)	13	1	7	**0**
DEPOT (22)	8	4	4	**0**
ASSEMBLY (30)	15	**0**	**0**	**0**
PIPESWORLD-NT (50)	23	12	9	**8**
PIPESWORLD-T (50)	32	29	25	**20**
TRUCKS (30)	16	**13**	14	14
AIRPORT (50)	17	**14**	17	**14**
OPTICAL-TELEGRAPHS (48)	46	45	46	**44**
h^{cea}				
PATHWAYS (30)	23	18	16	**4**
TPP (30)	10	18	**9**	18
DEPOT (22)	11	3	8	**1**
ASSEMBLY (30)	20	8	12	**0**
PIPESWORLD-NT (50)	25	14	18	**12**
PIPESWORLD-T (50)	35	35	31	**24**
TRUCKS (30)	**17**	18	**17**	**17**
AIRPORT (50)	**11**	**11**	**11**	**11**
OPTICAL-TELEGRAPHS (48)	45	45	46	**44**
$\sum$	388	290	301	**232**

to 232. Although the ideas of useful (helpful or preferred) and useless actions are conceptually similar, the experimental evaluation suggests that different aspects are covered by these notions. Therefore, we are able to identify "useful" actions, i. e., actions that are classified as useful by the corresponding distance heuristic, that are actually not useful in practice. Exemplary, this shows up well in the PATHWAYS domain for the h^{cea} distance heuristic. The number of unsolved instances with preferred operators (16) and relatively useless actions (18) is similar, but with the combination of preferred operators and relatively useless actions, only 4 instances remain unsolved.

As already discussed, the quality of our technique generally depends on the quality of the applied distance heuristic. This observation also shows up in the planning setting. The results are best for the h^{FF} heuristic, and less significant for the less informed h^{max} heuristic. Overall, relatively useless transitions have shown their potential also in the context of AI planning, where the number of unsolved problem instances could be significantly reduced compared to plain heuristic search as well

Table 3.6. Number of unsolved instances by domain for the h^{add} and h^{max} distance heuristics. The best configuration is shown in bold fonts. The first column shows the problem domain and the total number of problem instances in parentheses.

	greedy	+ useless
h^{add}		
PATHWAYS (30)	23	**18**
TPP (30)	15	**5**
DEPOT (22)	12	**2**
ASSEMBLY (30)	20	**8**
PIPESWORLD-NT (50)	29	**11**
PIPESWORLD-T (50)	34	**29**
TRUCKS (30)	**17**	18
AIRPORT (50)	**17**	**17**
OPTICAL-TELEGRAPHS (48)	**47**	**47**
h^{max}		
PATHWAYS (30)	26	**24**
TPP (30)	22	**18**
DEPOT (22)	17	**13**
ASSEMBLY (30)	20	**8**
PIPESWORLD-NT (50)	33	**26**
PIPESWORLD-T (50)	41	**38**
TRUCKS (30)	20	**18**
AIRPORT (50)	**28**	**28**
OPTICAL-TELEGRAPHS (48)	**46**	**46**
$\sum$	467	**374**

as compared to other well-established search enhancements like helpful actions and preferred operators.

3.5 Related Work

Approaches to evaluate transitions during heuristic search have mostly been studied in the area of AI planning so far. Among these approaches, we have already discussed the notion of helpful actions and preferred operators that are used to select a set of promising actions in a state. Helpful actions have been proposed by Hoffmann and Nebel [42]. An action is considered as helpful if it is applicable and adds at least one goal at the first time step during the computation of h^{FF}. Vidal [74] extended the idea of helpful actions. Instead of preferring states that result from applying helpful actions, he proposed to extract a prefix of actions from the relaxed solution that are sequentially applicable in the current state. Helmert's preferred operators [36] are similar to helpful actions in the context of the causal graph heuristic. All these approaches identify useful actions during the computation of the distance heuristic. Contrarily, we identify useless transitions and useless actions

based on reduced planning instances. Therefore, relatively useless transitions can be combined with arbitrary distance heuristics.

Nebel et al. [61] removed *irrelevant facts and operators* from a planning task in a preprocessing step. They removed operators and variables that are possibly not needed in order to find a solution. Such operators and variables are identified by solving a relaxed problem. However, in contrast to the technique of relatively useless transitions, this method is not solution-preserving. In our work, relatively useless transitions and relatively useless actions are identified on-the-fly, where it depends on the current state if an operator is relatively useless.

Haslum and Jonsson [34] defined an operator as redundant if it can be simulated with a sequence of other operators. Based on this idea, they proposed an approach to compute redundant operators and reduced operator sets that is solution preserving. Therefore, contrarily to useless transitions, redundant operators can be pruned completely, and there is no need to maintain several open queues.

Bacchus and Ady [4] proposed to avoid certain sequences of actions that do not make sense. For example, a transporter that has to deliver an object to some place should not load and unload the object without moving in between. However, such action sequences are identified manually.

3.6 Conclusions

We have introduced the concept of useless and relatively useless transitions for directed model checking and for AI planning as heuristic search. We have given an intuitive, but computationally hard definition of uselessness, which we have then approximated appropriately with the given distance heuristic. Afterwards, we have presented an extended directed model checking algorithm based on this concept. We have observed that relatively useless transitions significantly improved the scalability of directed model checking and AI planning as heuristic search. Although these two areas are motivated differently and therefore also provide differently structured benchmark problems, useless transitions have shown to be a general concept that can be successfully applied in both areas. The general lesson to be learned is that additionally evaluating transitions, rather than only states, can serve as a strong criterion to improve the guidance of the search.

For the future, it will be interesting to further extend and refine the concept. First, as already outlined in Section 3.3, the adaptation of relatively useless transitions for pattern database heuristics will be an interesting and important topic for future research. Although a straight forward adaptation is technically possible, this would yield an approach that is probably not very efficient because a separate pattern database has to be computed for every reduced system. The question remains if there is a way to do this more efficiently. Ultimately, as for every heuristic technique, one would like to have a precise theoretical analysis where the technique

works best and where it does not work at all. We have done a first step in this direction with our discussion in Section 3.3, where we observed that relatively useless transitions work probably best for distance heuristics $d^{\#}$ that are computed on-the-fly, and that the precision of our technique strongly depends on the precision of $d^{\#}$. For the future, it will be interesting to analyze if more precise statements can be achieved in this direction. For example, are there classes of systems where the concept does not work? Are there classes of systems for which it works provably well? Such insights would allow us to further refine and improve the overall concept, e. g., by automatically enabling or disabling the check for relatively useless transitions depending on the specific problem structure.

4

Context-Enhanced Directed Model Checking

In this chapter, we introduce a technique to prioritize transitions based on *interference contexts*. It is based on joint work with Kupferschmid [77]. The motivation is to overcome the limitation of relatively useless transitions to be dependent on a distance heuristic. Roughly speaking, our technique is based on the following idea: if there is a transition t that is part of a shortest error trace π, then there often is a subsequent transition in π that *profits* from the effect of t. Therefore, we propose to preferably explore states that have been reached by a transition that profits from the effect of the previously applied transition. As a consequence, the search process avoids "jumping" while exploring the state space. We use the notion of interference context to determine how much a transition profits from the execution of another transition, and propose *context-enhanced directed model checking* based on this technique. We have implemented our algorithm and applied it to uninformed search as well as to directed model checking with several distance heuristics from the literature. In particular, we compare our algorithm to useless transitions and to *iterative context bounding* proposed by Musuvathi and Qadeer [58]. The experiments reveal that context-enhanced directed model checking scales better than previous approaches in many challenging problems.

The remainder of this chapter is organized as follows. After a short motivation in Section 4.1, we formally introduce interference contexts and context-enhanced directed model checking in Section 4.2. Afterwards, we discuss related work in Section 4.3 and empirically evaluate our technique in Section 4.4. Finally, we conclude the chapter and discuss future work in Section 4.5.

4.1 Motivation

In the previous chapter, we have introduced the useless transition technique to heuristically prioritize transitions during directed model checking. In particular, we

have observed that the precision of this technique was dependent on the precision of the applied distance heuristic.

The limitations of useless transitions motivate to investigate a technique to prioritize transitions which is independent of the quality of the distance heuristic. The technique that we present in this chapter is based on the notion of *transition interference*. As we will see, it is purely syntactic and assigns higher priorities to transitions that "profit" from the effect of previously applied transitions. In the following, we give an example for a model checking task where useless transitions fail, but an interference based technique as sketched above succeeds.

Example 4.1. Consider the example model checking task of Figure 4.1 that consists of a system $\mathcal{M}$ with processes $p_1, \ldots, p_n$. The transitions of $\mathcal{M}$ are interleaving transitions, i. e., $label(t) = \tau$ for all $t \in \mathcal{T}(\mathcal{M})$. For the sake of readability, these labels are omitted in Figure 4.1.

Fig. 4.1. Example system with n components

The initial global state of the system is $s_0 = \langle p_1 = l_0, \ldots, p_n = l_0 \rangle$ where every process is in its initial location. The global error state is given as $\langle p_1 = l_3, \ldots, p_n = l_3 \rangle$ where every process is in its error location. Suppose we apply directed model checking using the distance heuristic based on the maximal graph distance d^L as in Example 3.1. Recall that $d^L(s) = \max_{i=1,\ldots,n} dist_i(s)$, where $dist_i(s)$ denotes the local error distance for process i in global state s. Doing so, we first observe that d^L is a rather uninformed distance heuristic for this problem because the domain of d^L is rather small ($d^L(s) \in \{0, \ldots, 3\}$ for all global states s), and global states can hardly be distinguished by d^L. Second, we observe that relatively useless transitions for d^L also fail in many cases. For example, *every* applicable transition in the initial state s_0 is relatively useless for d^L. To see this, consider transition $t = \{p_1.l_0 \rightarrow l_1\}$. We have $d^L(\mathcal{M}_t, s_0) = 2$, and also $d^L(\mathcal{M}, t[s_0]) = 2$ because the local error distances in the other processes are still 2. Therefore, we have $d^L(\mathcal{M}_t, s_0) \leq d^L(\mathcal{M}, t[s_0])$, and hence, t is relatively useless for d^L in s_0. Moreover, we observe that transition $t' = \{p_1.l_1 \rightarrow l_3\}$ is not useless in the successor state $t[s_0]$, but relatively useless for d^L. Hence, t' cannot be distinguished from transitions that are actually useless, e. g., going back to l_0 from l_1.

However, apart from showing the limitations of directed model checking and relatively useless transitions with d^L, this simple example also motivates a technique to overcome these problems. The overall idea is to prefer transitions that *interfere* with a previously applied transition on a syntactic level. For example, t' would be such a preferred transition after applying t because the source location of t' is equal to the destination location of t. In the following sections, we make this idea precise. In particular, we present a multi-queue algorithm for directed model checking that is able to respect different levels of relevance.

4.2 Context-Enhanced Directed Model Checking

In this section, we introduce context-enhanced directed model checking based on *interference contexts*. To compactly describe the main ideas, we need the notion of *innocence*. Let $\Theta = (\mathcal{M}, \varphi)$ be a model checking task. A transition t is innocent in Θ if for each state s where t is applicable, there is no constraint of the error formula φ that is satisfied by $\mathit{effect}(t)$. This means, if there is a conjunct c of φ so that $t[s] \models c$, then c was already satisfied by s. Note that, if the initial state of a system is not an error state, then only applying innocent transitions will never lead to error states.

The main idea of context-enhanced directed model checking is the following. Let s be a state and let t be an innocent transition that is enabled at s. If t is the first transition of a shortest error trace starting from s, then there must be applicable transitions at $s' = t[s]$ that *profit* from the effect of t. Otherwise t cannot be part of a shortest error trace. In this chapter, we use *interference contexts* to determine whether a transition profits from a preceding transition.

4.2.1 Interference Contexts

Our notion of interference contexts is based on the well-known concept of interference. Similar to other work (for example, in the area of partial order reduction [32]), we use a definition that can be statically checked. Roughly speaking, two transitions interfere if they work on a common set of system variables. For a transition t, we will use the notation $\mathrm{var}(\mathit{guard}(t))$ and $\mathrm{var}(\mathit{effect}(t))$ to denote the set of variables occurring in the guard and the effect of t, respectively. These include both integer variables as well as the *process* variables stating in which processes the transition occurs.

Example 4.2. Consider the system $\mathcal{M}$ in Figure 4.2 consisting of two processes and two global transitions. The set of transitions of $\mathcal{M}$ is $\mathcal{T}(\mathcal{M}) = \{t_1, t_2\}$ with the interleaving transition $t_1 = \{p_1.l_1 \xrightarrow[i<1;i:=1]{\tau} l_2\}$ and the synchronized transition $t_2 = \{p_1.l_1 \xrightarrow[j<1]{a} l_2, p_2.l_0 \xrightarrow[k:=1]{a} l_1\}$.

Fig. 4.2. Example system and transitions

For transition t_1, $\mathrm{var}(guard(t_1)) = \{i, p_1\}$ contains the integer guard variable i and the process variable p_1, indicating that t_1's source and destination location belong to p_1. Analogously, the effect variables $\mathrm{var}(effect(t_1)) = \{i, p_1\}$ of t_1 contain the integer effect variable i and process variable p_1. For the synchronized transition t_2, we have $\mathrm{var}(guard(t_2)) = \{j, p_1, p_2\}$, and $\mathrm{var}(effect(t_2)) = \{k, p_1, p_2\}$. We remark that the synchronization label a is not a variable.

Having this notion of guard and effect variables, we define the notion of interference. Informally speaking, two transitions interfere if they work on a common set of variables.

Definition 4.3 (Interference). *Two transitions* t_1 *and* t_2 interfere *iff at least one of the following conditions holds:*

1. $\mathrm{var}(effect(t_1)) \cap \mathrm{var}(guard(t_2)) \neq \emptyset$,
2. $\mathrm{var}(effect(t_2)) \cap \mathrm{var}(guard(t_1)) \neq \emptyset$,
3. $\mathrm{var}(effect(t_1)) \cap \mathrm{var}(effect(t_2)) \neq \emptyset$.

Informally, this means that two transitions interfere if one writes a variable that the other is reading, or both transitions write a common variable. We next give a pruning criterion based on interference. To formulate this, we first define the notion of *interference contexts*. Roughly speaking, for a transition t and $n \in \mathbb{N}_0$, the interference context $C_n(t)$ contains all transitions for which there is a sequence of at most n transitions where successive transitions interfere.

Definition 4.4 (Interference Context). *Let* $\mathcal{M}$ *be a system,* $t \in \mathcal{T}(\mathcal{M})$ *be a transition and* $n \in \mathbb{N}_0$. *The* interference context $C_n(t)$ *is inductively defined as follows:*

$$C_0(t) = \{t\}$$
$$C_n(t) = C_{n-1}(t) \cup \{t' \in \mathcal{T}(\mathcal{M}) \mid \exists t'' \in C_{n-1}(t) : t'' \textit{interferes with } t'\}$$

As we are dealing with finite systems, there exists a smallest $N \leq |\mathcal{T}(\mathcal{M})|$, so that $C_N(t) = C_{N+1}(t)$ for all transitions $t \in \mathcal{T}(\mathcal{M})$. We will denote this context with $C(t)$. Note that this context induces an equivalence relation on the set of system transitions $\mathcal{T}(\mathcal{M})$. It partitions $\mathcal{T}(\mathcal{M})$ into subsets of transitions which operate on pairwise disjoint variables. Based on this notion, we now give a pruning criterion which is guaranteed to preserve completeness. Informally speaking, if a state is part of a shortest error trace and has been reached via an innocent transition t, then it suffices to only apply transitions from $C(t)$ at s.

Proposition 4.5. *Let $\Theta = (\mathcal{M}, \varphi)$ be a model checking task for a system $\mathcal{M} = (\mathcal{P}, V)$ and an error formula $\varphi = \bigwedge(v_i \bowtie c_i)$, where $\bowtie \in \{<, \leq, =, \geq, >\}$, $v_i \in V$ and $c_i \in \mathbb{Z}$. Let s be a state in $\mathcal{M}$ that is part of a shortest error trace from s_0 and has been reached by a transition sequence with last transition t. Further assume that t is innocent in Θ. Then there is a shortest error trace from s that starts with a transition from $C(t)$.*

Proof. As s is part of a shortest error trace from the initial state, and t is innocent, there is a shortest error trace π that starts in s which contains a transition t'_π that needs the effects of t, i. e., $\mathrm{var}(\mathit{effect}(t)) \cap \mathrm{var}(\mathit{guard}(t'_\pi)) \neq \emptyset$. Otherwise, t would not have been needed and s would not be part of a shortest error trace. We transform π into a shortest error trace π' so that the first transition in π' is contained in $C(t)$. Let $t_1, \ldots, t_n$ be the prefix of π so that t_n is the first transition in π that interferes with t, i. e., $t_1, \ldots, t_{n-1}$ do not interfere with t. Then π' is constructed by an inductive argument. For $n = 1$, i. e., if t_1 interferes with t, then $\pi' = \pi$ as $t_1 \in C(t)$. For the induction step, let t_{n+1} be the first interfering transition with t. If t_{n+1} and t_n do not interfere, then they can be exchanged in π' as non-interfering transitions can be applied in any order, leading to the same state. If they do interfere, then by definition $t_n \in C(t)$, which proves the claim.

From Proposition 4.5, we can derive the following pruning criterion: in a global state s that has been reached with transition t, it is sufficient to apply the transitions from $C(t)$ because at least one of these transitions starts a shortest error trace in s. As a consequence, transitions that are not contained in $C(t)$ can be pruned without losing completeness. We remark that the condition from Proposition 4.5 for a state s to be on a shortest error trace can be assumed without loss of generality: if s is *not* part of a shortest error trace, *every* successor can be pruned.

In practice, this pruning criterion does not seem to fire very often: Benefits are only obtained if the system's transitions can be partitioned into at least two sets of transitions that operate on disjoint variables. In such cases, also compositional model checking is applicable. These concepts are known and by no means new. In fact, the pruning criterion can be seen as a version of compositional model checking, where completely independent parts of the system are handled individually. However, the formulation of Proposition 4.5 lends itself to a way of how to approximate the pruning criterion. This leads to the heuristic approach to prioritize transitions that we are introducing next.

4.2.2 The Context-Enhanced Search Algorithm

In this section, we describe how we approximate the pruning criterion. The basic idea is actually quite simple: instead of considering the exact closure $C(t)$ of a transition t, we substitute it with the interference context $C_n(t)$ for some bound $n < N$,

where N is the smallest number such that $C_N(t) = C(t)$ for all transitions $t \in T$. Suppose for a moment that we know how to choose a good value for n. Obviously, Proposition 4.5 does not hold anymore if we replace $C(t)$ with $C_n(t)$. As a consequence thereof, states that are reached via transitions that are not contained in $C_n(t)$ cannot be pruned without loosing completeness. The approximation therefore becomes a heuristic criterion for the *relevance* of transitions. Instead of pruning these states, we suggest to defer their expansion. This can be done by extending the basic directed model checking algorithm introduced on page 16 in Figure 2.1 with a second open queue. States whose expansions we want to defer are inserted in this queue. All other states are inserted in the standard open queue. States from the second queue are only expanded if the standard open queue is empty. The question remains which value we should use for the parameter n. From Definition 4.4, we know that $C_i(t) \subseteq C_j(t)$ for all transitions t if and only if $i \leq j$. On the one hand, the higher the parameter, the better $C(t)$ is approximated. However, as we already argued and also describe in the experimental section, the exact pruning criterion does not fire very often in practice. On the other hand, the lower the parameter the more misclassifications may occur, which hampers the overall performance of the model checking process.

Instead of choosing a constant parameter, we propose to use a multi-queue directed model checking algorithm that maintains $N+1$ different open queues, where $N \in \mathbb{N}$ is defined as above. Overall, we obtain a family of open queues $q_0, \ldots, q_N$, where q_i is accessed (according to the given distance heuristic) iff $q_0, \ldots, q_{i-1}$ are empty, and q_i is not. The basic directed model checking algorithm from Figure 2.1 can be converted into our multi-queue search method by replacing the standard open queue with a multi-queue and the corresponding accessing functions as given in Figure 4.3. In these queues, states are maintained as follows.

1 **function** insert(s', priority):
2   **if** t is innocent **then**:
3   **if** $t' \notin C(t)$ **then**:
4   prune s'
5   **else**:
6   determine smallest $n \in \mathbb{N}$ such that $t' \in C_n(t)$
7   q_n.insert(s', priority)
8   **else**:
9   q_0.insert(s', priority)

1 **function** pop-minimum():
2   determine smallest n such that $q_n \neq \emptyset$
3   **return** q_n.pop-minimum()

Fig. 4.3. Multi-queue accessing functions for context-enhanced directed model checking

Let s be a state that was reached with transition t, and let $s' = t'[s]$ be the successor state of s under the application of the transition t'. If t is innocent and $t' \notin C(t)$, then we can safely prune s'. If t is innocent and $t' \in C(t)$, then the successor state s' is maintained in queue q_1 if $t' \in C_1(t)$, and maintained in q_i if $t' \in C_i(t)$ and $t' \notin C_{i-1}(t)$. If t is not innocent, then s' is stored in q_0.

According to the given distance heuristic, *pop-minimum* returns a state with best priority from the queue q_i with minimal index i that is not empty. The multi-queue is empty iff all queues are empty. Obviously, our algorithm remains complete, as only the order in which the states are explored is influenced. The advantage of this multi-queue algorithm is that we do not have to manually find a good value for n, which strongly depends on the system. By always expanding states from the lowest non-empty queue, the algorithm also respects the quality of the estimated relevance.

4.3 Related Work

In general, context-enhanced directed model checking shares the property with previous transition prioritizing techniques that it labels transitions and defers the expansion of states that are reached by less relevant transitions. In contrast to previous techniques, context-enhanced directed model checking does not assign a Boolean flag to a transition, but an integer value. It is based on a multi queue algorithm that is well-suited to respect the different levels of relevance. In the following, we compare our algorithm to the most relevant techniques in more detail.

First, we reconsider relatively useless transitions. As we discussed in the last chapter, the quality of this technique strongly depends on the accuracy of the applied distance heuristic. In particular, relatively useless transitions cannot be applied to uninformed search methods because no distance heuristic is applied there. Complementary to these limitations, context-enhanced directed model checking is independent of the distance heuristic. Furthermore, as we will see in the experimental section, it is successfully applicable to uninformed search.

Musuvathi and Qadeer work in the area of software model checking. They propose a technique for bug detection based on *iterative context bounding* [58]. For multithreaded programs, they propose an algorithm that limits the number of *context switches*, i. e., the number of execution points where the scheduler forces the active thread to change. Their algorithm is actually a kind of iterative deepening search, where the number of context switches that may occur in each trace is increased in each iteration. They define a context essentially as a thread, and restrict the context switches forced by the scheduler. Compared to context-enhanced directed model checking, such a context switch would correspond to a state where a transition is executed which is not contained in the interference context induced by the preceding transition t. In our setting, a context switch in the sense of Musuvathi

and Qadeer corresponds to two consecutive states s_i and s_{i+1}, where s_i is taken from queue q_n, and s_{i+1} is taken from queue q_m with $n < m$. However, our criterion is stricter than the context switch criterion proposed by Musuvathi and Qadeer: Exploring a state from a deferred queue with greater index corresponds to a context switch in the sense of Musuvathi and Qadeer, but not vice versa. Moreover, we handle the different levels of interference with a fine-grained multi-queue search algorithm. Musuvathi and Qadeer propose an algorithm that is guaranteed to minimize the number of context switches. Contrarily, our search algorithm does not necessarily minimize them. As a consequence, context-enhanced directed model checking performs better than iterative context bounding in systems with tight interaction of the processes.

4.4 Evaluation

We have implemented our algorithm based on interference contexts and empirically evaluate its potential on benchmarks as described in Section 2.4. These include the Single-Tracked Line Segment case study ($S_1, \ldots, S_9$), the Mutual-Exclusion case study ($M_1, \ldots, M_4$ and $N_1, \ldots, N_4$), the Fischer Protocols (F_5, F_{10}, F_{15}) as well as the Towers of Hanoi problems ($H_3, \ldots, H_9$). We evaluate our algorithm on a number of different search methods, including uninformed search as well as various heuristic search methods as proposed in the literature.

4.4.1 Experimental Setting

We have implemented our context-enhanced search algorithm (denoted with *CE* in the following) into our model checker MCTA [56]. All experiments have been performed on an AMD Opteron 2.3 GHz system with 4 GByte of memory. We set a timeout to 30 minutes.

We compare *CE* to classical directed model checking with the rather coarse distance heuristics d^L and d^U [23] as well as to the more informed distance heuristics h^L and h^U [52]. All of them are implemented in MCTA. As already described in more detail in Section 2.3, the d^L and d^U distance heuristics are based on the *graph distance* of automata; synchronization behavior and integer variables are completely ignored. The distance heuristics h^L and h^U are based on the *monotonicity abstraction*. Under this abstraction, variables can have multiple values simultaneously. The h^L heuristic performs a fixed-point iteration under this abstraction starting in the current state until an error state is reached, and returns the number of iterations as heuristic value. Based on this fixed-point iteration, h^U additionally extracts an abstract error trace starting from the abstract error state, and returns the number of abstract transitions as the estimate.

Furthermore, we compare *CE* to various related search algorithms, including iterative context bounding (denoted with *IB*) and useless transitions (denoted with *UT*). As threads correspond to processes in our setting, we have implemented *IB* with process contexts as proposed by Musuvathi and Qadeer [58], denoted with IB_P. This means that a context switch occurs if two consecutive transitions belong to different processes.

Moreover, we implemented *IB* with our definition of interference contexts, denoted with IB_I. Here, a context switch occurs if a transition t' does not belong to $C_n(t)$, where t is the preceding transition. After some limited experiments, we set the bound n to 2 because we achieved the best results in terms of explored states for this value. For larger values, no context switches occurred, and hence IB_I behaves like greedy search.

4.4.2 Experimental Results

We give detailed results for *CE* with the coarse distance heuristic d^U, with the more informed distance heuristic h^L, as well as for the uninformed search method breadth-first search. Moreover, we additionally provide average performance results for depth-first search as well as for the distance heuristics d^L and h^U.

Let us start to discuss the results for *CE* with the d^U heuristic from Table 4.1. First, we observe that for the hard problems, the number of explored states with *CE* is significantly lower than with classical directed model checking (i. e., greedy search without search enhancements) and than with the other, related approaches. In the smaller instances, e. g., the F examples and the small S examples, the performance is comparable. Compared to classical directed model checking, we could also solve more problems when *CE* is applied. This mostly also pays off in better runtime. Finally, the length of the found error trace is comparable in the F and the small H instances, but mostly much shorter than with classical directed model checking and than with *UT*. Overall, when using d^U, *CE* significantly improves directed model checking, and also outperforms related approaches in terms of scalability and error trace length.

The results for *CE* and the h^L heuristic are given in Table 4.2. First, we observe that *UT* performs much better with h^L than with d^U and is often the best configuration. As already outlined, *UT* uses the distance heuristic itself to estimate the quality of a transition. Hence, the performance of *UT* strongly depends on the quality of the applied distance heuristic. This also shows up in our experiments, where *UT* outperforms the other approaches in the F and S examples. On the other hand, *CE* performs best in M, N and H, and only marginally worse than *UT* in most of the F and S problems. Moreover, *CE* scales significantly better than the iterative context bounding algorithm for both context definitions. Overall, we observe that

Table 4.1. Experimental results for greedy search with the d^U heuristic. Abbreviations: plain: greedy search, *CE*: greedy search + interference contexts, IB_P: iterative context bound algorithm with process contexts, IB_I: iterative context bound algorithm with interference contexts, *UT*: useless transitions. Dashes indicate out of memory (> 4 GByte) or out of time (> 30 min). Uniquely best results are given in bold fonts.

	explored states					runtime in s					trace length				
Inst.	plain	*CE*	IB_P	IB_I	*UT*	plain	*CE*	IB_P	IB_I	*UT*	plain	*CE*	IB_P	IB_I	*UT*
F_5	9	21	112	**8**	9	0.0	0.0	0.0	0.0	0.0	6	6	6	6	6
F_{10}	9	36	447	**8**	9	0.0	0.0	0.0	0.0	0.0	6	6	6	6	6
F_{15}	9	51	1007	**8**	9	0.0	0.0	0.0	0.0	0.0	6	6	6	6	6
S_1	11449	10854	44208	14587	**9796**	**0.0**	0.1	0.2	0.2	0.1	823	192	**59**	1022	842
S_2	33859	31986	163020	93047	**31807**	**0.1**	0.3	0.8	0.5	0.5	1229	223	**59**	134	1105
S_3	51526	**43846**	199234	80730	52107	**0.2**	0.3	1.1	0.5	0.8	1032	184	**59**	1584	1144
S_4	465542	**402048**	3.0e+6	1.8e+6	504749	**2.0**	2.1	18.0	8.6	7.5	3132	1508	**60**	783	5364
S_5	4.6e+6	**2.8e+6**	3.8e+7	2.4e+7	4.6e+6	22.6	**18.8**	257	137	70.6	14034	886	**65**	1014	14000
S_6	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
M_1	185416	**5360**	100300	56712	7557	156	0.1	2.5	1.1	0.1	106224	246	**65**	94	923
M_2	56240	**15593**	608115	82318	294877	1.1	**0.2**	22.5	1.7	67.1	13952	520	**63**	3378	51541
M_3	869159	**15519**	623655	281213	26308	1730	**0.2**	23.8	7.2	0.5	337857	562	**82**	3863	1280
M_4	726691	**21970**	3.4e+6	–	100073	428	**0.4**	221	–	1.8	290937	486	**80**	–	4436
N_1	10215	**5688**	92416	68582	19698	0.3	**0.1**	5.6	3.3	0.7	2669	162	**79**	109	1855
N_2	–	**11939**	645665	161800	96307	–	**0.3**	71.7	8.2	5.1	–	182	**75**	277	8986
N_3	–	**14496**	616306	417181	29254	–	**0.4**	62.4	25.8	1.2	–	623	**84**	1586	784
N_4	330753	**25476**	3.7e+6	889438	239877	23.0	**0.8**	761	91.6	17.8	51642	949	**89**	361	1969
H_3	**222**	279	1000	479	244	0.0	0.0	0.0	0.0	0.1	44	**32**	52	82	60
H_4	2276	2133	11726	7631	**545**	0.0	0.1	0.0	0.1	0.5	184	132	116	190	**92**
H_5	20858	**5056**	142359	116200	15435	**0.1**	0.2	0.3	0.4	13.9	708	**238**	266	422	400
H_6	184707	**122473**	1.5e+6	1.1e+6	130213	**0.6**	0.8	4.1	3.2	129	2734	1670	**522**	918	1242
H_7	1.6e+6	**876847**	1.5e+7	1.4e+7	1.2e+6	5.7	**5.4**	52.0	42.0	1268	11202	4456	**1136**	2100	3922
H_8	1.4e+7	**2.3e+6**	–	–	–	53.8	**17.1**	–	–	–	45280	**5666**	–	–	–
H_9	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–

CE with h^L performs similarly to *UT* with h^L, and scales often significantly better than the other approaches.

We also applied our context enhanced search algorithm to uninformed search. The results for breadth-first search are given in Table 4.3. First, we want to stress that breadth-first search is *optimal* in the sense that it returns shortest possible error traces. This is not the case for *CE*. However, the results in Table 4.3 show that the length of the found error traces by *CE* is mostly only marginally longer than the optimal one (around a factor of 2 in the worst case in S_1 and S_2, but mostly much better). Contrarily, the scaling behavior of *CE* is much better than that of breadth-first search. This allows us to solve two more problems on the one hand, and also allows us to solve almost all of the harder problems much faster. In particular, this shows up in the M and N examples, that could be solved within seconds with *CE* (compared to sometimes hundreds of seconds otherwise). Finally, we remark that

Table 4.2. Experimental results for greedy search with the h^L heuristic. Abbreviations: plain: greedy search, *CE*: greedy search + interference contexts, IB_P: iterative context bound algorithm with process contexts, IB_I: iterative context bound algorithm with interference contexts, *UT*: useless transitions. Dashes indicate out of memory ($>$ 4 GByte) or out of time ($>$ 30 min). Uniquely best results are given in bold fonts.

	explored states					runtime in s					trace length				
Inst.	plain	*CE*	IB_P	IB_I	*UT*	plain	*CE*	IB_P	IB_I	*UT*	plain	*CE*	IB_P	IB_I	*UT*
F_5	179	21	112	216	**7**	0.0	0.0	0.0	0.0	0.0	12	6	6	12	6
F_{10}	86378	36	447	95972	**7**	3.3	0.0	0.0	3.8	0.0	22	6	6	22	6
F_{15}	–	51	1007	–	**7**	–	0.0	0.0	–	0.0	–	6	6	–	6
S_1	1704	4903	24590	1971	**1537**	0.0	0.2	0.4	0.1	0.1	84	68	**62**	77	76
S_2	3526	11594	81562	4646	**1229**	0.1	0.4	1.2	0.2	0.1	172	70	**62**	166	76
S_3	4182	5508	89879	6118	**1061**	0.1	0.3	1.4	0.2	0.1	162	73	**62**	109	76
S_4	29167	7728	1.2e+6	108394	**879**	0.7	0.5	14.1	1.2	**0.2**	378	198	**60**	653	77
S_5	215525	2715	1.4e+7	870603	**1116**	5.2	0.4	143	7.4	**0.3**	1424	85	**65**	2815	78
S_6	1.7e+6	9812	–	1.3e+7	**1116**	37.4	1.1	–	85.5	**0.4**	5041	130	–	8778	**78**
S_7	1.6e+7	15464	–	–	**1114**	333	2.0	–	–	**0.6**	15085	130	–	–	**78**
S_8	7.1e+6	2695	–	–	**595**	129	0.5	–	–	**0.3**	5435	81	–	–	**76**
S_9	9.6e+6	4.0e+6	–	–	**2771**	201	193	–	–	**1.3**	5187	1818	–	–	**94**
M_1	4581	**1098**	102739	3230	4256	0.1	**0.0**	3.1	0.1	0.2	457	89	**66**	433	97
M_2	15832	**1904**	605001	53206	7497	0.3	**0.0**	27.8	1.2	0.5	1124	123	**61**	113	104
M_3	**7655**	8257	622426	141247	10733	**0.1**	0.2	26.1	2.8	0.7	748	180	**86**	100	91
M_4	71033	18282	3.3e+6	525160	**16287**	1.6	**0.7**	240	14.5	1.7	3381	334	**81**	118	98
N_1	50869	**1512**	93750	74307	5689	39.0	**0.0**	7.2	59.9	0.3	26053	93	**79**	26029	108
N_2	30476	**2604**	634003	82158	22763	1.2	**0.1**	77.3	3.8	1.5	1679	127	**75**	97	259
N_3	11576	**10009**	607137	177899	35468	0.4	**0.4**	77.1	9.6	2.7	799	224	**86**	106	204
N_4	100336	**20248**	3.7e+6	971927	142946	5.3	**1.1**	756	103	14.9	2455	396	**85**	134	792
H_3	**127**	256	1017	164	190	0.0	0.0	0.0	0.0	0.0	48	48	48	86	62
H_4	2302	764	11830	7488	**620**	0.0	0.1	0.0	0.1	0.0	300	**94**	124	438	114
H_5	20186	**15999**	144668	121027	31553	**0.2**	0.4	0.5	0.6	1.2	1458	478	**252**	878	890
H_6	230878	**85947**	1.5e+6	1.5e+6	281014	2.2	**2.0**	5.8	5.8	13.4	7284	1350	**558**	2070	2766
H_7	2.0e+6	**622425**	1.5e+7	1.6e+7	2.7e+6	21.4	**17.9**	69.0	67.1	155	18500	14314	**1086**	5164	7176
H_8	1.8e+7	**2.1e+6**	–	–	–	207	**78.3**	–	–	–	**70334**	74594	–	–	–
H_9	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–

UT is not applicable with breadth-first search, as no distance heuristic is applied. Overall, we conclude that if *short*, but not necessarily *shortest* error traces are desired, breadth-first search should be applied with *CE* because of the better scaling behavior of *CE* on the one hand, and the still reasonable short error traces on the other hand.

To get a deeper insight into the performance of our context enhanced directed model checking algorithm in practice, we report average results for for *CE* applied to various previously proposed distance heuristics in Table 4.4. For each distance heuristic as well as breadth-first and depth-first search, we average the data on the

Table 4.3. Experimental results for breadth-first search. Abbreviations: plain: breadth-first search, *CE*: breadth first search + interference contexts, IB_P: iterative context bound algorithm with process contexts, IB_I: iterative context bound algorithm with interference contexts. Dashes indicate out of memory (> 4 GByte) or out of time (> 30 min). Uniquely best results are given in bold fonts.

	explored states				runtime in s				trace length			
Inst.	plain	*CE*	IB_P	IB_I	plain	*CE*	IB_P	IB_I	plain	*CE*	IB_P	IB_I
F_5	333	**29**	129	273	0.0	0.0	0.0	0.0	6	6	6	6
F_{10}	5313	**44**	484	3868	0.0	0.0	0.0	0.0	6	6	6	6
F_{15}	34068	**59**	1064	20538	0.4	0.0	0.0	0.3	6	6	6	6
S_1	41517	**19167**	50361	58379	0.1	0.1	0.2	0.3	54	109	59	54
S_2	118075	**52433**	181941	217712	0.4	**0.3**	0.8	0.9	54	109	59	54
S_3	149478	**33297**	230173	264951	0.5	**0.2**	1.1	1.1	54	71	59	54
S_4	1.3e+6	**146094**	3.5e+6	3.7e+6	5.4	**0.7**	18.7	16.2	55	67	60	55
S_5	1.1e+7	**569003**	4.6e+7	4.4e+7	49.8	**3.2**	258.5	227.3	56	73	56	56
S_6	–	**2.5e+6**	–	–	–	**16.2**	–	–	–	**74**	–	–
S_7	–	–	–	–	–	–	–	–	–	–	–	–
M_1	192773	**32593**	273074	116682	4.6	**0.4**	7.4	1.5	**47**	51	50	50
M_2	680288	**75159**	1.7e+6	1.6e+6	19.1	**1.1**	95.2	56.6	50	52	53	50
M_3	740278	**94992**	1.7e+6	1.7e+6	22.0	**1.6**	80.9	76.9	50	54	63	50
M_4	2.6e+6	**189888**	9.7e+6	8.0e+6	87.6	**3.5**	893.7	506.7	53	55	66	53
N_1	361564	**34787**	310157	157610	20.0	**0.7**	15.6	4.3	**49**	56	58	55
N_2	2.2e+6	**81859**	2.4e+6	3.9e+6	399.0	**1.8**	357.6	622.2	52	57	72	52
N_3	2.4e+6	**92729**	2.3e+6	4.2e+6	442.8	**2.5**	275.8	677.9	52	54	71	52
N_4	–	**201226**	–	–	–	**5.9**	–	–	–	**57**	–	–
H_3	448	**315**	1062	954	0.0	0.0	0.0	0.0	22	22	30	24
H_4	4040	**2689**	11702	7228	0.0	0.1	0.0	0.1	**52**	54	64	54
H_5	42340	**19158**	146955	121807	**0.1**	0.2	0.3	0.4	**114**	116	148	148
H_6	377394	**161747**	1.5e+6	1.2e+6	**0.7**	0.8	3.0	2.8	**240**	300	294	278
H_7	3.4e+6	**1.2e+6**	1.5e+7	1.3e+7	7.2	**5.3**	36.8	33.7	**494**	520	634	636
H_8	3.0e+7	**9.2e+6**	–	–	67.2	**44.0**	–	–	**1004**	1302	–	–
H_9	–	–	–	–	–	–	–	–	–	–	–	–

instances that could be solved by all configurations. Furthermore, we give the total number of solved instances.

First, we observe that also on average, *CE* compares favorably to classical directed model checking. The average number of explored states could sometimes by an order of magnitude be reduced, which mostly also pays off in overall runtime. Furthermore, except for breadth-first search which returns shortest possible error traces, the length of the error traces could be reduced. Compared to the related approaches IB_P, IB_I and *UT*, we still observe that the average number of explored states is mostly lower with *CE* (except for the median with *UT* for h^L and h^U). In almost all cases, the average runtime is still much shorter. Apart from breadth-first search, the shortest error traces are obtained with iterative context bounding. However, *CE* scales better than iterative context bounding on average, and could solve

Table 4.4. Summary of experimental results. Abbreviations: plain: (uninformed or informed) search, *CE*: breadth first search + interference contexts, IB_P: iterative context bound algorithm with process contexts, IB_I: iterative context bound algorithm with interference contexts, *DFS*: depth-first search, *BFS*: breadth-first search, average: arithmetic means, solved: number of solved instances (out of 27). Uniquely best results in bold fonts for each configuration.

	explored states		runtime in s		trace length		solved
	average	median	average	median	average	median	
BFS							
plain	1266489.9	277168.5	53.0	2.7	**78.3**	**52.0**	21
CE	**140644.4**	**43610.0**	**1.1**	**0.6**	91.9	55.5	**23**
IB_P	4248527.1	291615.5	102.3	5.2	95.7	59.0	20
IB_I	4133416.8	241331.5	111.5	2.2	89.7	53.5	20
DFS							
plain	445704.1	63712.5	3.6	0.5	37794.3	9624.5	21
CE	**197295.0**	**15688.0**	**1.1**	**0.2**	2308.8	795.5	**22**
IB_P	3397688.6	409480.0	34.8	2.4	**148.7**	**71.0**	21
IB_I	2371171.5	155694.0	12.9	1.5	16477.9	1819.5	21
d^U							
plain	469310.8	42692.5	107.9	0.3	30415.8	1949.0	20
CE	**245070.9**	**13186.5**	**1.6**	**0.2**	665.4	230.5	**22**
IB_P	3503904.9	152689.5	63.8	1.8	**155.0**	**64.0**	21
IB_I	2380880.4	81524.0	16.5	0.8	892.9	391.5	20
UT	394999.4	23003.0	87.6	0.6	4764.3	1124.5	21
d^L							
plain	701962.7	54742.0	14.9	0.5	16414.1	7725.0	21
CE	**189144.3**	**12181.0**	**1.3**	**0.2**	1391.6	562.0	**22**
IB_P	2783118.2	199548.0	72.7	3.7	**100.6**	**67.0**	21
IB_I	1918955.4	105246.0	17.5	0.9	2642.1	798.0	19
UT	344885.2	46509.0	23.8	1.4	13664.1	2521.0	20
h^L							
plain	143536.3	18009.0	4.1	0.4	3327.0	773.5	25
CE	**41090.5**	5205.5	**1.2**	0.3	917.8	108.5	**26**
IB_P	2075198.1	374834.5	72.5	6.5	**150.5**	**70.5**	21
IB_I	1052988.1	89065.0	14.1	2.0	1981.0	126.0	21
UT	164821.9	**4972.5**	9.7	0.3	657.8	97.5	25
h^U							
plain	138325.2	12938.0	3.1	0.3	419.6	112.0	26
CE	**22956.1**	6523.0	**1.0**	0.3	213.0	92.0	**27**
IB_P	1953710.5	145351.0	73.7	7.3	**147.7**	**66.0**	21
IB_I	965865.2	72437.0	12.7	1.6	302.1	116.0	21
UT	106837.0	**2537.0**	9.6	**0.2**	436.3	80.0	26

the most problem instances in *every* configuration. Furthermore, the only configuration that could solve *all* instances is *CE* with the h^U distance heuristic.

Finally, we also compared *CE* with the exact pruning criterion given by Proposition 4.5. However, it turned out that this exact criterion is very strict in practice, and did not fire in any case. This effectively means that the results for this criterion are the same as the results for plain search, except for a slightly longer runtime due to the preprocessing.

Overall, our evaluation impressively demonstrated the power of directed model checking with interference contexts and multiple open queues. We have observed that the overall model checking performance could often significantly be increased for various search methods and distance heuristics on a range of real world problems. The question remains on which parameters the performance of the different search algorithms depend, and in which cases they perform best. We will discuss these points in the next section.

4.4.3 Discussion

Our context-enhanced directed model checking algorithm uses several open queues to handle the different levels of interference. In Table 4.5, we provide additional results about the number of queues and queue accesses for *CE* and *UT*, as well as the number of context switches for iterative context bounding. The data is averaged over all instances of the specific case study.

For our context-enhanced directed model checking algorithm, we report the number of queues that are needed for the different problems, as well as the number of pushed and popped states of these queues. First, we observe that in the F examples, only one deferred queue (q_1) is needed, which is accessed very rarely. However, the situation changes in the S, M and N examples, where three deferred queues are needed. Deferred states are explored from up to two deferred queues (for d^U and breadth-first search), whereas the last non-empty deferred queue is never accessed. Most strikingly, in the H examples, we need six deferred queues, from which most of them are never accessed. Overall, the performance of *CE* also depends on the number of explored deferred states. If there are deferred queues that never have to be accessed at all, the corresponding states do not have to be explored, and the branching factor of the system is reduced.

We applied the useless transitions technique in a multi-queue setting, where *useless* states, i. e., states reached by a relatively useless transition, are maintained in a separate queue. Table 4.5 shows the number of explored non-useless states (q_0) as well as the number of explored useless states (q_1). Obviously, the number of explored useless states also indicates the precision of *UT* for the particular problem. We observe that for the well-informed h^L distance heuristic, q_1 is never accessed except for the H examples, which explains the good performance of *UT* in this case. However, for the coarser d^U distance heuristic, q_1 is accessed very often, which explains the favorable performance of *CE* over *UT* with d^U.

Table 4.5. Average number of queue accesses for queues $q_0, \ldots, q_6$ for *CE* and *UT*, and average number of context switches for *IB* per case study. Number of pushed states at the top, number of popped states at the bottom.

	Accesses with *CE*							Accesses with *UT*		Context Switches	
	q_0	q_1	q_2	q_3	q_4	q_5	q_6	q_0	q_1	IB_P	IB_I
h^L heuristic											
F	38	114	0	0	0	0	0	13	24	2	0
	35	1	0	0	0	0	0	7	0		
S	169161	742370	87766	136105	0	0	0	1702	2440	31.6	0
	169144	277922	0	0	0	0	0	1268	0		
M	5585	5687	8598	3455	0	0	0	12308	9137	21	3
	5584	1801	0	0	0	0	0	9693	0		
N	6654	6547	9384	4124	0	0	0	69534	4332	22.3	3
	6652	1941	0	0	0	0	0	51716	0		
H	377962	294737	230682	297175	345419	445245	612451	404205	308105	134.8	15.6
	377935	93051	0	0	0	0	0	404186	204806		
d^U heuristic											
F	38	114	0	0	0	0	0	14	24	2	0
	35	1	0	0	0	0	0	9	0		
S	287161	395163	65564	273596	0	0	0	12746	1042207	31.6	3
	287149	379679	520	0	0	0	0	12746	1030510		
M	9712	7734	11949	5043	0	0	0	23355	115058	21	4
	9708	4902	0	0	0	0	0	23354	83849		
N	10006	7943	11920	6614	0	0	0	16204	109769	21.5	4.8
	10000	4399	0	0	0	0	0	16204	80079		
H	379149	289671	222272	289380	323717	385445	498216	117141	220791	134.8	15.2
	379100	164472	0	0	0	0	0	117141	143929		
BFS											
F	56	114	0	0	0	0	0	n/a	n/a	2	0
	43	1	0	0	0	0	0	n/a	n/a		
S	323745	607706	189050	447035	0	0	0	n/a	n/a	31.6	0
	323704	229784	1111	0	0	0	0	n/a	n/a		
M	50251	56932	86116	30730	0	0	0	n/a	n/a	21	1.3
	50249	47908	0	0	0	0	0	n/a	n/a		
N	55105	55314	92173	39908	0	0	0	n/a	n/a	20.7	1.7
	55103	47546	0	0	0	0	0	n/a	n/a		
H	894952	925034	649694	843886	861079	980240	1476125	n/a	n/a	134.8	20.8
	894927	874677	0	0	0	0	0	n/a	n/a		

Finally, let us give some explanations of the performance of the iterative context bound algorithm *IB* compared to *CE*. The *IB* approach is guaranteed to minimize the number of context switches, and obviously performs best in systems where not many context switches are needed. Contrarily, if the context has to be switched n times, the whole state space for all context switches smaller than n has to be explored until an error state is found, which could be problematic if n is high. Table 4.5 shows the average number n of context switches needed to find an error state in our examples.[1] We observe that, except for the F examples, n is pretty high for IB_P. Contrarily, in the F examples where the context has to be switched only two times, iterative context bounding performs very well. Overall, we conclude from our experiments that the approach proposed by Musuvathi and Qadeer works well for programs with rather loose interaction where not many context switches are required. However, in protocols with tight interaction and many required changes of the active threads, directed model checking with interference contexts performs much better.

4.5 Conclusions

In this chapter, we have introduced a fine-grained heuristic technique based on interference contexts to prioritize transitions. Based on this technique, we have presented a context-enhanced directed model checking algorithm that respects the different levels of transition relevance. The experimental evaluation impressively demonstrated the practical potential of our algorithm for various distance heuristics from the literature on large and realistic case studies. We could improve the overall search behavior compared to various related search algorithms in terms of number of explored states and search time.

For the future, it will be interesting to refine and extend our algorithm based on interference contexts. This includes, for example, to take into account the structure of our computational model more explicitly. In particular, we plan to better adapt our technique to timed systems. Although we are able to handle such systems, our technique is not yet optimized for them as clocks are currently ignored. We expect that taking clocks into account will lead to further improvements for that class of systems. Furthermore, the multi-queue search algorithm with interference contexts could also be adapted to the area of AI planning as we have done with useless transitions. As pointed out in the previous chapters, techniques to alleviate the state explosion problem are also strongly required in the area of AI planning. It will be interesting to investigate if performance improvements with techniques based on interference contexts can also be achieved in that area. Finally, it will also be in-

[1] As a side remark, the different number of context switches for h^L, d^U and *BFS* with IB_P and IB_I are due to the different number of solved instances of these configurations.

teresting to combine our algorithm with related transition prioritizing techniques like useless transitions. In particular, this includes to investigate on which parameters the overall performance depends when combining context-enhanced directed model checking with useless transitions, and which combination performs best. We expect that further synergy effects of these techniques are possible, and even better scaling behavior will be achieved.

5

The Causal Graph Revisited for Directed Model Checking

In the previous chapters, we have introduced the concepts of useless transitions and interference contexts to prioritize transitions for directed model checking. However, for both of these concepts, there is no theoretical guarantee that transitions that are expected to be not needed can be pruned. Therefore, in order to not miss any error state, we assigned lower priorities to such transitions, but have not pruned them completely. Complementary to these concepts, we propose a criterion to prioritize transitions in this chapter that allows for pruning transitions. Furthermore, we propose a distance heuristic for directed model checking that is an adaptation of the *causal graph heuristic* from the area of AI planning. Both contributions are based on the *causal graph* structure, which is a well known structure in AI planning.

The overall chapter is based on joint work with Helmert [76]. It is organized as follows. We start by giving additional notation in Section 5.1, and formally introduce the causal graph structure in Section 5.2. Based on causal graph analysis, we present our transition prioritizing technique in Section 5.3. Furthermore, we propose the adaptation of the causal graph heuristic from AI planning to directed model checking in Section 5.4. After an empirical evaluation of our techniques in Section 5.5, we finally conclude the chapter and discuss future work in Section 5.6.

5.1 Notation

In order to concisely present our techniques based on causal graph analysis in this chapter, we first introduce some new terminology. In particular, we come up with the notions of *variable domain processes* and *variable domain systems*, and describe how systems as considered so far can be translated to this formalism. This will allow us to concisely present the central ideas of this chapter. Roughly speaking, for a given system $\mathcal{M}$, the idea is to represent $\mathcal{M}$ by modeling $\mathcal{M}$'s processes and integer variables as variable domain processes that run in lockstep. The location set of a variable domain process corresponds to the domain of the modeled process or

integer variable. Transitions $t \in \mathcal{T}(\mathcal{M})$ correspond to synchronization labels from a global synchronization alphabet Σ. For a system $\mathcal{M}$, we will use the term *system variables* to denote both processes and integer variables of $\mathcal{M}$.

Definition 5.1 (Variable domain process).
Let $\mathcal{M} = (\mathcal{P}, V)$ be a system, $\mathcal{T}(\mathcal{M})$ be the transition set of $\mathcal{M}$, and $\Sigma = \{c_t \mid t \in \mathcal{T}(\mathcal{M})\}$ be the synchronization alphabet containing a label for each transition in $\mathcal{M}$. Let v be a system variable of $\mathcal{M}$, i. e., either a process in $\mathcal{P}$ or an integer variable in V. The variable domain process *for v is defined as the labeled directed graph $p = (L, E)$, where L is a finite set of* locations*, and $E \subseteq L \times \Sigma \times L$ is the set of* edges *of p, where*

- *$L = L_i$ if v corresponds to a process $p_i = (L_i, l_i, E_i)$ in $\mathcal{P}$, or $L = \mathrm{dom}(v)$ if v corresponds to an integer variable in V, and*
- *there is an edge $(v_1, c_t, v_2) \in E$ if there is a transition $t \in \mathcal{T}(\mathcal{M})$ such that the two following conditions hold.*
  - *The transition guard of t is consistent with the constraint $v = v_1$, i. e., $guard(t) \wedge v = v_1 \not\models \bot$.*
  - *If there is an assignment $v := v' \in effect(t)$, then $v' = v_2$. If there is no such assignment, then $v_1 = v_2$.*

The above definition gives a concise description of variable domain processes to model the domain of processes and integer variables and to model under which circumstances the values of these domains can change. There is an edge between locations l_1 and l_2 if there is a transition $t \in \mathcal{T}(\mathcal{M})$ that could possibly change the value of v from l_1 to l_2. Self loops encode the implicit effect that the value of a variable remains unchanged if it is not set otherwise. We remark that variable domain processes are finite because the domains of systems $\mathcal{M}$, i. e., the location sets of processes and the domain of integer variables, are finite as well. Furthermore, we remark that variable domain processes correspond to the notion of *domain transition graphs* as used in the area of AI planning [36].

A system consisting of variable domain processes is called *variable domain system*. Such a system is defined as the parallel composition of the variable domain processes of a given system $\mathcal{M}$.

Definition 5.2 (Variable domain system).
Let $\mathcal{M} = (\mathcal{P}, V)$ be a system with processes $\mathcal{P} = p_1 \parallel \ldots \parallel p_n$ and integer variables $V = \{v_1, \ldots, v_m\}$. Let $p'_1, \ldots, p'_n$ be the variable domain processes for $p_1, \ldots, p_n$, and $p'_{n+1}, \ldots, p'_{n+m}$ be the variable domain processes for $v_1, \ldots, v_m$. The variable domain system *for $\mathcal{M}$ is defined as*

$$\mathcal{S}(\mathcal{M}) = p'_1 \parallel_{\mathcal{S}} \ldots \parallel_{\mathcal{S}} p'_{n+m},$$

where $\parallel_{\mathcal{S}}$ is a parallel composition symbol for which the semantics is defined below.

In the following, we use the notation $\mathcal{S}(\mathcal{M})$ for a variable domain system corresponding to the system $\mathcal{M}$. The semantics of $\mathcal{S}(\mathcal{M})$ is defined as a system of processes that run in lockstep using global synchronization labels. Whenever a given variable domain process performs an edge from location l to l' with associated label $c_t \in \Sigma$, then all other variable domain processes must simultaneously perform an edge with the same label c_t, or else the edge is not permitted. Overall, this gives rise to the following definition of the semantics of $\mathcal{S}(\mathcal{M})$. We first define the semantics for variable domain systems with two variable domain processes.

Definition 5.3 (Semantics of variable domain systems).
Let $\mathcal{S}(\mathcal{M}) = p_1 \parallel_{\mathcal{S}} p_2$ be a variable domain system for a system $\mathcal{M}$ and variable domain processes $p_1 = (L_1, E_1)$ and $p_2 = (L_2, E_2)$. The semantics of $\mathcal{S}(\mathcal{M})$ is defined as a variable domain process (L, E) with location set $L = L_1 \times L_2$ and the set of edges $E = \{((l_1, l_2), c_t, (l_1', l_2')) \mid (l_1, c_t, l_1') \in E_1 \wedge (l_2, c_t, l_2') \in E_2\}$.

A global state of $\mathcal{S}(\mathcal{M})$ is given as a function that maps p_1 and p_2 to one of its locations. The initial global state of $\mathcal{S}(\mathcal{M})$ is determined by the initial values of the processes and integer variables of $\mathcal{M}$.

A trace $\pi = s_0, c_{t_0}, s_1, c_{t_1}, \ldots, c_{t_{n-1}}, s_n$ of $\mathcal{S}(\mathcal{M})$ is an alternating sequence of global states and synchronization labels starting from the initial global state such that $(s_{i-1}, c_{t_{i-1}}, s_i) \in E$ for all $i \in \{1, \ldots, n\}$. The length of a trace π, denoted with $|\pi|$, is defined as the number of synchronization labels in π, i. e., $|\pi| = n$ for the trace given above.

To distinguish between (local) edges of variable domain processes and (global) edges of variable domain systems, we will call the latter also global transitions. We observe that the semantics of $\parallel_{\mathcal{S}}$ is defined as a form of parallel composition which is an associative and commutative operation, up to isomorphism. For example, we can obtain $p_2 \parallel_{\mathcal{S}} p_1$ from $p_1 \parallel_{\mathcal{S}} p_2$ by renaming locations (l_1, l_2) to (l_2, l_1). The semantics of systems with more than two variable domain processes is defined accordingly. Overall, a variable domain system differs in two ways from systems $\mathcal{M}$ as defined in this thesis so far. First, *global* synchronization is needed to perform a global transition (instead of interleaved or binary synchronized transitions). Second, there are no explicit integer variables in $\mathcal{S}(\mathcal{M})$, but they are modeled as variable domain processes as well. To illustrate these definitions, consider the following example model checking task in Figure 5.1.

The example model checking task $\Theta = (\mathcal{M}, \varphi)$ consists of an asynchronous system $\mathcal{M}$ with three parallel processes P_1, P_2 and P_3. The transitions of $\mathcal{M}$ are interleaving transitions, i.e., $\mathit{label}(t) = \tau$ for all $t \in \mathcal{T}(\mathcal{M})$. For the sake of readability, these labels are omitted in Figure 5.1. Furthermore, $\mathcal{M}$ consists of integer variables v and w with initial value $v = 0$ and $w = 0$ and domain $\mathrm{dom}(v) = \mathrm{dom}(w) = \{0, 1, 2, 3\}$. Global error states in $\mathcal{M}$ are global states where P_3 is in the double circled location l_3. The edges in P_1 and P_2 are abbreviations

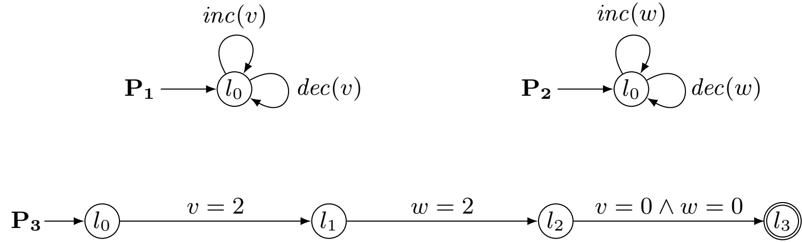

Fig. 5.1. An example system with three processes.

for a set of edges that increase and decrease v or w modulo the size of the domain of v and w, respectively. To be more precise, $inc(v) = v + 1 \mod 4$ and $dec(v) = v - 1 \mod 4$. The model checking task with corresponding variable domain system is depicted in Figure 5.2.

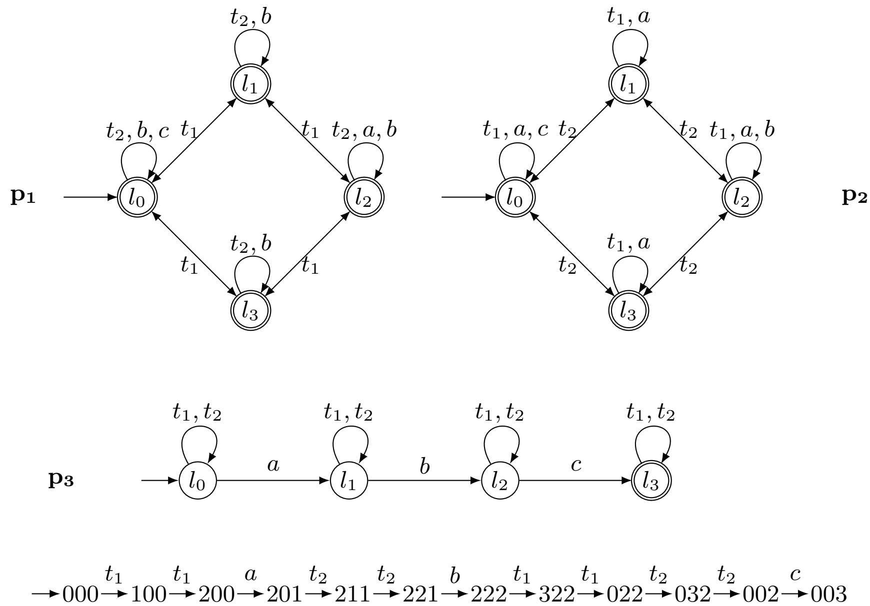

Fig. 5.2. The variable domain system for the example system of Figure 5.1 and an example for an error trace. An edge with more than one label is an abbreviation for several parallel edges, one for each label.

In the variable domain system $\mathcal{S}(\mathcal{M})$, the integer variables v and w from $\mathcal{M}$ are translated to variable domain processes p_1 and p_2. Global error states in $\mathcal{S}(\mathcal{M})$ are defined as states where every variable domain process is in one of its error lo-

cations, where error locations are determined by the error formula φ; in variable domain processes that correspond to system variables in $\mathcal{M}$ that are not specified in φ, *every* location is an error location. In the above example, error locations are double circled. Transitions to manipulate v and w in $\mathcal{M}$ are represented with synchronization labels t_1 and t_2 in $\mathcal{S}(\mathcal{M})$[1]. The transitions induced by P_3 in $\mathcal{M}$ are represented by the synchronization labels a, b and c in $\mathcal{S}(\mathcal{M})$. Figure 5.2 also shows a shortest error trace in $\mathcal{S}(\mathcal{M})$. To reach a global error state in $\mathcal{S}(\mathcal{M})$, we first have to move from l_0 to l_2 in p_1 in order to synchronize with a in p_3. This corresponds to increasing the integer variable v twice in $\mathcal{M}$. Similarly, in $\mathcal{S}(\mathcal{M})$, we have to move from l_0 to l_2 in p_2 in order to be able to synchronize with b. This corresponds to increasing w twice in $\mathcal{M}$. Finally, in $\mathcal{S}(\mathcal{M})$, we have to move back to l_0 in p_1 and p_2 in order to synchronize with c in p_3. This corresponds to decreasing v and w to zero in $\mathcal{M}$.

Overall, we have seen that a variable domain system $\mathcal{S}(\mathcal{M})$ of a system $\mathcal{M}$ is a different encoding for $\mathcal{M}$. In particular, it has the property that every error trace of $\mathcal{M}$ corresponds to an error trace of $\mathcal{S}(\mathcal{M})$ with the same length, and vice versa. Again, we remark that we do this effort of introducing such an additional notation for ease of presentation. We will see that the central ideas and definitions of our contributions of this chapter are presented very intuitively within this model. As we have defined the two formalisms in an isomorphic way, we directly work on variable domain systems in this chapter. A model checking task $\Theta = (\mathcal{M}, \varphi)$ translates to the new formalism in a straight forward way: the goal is to find a trace in $\mathcal{S}(\mathcal{M})$ that leads to a global state s such that the relevant processes in $\mathcal{S}(\mathcal{M})$ satisfy the error condition in s with respect to φ.

Definition 5.4 (Model checking task for variable domain systems).
Let $\Theta = (\mathcal{M}, \varphi)$ be a model checking task. The model checking task for variable domain system *corresponding to Θ is defined as $(\mathcal{S}(\mathcal{M}), L^*)$, where $\mathcal{S}(\mathcal{M})$ is the variable domain system for $\mathcal{M}$. The error locations $L^* = (L_1^*, \dots, L_n^*)$ with $L_i^* \subseteq L_i$ for all $i \in \{1, \dots, n\}$ for variable domain process $p_i = (L_i, E_i)$ in $\mathcal{S}(\mathcal{M})$ are determined by the values specified by φ for p_i, or $L_i^* = L_i$ if no value is specified. An error trace of Θ is a trace of $\mathcal{M}$ that ends in a global state $s \in L_1^* \times \cdots \times L_n^*$.*

If the meaning is clear from the context, then variable domain processes, variable domain systems and model checking tasks for variable domain systems will often be shortly called processes, systems and model checking tasks, respectively.

[1] To keep things as simple as possible in this example, we do not represent the original processes P_1 and P_2 from $\mathcal{M}$ in $\mathcal{S}(\mathcal{M})$ because they only have one location and therefore represent constants. Furthermore, we use only one synchronization label for transitions that manipulate v and w, i. e., one label for each variable (the formal translation would require a separate label for each transition).

5.2 The Causal Graph

In this section, we introduce the central concepts based on causal graph analysis, namely the *causal graph* and the *local subsystems* it induces. The causal graph represents a dependency structure of the given system that reflects how processes in variable domain systems depend on other processes in a certain sense. Our contributions in this chapter exploit the information gained from the structural analysis given by the causal graph. To provide some intuition for our definitions, in particular for the causal graph distance heuristic, we illustrate them with the example model checking task from Figure 5.2 as a running example. Recall that this system consists of three processes p_1, p_2 and p_3, each with locations $\{l_0, l_1, l_2, l_3\}$ and edges as shown in the figure. In the initial global state s_0, all processes are in location l_0. Global error states are defined by the property that process p_3 is in location l_3 (i. e., $L_1^* = L_2^* = \{l_0, l_1, l_2, l_3\}$ and $L_3^* = \{l_3\}$). The shortest error traces for this example have length 11 as we have discussed in the last section, and indeed this is the distance estimate that the causal graph distance heuristic will assign in this case. However, other distance heuristics considered in the directed model checking literature underestimate the true error distance. We briefly discuss this observation with respect to the new formalism in the running example.

- The d^L and d^U distance heuristics [23, 25] measure the graph distance to the nearest location L_i^* in each automaton p_i without taking into account synchronization labels. The d^L heuristic maximizes over the individual distances, whereas d^U sums these values. We obtain $d^L(s_0) = \max\{0, 0, 3\} = 3$ and $d^U(s_0) = 0 + 0 + 3 = 3$ because only p_3 needs to move to a different location (l_3), which can be reached from l_0 in three steps.
- The h^L and h^U distance heuristics [52] compute abstract error traces under the *monotonicity abstraction*. In the context of our running example, h^U considers an abstracted problem where each process can "jump back to" a previously visited location at every step free of cost. In this case, we obtain $h^U(s_0) = 7$ because h^U fails to take into account that processes p_1 and p_2 must return to location l_0 from l_2 in order to support the edge of p_3 from l_2 to l_3 via synchronization label c. The h^L heuristic has the same weakness as h^U but additionally assumes in its abstraction that the required edges of p_1 and p_2 from l_0 to l_2 can be performed simultaneously, leading to an estimate of $h^L(s_0) = 5$.

A common weakness of all these distance heuristics, which causes the imperfect distance estimates, is that they fail to take into account that reaching a certain location of p_3 has a *side effect* on p_1 and p_2. In particular, they assume that as soon as p_3 has reached l_2, the error location l_3 can be reached immediately in one step. The edge of p_3 from l_2 to l_3 requires p_1 and p_2 to follow an edge with label c, and the

initial locations of p_1 and p_2 have outgoing edges with this label from their locations in s_0, which is good enough for h^U and h^L (recall that d^L and d^U do not care about synchronization at all). The distance heuristics do not recognize that p_1 and p_2 must initially *move away* from l_0 (to location l_2) before p_3 reaches l_2 in order to synchronize on the labels a (for p_1) and b (for p_2).

We will see that the causal graph distance heuristic overcomes this limitation by finding error traces in simple cases like this example *directly*, without further abstraction, while distances in "larger" systems are computed by combining information from smaller subsystems. To make this more precise, we must introduce the notion of causal graph. To motivate the following definition, observe that the labels from $\Sigma = \{t_1, t_2, a, b, c\}$ play very different roles for the three processes in the example system:

- Label t_1 is very important for process p_1 because all proper (non-looping) edges between locations of p_1 must synchronize on this label. We say that a label $a \in \Sigma$ *affects* a process (L, E) if $(l, a, l') \in E$ for some $l \neq l'$. In the example, t_1 affects p_1, t_2 affects p_2 and a, b and c affect p_3.
- Labels a and c do not cause non-looping edges in p_1, but they are still relevant for the process because the current location of p_1 influences whether or not the overall system can synchronize on these labels. For example, the system cannot synchronize on a unless p_1 is in location l_2. We say that a label $a \in \Sigma$ *restricts* a process (L, E) if there exists a location $l \in L$ such that for all $l' \in L$, $(l, a, l') \notin E$. In the example, a restricts p_1 and p_3, b restricts p_2 and p_3, and c restricts all processes.
- Finally, labels t_2 and b are completely irrelevant for process p_1: no matter in which location the process is, it can synchronize on these labels, and they cannot cause a change in location. We say that a label $a \in \Sigma$ is *irrelevant* for a process (L, E) if it does not affect or restrict the process. In the example, t_1 is irrelevant for p_2 and p_3, t_2 is irrelevant for p_1 and p_3, a is irrelevant for p_2, and b is irrelevant for p_1.

Using these different roles for labels and processes, we define the *causal graph* of a variable domain system as follows.

Definition 5.5 (Causal graph).

For a system $\mathcal{M}$, let $\mathcal{S}(\mathcal{M}) = p_1 \parallel_{\mathcal{S}} \ldots \parallel_{\mathcal{S}} p_n$ be a variable domain system with synchronization alphabet Σ. The causal graph $CG(\mathcal{S}(\mathcal{M}))$ *of $\mathcal{S}(\mathcal{M})$ is the directed graph whose vertices are the component processes $p_1, \ldots, p_n$ of $\mathcal{S}(\mathcal{M})$. It contains an arc from p_i to p_j iff $i \neq j$ and there exists a label that restricts or affects p_i and affects p_j.*

Intuitively, the causal graph contains an arc from variable domain process p_i to variable domain process p_j if there may be a need to change the location of p_i in

order to change the location of p_j. There is *no* arc if p_i can change every location independently of the current location of p_j, and changing locations in p_i has no side effect on the current location in p_j either. The causal graph of the running example from Figure 5.2 is shown in Figure 5.3. We have an arc from p_1 to p_3 because the synchronization label a restricts p_1 and also affects p_3. As location l_2 is the only location in p_1 where we can synchronize with a, we possibly have to move to that location in order to be able to synchronize with a in p_3. A symmetrical argument holds for processes p_2 and p_3 for synchronization label b. Finally, we observe that there is no arc between p_1 and p_2 because these processes can change their locations independently of the current locations of each other.

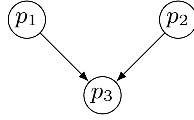

Fig. 5.3. The causal graph for the running example system

To translate this intuition into a formal result, we first introduce the notion of *subsystems*. Informally speaking, for a given system $\mathcal{M}$ and corresponding variable domain system $\mathcal{S}(\mathcal{M})$, a subsystem of $\mathcal{S}(\mathcal{M})$ induced by some processes is defined as the variable domain system where these processes are removed from $\mathcal{S}(\mathcal{M})$.

Definition 5.6 (Subsystem).
For a system $\mathcal{M}$, let $\mathcal{S}(\mathcal{M}) = p_1 \parallel_\mathcal{S} \ldots \parallel_\mathcal{S} p_n$ be a variable domain system with initial global state s_0, and let $\Theta = (\mathcal{S}(\mathcal{M}), (L_1^*, \ldots, L_n^*))$ be a model checking task for $\mathcal{S}(\mathcal{M})$. Let $P = \{p_{i_1}, \ldots, p_{i_k}\}$, $1 \leq i_1 < \cdots < i_k \leq n$ be a subset of the component processes of $\mathcal{S}(\mathcal{M})$. The variable domain system

$$\mathcal{S}(\mathcal{M})[P] := p_{i_1} \parallel_\mathcal{S} \cdots \parallel_\mathcal{S} p_{i_k}$$

with initial global state $s_{0_P} = \langle s_{0i_1}, \ldots, s_{0i_k} \rangle$ is called the subsystem *of $\mathcal{S}(\mathcal{M})$ induced by P, and the model checking task $\Theta[P] := (\mathcal{S}(\mathcal{M})[P], (L_{i_1}^*, \ldots, L_{i_k}^*))$ is called the* subtask *of Θ induced by P.*

We remark that removing a variable domain process p in $\mathcal{S}(\mathcal{M})$ corresponds to removing the corresponding variable in $\mathcal{M}$ in the following sense. Let v be the corresponding variable to p in $\mathcal{M}$. If v corresponds to a process p_v in $\mathcal{M}$, then p_v is "shrunken" such that the resulting process only contains one location, and there are looping edges that assure that every integer assignment and binary synchronization label in p_v is preserved. If v corresponds to an integer variable in $\mathcal{M}$, every integer constraint and assignment containing v is removed in $\mathcal{M}$.

It is easy to see that for any choice of P, $\mathcal{S}(\mathcal{M})[P]$ is an *over-approximation* of $\mathcal{S}(\mathcal{M})$: every trace π of $\mathcal{S}(\mathcal{M})$ induces a corresponding trace of $\mathcal{S}(\mathcal{M})[P]$, which can be obtained from π by projecting all states to the components in P. Moreover, every error trace for Θ is an error trace for $\Theta[P]$. Of course, the converse is not true in general, and the existence of error traces for $\Theta[P]$ does not imply that there are error traces for Θ. However, there is a simple sufficient criterion under which all error traces of $\Theta[P]$ *do* correspond to error traces of Θ with the same synchronization sequence: namely, if P includes all processes with a non-trivial error location set (i. e., processes p_i for which not all locations are in the set L_i^*), as well as all causal graph *ancestors* of such processes. This is essentially the idea of *cone-of-influence reduction* [16].

In fact, cone-of-influence reduction is still error-preserving if we consider an alternative definition of causal graphs where we only introduce an arc from p_i to p_j if some label *restricts* p_i and affects p_j. The reason why we also include arcs from p_i to p_j if some common label *affects* both of them is that this gives an additional decomposition result, which is essential for the safe abstraction technique that is introduced in the next section.

As a side remark, under our definition, if the causal graph consists of more than one weakly connected component, then there exists an error trace iff each subtask induced by a weakly connected component has an error trace. (The overall error trace is then essentially the concatenation of these "subtraces".) The intuitive reason for this property is that if two sets of processes P and P' are causally disconnected, then all state transitions that affect the locations of processes in P are not restricted by or affect the locations of processes in P' (and vice versa), and hence the corresponding subtasks can be addressed independently. A similar decomposition result does not hold under the alternative definition of causal graphs, where traces that affect the processes P are not restricted by the processes P', but can still change the locations of P' as a side effect.

5.3 Safe Abstraction

In this section, we introduce the safe abstraction technique which is guaranteed to not introduce spurious error states. In particular, we will see that every error trace in an abstracted system corresponds to a concrete error trace in the original system. The approach is based on *independent* variable domain processes that do not have any predecessors in the causal graph. We call such processes independent because by definition of causal graphs, they can change locations independently of and without affecting the locations of other processes. An independent process p is called *safe* if there are no dead ends in p, i. e., if there is a path from every location in p to every other location.

Definition 5.7 (Safe process).

For a system $\mathcal{M}$, let $\Theta = (\mathcal{S}(\mathcal{M}), L^*)$ be a model checking task with variable domain system $\mathcal{S}(\mathcal{M})$ and error locations L^*. A variable domain process p in $\mathcal{S}(\mathcal{M})$ is safe *iff p has no predecessors in the causal graph $CG(\mathcal{S}(\mathcal{M}))$, and p is strongly connected.*

Let p be a safe process. For the following considerations, we assume without loss of generality that

- the error location set for p is not empty (otherwise, trivially there exist no error states, since the error states are formed as the Cartesian product of the error location sets of the processes), and
- each label a that occurs in an edge of any process also occurs in an edge of p (otherwise edges with label a can never be synchronized, and we can remove all such edges in a preprocessing step).

In this case, it is possible to separate the edges for p from the rest of the model checking task completely. Let $\mathcal{S}(\mathcal{M})[P']$ be the subsystem induced by all processes of $\mathcal{S}(\mathcal{M})$ except p, i. e., $P' = \{p_1, \ldots, p_n\} \setminus \{p\}$. Given a global state s of $\mathcal{S}(\mathcal{M})$ and an error trace π' for $\mathcal{S}(\mathcal{M})[P']$ that starts in the projection of s to P', we can compute an error trace for $\mathcal{S}(\mathcal{M})$ starting from s with a simple polynomial algorithm:

- If $|\pi'| = 0$ (i. e., π' is the empty trace), then the projection state of s to P' is already an error state for $\mathcal{S}(\mathcal{M})[P']$, and hence all processes except possibly p are in an error location in state s. Because p is strongly connected and has at least one error location, we can find a sequence of edges of p that lead from its location in s to an error location of p. Because p is independent, these edges are not restricted by the other processes and do not affect their locations. By following these edges, we can go from state s to a global error state of $\mathcal{S}(\mathcal{M})$.
- If $|\pi'| = n \geq 1$, then the trace starts with some global edge (s', a, t') of $\mathcal{S}(\mathcal{M})[P']$. Because p is strongly connected and has at least one location with an outgoing edge labeled a, we can find a sequence of edges of p that lead from its location in s to a location in which p can synchronize on a. Because p is independent, these edges are not restricted by the other processes and do not affect their locations. By following these edges, we can go from state s to a state $\tilde{s}$ whose projection to P' is s' and in which all processes can synchronize on a, and from there to a state t whose projection to P' is t'. Since t' starts an error trace of length $n - 1$ in $\mathcal{S}(\mathcal{M})[P']$, we can reach an error state of $\mathcal{S}(\mathcal{M})$ from t (and hence, from s) by an inductive argument.

The analysis shows that if the independent process p is strongly connected, there exists a *safe abstraction* of $\mathcal{S}(\mathcal{M})$ to P': any error trace of $\mathcal{S}(\mathcal{M})[P']$ induces an error trace of $\mathcal{S}(\mathcal{M})$, and of course the converse is also true because subsystems are

always over-approximations. Under these circumstances, we can run the directed model checking algorithm directly on the subsystem $\mathcal{S}(\mathcal{M})[P']$ instead of $\mathcal{S}(\mathcal{M})$, and then apply the above procedure to convert the error trace for the abstracted problem into a concrete one. Of course, the abstraction may cascade, as $\mathcal{S}(\mathcal{M})[P']$ may admit further safe abstractions, even for processes that were not originally independent in P.

In our running example from Figure 5.2, we observe that the processes p_1 and p_2 are safe since they are independent and strongly connected. The resulting subsystem consists of p_3. Performing directed model checking on that subsystem results in the simple spurious error trace $\pi = l_0, a, l_1, b, l_2, c, l_3$ that leads from location l_0 to l_3 in p_3. Spurious edges in π can only be caused by the safe processes p_1 and p_2. These are resolved according to the above procedure: for example, the first spurious edge (l_0, a, l_1) in π is resolved by first moving in p_1 from location l_0 to location l_2 to enable synchronization with a.

5.4 The Causal Graph Heuristic for Directed Model Checking

In this section, we introduce the *causal graph distance heuristic* to directed model checking [76]. It has originally been introduced in the area of AI planning [35, 36] and was an important ingredient of FAST DOWNWARD, the winner in the track for classical planning of the international planning competition 2004.

For a variable domain system $\mathcal{S}(\mathcal{M})$, the causal graph heuristic estimates the cost of reaching a global error state in $\mathcal{S}(\mathcal{M})$ by computing distance estimates for a number of subtasks which are derived by looking at small "windows" of the causal graph. In this section, we describe this procedure conceptually as a bottom-up computation along a topological sorting of the causal graph. (In a practical implementation, a top-down implementation is more efficient, but both approaches lead to the same distance estimates.) Since we require a topological sorting of the causal graph, the procedure only works for acyclic causal graphs; we will later explain how to deal with the cyclic case. For now, let us just remark that deciding the existence of error traces is already PSPACE-complete for systems with acyclic causal graphs, even under the further restriction that all processes have only two locations [11].

Throughout this section, we assume that we are given a model checking task $\Theta = (\mathcal{S}(\mathcal{M}), (L_1^*, \dots, L_n^*))$ for a variable domain system $\mathcal{S}(\mathcal{M}) = p_1 \parallel_{\mathcal{S}} \dots \parallel_{\mathcal{S}} p_n$ with initial global state s_0. Our objective is to compute a distance estimate for a given global state s of $\mathcal{S}(\mathcal{M})$, which we denote as $h^{CG}(s)$. For each process $p_i = (L_i, E_i)$ and each pair of locations $l_i, l_i' \in L_i$, the causal graph heuristic computes a distance estimate $cost_{p_i}(l_i, l_i')$ for the cost of changing the location of p_i from l_i to l_i'. The overall distance estimate of s is then defined as the sum of the costs of reaching the nearest error location in each process, i. e.,

$h^{CG}(s) = \sum_{i=1}^{n} \min_{l_i^* \in L_i^*} cost_{p_i}(l_i, l_i^*)$ for $s = \langle l_1, \dots, l_n \rangle$. Note that the d^U estimate, due to Edelkamp et al. [23, 25], is defined by the same equation, but using a different estimate for $cost_{p_i}(l_i, l_i^*)$, which is simply the graph distance from l_i to l_i^*. In contrast, the cost estimates for h^{CG} take synchronization labels into account and usually provide larger (and, as we will see in the experimental evaluation in Section 5.5.2, more accurate) estimates than the graph distance.

5.4.1 Independent Processes

In this section and the following, we describe how the $cost_p(l, l')$ estimates are computed. We begin with the base case of independent processes with no predecessors in the causal graph.

Let $p = (L, E)$ be an independent variable domain process. In this case, like in the case of the d^U heuristic, we define $cost_p(l, l')$ as the graph distance from l to l' in p. For independent processes, this is an appropriate definition because their edges are not restricted by any other processes. Hence, in any global state of the system, a sequence of synchronization labels leading from l to l' does correspond to an executable trace that changes the location of p from l to l', without affecting the locations of other processes.

In our running example, Figure 5.3 shows that processes p_1 and p_2 do not have predecessors in the causal graph, and indeed, Figure 5.2 shows that these are independent processes, as the only labels affecting them – t_1 for p_1, t_2 for p_2 – are irrelevant for the other processes. Therefore, the cost estimates for these processes equal the graph distances. For example, $cost_{p_1}(l_0, l_2) = 2$ and $cost_{p_2}(l_2, l_3) = 1$.

5.4.2 Processes with Causal Predecessors

For variable domain processes p which do have predecessors in the causal graph, cost estimates are also computed by searching for paths in the labeled directed graph defined by the process. However, here we improve on the d^U approach by taking into account the synchronization labels on the edges: in addition to counting the number of edges of p required to reach a given location, we also consider the costs for moving the other processes of the system into locations which can synchronize with these edges. Note that by the definition of causal graphs, the *only* processes which can potentially restrict the non-looping edges of p are its causal predecessors, which we denote as $pred(p)$. Because we compute costs in a bottom-up order along a topological sorting of the causal graph, we have already computed all cost estimates for these processes. Hence, the computation of $cost_p(l, l')$ is based on finding traces from l to l' in the subsystem of $\mathcal{S}(\mathcal{M})$ induced by $\{p\} \cup pred(p)$, taking into account the known cost estimates for the processes $pred(p)$.

The algorithm for computing the cost values $cost_p(l, l')$ is shown in Figure 5.4. It is a modification of Dijkstra's algorithm for finding shortest paths in weighted

```
1  function compute-costs(S(M), s, p, l):
2      Let pred(p) be the set of immediate predecessors of p in CG(S(M)).
3      (L, E) := p
4      cost_p(l, l) := 0
5      cost_p(l, l') := ∞ for all l' ∈ L \ {l}
6      context(l, p_i) := location of p_i in s, for all p_i ∈ pred(p)
7      unreached := L
8      while unreached contains a location l' ∈ L with cost_p(l, l') < ∞:
9          Choose such a location l' ∈ unreached minimizing cost_p(l, l').
10         unreached := unreached \ {l'}
11         for each edge (l', a, l'') ∈ E from l' to some l'' ∈ unreached:
12             target-cost := cost_p(l, l') + 1
13             target-context := ∅
14             for each process p_i = (L_i, E_i) ∈ pred(p):
15                 m := context(l', p_i)
16                 Choose (m', m'') ∈ L_i × L_i such that (m', a, m'') ∈ E_i
                       and cost_{p_i}(m, m') is minimized.
17                 target-cost := target-cost + cost_{p_i}(m, m')
18                 target-context(p_i) := m''
19             if target-cost < cost_p(l, l''):
20                 cost_p(l, l'') := target-cost
21                 context(l'', p_i) := target-context(p_i) for all p_i ∈ pred(p)
22     return cost_p(l, l') for all l' ∈ L
```

Fig. 5.4. Modified Dijkstra algorithm for computing $cost_p(l, l')$.

directed graphs, applied to the process $p = (L, E)$. Like Dijkstra's algorithm, it is a one-to-all procedure, i. e., for a given start location l, it computes $cost_p(l, l')$ for all $l' \in L$. The only difference to Dijkstra's algorithm is that we do not define the cost of an edge $(l', a, l'') \in E$ before applying the algorithm. Instead, the edge cost is computed as soon as location l' is expanded by the algorithm, and it depends on the current locations of $pred(p)$ in the situation where l' is reached.

In detail, the cost of reaching $l'' \in L$ through edge $(l', a, l'') \in E$ is computed as the cost of reaching l' plus the *setup cost* required to take $pred(p)$ into locations that allow synchronization on the label a, plus 1 for taking the actual edge with label a that takes p from l' to l'' (lines 12–18). To estimate the setup cost for each predecessor $p_i \in pred(p)$, we associate each location $l' \in L$ with locations $context(l', p_i)$ for each $p_i \in pred(p)$, with the interpretation that when $l' \in L$ is first reached, we assume that process p_i is in location $m = context(l', p_i)$[2]. The setup cost for a given process is then the cheapest cost, according to the previously computed $cost_{p_i}$ values, for taking process p_i from m to a location m' where it can synchronize on the label a (lines 15–17).

[2] To avoid confusion, note that *contexts* in the causal graph setting are not related to the notion of contexts that has been used in the previous chapter. In this chapter, the term *context* is mainly used to be consistent with the literature.

If it turns out that $(l', a, l'') \in E$ reaches l'' more cheaply than the previously considered edges (line 19), then the cost of l'' is updated accordingly (line 20, as in Dijkstra's algorithm). At the same time, the context of l'' is set so that it reflects the way in which we have reached the location: by performing appropriate setup edges for $\mathit{pred}(p)$ and then synchronizing on label a (lines 16, 18, 21).

We remark that the algorithm is not guaranteed to find a globally shortest trace in the subsystem induced by $\{p\} \cup \mathit{pred}(p)$. Indeed, it may fail to find *any* path to a given location $l' \in L$ even though it is reachable. The reason for this is that the setup for each edge (l', a, l'') is performed greedily, without backtracking on the choice of *how* to modify the current context in order to allow synchronization on label a: we always pick a *locally cheapest* setup sequence. While it would of course be preferable to guarantee the success of the *compute-costs* algorithm, unfortunately this is not possible to do in polynomial time if $P \neq NP$: if we could, this would decide the existence of error traces in the model checking task induced by $\{p\} \cup \mathit{pred}(p)$. However, it is known that error detection for the subtask induced by a single process and its direct causal graph predecessors is NP-complete [35].

Returning to our running example from Figure 5.2, the algorithm computes the following cost estimates $\mathit{cost}_{p_3}(l_0, l')$ for the global state $\langle l_0, l_0, l_0 \rangle$:

- $\mathit{cost}_{p_3}(l_0, l_0) = 0$: this is due to the initialization step (line 4).
- $\mathit{cost}_{p_3}(l_0, l_1) = 0 + 1 + 2 = 3$: the three terms correspond to the cost of location l_0, the constant term 1, and the setup cost to reach locations of p_1 and p_2 in which we can synchronize on label a. In this case, we need to change p_1 from location l_0 to l_2, for a setup cost of 2.
- $\mathit{cost}_{p_3}(l_0, l_2) = 3 + 1 + 2 = 6$: cost of location l_1, constant term 1, setup cost to reach locations of p_1 and p_2 in which we can synchronize on label b. In this case, we need to change p_2 from location l_0 to l_2, for a setup cost of 2.
- $\mathit{cost}_{p_3}(l_0, l_3) = 6 + 1 + 4 = 11$: cost of location l_2, constant term 1, setup cost to reach locations of p_1 and p_2 in which we can synchronize on label c. In this case, we need to change both processes from location l_2 to l_0, for a setup cost of $2 + 2$.

5.4.3 Causal Graphs with Cycles

Up to this point, we have given a complete description of how to compute $h^{CG}(s)$ for systems with acyclic causal graphs. Unfortunately, many practical systems tend to have causal graphs with cycles. In this work, we use a rather simple idea to extend the definition of the heuristic to the general case (for an alternative approach, see Section 5.6).

If $\mathit{CG}(\mathcal{S}(\mathcal{M}))$ is not acyclic, we impose a total order $p'_1 \prec \cdots \prec p'_n$ on the processes of $\mathcal{S}(\mathcal{M})$. The computation of cost values then proceeds as previously described, except that for process p'_i, the *compute-costs* function does not consider

all causal predecessors $pred(p'_i)$ of p'_i, but only those which are ordered before p'_i in the ordering. Semantically, this means that we do not consider the synchronization costs for *all* processes, but only a subset of them. Of course, different total orders lead to different synchronization aspects being respected by this abstraction. Therefore, in practice one would prefer an order which is “close” to a topological sorting in some sense, i. e., loses as few arcs of the causal graph as possible. In our experiments, we use some simple greedy criteria to compute a reasonable ordering which are explained in more detail in Section 5.5.1.

5.4.4 Discussion: Sources of Imprecision

As distance heuristics *estimate* the distance to error states, we discuss reasons where precision is lost in the distance estimation of the causal graph heuristic. We will identify several sources of imprecision. Possible ways to overcome these limitations are described as future work in Section 5.6.

Turning the causal graph acyclic

In the previous sections, we have described a procedure to compute distance estimation values for acyclic causal graphs. However, in practice, almost all causal graphs have cycles (and may even consist of one single strongly connected component). The first (and most obvious) reason why precision is lost by h^{CG} is that the causal graph is statically turned acyclic in such cases. Therefore, some of the variable dependencies are not considered for the computation of the heuristic values. Semantically, this means that some setup costs of variables are not taken into account for the distance computation, which obviously might influence and worsen the quality of the resulting distance estimates.

The greedy algorithm for the *cost*-computation

To compute the *cost*-estimates as described in Figure 5.4, we use a greedy search algorithm. As argued in Section 5.4.2, this algorithm is not complete, and even may return infinity although a concrete solution exists. This is because the location contexts, once they are set, are *not changed* again later unless the location is reached more cheaply than before. A way that has looked shortest may turn out to be a dead end, and therefore, the heuristic may fail to find a solution even if the causal graph is acyclic.

Example 5.8. Consider the example system in Figure 5.5 consisting of two parallel processes with synchronization labels a, b and c.

The causal graph for this system is acyclic: we have an arc in the causal graph from p_2 to p_1 because a and b restrict p_2 and affect p_1. A simple trace to reach the

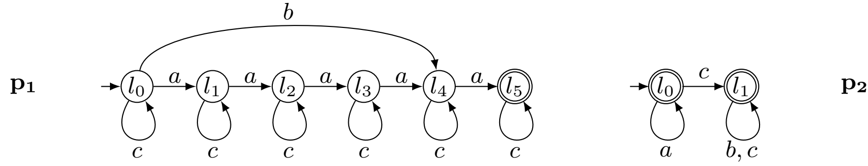

Fig. 5.5. Example where h^{CG} fails because of the greedy search algorithm

global error state is obtained by successively applying the edges in p_1 that synchronize on a. However, the causal graph distance heuristic also checks the edge from l_0 to l_4 in p_1 that is labeled with b. The modified Dijkstra algorithm for computing the cost estimates checks every outgoing edge of p_1, and reaches l_4 within two steps from l_0 (and therefore, $cost_{p_1}(l_0, l_4) = 0 + 1 + 1 = 2$). In particular, this enforces p_2 to go to location l_1 to synchronize with b. Hence, the context in p_1 of location l_4 is set to $p_2 = l_1$, from which the global error state is no longer reachable as there is no possibility to go back in p_2 to its initial location. As this context is not changed again, h^{CG} fails to find a solution and returns infinity as estimated error distance.

Limited context information

The previous section showed that the causal graph algorithm may fail to find a solution because of the greedy search algorithm. In this section, we investigate another source of imprecision that arises because the context information of each process is limited to one *level* in the causal graph: The contexts only store the configurations of the direct causal predecessors, whereas configurations of other predecessors are not taken into account for the cost computation. As a consequence, error traces found by h^{CG} might be arbitrarily longer or shorter than the shortest concrete one. To see this, consider the following example.

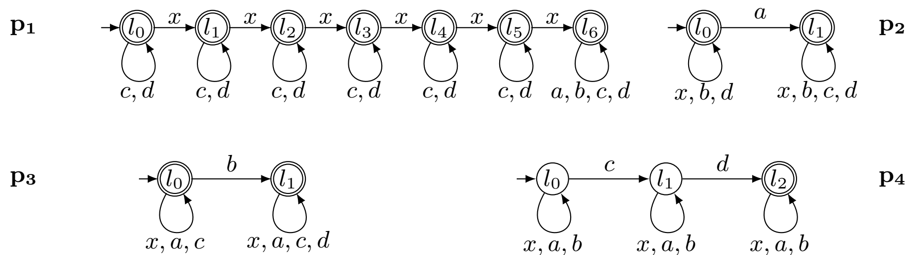

Fig. 5.6. Example system where h^{CG} overestimates the true error distance

Example 5.9. The example system depicted in Figure 5.6 consists of four parallel processes and label set $\{a, b, c, d, x\}$. The causal graph of this system is acyclic and

shown in Figure 5.7. In order to reach the global error state, p_4 has to synchronize with c and d, which enforces p_2 to first synchronize with a and p_3 to synchronize with b. To achieve this, p_1 has to reach its "last" location l_6. In particular, it is important to note that p_1 has to go from its initial location l_0 to l_6 *only once*. Overall, the true error distance in the initial global state where every process is in its initial location is equal to 10. However, this is not recognized by the causal graph distance heuristic because for p_4, only context information for p_2 and p_3 is stored. In more detail, we obtain the following cost estimates: $cost_{p_1}(l_0, l_6) = 6$, $cost_{p_2}(l_0, l_1) = 1 + 6 = 7$, $cost_{p_3}(l_0, l_1) = 1 + 6 = 7$, and $cost_{p_4}(l_0, l_2) = 1 + 7 + 1 + 7 = 16$. The resulting overestimation for p_4 is due to the fact that the costs to reach l_6 in p_1 have been counted twice.

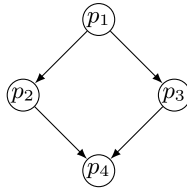

Fig. 5.7. The causal graph for the system in Example 5.9

Analogously, the causal graph distance heuristic may find an abstract error trace that is arbitrarily shorter than the shortest concrete one. To see this, consider the process p'_1 as shown in Figure 5.8, and assume that process p_1 in Example 5.9 is replaced by p'_1.

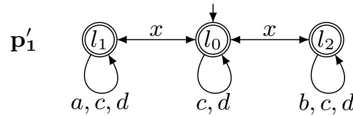

Fig. 5.8. Process for system in Figure 5.6 where h^{CG} underestimates the true error distance

In this example, we obtain a shortest error distance of 7 for the initial global state. However, in this case, h^{CG} underestimates this distance, as the cost estimates computed by h^{CG} to reach l_1 and l_2 in p'_1 are equal to 1. However, as we have to initially move to l_1, and then to move to l_2 *from* l_1, the true costs to reach l_2 from l_1 are equal to 2. Again, this is not recognized by the causal graph distance heuristic since the context information is limited, and the contexts in p_4 do not contain any information about p_1. Overall, this results in a distance estimate of 6 for the initial global state, underestimating the true error distance by one.

5.5 Evaluation

We implemented the causal graph distance heuristic h^{CG} and the safe abstraction technique in our model checker MCTA and evaluated it on benchmarks from the benchmark set as described in Section 2.4 on page 18 of this thesis. These include the Single-Tracked Line Segment case study ($S_1, \ldots, S_9$), the Mutual-Exclusion case study ($M_1, \ldots, M_4$ and $N_1, \ldots, N_4$), the Fischer Protocols (F_5, F_{10}, F_{15}) as well as the Towers of Hanoi problems ($H_3, \ldots, H_9$). The experimental results were obtained on a system with a 3 GHz Intel Pentium 4 CPU, using a memory bound of 1 GByte.

5.5.1 Implementation Details

As described in Section 2.4, our benchmark models consist of parallel processes with interleaving and binary synchronization semantics with bounded integer variables. Essentially, in our implementation, each process and bounded integer variable is identified with a variable domain process, and systems are translated to variable domain systems. In addition, recall that some benchmarks feature clock variables and actually represent timed automata [3]. Currently, clocks are ignored for the distance computation. (In fact, abstracting clocks away is the easiest way to deal with them for the computation of distance heuristics, and has already successfully been done in other approaches [21, 52].) For the safe abstraction technique, we additionally check that clocks do not affect any process.

As we have already pointed out, the procedure to compute the causal graph distance heuristic is defined for acyclic causal graphs. For systems with cyclic causal graphs, we greedily impose an ordering $\prec$ on the processes such that as much as possible of the important synchronization behavior is respected. For the following description, let $\mathcal{M}$ be a system and $\mathcal{S}(\mathcal{M})$ be the corresponding variable domain system. For the computation of the ordering $\prec$, first recall that arcs in the causal graph *CG*($\mathcal{S}(\mathcal{M})$) are generally induced by several synchronization labels. Essentially, in our implementation, arcs in *CG*($\mathcal{S}(\mathcal{M})$) are preferably ignored if they are induced by as few synchronization labels as possible. This is the same procedure as applied in the FAST DOWNWARD planning system [36]. Furthermore, variable domain processes in $\mathcal{S}(\mathcal{M})$ that correspond processes in $\mathcal{M}$ are treated in a special way. As such variable domain processes play a dedicated role in $\mathcal{S}(\mathcal{M})$ (and hence, so do the corresponding processes in $\mathcal{M}$), we order them after variable domain processes that correspond to integer variables. In more detail, we require for all variable domain processes p and p' that if p corresponds to a process in $\mathcal{M}$ and $p \prec p'$, then p' also corresponds to a process in $\mathcal{M}$.

5.5.2 Experimental Results and Discussion

We compare h^{CG} and h^{CG} with safe abstraction with the distance heuristics d^L, d^U, h^L and h^U as implemented in MCTA. Recall that a detailed description of these distance heuristics is provided in Section 2.3 on page 16 of this thesis. We report the number of explored states, the runtime and the error trace length in Tables 5.1, 5.2 and 5.3.

Table 5.1. Number of explored states for h^{CG} and h^{CG}_{safe} compared to the other distance heuristics. Abbreviations: #a: number of parallel automata, #v: number of variables, #s: number of variables removed by safe abstraction, dashes indicate out of memory ($>$ 1 GByte). Best results are given in bold fonts.

				explored states					
Instance	#a	#v	#s	d^L	d^U	h^L	h^U	h^{CG}	h^{CG}_{safe}
S_1	5	15	0	12780	11449	1704	**429**	2582	2582
S_2	6	17	1	38158	33859	3526	**828**	2203	1102
S_3	6	18	1	54871	51526	4182	**1033**	2668	1330
S_4	7	20	2	519921	465542	29167	12938	17566	**2535**
S_5	8	22	3	4763592	4617860	215525	65506	150176	**7247**
S_6	9	24	4	–	–	1710432	453763	1410955	**23383**
S_7	10	26	5	–	–	–	4230394	–	**86900**
S_8	10	27	5	–	–	7128947	3403395	–	**126160**
S_9	10	28	5	–	–	–	–	–	**600557**
M_1	3	15	0	12277	185416	**4581**	7668	6245	6245
M_2	4	17	1	43784	56240	15832	18847	18988	**8472**
M_3	4	17	1	54742	869159	**7655**	19597	27365	10632
M_4	5	19	2	202924	726691	71033	46170	96418	**18574**
N_1	3	18	0	15732	10215	50869	9117	**8171**	**8171**
N_2	4	20	0	102909	642660	30476	**23462**	30540	30540
N_3	4	20	0	131202	1155574	**11576**	43767	40786	40786
N_4	5	22	0	551091	330753	**100336**	152163	252558	252558
F_5	5	6	0	496	9	179	**7**	9	9
F_{10}	10	11	0	–	9	86378	**7**	9	9
F_{15}	15	16	0	–	9	–	**7**	9	9
H_3	4	0	0	216	222	127	**121**	254	254
H_4	5	0	0	2732	2276	2302	**839**	2398	2398
H_5	6	0	0	22226	20858	**20186**	21262	26410	26410
H_6	7	0	0	**76088**	184707	230878	235474	256279	256279
H_7	8	0	0	**443403**	1607485	1984213	2245588	1880203	1880203
H_8	9	0	0	–	–	–	–	**7641230**	**7641230**
H_9	10	0	0	–	–	–	–	–	–

Overall, we observe that the causal graph distance heuristic h^{CG} is competitive with previously proposed distance heuristics. First, we observe that h^{CG} mostly outperforms d^L and d^U that are based on the plain graph distance. This is reflected in the number of explored states, the runtime and the length of the found error

Table 5.2. Runtime for h^{CG} and h^{CG}_{safe} compared to the other distance heuristics. Abbreviations: #a: number of parallel automata, #v: number of variables, #s: number of variables removed by safe abstraction, dashes indicate out of memory ($>$ 1 GByte). Best results are given in bold fonts.

				runtime in seconds					
Instance	#a	#v	#s	d^L	d^U	h^L	h^U	h^{CG}	h^{CG}_{safe}
S_1	5	15	0	0.1	0.1	0.1	**0.0**	0.4	0.4
S_2	6	17	1	0.2	0.2	0.1	0.1	0.3	0.2
S_3	6	18	1	0.3	0.3	0.2	**0.1**	0.3	0.2
S_4	7	20	2	3.2	2.8	1.1	0.8	0.4	**0.1**
S_5	8	22	3	31.3	30.0	7.3	3.9	2.4	**0.3**
S_6	9	24	4	–	–	53.5	23.8	37.7	**1.6**
S_7	10	26	5	–	–	–	201.5	–	**5.3**
S_8	10	27	5	–	–	182.9	157.5	–	**10.3**
S_9	10	28	5	–	–	–	–	–	**51.2**
M_1	3	15	0	0.3	253.5	0.2	0.3	**0.1**	**0.1**
M_2	4	17	1	1.1	2.5	0.5	0.8	0.4	**0.2**
M_3	4	17	1	1.5	1938.5	**0.2**	0.7	0.5	0.3
M_4	5	19	2	6.3	506.2	2.3	2.0	2.1	**0.5**
N_1	3	18	0	0.8	0.4	46.3	0.5	**0.3**	**0.3**
N_2	4	20	0	7.1	2238.9	1.7	1.6	**1.4**	**1.4**
N_3	4	20	0	11.3	8401.6	**0.5**	3.1	1.9	1.9
N_4	5	22	0	71.4	31.2	**7.1**	18.1	21.7	21.6
F_5	5	6	0	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**
F_{10}	10	11	0	–	**0.0**	4.2	**0.0**	**0.0**	**0.0**
F_{15}	15	16	0	–	**0.0**	–	**0.0**	**0.0**	**0.0**
H_3	4	0	0	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**
H_4	5	0	0	**0.0**	**0.0**	**0.0**	**0.0**	0.1	0.1
H_5	6	0	0	**0.1**	**0.1**	0.3	0.5	0.6	0.6
H_6	7	0	0	**0.3**	0.7	3.3	5.7	5.5	5.5
H_7	8	0	0	**2.1**	6.6	30.9	59.0	45.5	45.6
H_8	9	0	0	–	–	–	–	**190.1**	**193.3**
H_9	10	0	0	–	–	–	–	–	–

traces. In particular, this is interesting because h^{CG} can be considered as a direct extension of the graph distance heuristic d^U (recall that h^{CG} additionally takes the local context information of causal predecessors into account). The benefit of this conceptual extension is clearly indicated by the experimental results, and also motivates to consider more sophisticated context information as future work (see also Section 5.6). Furthermore, we observe that the performance of h^{CG} is similar to that of h^L in terms of the number of explored states until an error state in found. The runtimes are also comparable. However, h^{CG} solves one more problem than h^L, and the length of the error traces found with h^{CG} is mostly shorter that with h^L. Finally, we also observe that h^{CG} is somewhat less informed than h^U, which can be observed particularly in the Single-Tracked-Line-Segment case study.

Table 5.3. Error trace length for h^{CG} and h^{CG}_{safe} compared to the other distance heuristics. Abbreviations: #a: number of parallel automata, #v: number of variables, #s: number of variables removed by safe abstraction, dashes indicate out of memory ($>$ 1 GByte). For h^{CG}_{safe}, trace lengths reported as $x + y$ denote trace length x for the abstract error trace and $x + y$ for the concrete error trace. Best results are given in bold fonts.

Instance	#a	#v	#s	error trace length d^L	d^U	h^L	h^U	h^{CG}	h^{CG}_{safe}
S_1	5	15	0	922	823	84	**67**	112	112
S_2	6	17	1	1386	1229	172	**83**	134	112 + 0
S_3	6	18	1	1524	1032	162	**79**	138	112 + 0
S_4	7	20	2	7725	3132	378	**112**	313	119 + 5
S_5	8	22	3	25647	14034	1424	176	773	**124 + 11**
S_6	9	24	4	–	–	5041	432	4480	**159 + 21**
S_7	10	26	5	–	–	–	924	–	**250 + 45**
S_8	10	27	5	–	–	5435	2221	–	**124 + 16**
S_9	10	28	5	–	–	–	–	–	**124 + 23**
M_1	3	15	0	2779	106224	457	**71**	231	231
M_2	4	17	1	11739	13952	1124	**119**	395	240 + 3
M_3	4	17	1	12701	337857	748	**124**	361	205 + 4
M_4	5	19	2	51402	290937	3381	**160**	642	219 + 7
N_1	3	18	0	3565	2669	26053	**99**	243	243
N_2	4	20	0	18180	415585	1679	**154**	376	376
N_3	4	20	0	20021	262642	799	**147**	232	232
N_4	5	22	0	90467	51642	2455	**314**	478	478
F_5	5	6	0	79	**6**	12	**6**	**6**	**6**
F_{10}	10	11	0	–	**6**	22	**6**	**6**	**6**
F_{15}	15	16	0	–	**6**	–	**6**	**6**	**6**
H_3	4	0	0	94	**44**	48	52	52	52
H_4	5	0	0	1108	184	300	**102**	186	186
H_5	6	0	0	4442	708	1458	402	**320**	**320**
H_6	7	0	0	30980	2734	7284	1544	**1228**	**1228**
H_7	8	0	0	138584	11202	18500	4988	**2954**	**2954**
H_8	9	0	0	–	–	–	–	**5012**	**5012**
H_9	10	0	0	–	–	–	–	–	–

Moreover, safe abstraction leads to strong performance improvements when applicable. This is reflected in the S and M examples, where the model reduction obtained by safe abstraction further significantly reduced the number of states to be explored. In particular, this is the case in the large S examples. Furthermore, we observe that the computation of the causal graph structure and the check for safe variables is very cheap, and the runtime overhead is negligible (a fraction of a second in most of the examples). Therefore, the overall model checking performance is almost not affected by this precomputation if safe abstraction is not applicable and the system cannot be reduced.

Finally, we remark that safe abstraction is applicable to directed model checking with arbitrary distance heuristics. Currently, our implementation only handles

safe abstraction with h^{CG}, for which we have shown detailed results. A more general implementation is subject to future work. However, we can announce that a preliminary implementation for other distance heuristics is available and indicates that similar performance improvements will be achieved also in the general case for arbitrary distance heuristics.

5.6 Conclusions

In this chapter, we have introduced the causal graph structure to directed model checking. Based on causal graph analysis, we have first proposed the safe abstraction principle. This abstraction technique guarantees to preserve error traces in the sense that every concrete error trace corresponds to an abstract error trace, and vice versa. Furthermore, we adapted the causal graph distance heuristic from AI planning to our context of directed model checking. We have also discussed reasons why and where precision is lost when abstract error traces are computed. The experimental results show that the causal graph heuristic is competitive with previously proposed heuristics. Moreover, safe abstraction has shown to significantly improve the directed model checking performance when applicable, while needing only little computational overhead when not. Overall, the contributions based on causal graph analysis show to be promising also in directed model checking. Once more, this reflects the close relationship between directed model checking and AI planning as heuristic search.

In the future, the precision of the causal graph distance heuristic could be improved in several ways, where the sources of imprecision that we have discussed in Section 5.4.4 can serve as a starting point for improvements. As an example, we have observed that pruning arcs in the causal graph affects the distance estimates. In this work, we have applied a rather simple strategy to prune them, and the question arises if there are more sophisticated pruning strategies that lead to better distance estimates. Moreover, it will be interesting to investigate if there are classes of systems where specific pruning strategies work better than others. A problem that should also be addressed in the future is the treatment of cycles in general. More sophisticated ways than just statically pruning arcs are likely to further improve the accuracy of the distance heuristic [38]. Finally, as we have observed that the limited context information is another source of imprecision of h^{CG}, it will also be interesting to consider "larger" local subproblems for the h^{CG} computation. This means to extend the context information to $n \geq 1$ levels, where the current version of h^{CG} will be a special case for $n = 1$.

6

The Model Checker MCTA

In this chapter, we present our directed model checking tool MCTA [56]. The corresponding tool paper is based on joint work with Kupferschmid, Nebel, and Podelski. MCTA provides various directed model checking techniques, including the transition-based techniques presented in this thesis. In addition to the computational model as considered so far, MCTA is a model checker for real-time systems modeled as *timed automata* [3]. We will give a brief and intuitive introduction to that formalism in Section 6.1. Thereafter, we describe MCTA, its features and an experimental comparison with the UPPAAL model checker in Section 6.2.

6.1 Timed Automata

The theory of timed automata has been introduced by Alur and Dill [3] and has been studied in different variants [7, 18, 41, 63]. In this section, we focus on giving a high level description of this theory that is sufficient to capture the basic ideas and the description of MCTA in Section 6.2. For more technical details, the reader is referred to the literature.

Real time systems modeled as timed automata consist of finite state automata (as the computational model we have considered in this thesis) that are augmented with real valued *clock* variables (*clocks* for short). As one would expect, clocks have a special semantics and increase synchronously with the same rate. The initial value of each clock is zero. Transitions can additionally be guarded by clock constraints, and can reset clock values to zero in the effect. Moreover, locations l contain *invariants*, i. e., clock constraints that specify for which clock values it is allowed to stay in l.

The operational semantics of systems of timed automata $\mathcal{M}$ is given as a transition system. In addition to the definition of global states so far, global states of $\mathcal{M}$ also reflect the current values of the clocks. As the domain of clocks is the set that consists of the non-negative real numbers, the explicit representation of states leads

to an uncountable size of the induced transition system. However, for certain classes of timed automata, there is a finite symbolic representation of the state space with the property that every global state that is reachable in the original transition system is also reachable in the symbolic representation, and vice versa. This representation is based on the idea that in every global state s, it is sufficient to determine if the clock values in s satisfy the clock constraints that occur in the system. Therefore, little differences of clock values are often irrelevant because they cannot be distinguished by any clock constraint. For example, if the only clock constraint of the system is $x \leq 1$ for a clock x, it is only important to determine if x is less than or equal to one or not; given this information, the exact value of x is not relevant. Such symbolic representations of clock values are called *zones*. In general, a zone is defined as a conjunction of clock constraints. The symbolic representation of the state space is called the *zone graph*. For a more detailed introduction to timed automata, the reader is referred to the literature [7]. We conclude the section with a simple example.

Example 6.1. Consider the simple system $\mathcal{M}$ of Figure 6.1, consisting of one timed automaton p with 3 locations, initial location l_1, no integer variables and one clock x.

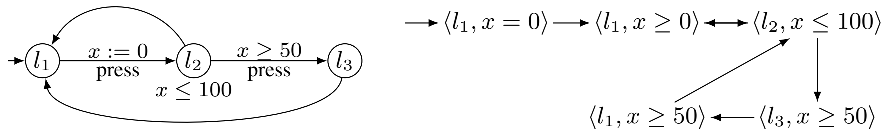

Fig. 6.1. Example timed automaton and corresponding zone graph

The timed automaton models a controller of a simple keyboard of a mobile phone that only consists of one button. The objective is to write a text message consisting of the letters A and B. If the button is pressed, then A is retrieved. If the button is pressed again between 50 and 100 milliseconds, A changes to B. In this example, pressing the button corresponds to receiving a *press* event. To model the timing behavior, a clock x is reset to zero when the button is pressed for the first time. The l_2 location (“retrieve A”) has an invariant that specifies that x must not exceed 100 milliseconds while staying in l_2. If the *press* event is received again and within 50 and 100 milliseconds, then l_3 is reached (“retrieve B”). In both cases, we finally have to return to l_1 in order to receive the next *press* event, i. e., in order to write the next letter.

The zone graph of $\mathcal{M}$ is depicted on the right of Figure 6.1. A symbolic state consists of the current location of p and the zone that describes the possible values for x. The initial symbolic state is $\langle l_1, x = 0 \rangle$, from which we reach $\langle l_1, x \geq 0 \rangle$ by

letting time pass. As there is no invariant in l_1 that restricts x, x can have every non-negative real value. By pressing the button, we reach state $\langle l_2, x \leq 100\rangle$, where the clock values are restricted to less than or equal to hundred because of l_2's invariant. From there, when pressing the button again, we reach state $\langle l_3, x \geq 50\rangle$. In this state, x is larger or equal to fifty because of the corresponding guard $x \geq 50$, and there is no upper bound on the values of x any more as there is no invariant in l_3. By returning to the initial location, we end up in the state $\langle l_1, x \geq 50\rangle$, where the values of x are still larger or equal to fifty as no clock reset has been performed. Finally, the whole procedure can restart from the beginning, and the button can be pressed again for the next letter. Overall, we have a symbolic representation of the infinite state space that consists of five symbolic states.

6.2 Description of MCTA

MCTA is a directed model checking tool for timed automata. The input of MCTA is a system of timed automata and a formula that describes the error states. Given a model checking task $\Theta = (\mathcal{M}, \varphi)$, MCTA performs a symbolic reachability analysis, which means that directed model checking is performed directly on the zone graph of $\mathcal{M}$. Several types of traces can be generated, including options to find (guaranteed) shortest error traces. There is also the possibility to examine MCTA's traces with UPPAAL's [6] graphical user interface.

MCTA accepts input models in the form of the UPPAAL input language [6]. So far, only a fraction thereof is supported, e. g., there is no support for urgent channels, arrays, etc. yet. Internally, UPPAAL's timed automata parser library is used. 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. All other data structures, algorithms and their implementation are genuine to MCTA.

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

6.2.1 Ingredients in a Nutshell

MCTA is written in C++ and provides a flexible platform to apply and implement directed model checking techniques. It offers various search algorithms, distance heuristics and search enhancements for directed model checking. First, MCTA provides an implementation of breadth-first and depth-first search as well as greedy search and A*. Breadth-first search and A* with admissible distance heuristics are guaranteed to find shortest possible error traces [64]. Furthermore, the distance heuristics d^L, d^U, h^L, h^U and h^{CG} are implemented. In addition to the distance heuristics, the transition-based techniques from this thesis are implemented in

Table 6.1. Experimental results with UPPAAL and MCTA. Dashes indicate out of memory (> 1 GByte). Best results are given in bold fonts.

Instance	explored states		runtime in seconds		error trace length	
	UPPAAL	MCTA	UPPAAL	MCTA	UPPAAL	MCTA
S_1	14462	**243**	0.26	**0.05**	1037	**54**
S_2	32677	**212**	0.50	**0.06**	894	**54**
S_3	45427	**198**	0.63	**0.06**	552	**54**
S_4	338814	**174**	5.35	**0.08**	1195	**55**
S_5	2655586	**147**	48.66	**0.10**	3086	**61**
S_6	22044252	**147**	448.88	**0.13**	3019	**61**
S_7	–	**143**	–	**0.15**	–	**61**
S_8	–	**1466**	–	**1.28**	–	**56**
S_9	–	**1575**	–	**1.49**	–	**69**
M_1	9888	**4366**	0.65	**0.24**	851	**73**
M_2	29858	**2018**	2.28	**0.21**	3456	**81**
M_3	23765	**17315**	1.82	**1.60**	2938	**163**
M_4	88622	**15349**	7.88	**2.23**	11992	**91**
N_1	9675	**5191**	1.30	**0.33**	787	**80**
N_2	48420	**3260**	8.47	**0.32**	3870	**136**
N_3	30028	**19271**	5.00	**1.55**	3302	**149**
N_4	174206	**15102**	34.38	**1.88**	11652	**377**
A_2	57	**20**	0.01	**0.00**	34	**18**
A_3	4365	**27**	0.07	**0.02**	147	**17**
A_4	6598	**34**	0.17	**0.13**	298	**22**
A_5	–	**42**	–	**0.66**	–	**27**
A_6	–	**50**	–	**3.26**	–	**32**

MCTA. These include useless transitions, context-enhanced directed model checking, and the safe abstraction technique. We also implemented caching strategies to avoid the recomputation of heuristic values for the same state. In particular, this is useful in the context of useless transitions, where heuristic values have to be accessed several times to decide if a given transition is useless in a given state.

6.2.2 Results

To emphasize MCTA's performance, we compare MCTA with UPPAAL (version 4.0.6). UPPAAL is a state-of-the-art model checker for timed automata systems and provides a very efficient implementation of uniformed search methods [6]. The experimental results were obtained on a system with a 3 GHz Intel Pentium 4 CPU, using a memory bound of 1 GByte. We show that the application of directed model checking (MCTA) pays off compared to uninformed search (UPPAAL) when the aim is to find short error traces. For UPPAAL, we report the results obtained with its most efficient search algorithm which is randomized depth first search (averaged over 10 runs). For MCTA, we exemplary report the results obtained with the h^U distance heuristic applied with relatively useless transitions and a caching strategy for the heuristic values. The results in Table 6.1 impressively demonstrate the power of directed model checking when the aim is to find short error traces.

Compared to the very efficient implementation of uninformed search provided by the UPPAAL model checker, we observe that MCTA finds significantly shorter error traces and explores less states. As a consequence, MCTA can handle much larger systems and is often by orders of magnitude faster. Overall, we observe that directed model checking with our transition prioritizing techniques is able to handle very large state spaces. This allows us to handle large and realistic systems very efficiently. In particular, we have demonstrated that we are able to efficiently handle systems that cannot be handled by UPPAAL at all.

6.2.3 Future Work

In the future, MCTA will evolve by supporting more language constructs for defining (extensions of) timed automata and by providing more distance heuristics. Furthermore, MCTA will also provide more search enhancements to prune the state space. The long term vision is that the results and the practical experience with MCTA for analyzing incorrect specifications will also flow into tools that were originally geared towards analyzing *correct* timed automata. We have done a first step in this direction with our pattern database approach based on downward pattern refinement [55], which we will discuss in more detail in the next chapter.

7

Conclusions and Future Research

In this thesis, we have introduced the concept of *transition-based* directed model checking, which advances classical directed model checking by prioritizing transitions rather than only prioritizing states. All of the presented techniques are fully automated and do not need any further user input. To estimate if a transition is needed to find an error state, we have first presented the concept of useless transitions. This concept has been further adapted to the area of AI planning as heuristic search. Moreover, based on the notion of interference contexts, we have presented an alternative technique to compute transition priorities that is more fine-grained than the Boolean property of uselessness. As there is no theoretical guarantee that transitions identified with these concepts can *always* be pruned, we have presented search algorithms with the property that such transitions are less preferred over others. Furthermore, we have proposed the safe abstraction technique to identify transitions that can *provably* be pruned without affecting completeness. The experimental evaluation of our techniques in the MCTA model checker revealed significant performance improvements compared to existing directed model checking techniques on large and realistic case studies. We have also demonstrated that the adaptation of useless transitions to the area of AI planning improves existing planning techniques on a large number of benchmarks from the international planning competitions. Finally, we have demonstrated that transition-based directed model checking enables us to handle systems that have been out of scope for non-heuristic search algorithms provided by the UPPAAL model checker. Let us conclude the summary of this thesis with a statement of one of the 2007 ACM Turing Award Winners Edmund M. Clarke: in the *25 Years of Model Checking* Festschrift [14], he described challenges for the future, including the challenge to "scale up even more", and the challenge to find efficient model checking techniques for timed automata. In this thesis, we have demonstrated that transition-based directed model checking improves classical directed model checking for various distance heuristics, and the implementation in MCTA shows that transition-based directed model checking is also successfully

applicable to timed automata. In this sense, we conclude that we have done an important step in this direction.

For the future, it will be important to further investigate (transition-based) search techniques to advance classical directed model checking. The first reason to do so is that such techniques provide much potential as we have demonstrated in this thesis. The second reason is that there are impressive empirical results for a recently proposed distance heuristic that indicate that not much more room for improvements with pure heuristic search is achievable, at least not with distance heuristics based on the monotonicity abstraction in the context of optimal search [37]. Hence, further search enhancement techniques have to be considered. In a suboptimal setting, a first step in this direction could be to investigate the notion of *landmarks* in our directed model checking setting. Landmarks are a successful technique in AI planning [43, 67]. Adapted to directed model checking, they provide information about predicates that have to be true on *every* error trace at some time point. Considering this information, e. g., by preferring states that are reached on a trace where more landmarks have already been satisfied, could lead to further guidance improvements.

In this thesis, we have intensively discussed strategies to evaluate *transitions* during the search. Closely related to the problem of evaluating transitions is the problem of evaluating *variables* or *predicates* of the system under consideration. In particular, this is important in the context of *pattern database heuristics* [17, 22]. Pattern database heuristics are an important class of admissible distance heuristics that are guaranteed to find shortest possible error traces with the A^* search algorithm [64]. Typically, such distance heuristics are built in a preprocessing step prior to model checking by an exhaustive state space exploration of an abstracted system. Abstract system states are stored in a table (the *pattern database*) together with the corresponding abstract error distances. During directed model checking, the distance estimation for a concrete state s corresponds to the abstract error distance of the corresponding abstract state of s. The abstract system is determined by a set of variables, the *pattern*. On the one hand, the abstract system should appropriately reflect the original system behavior. Therefore, the pattern should contain the most important variables of the system. On the other hand, the pattern should also yield abstractions that are as small as possible to be able to handle large systems. On this point, we emphasize the importance of being able to appropriately evaluate variables. Obviously, the choice of the pattern is the most crucial part in the design of a pattern database heuristic because it determines the overall quality of the resulting distance heuristic. However, although various approaches for the automatic pattern selection have been proposed in the literature [33, 44, 54, 55, 72], the problem of how to automatically select good patterns is still not considered as solved. More precisely, the questions remain if there are variables that lead to better patterns and more informed distance heuristics than others, and if these variables can

be identified automatically. In this context, our recently proposed approach based on *downward pattern refinement* to evaluate the importance of processes and variables seems to be very promising [55]. It is based on techniques that have been adapted from this thesis and appears to be a good starting point for future research. In particular, the treatment of clocks in the context of timed systems remains as an important research topic.

For the class of pattern database heuristics, we finally mention that there also is potential in *verifying correct* systems with heuristic search. To see this, we first point out that admissible distance heuristics h, i. e., distance heuristics that never overestimate the real error distance, can be used as a pruning technique: if $h(s) = \infty$ for a global state s, then s can safely be pruned as there is no concrete error trace from s. Therefore, the absence of error states might be shown without actually exploring the entire reachable state space. As far as we are aware, the pruning capabilities of admissible distance heuristics like h^L have not been powerful enough to obtain spectacular results. However, the situation changes with more sophisticated admissible distance heuristics like the above explained pattern database heuristics, which have the potential to achieve much more powerful results. In this direction, Kupferschmid [50] presents encouraging results for the pattern database heuristic based on Russian Doll abstraction. Furthermore, we have successfully verified large systems with our recently proposed approach based on downward pattern refinement [55]. These results demonstrate the potential of applying directed model checking also for verification, and show that this area should be further explored in the future.

References

1. The international planning competition 2008. http://ipc.informatik.uni-freiburg.de/. [Online; accessed 28-February-2011].
2. Journal of artificial intelligence research. http://www.jair.org/. [Online; accessed 28-February-2011].
3. Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.
4. Fahiem Bacchus and Michael Ady. Planning with resources and concurrency: A forward chaining approach. In Bernhard Nebel, editor, *Proceedings of the 17th International Joint Conference on Artificial Intelligence (IJCAI 2001)*, volume 1, pages 417–424. Morgan Kaufmann, 2001.
5. Jorge Baier and Adi Botea. Improving planning performance using low-conflict relaxed plans. In Alfonso Gerevini, Adele E. Howe, Amedeo Cesta, and Ioannis Refanidis, editors, *Proceedings of the 19th International Conference on Automated Planning and Scheduling (ICAPS 2009)*, pages 10–17. AAAI Press, 2009.
6. 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 Systems (SFM-RT 2004)*, volume 3185 of *Lecture Notes in Computer Science*. Springer-Verlag, 2004.
7. 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.
8. Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking. *Advances in Computers*, 58:118–149, 2003.
9. 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.
10. Blai Bonet and Héctor Geffner. Planning as heuristic search. *Artificial Intelligence*, 129(1–2):5–33, 2001.
11. Ronen I. Brafman and Carmel Domshlak. Structure and complexity in planning with unary operators. *Journal of Artificial Intelligence Research*, 18:315–349, 2003.
12. Tom Bylander. The computational complexity of propositional STRIPS planning. *Artificial Intelligence*, 69(1-2):165–204, 1994.
13. 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.

14. 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.
15. 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.
16. Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, 2000.
17. Joseph C. Culberson and Jonathan Schaeffer. Pattern databases. *Computational Intelligence*, 14(3):318–334, 1998.
18. Henning Dierks. PLC automata: A new class of implementable real-time automata. *Theoretical Computer Science*, 253(1):61–93, 2001.
19. Henning Dierks. Comparing model-checking and logical reasoning for real-time systems. *Formal Aspects of Computing*, 16(2):104–120, 2004.
20. 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 (SPIN 2006)*, volume 3925 of *Lecture Notes in Computer Science*, pages 19–34. Springer-Verlag, 2006.
21. 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.
22. Stefan Edelkamp. Planning with pattern databases. In Amedeo Cesta and Daniel Borrajo, editors, *Proceedings of the 6th European Conference on Planning (ECP 2001)*, pages 13–24, 2001.
23. 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.
24. Stefan Edelkamp, Stefan Leue, and Alberto Lluch-Lafuente. Partial-order reduction and trail improvement in directed model checking. *International Journal on Software Tools for Technology Transfer*, 6(4):277–301, 2004.
25. 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 (SPIN 2001)*, volume 2057 of *Lecture Notes in Computer Science*, pages 57–79. Springer-Verlag, 2001.
26. 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.
27. Stefan Edelkamp, Viktor Schuppan, Dragan Bošnački, Anton Wijs, Ansgar Fehnker, and Husain Aljazzar. Survey on directed model checking. 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 65–89. Springer-Verlag, 2009.
28. 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.
29. 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.
30. Alfonso Gerevini, Derek Long, Patrik Haslum, Alessandro Saetti, and Yannis Dimopoulos. Deterministic planning in the fifth international planning competition: PDDL3 and experimental evaluation of the planners. *Artificial Intelligence*, 173:619–668, 2009.
31. 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.

32. Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems — An Approach to the State-Explosion Problem*, volume 1032 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.
33. 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.
34. Patrik Haslum and Peter Jonsson. Planning with reduced operator sets. In Steve Chien, Subbarao Kambhampati, and Craig A. Knoblock, editors, *Proceedings of the 5th International Conference on Artificial Intelligence Planning Systems (AIPS 2000)*, pages 150–158. AAAI Press, 2000.
35. Malte Helmert. A planning heuristic based on causal graph analysis. In Shlomo Zilberstein, Jana Koehler, and Sven Koenig, editors, *Proceedings of the 14th International Conference on Automated Planning and Scheduling (ICAPS 2004)*, pages 161–170. AAAI Press, 2004.
36. Malte Helmert. The Fast Downward planning system. *Journal of Artificial Intelligence Research*, 26:191–246, 2006.
37. Malte Helmert and Carmel Domshlak. Landmarks, critical paths and abstractions: What's the difference anyway? In Alfonso Gerevini, Adele E. Howe, Amedeo Cesta, and Ioannis Refanidis, editors, *Proceedings of the 19th International Conference on Automated Planning and Scheduling (ICAPS 2009)*, pages 162–169. AAAI Press, 2009.
38. Malte Helmert and Héctor Geffner. Unifying the causal graph and additive heuristics. 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 140–147. AAAI Press, 2008.
39. Malte Helmert, Patrik Haslum, and Jörg Hoffmann. Flexible abstraction heuristics for optimal sequential planning. In Mark Boddy, Maria Fox, and Sylvie Thiébaux, editors, *Proceedings of the 18th International Conference on Automated Planning and Scheduling (ICAPS 2007)*, pages 176–183. AAAI Press, 2007.
40. 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.
41. Jochen Hoenicke. *Combination of Processes, Data, and Time*. PhD thesis, University of Oldenburg, July 2006.
42. Jörg Hoffmann and Bernhard Nebel. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14:253–302, 2001.
43. Jörg Hoffmann, Julie Porteous, and Laura Sebastia. Ordered landmarks in planning. *Journal of Artificial Intelligence Research*, 22:215–278, 2004.
44. 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 Lomuscio, 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.
45. Gerard J. Holzmann. *The SPIN Model Checker – Primer and Reference Manual*. Addison-Wesley, 2003.
46. 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.
47. Erez Karpas and Carmel Domshlak. Cost-optimal planning with landmarks. In Craig Boutilier, editor, *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI 2009)*, pages 1728–1733, 2009.

48. Henry Kautz and Bart Selman. Pushing the envelope: Planning, propositional logic, and stochastic search. In *Proceedings of the 13th National Conference on Artificial Intelligence (AAAI 1996)*, pages 1194–1201, 1996.
49. 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.
50. Sebastian Kupferschmid. *Directed Model Checking for Timed Automata*. PhD thesis, Albert-Ludwigs-Universität Freiburg, 2009.
51. 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.
52. 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 (SPIN 2006)*, volume 3925 of *Lecture Notes in Computer Science*, pages 35–52. Springer-Verlag, 2006.
53. Sebastian Kupferschmid, Jörg Hoffmann, Henning Dierks, and Gerd Behrmann. Adapting an AI planning heuristic for directed model checking. Technical Report 222, University of Freiburg, Department of Computer Science, Freiburg, Germany, 2006. Available at `http://www.informatik.uni-freiburg.de/tr/2006/Report222/`.
54. 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*. Springer-Verlag, 2008.
55. Sebastian Kupferschmid and Martin Wehrle. Abstractions and pattern databases: The quest for succinctness and accuracy. In Parosh A. Abdulla and K. Rustan M. Leino, editors, *Proceedings of the 17th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2011)*, volume 6605 of *Lecture Notes in Computer Science*, pages 276–290. Springer-Verlag, 2011.
56. 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.
57. Leslie Lamport. A fast mutual exclusion algorithm. *ACM Transactions on Computer Systems*, 5(1):1–11, 1987.
58. Madanlal Musuvathi and Shaz Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In Jeanne Ferrante and Kathryn S. McKinley, editors, *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI 2007)*, pages 446–455. ACM Press, 2007.
59. Glenford J. Myers. *The Art of Software Testing*. John Wiley & Sons, Inc., 1 edition, 1979.
60. Dana Nau, Malik Ghallab, and Paolo Traverso. *Automated Planning: Theory and Practice*. Morgan Kaufmann Publishers, San Fransisco, CA, 2004.
61. Bernhard Nebel, Yannis Dimopoulos, and Jana Koehler. Ignoring irrelevant facts and operators in plan generation. In Sam Steel and Rachid Alami, editors, *Proceedings of the 4th European Conference on Planning (ECP 1997)*, volume 1348 of *Lecture Notes in Computer Science*, pages 338–350. Springer-Verlag, 1997.

62. 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.
63. Ernst-Rüdiger Olderog and Henning Dierks. *Real-Time Systems: Formal Specification and Automatic Verification*. Cambridge University Press, 2008.
64. Judea Pearl. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley, 1984.
65. 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.
66. Silvia Richter and Malte Helmert. Preferred operators and deferred evaluation in satisficing planning. In Alfonso Gerevini, Adele E. Howe, Amedeo Cesta, and Ioannis Refanidis, editors, *Proceedings of the 19th International Conference on Automated Planning and Scheduling (ICAPS 2009)*, pages 273–280. AAAI Press, 2009.
67. 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.
68. Jussi Rintanen. Introduction to automated planning. manuscript, 2006.
69. Jussi Rintanen, Keijo Heljanko, and Ilkka Niemelä. Planning as satisfiability: Parallel plans and algorithms for plan search. *Artificial Intelligence*, 170(12–13):1031–1080, 2006.
70. Nathan Robinson, Charles Gretton, Duc Nghia Pham, and Abdul Sattar. SAT-based parallel planning using a split representation of actions. In Alfonso Gerevini, Adele E. Howe, Amedeo Cesta, and Ioannis Refanidis, editors, *Proceedings of the 19th International Conference on Automated Planning and Scheduling (ICAPS 2009)*. AAAI Press, 2009.
71. Charles L. Seitz. Ideas about arbiters. *Lambda*, 1:10–14, 1980.
72. 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 147–165. Springer-Verlag, 2009.
73. Antti Valmari. Stubborn sets for reduced state space generation. In Grzegorz Rozenberg, editor, *Proceedings of the 10th International Conference on Applications and Theory of Petri Nets (APN 1989)*, volume 483 of *Lecture Notes in Computer Science*, pages 491–515. Springer-Verlag, 1991.
74. Vincent Vidal. A lookahead strategy for heuristic search planning. In Shlomo Zilberstein, Jana Koehler, and Sven Koenig, editors, *Proceedings of the 14th International Conference on Automated Planning and Scheduling (ICAPS 2004)*, pages 150–159. AAAI Press, 2004.
75. Willem Visser, Klaus Havelund, Guillaume P. Brat, Seungjoon Park, and Flavio Lerda. Model checking programs. *Automated Software Engeneering*, 10(2):203–232, 2003.
76. Martin Wehrle and Malte Helmert. The causal graph revisited for directed model checking. In Jens Palsberg and Zhendong Su, editors, *Proceedings of the 16th International Symposium on Static Analysis (SAS 2009)*, volume 5673 of *Lecture Notes in Computer Science*, pages 86–101. Springer-Verlag, 2009.
77. Martin Wehrle and Sebastian Kupferschmid. Context-enhanced directed model checking. In Jaco van de Pol and Michael Weber, editors, *Proceedings of the 17th International SPIN Workshop (SPIN 2010)*, volume 6349 of *Lecture Notes in Computer Science*, pages 88–105. Springer-Verlag, 2010.
78. 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.

79. 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.
80. 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.