UNIVERSITY OF NEW SOUTH WALES
SCHOOL OF COMPUTER SCIENCE & ENGINEERING

FORMAL SYMBOLIC VERIFICATION USING
HEURISTIC SEARCH AND ABSTRACTION
TECHNIQUES

KAIRONG QIAN

A dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science & Engineering

Kensington Campus, UNSW 2052 Australia

August 2006
FORMAL SYMBOLIC VERIFICATION USING HEURISTIC SEARCH AND ABSTRACTION TECHNIQUES

Copyright © 2006

by

KAIRONG QIAN
Abstract

Computing devices are pervading our everyday life and imposing challenges for designers that have the responsibility of producing reliable hardware and software systems. As systems grow in size and complexity, it becomes increasingly difficult to verify whether a design works as intended. Conventional verification methods, such as simulation and testing, exercise only parts of the system and from these parts, draw conclusions about the correctness of the total design. For complex designs, the parts of the system that can be verified are relatively small. Formal verification aims to overcome this problem. Instead of exercising the system, formal verification builds mathematical models of designs and proves whether properties hold in these models. In doing so, it at least aims to cover the complete design. Model checking is a formal verification method that automatically verifies a model of a design, or generates diagnostic information if the model cannot be verified. It is because of this usability and level of automation that model checking has gained a high degree of success in verifying circuit designs.

The major disadvantage of model checking is its poor scalability. This is due to its algorithmic nature: namely, every state of the model needs to be enumerated. In practice, properties of interest may not need the exhaustive enumeration of the model state space. Many properties can be verified (or falsified) by examining a small number of states. In such cases, exhaustive algorithms can be replaced with search algorithms that are directed by heuristics. Methods based on heuristics generally scale well.

This thesis investigates non-exhaustive model checking algorithms and focuses on error detection in system verification. The approach is based on a novel integration of symbolic model checking, heuristic search and abstraction techniques to produce a framework that we call abstraction-directed model checking. There are 3 main components in this framework. First, binary decision diagrams (BDDs) and heuristic search are combined to develop a symbolic heuristic search algorithm. This algorithm is used to detect errors. Second, abstraction techniques are applied in an iterative way. In the initial phase, the model is abstracted, and this model is verified using exhaustive algorithms. If a definitive verification result cannot be obtained, the same abstraction is re-used to generate a search heuristic. The heuristic in turn is used to direct a search algorithm that searches for error states in the concrete model. Third, a model transformation mechanism converts an arbitrary branching-time property to a reachability property. Essentially, this component allows this framework to be applied to a more general class of temporal property. By amalgamating these three components, the framework offers a new verification methodology that speeds up error detection in formal verification. The current implementation of this framework indicates that it can outperform existing standard techniques both in run-time and memory consumption, and scales much better than conventional model checking.
This thesis is dedicated to my wife, Jing, and our daughter, Rachael.
Contents

Contents  ii

List of Figures  vi

List of Tables  viii

Acknowledgements  ix

1 Introduction  1
  1.1 Overview  1
  1.2 Aims  4
  1.3 The Approach in A Nutshell  5
  1.4 Contribution  6
  1.5 Thesis Organization  8

2 Background and Related Work  10
  2.1 Abstraction Based Verification  10
    2.1.1 Symbolic model checking  10
    2.1.2 Abstractions in verification  12
    2.1.3 CEGAR: counter-example guided abstraction refinement  14
    2.1.4 Efficiency considerations  16
  2.2 Directed Model Checking  17
    2.2.1 Heuristic search  17
    2.2.2 Directed model checkers  21
  2.3 Summary  26
## 3 BDD-Based Heuristic Search

3.1 Introduction .................................................. 27
3.2 Binary Decision Diagrams .................................... 28
  3.2.1 Basic BDD algorithms ................................... 33
  3.2.2 BDD-based graph search ................................. 38
3.3 A*-Based Heuristic Search ................................. 39
  3.3.1 The explicit-state A* algorithm ......................... 39
  3.3.2 A BDD-based A* Algorithm ............................. 40
  3.3.3 Application of SA*: an example ....................... 42
3.4 Heuristic Strength Experiments .......................... 44
  3.4.1 Set-up .................................................. 44
  3.4.2 Measuring informedness of heuristics ................. 47
  3.4.3 Run-time and memory comparison ...................... 48
3.5 Related Work .................................................. 52
3.6 Summary ....................................................... 53

## 4 Abstraction Databases

4.1 Introduction .................................................. 55
4.2 Abstractions in Model Checking ........................... 56
4.3 The Symbolic Abstraction Databases ...................... 58
  4.3.1 The symbolic pattern databases ....................... 58
  4.3.2 Symbolic abstraction databases ...................... 60
4.4 Abstraction-Directed Verification ......................... 63
4.5 Single Abstraction Database ............................... 64
  4.5.1 The BFS algorithm .................................... 64
  4.5.2 The directed algorithm: an example ................. 65
  4.5.3 The directed invariant verification algorithm ....... 67
  4.5.4 Experimental evaluation ............................. 69
4.6 Multiple Abstraction Databases .......................... 72
  4.6.1 Experiment set-up .................................... 74
  4.6.2 Heuristic distribution experiment .................. 75
  4.6.3 Mean heuristic value experiment .................... 76
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.6.4</td>
<td>Multiple SADB experiment</td>
<td>78</td>
</tr>
<tr>
<td>4.6.5</td>
<td>Merge strategy experiment</td>
<td>79</td>
</tr>
<tr>
<td>4.6.6</td>
<td>Evaluation</td>
<td>80</td>
</tr>
<tr>
<td>4.7</td>
<td>Summary</td>
<td>82</td>
</tr>
<tr>
<td>5</td>
<td>Heuristic Synthesis</td>
<td>84</td>
</tr>
<tr>
<td>5.1</td>
<td>Introduction</td>
<td>84</td>
</tr>
<tr>
<td>5.2</td>
<td>DFS-based Heuristic Search</td>
<td>86</td>
</tr>
<tr>
<td>5.2.1</td>
<td>BDD-IDA* algorithm</td>
<td>87</td>
</tr>
<tr>
<td>5.3</td>
<td>The 3-Phase Heuristic Synthesis Algorithm</td>
<td>88</td>
</tr>
<tr>
<td>5.3.1</td>
<td>Automated abstraction</td>
<td>88</td>
</tr>
<tr>
<td>5.3.2</td>
<td>Approximation of abstraction</td>
<td>90</td>
</tr>
<tr>
<td>5.3.3</td>
<td>Constructing abstraction database</td>
<td>91</td>
</tr>
<tr>
<td>5.4</td>
<td>Experimental Results</td>
<td>93</td>
</tr>
<tr>
<td>5.4.1</td>
<td>Small models</td>
<td>93</td>
</tr>
<tr>
<td>5.4.2</td>
<td>Real-world models</td>
<td>95</td>
</tr>
<tr>
<td>5.5</td>
<td>Related Work</td>
<td>99</td>
</tr>
<tr>
<td>5.6</td>
<td>Summary</td>
<td>100</td>
</tr>
<tr>
<td>6</td>
<td>Ordered Abstractions</td>
<td>102</td>
</tr>
<tr>
<td>6.1</td>
<td>Introduction</td>
<td>102</td>
</tr>
<tr>
<td>6.2</td>
<td>Ordered Abstractions</td>
<td>103</td>
</tr>
<tr>
<td>6.3</td>
<td>Verification Guided by Ordered Abstractions</td>
<td>105</td>
</tr>
<tr>
<td>6.3.1</td>
<td>Static abstraction-guided verification</td>
<td>105</td>
</tr>
<tr>
<td>6.3.2</td>
<td>Example</td>
<td>106</td>
</tr>
<tr>
<td>6.3.3</td>
<td>The SAG algorithm</td>
<td>108</td>
</tr>
<tr>
<td>6.4</td>
<td>Choosing an Abstraction Order</td>
<td>110</td>
</tr>
<tr>
<td>6.5</td>
<td>Experiments</td>
<td>113</td>
</tr>
<tr>
<td>6.6</td>
<td>Summary</td>
<td>114</td>
</tr>
<tr>
<td>7</td>
<td>Dealing with Full CTL</td>
<td>115</td>
</tr>
<tr>
<td>7.1</td>
<td>Introduction</td>
<td>115</td>
</tr>
<tr>
<td>7.2</td>
<td>Alternating Tree Automata</td>
<td>116</td>
</tr>
</tbody>
</table>
List of Figures

1.1 The flow of the approach ............................................. 5
2.1 The CEGAR framework ..................................................... 15
2.2 The 15-puzzle game ....................................................... 20
2.3 (a) The visited search space before and (b) after target enlargement. .... 22
3.1 The binary decision tree for a Boolean expression .......................... 31
3.2 The “merge” and “eliminate” operation .................................... 32
3.3 A BDD for $f = (x \land y) \lor z$ .......................................... 33
3.4 Two BDDs for the same function with different variable ordering .......... 34
3.5 The $A^*$ algorithm ....................................................... 40
3.6 The symbolic $A^*$ algorithm ............................................ 41
3.7 The state space of the marker game on a $2 \times 2$ board ....................... 43
3.8 The Open BDD after one iteration of state exploration ....................... 44
3.9 Part of the search graph for the sliding tile puzzle ......................... 46
3.10 The relative strength .................................................... 47
3.11 (a) The length of the Closed list in $A^*$ for all heuristics (b) The length of
Closed list in SA* for all heuristics ....................................... 48
3.12 Relative speed for the weakest heuristic, $h_1$, with an inset on the right ... 49
3.13 Relative speed for heuristic $h_2$ with an inset on the right .................. 50
3.14 Relative speed when the Manhattan distance $h_3$ is employed ................ 50
3.15 Memory consumption of both the algorithms directed by the 3 heuristics .. 52
4.1 A homomorphic abstraction of a transition system .......................... 58
4.2 An example of symbolic pattern database ................................ 60
4.3 Abstraction-directed symbolic model checking ................................ 63
### List of Tables

3.1 Truth tables for Boolean connectives .................................. 29
3.2 A BDD implementation of Set operations ............................... 39
4.1 An example of pattern database ........................................... 59
4.2 Experimental Results for the tree arbiter and DME ..................... 71
5.1 Model used in the experimentation ....................................... 96
5.2 Experimental results for BDD-IDA*, BDD-BFS ......................... 97
5.3 Experimental results for BDD-IDA*, BDD-BFS and SAT .............. 98
6.1 Run-time comparison for SAG algorithm ................................. 113
Acknowledgements

At the end of this long journey, I would like to express my sincere gratitude to people for their support and encouragement.

My foremost thank must go to my supervisor A/Professor Albert Nymeyer. Without his patient guidance, enthusiastic discussions and thoughtful planning, I would have been still wandering outside the main hall of this research. He introduced me to this graduate thesis project and taught me many research skills over the years. I especially thank him for reading my early-stage manuscripts and papers. His tolerance on my slow improvement in writing encouraged me to keep on track of this fantastic research project. My communication skills have been improved a lot by just talking to him. I have to admit that he is one of the best people I have ever met and make friends with. I wish I could have opportunity to collaborate with Prof. Nymeyer in the future.

I am in deep debt to my family. My wife, Jing, has been always supporting me both spiritually and financially. With a slim income, I could not have survived without her constant support. Most importantly, she gave me a little angel, Rachael, in the middle of this research. Jing and Rachael, this thesis is for you and I love you two.

I also would like to acknowledge the financial support from my parents-in-law. I could not have come to Australia in the first place without their aid. My mum, dad and sister have all been supportive to me. Thank you all!

I would like to thank the three examiners of my thesis, Stefan Edelkamp, Ansgar Fehnkers and William Visser. I am grateful for the time you spent in reading my thesis and writing valuable report. I could not have fixed many little errors without your detailed comments.

Last but not least, Rob van Galbbeek helped in understanding bisimulation in process algebra. Steven Susanto helped me develop the code. Peng Gao put tremendous work on helping me for submission of the thesis while I was away from the campus. I also would like to thank Dr. Dongmo Zhang, a home-town fellow, who has offered the help in writing up the thesis.

I am sure there are many people that helped me in the past and aren’t mentioned here. I thank you all.

I love Australia. I love Sydney.

Thanks, mate . . .
Chapter 1

Introduction

1.1 Overview

The scale of computer systems continues to grow rapidly. The roles that computer sys-
tems play in society is also ever-widening and ever-critical, and the requirement for com-
plete trustworthiness and correctness is ever-increasing. Traditional methods of software
development lacks any formal reasoning framework, and hence are incapable of guaran-
teeing correctness. Formal methods of system development, in contrast, use mathematics
to formally analyze systems, and to varying degrees, are able to guarantee program cor-
rectness. In general, formal methods can be applied in the specification, implementation
and verification of system development. We concentrate in this thesis on one particular
formal method in verification called model checking, a method that Amir Pneuli, recipient
of the 1996 ACM Turing Award, called a cornerstone of the emerging field of verification
engineering [Pnu97].

Verification engineering is the study of techniques to verify that a system is correct.
This field is expanding rapidly as society’s expectation of system infallibility and reliance
on computer systems and communication devices increases. The results of a pan-European
survey of electronics companies\(^1\) have shown that system verification is the single biggest
problem in electronics design today, and that incomplete verification of the design is by
far the greatest single cause of severe bugs. Moore’s law [Moo65, Moo00] is resulting in
huge strides in computer power. Engineers today are creating systems that are far harder

\(^1\)Hyperactive Electronics Weekly Online: “What’s wrong with design” by Pete Davy, Oct. 2001
to verify than ever before. It is now typical in a majority of design groups to spend 50-70 per cent of the design time on verification. The IBM Research Center in Haifa state:

It is widely recognized that functional verification emerges as the bottleneck of the design development cycle. This is due to a combination of correlated factors: exponential increase in design complexity, tighter time-to-market requirements, and higher quality expectations. In parallel, verification means are not evolving at a matching pace. The cost of the late discovery of bugs is enormous. Justifying the fact that, for a typical microprocessor design project, up to half of the overall resources spent, are devoted to its verification.

The most successful verification technique is model checking, as witnessed by the very large number of public-domain and commercial model-checking tools that are available on the market. In comparison, traditional verification techniques like simulation and testing are inadequate: they are, in effect, hit and miss techniques that rely on the appropriate input being chosen that will reveal errors in the system. While these methods are widely understood and relatively easy to use, they have serious limitations when used on real-world, concurrent systems that have high complexity:

- The greater the complexity of the system the less effective these techniques are [KG99]. For example, using traditional simulation, a 500,000 gate chip design could take up to a week to verify, while formal verification can accomplish this task in about half an hour.

- Concurrency flaws cannot be easily detected by simulation because these flaws may be non-reproducible, i.e. they do not manifest themselves for the same input. This makes it extremely difficult for simulation to catch these bugs [CW96, CGP99].

When using a formal method, we do not simulate system behaviors. Instead, mathematical models are employed to establish the validity of specifications by proving theorems. By distinguishing the way the theorems are proved, formal methods can be divided into two classes: theorem proving, where the validity is proved mainly manually, and model checking, where it is proved algorithmically.

Model checking [CES86, VW86, CGP99] is an automatic method for proving that a model of a system satisfies its specification, which is represented by a temporal-logic formula. Model checking can be used to verify concurrent systems such as sequential

---

2 http://en.wikipedia.org/wiki/Model_checking

3 While there are automatic theorem provers, they generally need human interaction to guide the proving process.
circuits and communication protocols, but also software. Model checking is an example of a lightweight formal tool [Hei98]. For years, the need for more lightweight formal tools of this nature has been acknowledged (e.g. Jackson and Wing [JW96]). In contrast, heavy-duty tools, like theorem provers and proof assistants, require skilled use, which is a barrier to their adoption by industry. Indeed, almost universally, experiments (e.g., [CS96a]) have concluded that the major hindrance to the adoption of formal methods is the lack of work-force skills. A thorough survey on formal hardware verification can be found in [Gup92, KG99, Dre04]. More recently, model checking has been applied to software design validation and currently there is intensive activity in this area (see e.g. the Computer-Aid Verification Conference series 4 and the SPIN Software Model Checking Workshop series 5).

Model checking involves examining all the states of a model of a system to determine whether the behaviors of the system satisfy the specification. Model checking is hence by nature an exhaustive technique requiring a search algorithm to determine all the states of a system. The search is done algorithmically on a formalism that represents the model and specification together. If a state is found that violates the specification, the underlying search algorithm is able to produce a diagnosis, usually called a counter-example or witness.

Model checkers can represent the states in a system either explicitly, or symbolically. Explicit-state model checking manage model states explicitly usually by storing a state as an array of bits. The best-known example of an explicit-state model checker is SPIN [Hol04]. Symbolic model checking [McM93] represents the model states by data structures, called binary decision diagrams (BDDs) [Bry86], or by satisfiability clauses [BCCZ99]. A well-known example of a symbolic model checker is NuSMV [CCGR99], which is an improved implementation of SMV [McM93]. A symbolic model checker can reduce the size of the state space dramatically by verifying sets of states, instead of individual states. An extreme example is [CGL92] where SMV was used to verify specifications that contain $10^{1300}$ states. It is not predictable, however, when a symbolic model checker will work better than an explicit model checker. This depends mainly on the nature of the model and the relationship between the variables whose valuations constitute a state of the system.

However the states are represented in model checking, the exhaustive search algorithms are major obstacles for verifying large systems. System state space often grows exponentially with the number of components, causing model checking complexity also exponentially increasing. This is usually referred to as the “state explosion” problem. This is an inherent problem for model checking as all states must be examined before a

---

4http://www.informatik.uni-trier.de/~ley/db/conf/cav/index.html
5http://www.informatik.uni-trier.de/~ley/db/conf/spin/index.html
conclusion can be given. However, there are particular situations where it is not necessary to enumerate all states to give a conclusion. For example, when used to find flaws in a system, a model checker only needs to check whether there is a path that leads to the flaw. In this situation, it suffices to enumerate all states that are on the path. There are many situations like this in verification where exhaustive algorithms can be avoided. Many research in the past strive to define such situations and then develop non-exhaustive algorithms to deal with them. Our research aim in this thesis is also in-lined with this path and we base our work on symbolic model checking using BDDs.

1.2 Aims

This thesis is about development of model checking algorithms for verification scenarios where exhaustive algorithm can be avoided. Exhaustive algorithms use brute-force strategy to explore system state space and waste a lot of computation resources. The brute-force strategy can also prevent model checking from being applied to large models. We are interested in modifying and developing a new set of algorithms that minimize the total amount of computation for a given verification task. Our basic idea in the thesis is to add heuristics to model checking. With the guidance from heuristics, the algorithm does the computation only when the heuristic instructs it to do so. The heuristics should be derived systematically to avoid much user interaction. We consider in this thesis to use abstractions of the original system to provide partial information and create heuristics. Specifically, our primary aims in this thesis are to develop:

- a search algorithm that uses heuristics and can be easily integrated to our BDD-based model checking infrastructure.
- a systematic method to derive heuristics from abstractions, as ad hoc heuristics need user interaction and undermine usability.
- an algorithm that synthesizes useful abstractions automatically.
- a compositional heuristic model checking algorithm when multiple abstractions are ordered by simulation relation.
- a mechanism that can make our approach applicable to more temporal properties.
- a prototype tool that implements all algorithms in this thesis.
1.3 The Approach in A Nutshell

The basic model checking algorithm is simple: given a model and a property, a model checker determines whether the property holds for the model. Hence, a model checker can be treated as a black-box which takes an input and generates an output. The input is a model and a property, together often referred to as model under verification (MUV). The output is a Boolean value that shows whether the property holds. When the returned value is false, a model checker may generate diagnostic information that can be used to detect what went wrong. Our approach follows the same basic algorithm as other model checking approaches. However, what sets our approach apart is that we use heuristic-directed model checking algorithms and abstractions during the checking process. A brief overview of our approach is depicted in Figure 1.1.

![Diagram](image)

**Figure 1.1.** The flow of the approach

The process of checking a MUV in our approach consists of several stages, each of which
is represented by a rectangle in the diagram. The outputs are represented by circles. Each stage of the process is explained as follows.

**Model Transformation** Our approach is able to handle general temporal properties. However, the core algorithm that uses heuristics can deal with only certain types of properties. The MUV must therefore be transformed into a formalism that can be handled by our core algorithm.

**Data Abstraction** At this stage, (irrelevant) information is abstracted away. The MUV becomes an abstract MUV (AMUV) as a result of abstraction.

**Abstract Model Checking** After abstraction, the AMUV has a much smaller state space than the MUV. We can therefore use an exhaustive algorithm to check the AMUV. If the property holds, then the process is terminated and true is returned. If the property does not hold, then we do not know whether the property holds in the MUV. The reason for this is that our data abstraction stage only preserves the truthfulness of the property. A false negative is also possible.

**Directed Model Checking** In this stage, a heuristic algorithm is used to check the MUV. The heuristic is derived from the AMUV using an automated synthesis process. The final output by the heuristic algorithm is either true or false. Since the MUV is checked during this stage, these output are sound and complete if the process terminates. Therefore they are the final output of the model checking process.

As depicted in the diagram, heuristics and abstractions play important roles in our approach. Essence aspects of our approach are the following:

1. Check the abstraction whenever possible as it has a relatively small state space.
2. If truthfulness cannot be determined in the abstraction, use heuristic algorithms to avoid exhaustive computation.

The novelty of this approach is that the abstraction is essentially used twice, but in different ways. This ‘double-dipping’ of abstraction makes the approach scale better for large systems.

### 1.4 Contribution

In this thesis, model checking, heuristic search and abstraction are combined to form a novel formal verification paradigm, *abstraction-directed verification*. Unlike conventional
model checking, where exhaustive algorithms are often used, we explore the possibility of using heuristic algorithms to avoid heavy computational overhead. The abstraction is doubly effective in our approach: it not only reduces the model size, it provides guidance for heuristic algorithms. More specific contributions made by this thesis are:

- We have developed a heuristic search algorithm that is based on BDDs. Our algorithm uses a mechanism called heuristic partition to handle the heuristic function symbolically using BDDs. We also conducted an experiment called “heuristic strength” that showed the difference in performance between the explicit-state and symbolic heuristic search algorithms. The experiment showed that our BDD-based algorithm performs well when the heuristic is weak.

- Using abstractions, we have developed a heuristic called abstraction database. The BDD-based heuristic algorithm is integrated into symbolic model checking to verify safety properties using the abstraction database heuristic. Experiments show that up to 2 orders of magnitude speed-up can be achieved by our algorithm against the conventional algorithm. An experiment in using multiple abstraction databases reveals that only under certain conditions does our algorithm benefit from using multiple abstraction databases.

- We have also developed a three-phase automatic heuristic synthesis algorithm. This algorithm abstracts the model and generates the abstraction database automatically. As well, we have proposed another heuristic model checking algorithm that is based on a depth-first search (DFS) heuristic search algorithm. The synthesis makes our verification algorithm fully automatically, i.e. no user interaction is required.

- We have proposed an algorithm that uses a list of ordered abstractions. When multiple abstractions are ordered by a relation, we iteratively apply our abstraction database heuristic algorithm to different levels of the abstraction. This algorithm uses abstraction to eliminate the proportion of the state space that cannot contain a state that violates the property. This is a compositional method and can also outperform the conventional algorithm.

- We have contributed a model transformation algorithm to transform a general property to a safety property. This is essentially a symbolic search algorithm for AND/OR graphs.

- Finally, we have implemented all algorithms in the prototype tool called GOLFER and evaluated it using benchmarks from the public domain.
1.5 Thesis Organization

The thesis organization reflects the chronological development of this research very closely. After reviewing related literature in Chapter 2, we present the research as a series of publications, basically in the order they have appeared at conferences. Most chapters comprise a single publication; one chapter comprises two publications.

Each chapter is prefaced (shown in italics) by a reference to the publication in the literature, and a list of changes that have been made to the publication to make it up to date, to expand the work, or to make it fit better in this thesis. Each chapter is concluded (also shown in italics) with an explanation that motivates the next stage (i.e. chapter) of the research.

The work follows the following schema:

**Chapter 2** introduces background knowledge and reviews related work. The focus here is on verification algorithms that use abstraction and heuristic search. The counter-example guided abstraction refinement framework by Kurshan and Clark is described first, followed by the direct model checking paradigm by Edelkamp et al.

**Chapter 3** describes the modified BDD-based heuristic search algorithm and heuristic strength study.

**Chapter 4** describes the abstraction database heuristic first, then followed by the integration to model checking. The second part of this chapter describes an experimental study of heuristic model checking using multiple abstraction databases. We show the experimental results to support our conjectures.

**Chapter 5** presents another DFS heuristic algorithm that uses abstraction databases. A three-phase automatic heuristic synthesis algorithm is then presented. This chapter also presents and evaluates experimental results.

**Chapter 6** applies our abstraction database heuristic to a set of abstractions. These abstractions must be ordered by a relation called simulation. Our approach is applicable to any two consecutive abstraction in the given order. We also show experimental results in this chapter.

**Chapter 7** illustrates the model transformation mechanism. The focus is to establish the theoretical foundation of our transformation algorithm. We have implemented the core verification algorithm and experimented with a small model for proof of concept.
Chapter 8 describes our prototype tool. A users’ manual is also presented.

Chapter 9 summaries the research and draws conclusions. We have further proposed three directions for future research.
Chapter 2

Background and Related Work

This thesis develops two well-known model-checking paradigms: abstraction-based verification and directed model checking. Informally speaking, the process of abstraction reduces the model size while preserving the property of interest. For large finite state systems, the number of states is usually too large for any model-checking tool. Abstraction is used in these situations to generate a so-called abstract model whose state space is much smaller than the concrete model. If the model has an infinite state space, then abstraction is an essential step as model checking generally deals with finite state models. The second paradigm, directed model checking, involves using heuristic algorithms from artificial intelligence to improve the search-efficiency of the model checker. The heuristic algorithm is usually one of the variants of the $A^*$ algorithm. This chapter reviews these two verification paradigms and establishes the connection between our work and the literature.

2.1 Abstraction Based Verification

2.1.1 Symbolic model checking

Model checking algorithms that use BDDs [Bry86] or SAT [MMZ+01] solvers are referred to as symbolic model checking [McM93, BCCZ99]. Symbolic model checking has been very successful in hardware verification [BCL+94, CK96]. The main performance gain of symbolic model checking lies in the two features of BDDs: namely a compact state representation and efficient manipulation algorithms [BRB90]. In essence, BDDs can be used
to compress a set of states that share common portions of their representation. As well, most algorithms for manipulating Boolean functions work directly on BDDs, i.e. there is no need to decompress the representation to perform the algorithms. These features make most algorithms polynomial in the size of BDDs. Recent years have also seen the advance of research in SAT solving area, making verification using SAT-based method also possible. SAT-based methods are mainly based on translating the verification problem to a Boolean formula. The SAT solvers are employed to solve the formula and verification result can then be obtained from the fact whether the formula is satisfiable. While we will occasionally mention SAT-based procedures, our work is built on BDD-based symbolic model checking infrastructures, and in general we use the term symbolic model checking to refer to BDD-based techniques.

In general, model checking is concerned with answering the simple question: given a model \( M \) and a property \( \varphi \), can we determine whether \( M \) satisfies \( \varphi \) or not, written \( M \models \varphi \)? Here the property \( \varphi \) is usually given in some specification language, for example computational tree logic (CTL) or linear time logic (LTL). The model \( M \) is often considered as a transition system and the semantics of the model is represented by its state graph. Model checking algorithms check the validity of \( \varphi \) w.r.t. \( M \) by exhaustively searching all possible states in the graph representing the model.

In symbolic model checking, a set of states of a model is encoded by its Boolean characteristic function represented by a BDD. The transition relation of the model is handled in a similar way except that two sets of state variables are used. Given a BDD representing a set of current states, the traversal of the state graph in symbolic model checking is carried out implicitly by computing a new BDD representing all successors. This operation is often referred to as image computation, and this operation plays a very important role in symbolic model checking.

While we can obtain a compact representation of the state space by using BDDs, most algorithms in symbolic model checking still use a blind search strategy. The term 'blind' refers to the fact that all the 'latest' successor states (called the frontier states) are used to compute new successor (i.e. frontier) states during the search. Note that the frontier states forms a natural set of states that can be manipulated by BDD set operations. In general, this 'blind' search strategy is intrinsic to symbolic model checking. It is also a strategy that could lead to the exploration of irrelevant segments of the state space, and it can cause the BDD size to grow very large.

To minimize the resource usage, one should respect the principle that resources are used only when necessary. This principle in fact requires the model checking algorithms be intelligent enough to explore only the necessary portion of the state space in order
to determine the satisfaction of a property. Many approaches have been proposed to deal with the state space more intelligently. Abstraction techniques are one of the most promising techniques, and research is still very active in this area. (see e.g. [LBC05, BCP06]) Additionally, there exists a body of work that is dedicated to improve the efficiency of symbolic exploration of the state space of a model.

2.1.2 Abstractions in verification

The process of abstraction involves extracting essential aspects of a system, and omitting unnecessary detail. Formal verification in general and model checking in particular have long used abstraction techniques to reduce the large model size while preserving the property to be verified [CC77, CGL94, GS97]. When a finite state system has a very large number of states, model checking may become intractable computationally due to its exhaustive nature. Abstraction techniques can be used to derive an abstract system with a fewer number of states and verification can be carried out on the abstract system directly. Abstract systems are in principle more tractable than the original concrete system. When model checking infinite systems, the abstraction process becomes even more important. In order to make model checking work on infinite systems, one has to apply abstraction techniques that partition the infinite state space into a finite one. Again, the assumption is that the property of interest is preserved.

To statically analyze the run-time behaviors of a software system, Cousot and Cousot proposed a formally based framework called abstract interpretation in [CC77, CC92]. This framework allows many run-time properties of a program to be mathematically proved or disproved during compilation. The idea of abstract interpretation is that the semantics of a program is mapped to a abstract domain while preserving the properties of interest. For example, if we require the result of an assignment $c := a * b$ ($a$, $b$ and $c$ are integers) to be positive, we may abstract the variable $a$ and $b$ to a 3-element domain $\{+, 0, -\}$. Let $H$ be this mapping and define it as follows:

$$
H(a) = \begin{cases} 
+ & \text{if } a > 0; \\
0 & \text{if } a = 0; \\
- & \text{otherwise.}
\end{cases}
$$

Note that $b$ and $c$ are abstracted in the same manner as $a$. To check the property that $c$ is positive after the assignment, we only need to determine whether $H(a) = H(b) = +$, or $H(a) = H(b) = -$. So we do not need to compute the exact value for the variables to check the property. Usually computing these abstract values is much easier than running the program to compute the exact values. This is the basic idea of abstract interpretation.
This method is now widely used in software verification for deriving finite models from an infinite system. A notable application is the predicate abstraction for software model checking (see e.g. [GS97, BMMR01, HJMS02]).

Clarke et al. in [CGL94] proposed data abstraction for symbolic model checking. While their work is really just an application of abstract interpretation, using temporal logic to express the property sets this work apart. In proving the property preservation of their abstraction mechanism, they propose a simulation relation based on the operational semantics of abstract and concrete models. Given two models $M$ and $\hat{M}$, $\hat{M}$ simulates $M$ iff every behavior in $M$ has a corresponding mapping in $\hat{M}$. In other words, $\hat{M}$ contains all possible behaviors of $M$. It is proved in [CGL94] that this relation preserves the truth of all formulas from $\forall$CTL*, a universal segment of CTL*. In the paper they also described a method of deriving an abstract model directly from a declarative representation of the concrete model, hence avoiding dealing with a concrete model that has a large state space. Finally they proposed some abstraction mechanisms that can form simulation relations. They noted that for different classes of systems, specific abstraction methods could be used to generate appropriate abstractions.

The above work is actually closely related to Kurshan’s work [Kur94]. Kurshan proved the lifting lemma that can derive an abstraction of the concrete model based on language homomorphisms induced by Boolean algebra homomorphisms.

While these techniques reduce the effort required to verify a large model, they share a problem, namely they induce false negatives. In the situation where the abstraction model $\hat{M}$ does not satisfy the property, we cannot conclude that the concrete model $M$ violates the property. This problem is due to the fact that the simulation mentioned above is a partial order: the abstract model contains more behaviors than the concrete system contains. Certain behaviors of the abstract system may be artifacts of the abstraction process. When the standard model checking reports an error in the abstract system, it is not known whether the error is a real error or is spurious. The data abstraction mechanism is hence incomplete. To handle this problem, Kurshan [Kur94] proposed a technique localization reduction. Later Clarke et al. [CGJ+00] further develop this idea and proposed the counter-example guided abstraction refinement framework, usually referred to as CEGAR. These techniques are designed to target the false negative problem and make the abstraction-based verification sound and complete.
2.1.3 CEGAR: counter-example guided abstraction refinement

As discussed above, abstraction-based model checking may induce false negatives. False negatives are usually caused by the inability of the abstraction method to extract information from the model that will allow the property to be verified (or not): we say the abstraction is too coarse w.r.t. the property. To resolve this, the abstraction must be refined, and this will result in a more detailed (accurate) model. The CEGAR framework iterates this refinement process. The iteration is driven by a process that involves analyzing the counter-example. In [Kur94, CGJ+00, CGKS02], CEGAR is used to verify various hardware systems. Today it is often used for verifying software programs (see for example [BMMR01, HJMS02, CCG+04, BKY05]).

Figure 2.1 depicts the CEGAR model-checking framework. The process starts with the concrete model that is to be verified. An initial data abstraction is applied to this model and this results in an abstract model. This abstract model is in turn an input for a standard model checker and the outcome of this is either an indication that the property is satisfied in the abstract model, or a counter-example is generated. If the former case, we conclude that the concrete model must also satisfy the property as we have used a property-preserving abstraction. In the latter case, we do not know whether the counter-example is real or is spurious. The counter-example is passed to an analyzer to answer this question. If the counter-example is real, then we report the bug in the concrete system. Otherwise, the information from the counter-example analysis is passed back to the refinement unit and used to determine a new abstraction. This abstraction is guaranteed to eliminate the spurious counter-example. The whole process then repeats itself.

Unfortunately, it is often the case that much of the computational effort in CEGAR is used to analyze the counter-example. The abstract counter-example needs to be examined to see whether it corresponds to a real counter-example (in the concrete system). Clarke et al. [CGKS02] use a SAT-based method to simulate the concrete model constrained by the abstract counter-example. If the abstract counter-example is real, CEGAR reports the property is violated and outputs the counter-example in the concrete model. Otherwise CEGAR analyzes the abstract model to determine which abstract state caused the spurious counter-example. The abstract state is refined such that the same spurious counter-example does not re-occur.

While CEGAR offers a way of dealing with the false-negative problem, it still suffers from the following drawbacks:

- Splitting an abstract state to eliminate spuriousness is provably NP-hard [CGJ+00].
- Since the refinement is completely guided by a counter-example that is usually ran-
randomly generated from a set of counter-examples, the refinement process does not necessarily eliminate the irrelevant information. In other words, the refinement does not always lead to a more accurate model w.r.t the property that is being verified. Glusman et al. [GKMH+03] proposes an alternative scheme that uses multiple counter-examples for guiding the refinement process. In the worst case, however, it still suffers from this drawback.

- The entire framework is focused on the proof of the correctness. In practice, the generated abstract counter-example may be spurious even if there is a real concrete counter-example. Although the refinement process eventually catches the real counter-example, it has to go through many iterations, which consumes much time. CEGAR is not designed to quickly detect errors in a system: it is a verification tool.

One of the goals in this work is to develop an approach that is complementary to CEGAR, and that does not suffer the above drawbacks (or at least to a lesser degree). We repeat that our method is designed to find error states as quickly as possible. In a situation where there are no errors in the model, our approach does not have any practical advantage over existing techniques such as CEGAR.
2.1.4 Efficiency considerations

Another strategy to deal with large state space is to improve the efficiency of symbolic state-space exploration. One approach is to partition the search frontier into sets of states and rank them according to some metric, and always search the set with the highest rank first.

Ravi and Somenzi [RS95] proposed a novel method called “high density analysis” for BDD-based state-space exploration. They define the density of a BDD to be the ratio of the number of states over the number of nodes. As the size of most BDD-based algorithms have polynomial complexity, BDDs with high densities represent states more efficiently and are given higher priority. In [FKZ+00] the authors presented a similar strategy, and derive a metric that measures the representability of a BDD. Instead of statically calculating the density, their technique was based on a predicting function. During the exploration only BDDs with potentially high density are generated. Kuehlmann et al. [KMB99] presented a strategy that is based on a probabilistic ranking metric. The ranking is calculated based on the simulation statistics on a random walk model.

Another approach is based on the idea that intermediate hard-to-represent BDDs can be avoided by guiding the search. Bloem et al. observed that during the state-space exploration, the sizes of intermediate BDDs could “blow-up” (become very large). This would often cause main memory to be exhausted and the search to terminate. In [RS99, BRS00], they showed that this could be avoided by using so-called “hints” to guide the search. In essence, hints specify the ordering of layers for state-space exploration. By using appropriate hints the authors claim that while the search may need more iterations, the intermediate BDD “blow-up” is reduced.

Symbolic state-space exploration is a core operation in BDD-based model checking. While the above techniques may improve the efficiency of the exploration, they still carry out the exploration on the concrete model. The notion of hints are informal and lack any systematic treatment. While we also in a sense use “hints” to guide the search exploration, the hints come in the form of an abstract model derived from the concrete model. In our case, the hints are used to direct a heuristic search algorithm. This paradigm is often referred to as directed model checking.
2.2 Directed Model Checking

Directed model checking is an emerging field in formal verification where the focus is on detecting errors in a system. The basic idea is to use heuristic search techniques to avoid exhaustive search in system space. For pioneering work in this field, we refer to [YD98, ER98, ELLL04a]. Before delving into directed model checking, we first need to briefly describe the basic idea of heuristic search that is commonly used in artificial intelligence (AI).

2.2.1 Heuristic search

Search is a problem-solving technique commonly used in computer science. Many real-world problems can be modeled by a relational graph where nodes represent the distinctive states and edges represent the relation between nodes. The problem-solving process is then reduced to searching for a given goal state from a given initial state. Breadth-first search (BFS) and depth-first search (DFS) are two common search methods used. BFS and DFS are typical uninformed search techniques that do not make use of any domain knowledge to direct the search, in contrast to informed search techniques where the search algorithm is directed by some pre-defined heuristic. This heuristic can be derived by using domain knowledge of the problem, or some statistical information. The term heuristic search in AI refers to a class of search methods that use heuristics during the course of the search. For complex or computationally-intractable problems, heuristics are crucial to solve the problem or in certain situation to approximate the solution. Popular heuristic search algorithms used in AI are $A^\ast$, genetic algorithms and simulated annealing [RN03]. In this work we focus on $A^\ast$-based heuristic algorithms.

$A^\ast$-based heuristic search

The $A^\ast$ algorithm is one of the most prominent heuristic-search algorithms, and is used in many applications including planning problems as well as hard combinatorial puzzles [Nil80, RN03]. Most studies on the $A^\ast$ algorithm are conducted and evaluated in the context of these benchmarks. Here we sketch the basic $A^\ast$ algorithm and leave the formal description of the algorithm to Chapter 3. In the algorithm below, rank is called the heuristic function.
INPUT: A graph, an initial state and a goal state;
OUTPUT: An indication for whether the goal is found;

(1) Put the initial state with a rank into the open set;
(2) Pick a state with best rank from the open set;
(3) If the open is empty, terminate report failure;
(3) If the state is the goal, terminate and report success;
(4) Generate all new successors and put them back to open with rank
(5) Go back to step (2);

The rank is used to estimate the search effort required to go from a state to the goal. The search effort is usually measured by the “distance”, which in most applications corresponds to the total cost of some number of state transitions. The cost of a transition may be the same for all states, or it may vary. The heuristic function (i.e. rank) provides domain information that is used to direct the search, and generally prevent it from exploring parts of the state space that do not contain the goal node. Mathematically, the heuristic function used in A* is a mapping \( h : S \rightarrow \mathbb{R} \) where \( S \) is a set of states and \( \mathbb{R} \) is a set of values that can be used to measure the distance between states. A basic requirement for the heuristic function is that it should be efficient to compute and as accurate as possible, and (ideally) have the property that is does not over-estimate the cost of the path from any state to the goal state. A heuristic function that satisfies this property can guarantee the path that A* finds is the optimal path. Finding an optimal path is usually more important for problems in AI than in model checking.

While the A* algorithm uses a heuristic function to guide the search, it may still build a nearly complete search tree in the situation where the heuristic is be weak (very inaccurate). In other words, when the heuristic is weak, the search is usually poorly directed. In an extreme case, A* may perform just like a normal BFS when the heuristic function always return 0. This is the main problem for the A* algorithm that needs exponential memory for storing the states. To overcome this problem, Korf presented a memory-bound alternative algorithm, called IDA*, which only requires linear memory and often performs better than conventional A* [Kor85]. An overview of other work on memory-bounded heuristic search algorithms can be found in [RN03].

To use the A* algorithm or one of its derivatives in model checking, we must first study the different characteristics between model checking and AI problems. For problems in the AI domain, the solution generally needs to be optimal. For example, when we use heuristic search to perform planning, the aim is to determine an optimal plan with respect to some cost. In model checking, however, optimality is not necessarily required. For example,
when using heuristic search to detect bugs in a design, finding bugs is more important
than finding the shortest bug trace. This means in model checking, we may compromise
the quality of a heuristic in order to get efficiency to just find the bug.

Another problem is that model checking applications usually have extremely large state
spaces. In such cases, representing the state space using BDDs may make the problem
tractable. In that case, we need to modify the conventional A* algorithm so that it performs
its operations on BDDs and not explicit states. Other data structures such as the open
and close sets also need to be modified. There are a few BDD-based A* algorithm in the
literature, they handle open and close differently. Detail about them and our mechanism
are described in Chapter 3.

The heuristic function used by A* plays a crucial role in directing the search algorithm.
A heuristic is said to be admissible iff it never over-estimates the cost of a node. An
admissible heuristic guarantees A* algorithm finds the path with optimal cost if such path
exists. As its name suggests, heuristics are generally formulated in an ad hoc manner. In
AI literature, most heuristics are derived manually and in a specific way from the problem
domain itself. This makes the algorithm dependent on human interaction. For applications
that need high levels of automation, for instance model checking, this is a problem. In
such cases, an automated mechanism of generating the heuristic function is desired. This
can be provided by a technique that derives pattern database heuristics.

Pattern database heuristics

Heuristics can be derived by ‘relaxing’ (in a manner of speaking) the constraints on the
problem, i.e. removing some constraints. The pattern database heuristic uses this concept.
Pattern database heuristics were first introduced by Culberson and Schaeffer [CS96b]. The
creation of this heuristic is as follows:

1. Given a representation of a problem, a sub-problem is generated using a partial
   representation, called a pattern, of the original problem.
2. A brute-force method is used to solve the sub-problem.
3. During the course of solving the sub-problem, the distance from each state in the
   sub-problem to the goal in the sub-problem is recorded.
4. A hash table is used to store these states and distances. This hash table is called
   the pattern database.
A pattern database is an abstract representation of the original problem. The knowledge gained by ‘solving’ the pattern is used to direct A∗ as it is applied to the original problem.

We now use the 15-puzzle example in Figure 2.2 to illustrate an application of the pattern database heuristic [CS96b]. The 15-puzzle is an instance of \( n^2 \)-puzzle family and is a board game consisting of 16 squares and 15 tiles. The top row in the figure shows an initial and a goal configuration (i.e. state) for this game. Given these states, the objective of the game is to move the tiles around, starting in the initial state, until the goal state is reached. The only place a tile can be moved is onto the empty square on the board. For example, in the initial state, we may move any of the three tiles, 4, 15 and 12 to the empty square.

This puzzle can be easily modeled to a search problem by representing each square with a variable to contain values ranging from 0 to 15. To explain what we mean by the term pattern, suppose we are only interested in the subset of tiles \( \{1, 2, 3, 4, 5, 9, 13\} \), which is the set of tiles on the fringe of the goal state. We call this subset a pattern, and it is a partial representation of the original state (in this case the goal state). This induces a mapping from the original state space to a new state space that contains a (generally) smaller number of states than the original. It is generally smaller because any number of original states may map to the same pattern. The bottom row in Figure 2.2 shows the representation after mapping. In effect, in carrying out this mapping, we ignore the tiles that are not part of the pattern. Once we have this mapping, we can use the procedure described above to generate a pattern database. During the course of the search, this database is queried when the algorithm needs a heuristic value for a state. The query actually involves mapping the concrete state to an abstract one and look up
the heuristic value in the database. The pattern database heuristic has been the subject of intensive research in AI in recent years. Some interesting results can be found in, for example [Kor97, HNF+04, KF02, ZH05]. In this work, we place the concept of pattern databases in a symbolic setting.

2.2.2 Directed model checkers

While the work described in Section 2.1.4 uses a directed approach to state-space exploration, they do not offer a systematic approach to implementing heuristic search algorithms in formal verification. In this thesis we restrict ourselves to verification algorithms that are based on $A^\ast$. In this section, we survey research in model checking based on the $A^\ast$ algorithm. We first describe three representatives of $A^\ast$-based model checkers, SpotLight, HFS-SPIN and Java PathFinder, in an order by which they appeared in the literature. Other related $A^\ast$-based work is described together at the end of this section.

Note that the work we describe is by no means a complete list of all related work on directed model checking. Directed algorithms have also been used in other verification paradigms: for example, [LBB+01] uses this approach to perform reachability in timed automata for continuous system verification. However, we focus our attention on applications of $A^\ast$-based model checking.

SpotLight

The first implemented research in the formal verification literature that used the $A^\ast$ algorithm is Yang and Dill’s SpotLight system [YD98] and Yang’s doctorate thesis. This impact of this work is underrated in the literature: it was the first serious attempt at combating state explosion in model checking using what could be called a ‘smart’ paradigm (avoid doing unnecessary work). As they claim in their paper, model checkers are in practice more useful for hunting bugs than proving a property. A directed search strategy offers an intelligent and more efficient alternative for debugging.

The directed search algorithm proposed in [YD98] is similar to the $A^\ast$ we described previously. As well as three important heuristics, they presented a novel idea called target enlargement analysis. The idea of target enlargement is to help the search algorithm by ‘enlarging’ the goal. Actually, a goal cannot be ‘enlarged’: the nodes around the goal are identified and made known to the search algorithm. This is done by computing the pre-image of the goal, i.e. computing the set of predecessors of the goal. When using the

---

BFS and DFS search algorithms, target enlargement can reduce the height of the search tree. When used in A*, target enlargement not only reduces the height of the search tree, but also greatly reduces the number of states that need to be visited. Figure 2.3 illustrates this effect. We refer to the form of the part of the search space that has been visited as an ‘onion’ because of its familiar shape. In Figure 2.3, the black node is the initial state and the white node is the goal.

Figure 2.3. (a) The visited search space before and (b) after target enlargement.

The ‘onion’ in Figure 2.3(a) represents the visited search space before target enlargement. It is quite bulbous on top. The concentric circles in (b) indicate the target enlargement after 2 applications of the pre-image operation, and results in a new ‘leaner onion’ of visited search states. To indicate the change, we show the original onion with the dotted line. Enlarging the target can greatly increase the opportunity for the directed search to find a path to an error state and hence lead to a substantial saving in search effort.

Yang and Dill [YD98] presented 3 heuristics: the Hamming distance, tracks and guided-post. The authors claimed that these heuristics with the target enlargement can reduce the number of states that are needed to be searched by 1 to 3 orders of magnitude (for industrial benchmarks). The main problem however, as the authors pointed out, was that these heuristics generally needed user intervention, an exception being the Hamming distance. The Hamming distance was found to have inconsistent performance in practice however. While the heuristics tracks and guided-post performed better in the experiments, they both needed domain knowledge and user interaction; undesirable of course as we strive for push-button technology.

Another problem with their approach is that it only deals with safety properties. In fact, their algorithm only handles simple assertion violations. The authors did not discuss how their technique could be modified to handle liveness properties. The directed search
algorithm proposed in Yang and Dill’s work cannot be easily modified to handle this type of search.

HSF-SPIN

HSF-SPIN [ELL01, ELLL04a] is an explicit-state model checking built on top of the SPIN model checker [Hol97]. Like the SPIN, HSF-SPIN is also based on the automata-theoretic approach to linear-time temporal logic model checking. The basic principle of automata-theoretic model checking is as follows. First, the system and the negation of the property are represented with two automata. Second, build the product automaton on-the-fly to check whether an infinite word exists that belongs to the language of the product automaton. The SPIN implements a double DFS algorithm to search for a such word [CVWY92]. In the case that the algorithm finds a word, SPIN generates this word and outputs it to the user as a counter-example. Thus, there are actually 2 important components of the SPIN model checker: an automata formalism and a ‘smart’ search algorithm.

Note that, while the double DFS search algorithm presented in [CVWY92] is memory-efficient and works quite well in practice, it is still based on the blind search strategy. This blind search strategy often results in long counter-examples. This makes the interpretation of counter-example difficult for the user and hence less useful for debugging. HSF-SPIN implements different variants of $A^*$ algorithm and aims at shortening the length of the counter-example, as well as reducing the number of states that are needed to detect the error.

In HSF-SPIN, safety violation checking is handled directly by replacing the original naive blind search in SPIN with the $A^*$ algorithm. The heuristic function used in $A^*$ is recursively defined and derived from the minimum requirement to satisfy the property. For example, if the property is a formula $g \lor f$, then the heuristic value is $\min\{H(g), H(f)\}$. If the formula is $g \land f$, then the heuristic is $H(g) + H(f)$. Another way to look at this method of deriving the heuristic is that we relax the system in such a way that we satisfy the property with only necessary transitions. Heuristics created in this way is admissible, making the length of a counter-example shortest.

In order to handle liveness properties, HSF-SPIN uses a improved double DFS algorithm based on the classification of strongly connected components (SCCs). A SCC is a subgraph of a graph such that a node in the SCC is reachable from any other nodes in the SCC. In model checking, the violation of a liveness property can be captured by a SCC in the system graph. Checking liveness therefore boils down to search for that SCC.
This approach is similar to the notion of strength of a SCC presented in Kupferman and Vardi [KV98], who suggest that, depending on the strength of a SCC, different verification algorithms with different complexity can be chosen. In [ELLL04a] Edelkamp et al. presented an improved version of the double DFS algorithm. Again, in order to improve the performance of the search algorithm and shorten the counter-example, they used a hybrid algorithm that combined their improved version with A* algorithm. The experimental results showed that their improved version performed slightly better than the original but the hybrid algorithm performed rather inconsistently.

The advantage of HSF-SPIN is that it is an extension of SPIN, which, as it is well-known, provides a good platform to study how heuristic search algorithm can be incorporated in model checking. Like SPIN, HSF-SPIN is based on an explicit representation of the states, and this of courses limits the ability of the model checker to handle large state spaces. Partial-order reduction [ELLL04b] and bit-state hashing [ELLL04a] are used to address these limits to a certain extend. This work is an important development that showed that in certain cases a heuristic model checker can have better performance. The best performance improvement that the authors achieved in [ELLL04b] is about 2 orders of magnitude (factor of 237).

JAVA PathFinder

JAVA PathFinder (JPF) [VHB+00] is a software model checker that detects programming errors in Java programs. Unlike HFS-SPIN, which needs the model to be specified by a modeling language, JPF is works directly on the Java source code. JPF is encased in a virtual machine with a special mechanism to backtrack the program execution. When searching for an error in the program, JPF emulate executions by a depth-first search that traverses the control-flow graph of the program. Unlike real execution, JPF can backtrack from any point where a given condition is satisfied. JPF is an explicit-state model checker, where the state of the Java program is represented using a block of memory. As a result, state explosion is a major problem if an exhaustive search algorithm is used. The implementation of JPF uses the A* algorithm [GV02]. Instead of using a property-dependent heuristic, they proposed a set of heuristics that was based on the structural characteristics of the model. In the situation where the representation of the goal cannot be used to derive the heuristic estimation, the structural heuristics are complementary to the property-dependent heuristic.

Coupled with heuristic search techniques, JPF has been found useful in detecting errors

\[^2\]In fact, it works at Java byte code level.
in real-world software systems. A notable example was that JPF found a race violation in the software of NASA’s spacecraft called Deepspace 1 [GV02]. It was reported that JPF was used to successfully find concurrency bugs in parts of the enterprise software in Fujitsu Labs of America (see JPF homepage on SourceForge). JPF is also used as a backend of a Java software model checker, BANDERA [CDH+00], which verified modules of operating system kernel. While heuristic search greatly reduces the efforts for error detection in JPF, it is not known how well JPF scales for large systems. Experiments reported by JPF authors are based on software modules that have relatively small size. For example, in [GV02], the maximum number of states explored by JPF is around 1 million states.

**Other $A^\ast$-based model checkers**

Other work that extends the above work to liveness properties includes Tan et al. in [TAC+04], who presented a novel 2-stage algorithm to search for a counter-example of a liveness property in their tool FLAVERS. They divided the search into two stages and used a different heuristic for each stage: a structural heuristic is used for the first stage and a property-dependent heuristic for the second stage. The authors claim that this hybrid use of heuristics is able to speed up the search for a counter-example as well as to shorten the length of it.

Seppi et al. [SJL04] proposed a *meta-heuristic* that is based on the empirical Bayes method. A meta-heuristic revises a search estimation using statistical information. The results showed that the heuristic reduced the number of states explored by as much as 4 orders of magnitude. This work opens another door for designing better and more robust heuristics, which play such a vital role in heuristic search.

Recently, Dräger et al. [DFP06] proposed a method called *distance-preserving abstraction* to create heuristics for directed model checking. While this work uses a similar approach to ours in this thesis, it is based on explicit-state representation. Most experimental models reported in the paper have less than 1 million states. Also recently predicate abstraction has been used in AI planning to create heuristics for the real-time model checker Uppaal [HSR+06, KHDB06]. These works extends (partially) the idea of this thesis and apply it to AI planning domain.
2.3 Summary

In this chapter, we have described two important model-checking techniques that form the basis of the work presented in this thesis. The first, abstraction, is a powerful technique to reduce the model size. However, a poor abstraction can lead to the creation of spurious errors, and these may hinder the search for real errors. The second, heuristic search, can also suffer poor performance if the heuristic is not chosen wisely, and in that case the resulting search will be close to exhaustive. The above 2 approaches form the background to the research carried out in this thesis. These 2 approaches are in fact completely integrated in this work to create a single framework.

Before building this framework, we first need to study how heuristic search can be implemented in a system that is represented as BDDs. We furthermore need to study the effectiveness of BDD-based heuristic search when heuristics of different strengths are used.
Chapter 3

BDD-Based Heuristic Search

This chapter is based on the work [NQ03] presented at the Australian AI conference in 2003. The changes made are:

- A section introducing BDD basics has been added.
- The explicit-state $A^*$ algorithm is described in more detail.

3.1 Introduction

In heuristic search, heuristics are used to direct the search algorithm to a particular node of the graph, and thereby avoid traversing certain parts of the search graph. Heuristic search methods have been employed to solve a variety of problems in areas such as problem-solving tasks, AI planning as well as finding errors in formal verification. In a broad sense, heuristic search includes $A^*$-like algorithms, hill-climbing, simulated annealing and genetic algorithms [RN03]. In this thesis we focus on the $A^*$-like heuristic search algorithms. The heuristic search algorithm $A^*$ has been the subject of much research in AI literature. Traditionally, $A^*$ handles states explicitly. When a problem is modeled as a graph, $A^*$ uses a block of bits to explicitly represent a state and store all the explored states in memory. As the search grows, the memory required by $A^*$ increase exponentially. The practical use of $A^*$ is therefore handicapped by this exponential memory requirement.

Much work has been proposed to improve memory-efficiency of $A^*$ algorithm. Korf
Kor85] presented a variation, called IDA∗, that consumes a linear amount of memory w.r.t. its search depth. Bounded by a given depth, IDA∗ searches the graph in depth-first fashion but still guided by a heuristic. If the goal cannot be found within the bound, it increases the bound and starts the search over again. In effect, IDA∗ iteratively searches the graph using a directed depth-first search. While IDA∗ explores states many times, it only uses linear memory. As a result, it can handle problems that cannot be handled by A∗ [Kor95]. Russell et al. also proposed a memory-bounded variation called SMA∗ [RN03]. SMA∗ performs heuristic search in two distinct stages. The first stage uses A∗ search until the memory is used up. The second stage basically use IDA∗ as it requires very little memory. Recently in [KZTH05] Korf et al. proposed frontier heuristic search. The key idea is that the algorithm only stores the frontier nodes. The solution path needs to be constructed by using a divide-and-conquer algorithm. While these variations improves the performance of heuristic search, the results cannot be translated to formal verification problems.

The models we are interested in usually have very large state spaces (for example, $2^{50}$ states [BCM+90]). Binary decision diagrams (BDDs) [Bry86] are symbolic data structures that can represent a large set of states in a compact directed acyclic graph (DAG), and can be used as the underlying data structure for symbolic model checking [McM93]. In order to apply heuristic search techniques in symbolic model checking, the first obstacle is to modify the A∗ algorithm and base it on BDDs. The modified heuristic search algorithm behaves like the conventional A∗ but processes sets of states represented as BDDs.

In this chapter, we first introduce the basics of BDDs, and then describe both the explicit-state A∗ and the modified BDD-based A∗ algorithm, which we call SA∗. We also describe the results of experiments we have conducted.

3.2 Binary Decision Diagrams

In this section we introduce the basics of binary decision diagrams (BDDs). Note that we only describe concepts and algorithms that are related to our work. A thorough and detailed description of BDDs can be found in [Bry86].

BDDs are graphical representation for Boolean expressions. Boolean expressions are constructed by Boolean variables, Boolean connectives and two constants, 0 and 1. A variable $x$ is a Boolean variable if it ranges over the two constants. Boolean connectives include conjunction $\land$, disjunction $\lor$, negation $\neg$, implication $\rightarrow$ and bi-implication $\leftrightarrow$. 28
The negation \( \neg \) is a unary connective and others are binary connectives. All Boolean expressions are defined recursively as follows.

- A Boolean variable \( x \), the two constants \( 0 \) and \( 1 \) are Boolean expressions.
- If \( t_1 \) and \( t_2 \) are Boolean expressions, so are \( \neg t_1 \), \( t_1 \land t_2 \), \( t_1 \lor t_2 \), \( t_1 \rightarrow t_2 \) and \( t_1 \leftrightarrow t_2 \).

For example, \( x \land y \lor z \) is a legal Boolean expression with three Boolean variables \( x \), \( y \) and \( z \). Note that there is a convention that the Boolean connectives have their relative binding strength. The ordering is used to avoid an excess of parentheses. The order is \( \neg \), \( \land \), \( \lor \), \( \leftrightarrow \) and \( \rightarrow \), with the strongest first. Thus, \( x \land y \lor z \) is actually equivalent to \( (x \land y) \lor z \).

Having defined the syntax of Boolean expressions, we are ready to define the semantics of Boolean expressions. If a Boolean variable \( x \) gets assigned a constant value, for instance \( x := 1 \), we call the value of \( x \) a truth assignment. By definition, the set of truth assignments for Boolean variables consists of two elements, denoted \( \mathbb{B} = \{0, 1\} \). We denote a Boolean expression \( f \) with \( n \) variables by \( f(x_1, x_2, \ldots, x_n) \). A list of truth assignments for all variables \( (x_1, x_2, \ldots, x_n) \) is called a Boolean vector. Such a vector is called a valuation of the expression. For example, a Boolean vector \((0, 1, 0)\) is a valuation for the expression \( f(x, y, z) = x \land y \lor z \). It is easy to see that an expression \( f \) with \( n \) variables can be valuated to at most \( 2^n \) different Boolean vectors, denoted \( \mathbb{B}^n \). Hence, every Boolean expression essentially defines a function \( \mathbb{B}^n \rightarrow \mathbb{B} \). A table is called a truth table for a Boolean expression if it enumerates all possible Boolean vectors and corresponding evaluations. The truth tables of the basic connectives are shown in Table 3.1. Given a Boolean expression \( f \) with \( n \) variables, one can construct its truth table by evaluating each Boolean vector according to Table 3.1. While we have many connectives, it is not difficult to show that all other connectives can be encoded by \( \neg \) and \( \land \). For instance, \( x \lor y \) can be encoded to \( \neg(\neg x \land \neg y) \). This conversion is also referred as DeMorgan’s law.

![Table 3.1. Truth tables for Boolean connectives](image)

A Boolean expression is said to be a tautology if it evaluates all possible Boolean vectors to \( 1 \). Two expressions with same number of variables, \( f_1 \) and \( f_2 \), are said to be equivalent if they have a same truth table. Hence, a Boolean expression can only have one truth table and we call the truth table a canonical representation of the expression.
The main problem of this representation is that it is a pure enumerative representation and hence suffers from exponential “blow-up” when the number of variables increases. In practice such a representation is hardly useful as real-world applications usually have many variables and the truth tables are often excessively large. To manipulate such expressions, a compact representation and efficient algorithms are needed. BDDs are used to deal with this problem and have gained much success in the field of formal verification, especially in hardware verification.

Before delving into the details of BDDs, we introduce another Boolean connective, called if-then-else and denoted \( \text{ite}(t, x, y) \) where \( t, x \) and \( y \) are Boolean expressions. Different from \( \neg \) and \( \land \), which are either unary or binary, \( \text{ite} \) is a ternary connective that takes three arguments. The semantics of this operator is defined by

\[
\text{ite}(t, x, y) = (t \land x) \lor (\neg t \land y)
\]

An interpretation of \( \text{ite} \) is that, to determine the truth value of an \( \text{ite} \) formula, one first examines the value of \( t \), if \( t \) is true then the formula is equivalent to \( x \), and to \( y \) otherwise. Here \( t \) is called a decision expression. Together with two constants, 0 and 1, any Boolean expression can be formulated by the \( \text{ite} \) connective. For example, \( x = \text{ite}(x, 1, 0) = (x \land 1) \lor (x \land 0), \neg x = \text{ite}(x, 0, 1) = (x \land 0) \lor (\neg x \land 1) \) and \( x \land y = \text{ite}(x, y, 0) = (x \land y) \lor (\neg x \land 0) \), where \( x, y \) is an arbitrary Boolean expression. Since \( \neg \) and \( \land \) are sufficient to express any Boolean formula, the \( \text{ite} \) and two constants 0 and 1 are also sufficient to express all Boolean expressions. An expression containing only 0, 1 and \( \text{ite} \) is called ITE normal form.

The process of translating a Boolean expression to its ITE normal form is referred to as Shannon’s Expansion. Shannon’s expansion decomposes a Boolean expression by identifying the cofactors of a variable \( x \) of the expression. That is:

\[
f(\ldots, x, \ldots) = \text{ite}(x, f_x, f_{\neg x}) = (x \land f_x) \lor (\neg x \land f_{\neg x})
\]

where \( f_x \) and \( f_{\neg x} \) are the cofactor of \( x \). We denote \( f[t/x] \) as the substitution of all occurrence of \( x \) in \( f \) with \( t \). The positive cofactor \( f_x \) of \( x \) is defined as \( f_x = f[1/x] \) and similarly \( f_{\neg x} = f[0/x] \). By recursively applying Shannon’s expansion until each cofactor is either 0 or 1, any expression can be translated to ITE normal form. This translation can be viewed graphically as the process essentially generates a binary tree. Each node of the tree corresponds to a variable, and its two children to its positive and negative cofactor. Hence this tree is called the binary decision tree. An example of Shannon’s Expansion is shown below. Its corresponding binary decision tree is shown in Figure 3.1.
Example 3.2.1  Shannon’s expansion of the expression $f = (x \land y) \lor z$.

$$f = \text{ite}(x, f_1, f_0) \quad \text{where } f_1 = y \lor z, f_0 = z;$$

$$f_1 = \text{ite}(y, 1, f_{10}) \quad \text{where } f_{10} = z;$$

$$f_0 = \text{ite}(y, f_{01}, f_{00}) \quad \text{where } f_{01} = z, f_{00} = z;$$

$$f_{10} = \text{ite}(z, 1, 0);$$

$$f_{01} = \text{ite}(z, 1, 0);$$

$$f_{00} = \text{ite}(z, 1, 0);$$

During Shannon’s expansion, we must determine a variable ordering before the cofactors of each variable are evaluated. Different variable orderings may result in different binary decision trees. Later when we consider the compact representation of these trees, the choice of the variable ordering will play a crucial role in determining the size of the BDD.

Given a variable ordering, the binary decision tree is a graphical representation of a Boolean expression. Compared with the truth table representation, the tree representation is in general more compact. For example, in Figure 3.1, if the truth assignment for $x$ and $y$ are both 1, then the evaluation of the expression is 1 regardless of the value of the variable $z$. However, in a truth table two distinct rows of Boolean vectors would need to be enumerated.
Binary decision diagrams are based on binary decision trees, and uses two extra rules, *merge* and *eliminate*, to make the tree even more compact. Once the variable ordering is fixed, all nodes at every level of the decision tree are associated with the same variables except the 0 and 1 nodes. Since every Boolean expression can be evaluated to 0 or 1 for a given Boolean vector, we call these two special nodes *terminal* nodes. Other nodes are called *non-terminal* nodes.

The “merge” and “eliminate” rule are depicted in Figure 3.2. Since all decision trees only have two terminal nodes we use only two distinct nodes, called 0-node and 1-node, to represent them. The merge operation specifies that if, at the same level, two nodes have exactly the same positive and negative cofactors, we can merge them as they represent the same Boolean function. For example, consider the unshaded oval in Figure 3.2. Three nodes associated with the variable $z$ can be merged as they share the same cofactors. After this merge, there is a node whose positive and negative cofactor point to the same node. This means the function this node represents is independent of the variable that is associated with this node. Hence we apply the eliminate rule to delete this node and link all the incoming edges to its cofactor node as shown by the shaded oval in the figure. The merge and elimination process eventually terminates and the resulting directed graph is a BDD. In a BDD, no two nodes represent the same function and there are no redundant nodes. Such a BDD is called a Reduced Ordered BDD (ROBDD). In this thesis, unless otherwise specified, all BDDs will be ROBDDs, but for convenience sake, we will refer to them as BDDs. The resulting BDD for this example is shown in Figure 3.3.

A BDD is a *canonical* representation for Boolean expressions, which means the BDD
that represents a Boolean expression is unique, provided that the variable ordering is fixed. This canonical property of BDDs is useful for the efficiency of some algorithms, e.g. the testing of equivalence of two BDDs can be done in constant time. The size of a BDD is the total number of nodes. As most algorithms on BDDs are based on the graphical manipulation of the DAG in memory, the size is often a key for the efficiency. Algorithms in [Bry86] have polynomial time w.r.t. the size of input BDDs.

Changing a variable ordering does not change the expression that the BDD represents, but a good variable ordering can reduce the size of the BDD significantly. For example, we show two BDDs with different orderings for the function \( f = (x \land y) \lor z \) in Figure 3.4. The BDD in Figure 3.4 (a) has 6 non-terminal nodes whereas in (b) the BDD has 9 non-terminal nodes. In real-world applications, a good variable ordering can result in a BDD that has polynomial size w.r.t the number of variables. An ordering that results in the minimum BDD size is called optimal ordering. However, finding such an optimal ordering is provably NP-hard [BW96]. For specific application domains, heuristics exist to help us choose a near optimal ordering in order to reduce the size of a BDD.

3.2.1 Basic BDD algorithms

By representing a Boolean expression with a BDD, we are able to implement operations on the BDD by graphical algorithms. There are an array of algorithms that manipulate BDDs efficiently. We introduce only three algorithms that are related to our work.
To guarantee the canonicity, most BDD packages use a hash table called unique table to store all BDD nodes. The function \( \text{Create}(v, f_v, f_r) \) creates a new BDD node that is associated with variable \( v \) and cofactors \( f_v \) and \( f_r \). Every node in the unique table is indexed by \((v, l, r)\). When creating a new node, \( \text{Create} \) first checks in the unique table whether such a node already exists. \( \text{Create} \) returns the pointer of the node if the node is in the table. Otherwise, \( \text{Create} \) builds a new node and updates the hash table. \( f_-(u) \) and \( f_+(u) \) denote the negative and positive cofactor of \( u \). To simplify our description, we index all variables (levels) of a BDD by 1, 2, \ldots. The root node is indexed by 1 and two terminal nodes are indexed by the height of the BDD. \( \text{TopVar}(u) \) returns the top variable index of \( u \), which is a BDD. For efficiency concerns, dynamic programming is used for many BDD algorithms. A cache called \( \text{ComputeTable} \) is used to store the intermediate result of recursive computations.

Figure 3.4. Two BDDs for the same function with different variable ordering
The Apply algorithm

The **Apply** algorithm takes two BDDs, \( b_1 \) and \( b_2 \), and a Boolean operator \( op \) as input. This algorithm computes a new BDD representing the function \( b_1 \ op \ b_2 \). The operator is a binary Boolean connective such as \( \land \) or \( \rightarrow \).

Initialize \( ComputeTable \)

**Procedure Apply** \((b_1, b_2, op)\)

1. if \( ComputeTable(b_1, b_2) \neq \emptyset \) then
2. return \( ComputeTable(b_1, b_2) \)
3. if \( (b_1 \in \mathbb{B}) \land (b_2 \in \mathbb{B}) \) then
4. \( b \leftarrow op(b_1, b_2) \)
5. if \( TopVar(b_1) = TopVar(b_2) \) then
6. \( l \leftarrow Apply(f_-(b_1), f_-(b_2), op) \)
7. \( r \leftarrow Apply(f_+(b_1), f_+(b_2), op) \)
8. \( b \leftarrow Create(TopVar(b_1), l, r) \)
9. if \( TopVar(b_1) < TopVar(b_2) \) then
10. \( l \leftarrow Apply(f_-(b_1), b_2, op) \)
11. \( r \leftarrow Apply(f_+(b_1), b_2, op) \)
12. \( b \leftarrow Create(TopVar(b_1), l, r) \)
13. if \( TopVar(b_1) > TopVar(b_2) \) then
14. \( l \leftarrow Apply(b_1, f_-(b_2), op) \)
15. \( r \leftarrow Apply(b_1, f_+(b_2), op) \)
16. \( b \leftarrow Create(TopVar(b_2), l, r) \)
17. \( ComputeTable(b_1, b_2) \leftarrow b \)
18. return \( b \)

In lines 1-2, the algorithm first looks up the computing cache for the result. Lines 3-4 handle terminal cases. Note that this is a general algorithm for any binary Boolean operator. For a particular operator, handling terminal cases can be optimized. For instance, if the operator is \( \land \) and one of the operand is 0, then the algorithm can immediately return 0 without making any recursive calls. Lines 5-16 handle three possible cases according to the index of the top variable. Note that the algorithm always calls \( Create \) to build a new node in order to maintain the canonicity of the BDDs. Line 17 depicts the mechanism of dynamic programming, namely before returning the result it is stored in the cache. Due to the cache technique, the complexity of this algorithm is reduced to \( O(|b_1| \cdot |b_2|) \), where \(|b_1|\) and \(|b_2|\) are the sizes of the operands.
The Restrict algorithm

Recall that \( f[t/x] \) denotes the substitution all occurrences of \( x \) in \( f \) with \( t \). If \( t \) is a constant 0 or 1, \( f[t/x] \) is called a restriction on \( f \). In general, we can restrict a variable of a BDD to a constant value. The algorithm \textbf{Restrict} below achieves the restriction of a BDD.


code

Initialize \textit{ComputeTable}

Procedure \textbf{Restrict} \((b, i, t)\)

1. if \( \text{ComputeTable}(b) \neq \emptyset \) then
2. \hspace{0.5cm} return \( \text{ComputeTable}(b) \)
3. if \((\text{TopVar}(b) = i) \land (t = 0)\) then
4. \hspace{0.5cm} return \( \text{Restrict}(f_-(b), i, t) \)
5. if \((\text{TopVar}(b) = i) \land (t = 1)\) then
6. \hspace{0.5cm} return \( \text{Restrict}(f_+(b)), i, t \)
7. if \( \text{TopVar}(b) > i \) then
8. \hspace{0.5cm} return \( b \)
9. if \( \text{TopVar}(b) < i \) then
10. \hspace{1cm} \( l \leftarrow \text{Restrict}(f_-(b), i, t) \)
11. \hspace{1cm} \( r \leftarrow \text{Restrict}(f_+(b), i, t) \)
12. \hspace{1cm} \( c \leftarrow \text{Create}(\text{TopVar}(b), l, r) \)
13. \hspace{0.5cm} \( \text{ComputeTable}(b) \leftarrow c \)
14. return \( c \)

define

The input argument of the algorithm consists of a BDD \( b \), variable index \( i \) and \( t \in \mathbb{B} \). Once again, due to recursive calls, cache is used to reduce complexity. The structure of the algorithm is similar to \textbf{Apply}. It is not difficult to show that the complexity of \textbf{Restrict} is \( O(|b|) \).

One of the important applications of \textbf{Restrict} is to compute the quantification of Boolean functions. For example, the result of \( \exists x f(\ldots, x, \ldots) \) can be computed by restrict algorithms on the BDD for \( f \). By definition, \( \exists x f = f[0/x] \lor f[1/x] \). Hence, the existential quantification can be accomplished by two \textbf{Restrict} operations and an \textbf{Apply} operation. Universal quantification can be handled by using DeMorgan’s Law to convert it to an existential quantification.
The Simplify algorithm

This algorithm is due to [CBM90]. The purpose of this algorithm is to simplify a BDD, $b$, taken as input of the algorithm. Another input $d$ is also a BDD. The final return of the algorithm is a BDD $b'$ such that $d \land b = d \land b'$. The BDD $b'$ is not a simple result of $d \land b$. The size of $b'$ is no larger than $b$, otherwise $b' = b$. The BDD $d$ is often called a care set. In other words, the truth assignments of $b$ and $b'$ are exactly same when $d$ is true. We shall discuss the use of the algorithm in the context of state space search later. Note that the complexity of this algorithm is also $O(|d| \cdot |b|)$ due to cache techniques.

Initialize $\text{ComputeTable}$

Procedure Simplify $(d, b)$

1. If $\text{ComputeTable}(d, b) \neq \emptyset$ then
   2. Return $\text{ComputeTable}(d, b)$
3. If $d = 0 \lor b = 1$ then return $d$
4. If $b = 0 \lor d = 1$ then return $b$
5. If $\text{TopVar}(d) = \text{TopVar}(b)$ then
   6. If $f_-(d) = 0$ then
      7. $c \leftarrow \text{Simplify}(f_+(d), f_+(b))$
   8. Else if $f_+(d) = 0$ then
      9. $c \leftarrow \text{Simplify}(f_-(d), f_-(b))$
   10. Else
       11. $l \leftarrow \text{Simplify}(f_-(d), f_-(b))$
       12. $r \leftarrow \text{Simplify}(f_+(d), f_+(b))$
       13. $c \leftarrow \text{Create}(\text{TopVar}(b), l, r)$
       14. Return $c$
   15. If $\text{TopVar}(d) < \text{TopVar}(b)$ then
       16. $l \leftarrow \text{Simplify}(f_-(d), b)$
       17. $r \leftarrow \text{Simplify}(f_+(d), b)$
       18. $c \leftarrow \text{Create}(\text{TopVar}(d), l, r)$
   19. Else if $\text{TopVar}(d) > \text{TopVar}(b)$ then
       20. $l \leftarrow \text{Simplify}(d, f_-(b))$
       21. $r \leftarrow \text{Simplify}(d, f_+(b))$
       22. $c \leftarrow \text{Create}(\text{TopVar}(b), l, r)$
   23. $\text{ComputeTable}(d, b) \leftarrow c$
24. Return $c$
3.2.2 BDD-based graph search

To represent a graph symbolically and carry out a search on the graph, two key issues are involved: the encoding of states and the transition relation.

**Encoding states** We denote $|G|$ as the size of a graph $G$ and $|x|$ as the size of a Boolean vector $x$. A single state can be encoded by a Boolean vector $x$, where $|x|$ is determined by $|G|$. The size of the vector $|x|$ should satisfy the inequality $|G| \leq 2^{|x|}$. For example, if $|G| = 7$, then $|x|$ should be 3 which means 3 Boolean variables are sufficient to encode all the states of $G$. This encoding can generate 8 Boolean vectors and one of them is unused. Once we have encoded each state with a Boolean vector, we can use a characteristic function (CF) to represent the vector. The CF of a vector is a Boolean function consisting of $|x|$ Boolean variables. The CF evaluates the truth assignment of the vector to 1 and all other assignments to 0. For instance, $f = x_3 \land \neg x_2 \land \neg x_1$ is the CF for the vector $x = (100)$. A set of states can also be encoded in this way by representing them as a single BDD. If a BDD evaluates a Boolean vector to 1, the corresponding state is in the set. In this manner, BDDs offer potentially very compact representations of very large state spaces.

**Encoding the transition relation** The transition relation (TR) is a function that computes a set of successors given current state(s). In general a TR is a set of rules that specifies how the current state changes to its successor(s). In BDD-based graph search, the TR is encoded as BDDs as well. This is accomplished again using characteristic functions. The only difference is that we need two sets of Boolean variables to encode the TR: one for the current state and the other for the successors. For example, if we have a transition from $(100)$ to $(010)$, we could use a BDD $t = (x_3 \land \neg x_2 \land \neg x_1) \land (\neg x'_3 \land x'_2 \land \neg x'_1)$ to encode this transition, where $(x_3, x_2, x_1)$ and $(x'_3, x'_2, x'_1)$ represent the current state and its successor, respectively. The computation of the successor of a given state (a BDD) is done by using existential quantification. Suppose $s = x_3 \land \neg x_2 \land \neg x_1$ is the current state and we use the above transition relation $t$. The BDD representing the successor of $s$ is given by the expression

$$s' = \exists x_1, x_2, x_3(s \land t)[x'_1/x_1, x'_2/x_2, x'_3/x_3] = \neg x_3 \land x_2 \land \neg x_1$$

Note that $[x'_1/x_1, x'_2/x_2, x'_3/x_3]$ denotes the swapping of two variable sets, which essentially adds the expanded new states into the current state set.

The set operations can be performed by corresponding Boolean operations. Table 3.2 shows a list of set operations and their corresponding operations on BDDs. $CF_x$ is the
Table 3.2. A BDD implementation of Set operations

<table>
<thead>
<tr>
<th>Operations</th>
<th>Set $u_1$ and $u_2$</th>
<th>BDDs $b_1$ and $b_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Union</td>
<td>$u_1 \cup u_2$</td>
<td>$b_1 \lor b_2$</td>
</tr>
<tr>
<td>Intersection</td>
<td>$u_1 \cap u_2$</td>
<td>$b_1 \land b_2$</td>
</tr>
<tr>
<td>Subtraction</td>
<td>$u_1 - u_2$</td>
<td>$b_1 \land \neg b_2$</td>
</tr>
<tr>
<td>Complement</td>
<td>$\overline{u_1}$</td>
<td>$\neg b_1$</td>
</tr>
<tr>
<td>Membership</td>
<td>$x \in u_1$</td>
<td>$CF_x \land b_1 \neq 0$</td>
</tr>
<tr>
<td>Cardinality</td>
<td>$</td>
<td>u_1</td>
</tr>
</tbody>
</table>

characteristic function of the set member $x$, and SatCount($b$) is a function that counts all Boolean vectors that make $b$ evaluate to 1. All other operations can be derived by the core operations in the table.

3.3 $A^*$-Based Heuristic Search

3.3.1 The explicit-state $A^*$ algorithm

The conventional explicit-state $A^*$ algorithm is a generalization of best-first search, which computes a least-cost path through a search graph. Best-first search is an iterative technique that in each iteration selects and expands a least-cost successor node of all the current nodes (often referred to as Open nodes). If the selected successor has been expanded before, then we may need to propagate new costs to previously selected nodes. When a successor node is a goal node then we have found a least-cost path through the graph.

In $A^*$, we add to the actual cost of a node an estimate of the (future) cost to a goal node. Given a node $n$ in the search graph, we refer to the actual cost as $g(n)$ and the estimate of the cost to a goal, $h^*(n)$. The cost of a node is therefore $f(n) = g(n) + h^*(n)$.

The $A^*$ algorithm is shown in Figure 3.5 (the algorithm was derived from an algorithm given in [Pea84]). In this algorithm, Open and Closed are lists of nodes, where the Open list is ordered by the cost $f$. In lines 8 and 9 of the algorithm, we see that a procedure Propagation() is called when we encounter a successor node that has been visited before. The routine will update the Open and Closed sets if the cost of this successor node is less than on the previous least-cost visit. The details of this routine are not shown here.
Procedure $A^*$ ($Graph, N_{start}, N_{goal}$)

1. $Open.push() \leftarrow (0, Heu(N_{start}), N_{start})$
2. while ($Open \neq \phi$)
3. $(g, h, N) \leftarrow Open.pop()$
4. if ($N = N_{goal}$)
5. return $SUCCESS$
6. $Closed.push() \leftarrow (g, N)$
7. for each successor node $M$ of $N$ do
8. if $M \in (Open \lor Closed)$
9. Propagation($M, N, g$)
10. else
11. $g \leftarrow g + Cost(N, M)$
12. $h \leftarrow Heu(M)$
13. $Open.push() \leftarrow (g, h, M)$
14. if ($Open = \phi$) then $NoGoalExists$

Figure 3.5. The $A^*$ algorithm

3.3.2 A BDD-based $A^*$ Algorithm

Modifying the $A^*$ algorithm to use BDDs for its data structures requires the states of the system to be encoded as a Boolean function, called the characteristic function. We also need to compute the successor states (and hence a new characteristic function) using a Boolean function, called the transition function. Both functions can be represented as BDDs. As the BDD-based algorithm does not use the explicit states but a symbolic representation of these states in the form of Boolean functions, it is sometimes referred to as the symbolic $A^*$ algorithm, which we will refer to as $SA^*$. The $SA^*$ algorithm is shown in Figure 3.6.

The $SA^*$ algorithm has 4 input parameters:

**TP** The transition partition $TP$ refers to the partitions of the entire transition relation. The technique of partitioning the transition function has been widely used in symbolic model checking area and allows large transition relations, which cannot be manipulated as a single BDD, to be handled incrementally [JBV02]. The $i$th parti-
Procedure $\text{SA}^*$ ($TP, HP, CF_{\text{start}}, CF_{\text{goal}}$)

1. $\text{Open.push}(0, h_{\text{start}}, CF_{\text{start}})$

2. while ($\text{Open} \neq \emptyset$)

3. ($g, h, CF$) ← $\text{Open.pop}()$

4. if ($CF \land CF_{\text{goal}} \neq \text{False}$)

5. return $\text{SUCCESS}$

6. $\text{Closed.push}(g, CF)$

7. for $i = 0$ to $|TP| - 1$

8. $CF_{\text{next}} \leftarrow \exists x(CF \land TP(i)[x'/x])$

9. if ($CF_{\text{next}} \neq \text{False}$)

10. for $j = 0$ to $|HP| - 1$

11. $CF_{\text{tmp}} \leftarrow (CF_{\text{next}} \land HP(j))$

12. if ($CF_{\text{tmp}} \neq \text{False}$)

13. $g \leftarrow g + \text{Cost}(TP(i))$

14. $h \leftarrow \text{Heu}(HP(j))$

15. $\text{Open.push}(g, h, CF_{\text{tmp}})$

16. $\text{Open.union}()$

17. if ($\text{Open} = \emptyset$) then $\text{NoGoalExists}$

Figure 3.6. The symbolic $A^*$ algorithm

... [rest of the text]
Both algorithms iterate and test whether the Open list is empty or not: if not empty then the first element in Open is taken. As this element is a BDD that encodes many states, we will need to check the conjunction of this BDD and the goal BDD in order to determine whether the goal has been reached. If the goal has been found, the path can be extracted from the Closed list. Otherwise, we compute:

**successors** The iteration in line 8 computes the successors of all the states represented by the BDD CF. The computation \( \exists x(CF \land TP(i)[x'/x]) \) (in line 8) is called the relational production. The characteristic function CF encodes the set of current states. The transition relation TP(i) is a BDD in which half the variables (denoted by x) encode the current state, and the other half of the variables (denoted by x’) encode the successor states. To compute (the BDD of) the successors of CF, we compute the conjunction of CF and TP(i), and apply existential quantification to the result. The conjunction of two Boolean expressions can be computed by calling the algorithm Apply. The quantification can be done by calling both Restrict and Apply. Both algorithms were described in the previous section. As well, we need to change the names of the variables x’ to x and thereby generate a BDD for a new characteristic function.

**heuristic costs** The iteration in line 10 computes a new BDD that comprises the newly generated characteristic function and the heuristic cost. In line 11, a BDD, CF_{next}, is partitioned according to the heuristic function. The partition here can be done in one of the two ways. One way of doing this is to use the Apply algorithm to get the conjunction as shown. The other way is to use the Simplify algorithm. Hence, line 11 can also be \( CF_{tmp} \leftarrow \text{Simplify}(HP(j), CF_{next}) \). In line 15, this BDD and the actual and heuristic costs are added to Open.

The operation Open.union() merges all elements in Open that have the same cost \( g + h \). For the sake of clarity, we have omitted the propagation function (it can be included in a similar way to A*). Like A*, SA* guarantees it will find a least-cost path if the heuristic function is admissible.

### 3.3.3 Application of SA*: an example

We have applied the SA* algorithm to a marker game. We illustrate the computations performed by the SA* algorithm by determining a least-cost path in a simple game that moves a marker on a 2×2 board. The size of the state space in this game is 4, corresponding to the 4 board positions where the marker can sit. In the start state, the marker is in
position (0, 0), in the goal state it is in position (1, 1). A graph depicting the state space is shown in Figure 3.7. Each state (node in the graph) is labeled with the position of the marker. The marker can move between states along the edges in the graph. The labels on the edges are the costs of moves, and the labels in parentheses on the states are the heuristic costs of reaching the goal from that state.

To encode the state space we need 2 Boolean variables, \(x_0, x_1\). The characteristic function of the start state is \(CF_{\text{start}} = x_0 \land x_1\) and of the goal state is \(CF_{\text{goal}} = x_0 \land x_1\). We need 4 variables, \(x_0, x_1, x'_0, x'_1\), to encode the transition relation. The transition partitions are the following:

\[
\begin{align*}
TP(0) &= (x_0 \land \overline{x_1} \land x'_0 \land x'_1) \lor (x_0 \land \overline{x_1} \land x'_0 \land x'_1) \\
TP(1) &= x_0 \land x_1 \land x'_0 \land x'_1 \\
TP(2) &= \overline{x_0} \land x_1 \land x'_0 \land x'_1 \\
TP(3) &= (x_0 \land \overline{x_1} \land x'_0 \land x'_1) \lor (\overline{x_0} \land x_1 \land x'_0 \land x'_1)
\end{align*}
\]

and the costs of these 4 partitions are 3, 7, 14 and 16 resp. The heuristic partitions are:

\[
\begin{align*}
HP(0) &= x_0 \land x_1 \\
HP(1) &= x_0 \land \overline{x_1} \\
HP(2) &= (x_0 \land \overline{x_1}) \lor (\overline{x_0} \land x_1)
\end{align*}
\]

with heuristic values 0, 2 and 9 resp. In the first step of the algorithm, \(Open\) is set to the characteristic function of the start state \(CF_{\text{start}}\), together with its cost \(g = 0\) and heuristic value \(h = 9\). In the outer for loop iteration, we compute the successors of the start node using \(CF_{\text{next}} \leftarrow \exists x (CF \land TP(i)[x'/x])\) for \(i: 0\) to 3. For \(i = 0\) we have:

\[
CF_1 \leftarrow \exists (x_0, x_1) (x_0 \land x_1) \land ((x_0 \land x_1 \land \overline{x'_0} \land x'_1) \lor (x_0 \land x_1 \land x'_0 \land \overline{x'_1}))[x'/x]
\]

which simplifies to \(CF_1 = \overline{x_0} \land x_1\). In the inner for loop, we calculate the heuristic cost of this successor. If \(j = 2\) then \(\exists (\overline{x_0} \land x_1 \land HP(2))\) is true, so the heuristic cost \(h\) of \(CF_1\) is
given by $Heu(HP(2)) = 9$. The actual cost $g$ is given by $Cost(TP(0)) = 3$. In line 16 we add $(g, h, CF)$ to $Open$. After computing all the successors of the start node, the $Open$ set looks like:

<table>
<thead>
<tr>
<th>Successor</th>
<th>$f = g + h$</th>
<th>Characteristic Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>12 = 3 + 9</td>
<td>$CF_0 = \overline{x_0} \land x_1$</td>
</tr>
<tr>
<td>1</td>
<td>16 = 16 + 0</td>
<td>$CF_1 = x_0 \land x_1$</td>
</tr>
<tr>
<td>2</td>
<td>18 = 16 + 2</td>
<td>$CF_2 = x_0 \land \overline{x_1}$</td>
</tr>
</tbody>
</table>

and the $Open$ BDD is shown in Figure 3.8. We pick the element in $Open$ with minimum $f$ and continue iterating using the while iteration until we find the goal node.

![Figure 3.8](image)

**Figure 3.8.** The Open BDD after one iteration of state exploration

### 3.4 Heuristic Strength Experiments

#### 3.4.1 Set-up

To better appreciate the difference in performance between the $A^\star$ and $SA^\star$ algorithms we needed a problem that has a large but solvable state space. The largest solvable (by $A^\star$) $n^2 - 1$ sliding tile puzzle is the 8-puzzle. This puzzle consists of a $3 \times 3$ board and 8 tiles, numbered 1 to 8. Initially the tiles are placed in some configuration on the board. The aim is to re-arrange the tiles until some goal configuration is reached, making as few moves as possible.
To illustrate how different heuristics influence the performance of A* and SA* algorithm, we apply the search algorithms using 3 different heuristics:

**Odd tiles out of position** The first heuristic we apply is as follows:

\[ h_1(n) = \sum_{i=1}^{8} \begin{cases} 1 & \text{if } p^i \neq g^i \text{ and } i \text{ is odd} \\ 0 & \text{otherwise} \end{cases} \]

where \( p^i \) is the position of tile \( i \) in the present configuration and \( g^i \) is the position of tile \( i \) in the goal configuration.

**Tiles out of position** This heuristic is similar to the first one, except we now consider all tiles. The heuristic can also be expressed as:

\[ h_2(n) = \sum_{i=1}^{8} \begin{cases} 1 & \text{if } p^i \neq g^i \\ 0 & \text{otherwise} \end{cases} \]

where \( p^i \) and \( g^i \) have the same meanings as in \( h_1(n) \).

**Manhattan distance** This heuristic can be expressed as:

\[ h_3(n) = \sum_{i=1}^{8} |p^i_x - g^i_x| + |p^i_y - g^i_y| \]

where \( p^i_x \) is the x-coordinate of tile \( i \) in the present configuration (similarly \( p^i_y \)), and \( g^i_x \) is the x-coordinate of tile \( i \) in the goal configuration (similarly \( g^i_y \)). This heuristic computes the number of moves that each tile needs to make to reach its final position in the goal configuration, assuming there are no other tiles on the board. It never overestimates the actual number of moves that will be required.

Note that all 3 heuristics are admissible and therefore an optimal solution is guaranteed. Moreover, we can say:

\[ h_1(n) < h_2(n) < h_3(n) < h(n) \]

where \( h(n) \) is the minimum real cost from state \( n \) to the goal state. The Manhattan distance heuristic is of course the strongest heuristic, and the “odd tiles out of position heuristic” is the weakest one.

For simplicity, we let the cost of moving a tile be 1, so the cost of a configuration is the number of moves required to reach the configuration. Each configuration that we create when we move a tile can be represented by a node in a search graph. An edge in the graph represents the action of moving a tile to the vacant square on the board. As an example for what the search tree looks like, we show a part of the search graph for the
sliding tile puzzle is shown in Figure 3.9. Notice that the number of successors of a node varies between 1 and 3.

The first task is to encode all 9! configurations of the puzzle as a Boolean vector. Ideally we would like to use only 19 variables \(2^{18} < 9! < 2^{19}\) to encode the state space of the puzzle, but such an encoding requires a non-trivial computation. As each square can hold one of 8 tiles, in our implementation, we use 4 Boolean variables to represent one square and the binary assignment to them encode the tile number in it. For example, if we use \(x_0, x_1, x_2, x_3\) to represent left-upper corner square, \(x_0, x_1, x_2, x_3\) means tile number 5 is in that square. Although this kind of encoding will need more variables, it makes the transition function much easier to compute.

The next task is to partition the transition and heuristic functions.\(^1\) (We let the cost of a transition (i.e. moving a tile) be constant.) Partitioning the heuristic function involves grouping all states that have the same cost. Note that the transition and heuristic functions are computed \emph{a priori}. In the sliding tile puzzle, we keep all the transition relation and heuristic partitions in memory for convenience. In bigger applications, we would not do this as we only need this information for successor calculation and heuristic cost estimation.

We used the BDD package CUDD\(^2\) in this work. The package contains all standard BDD operations and reordering functions, as well as a set of statistical functions that help with experimentation. CUDD was written for symbolic model checking, and has been integrated with such tools as VIS\(^3\) and NuSMV\(^4\). The work was carried out on an Intel PIII 933Hz CPU with 512MB RAM.

\(^{1}\)Partitioning is actually not necessary in the puzzle but we do it for the sake of generality.
\(^{2}\)http://vlsi.colorado.edu/~fabio/
\(^{3}\)http://vlsi.colorado.edu/~vis/
\(^{4}\)http://nusmv.irst.itc.it/
We tested 500 randomly generated solvable start configurations of the sliding tile puzzle. We used the aforementioned heuristics in the test, and we studied their relative informedness (i.e. the relative strengths of each heuristic), the effect of clustering in SA*, the relative speeds of A* and SA*, and their memory consumption.

3.4.2 Measuring informedness of heuristics

A heuristic $h_a$ is said to be more informed than heuristic $h_b$ if, to find the optimal solution path, the algorithm directed by $h_a$ explores a smaller proportion of the state space than directed by $h_b$. For each test, we collected the total number of states explored by the algorithm, and we did this for each of the 3 heuristics\(^5\). From the graph in Figure 3.10 we can clearly see that when the Manhattan distance heuristic ($h_3$) is used, the algorithm explores less than 10% of the state space in order to find the solution path for all 500 different instances of the puzzle, whereas in the case of the weakest heuristic ($h_1$), the algorithm needs to explore up to 95% of the state space. Note that the tests are ordered by the results of $h_1$ and this naturally results in the perfect linear line for $h_1$ in the graph.

Figure 3.10. The relative strength
Figure 3.11. (a) The length of the Closed list in A* for all heuristics (b) The length of Closed list in SA* for all heuristics

3.4.3 Run-time and memory comparison

Clustering Effects

In contrast to A*, which processes each state individually, SA* processes states in sets, represented as Boolean functions. In our implementation we cluster those states together that have the same “g + h” cost. The graphs shown in Figure 3.11 (a) and (b) illustrate the effect of clustering.

In Figure 3.11(a) we see that in A* the length of the Closed list is linear in the explored state space, as expected. The reason for this is that A* handles the states explicitly. Note that the length of the curve is dependent on the informedness of the heuristic, and as described above, indicates the proportion of the search space that is examined.

In Figure 3.11(b) we see that for SA*, the Closed lists are more than 2 orders of magnitude shorter than those in Figure 3.11(a). This is direct consequence of clustering. We also see that the Closed lists for the strongest heuristic in SA* (Figure 3.11(b)) are longest, but also more erratic in behavior. The weakest heuristic has the shortest Closed lists and most consistent behavior.

The reason that the strong heuristic $h_3$ has longest Closed lists is that a well informed heuristic causes the search algorithm to quickly descend deep into the search space, and this results in a greater spread of costs (i.e. “g + h”) in the visited nodes. As the states in the BDDs are clustered by the cost, this greater spread result in longer Closed lists.

Note that both A* and SA* explore the same state space proportion if we apply the same heuristic.
Conversely, when the heuristic is weak, the costs of states are similar, and hence more states can be clustered, and the *Closed* list is shorter.

**Performance Comparison of A* and SA***

![Performance Comparison of A* and SA*](image)

Figure 3.12. Relative speed for the weakest heuristic, $h_1$, with an inset on the right

To understand the performance difference between the two algorithms directed by different heuristics, we plot in Figure 3.12, 3.13 and 3.14 the time taken to find the solution for both algorithms for all 500 different start configurations. The graph in Figure 3.12 shows the time consumed by both algorithms for weakest heuristic $h_1$. We order the run times according to the proportion of the explored state space. A dramatic performance difference can be observed in the graph when the explored state space is greater than 20%. To further illustrate the behavior, we magnify the left-lower corner of Figure 3.12. One can see that there is a cross-over at about 7%. A* takes less time to find the goal than SA* for all tests less than 7%, and vice versa above 7%.

The graph shown in Figure 3.13 depicts the performance of A* and A* using the next heuristic, $h_2$. We again magnify the left-lower corner area as we did above. The performance of both algorithms directed by $h_2$ is similar to the performance of $h_1$. There again is (or appears to be) a cross-over effect between A* and SA*, but this time at a slightly lower point close to 6%.

The time taken to find the goal using the third and strongest heuristic is plotted in Figure 3.14. Note that both algorithms only need to explore no more than 6% of the state space to find the goal. This contrasts with 70% for $h_2$, and roughly 95% for $h_1$. Most of
the data in Figure 3.14 lie in the area less than 3.5%. Though there are just a few data points above 3.5%, there is still a suggestion of a cross-over at above 5%.

One could expect the time taken to process the nodes in $A^*$ to be linear as each element in the Closed list represents one state. In Figures 3.12, 3.13 and 3.14 we see however that it is worse than linear for all 3 heuristics. The explanation for this behavior lies in the computation of successor nodes: a node is considered Closed when we have expanded all its successors and added them to the Open list. The branching factor is the average number of successors for each node that is visited. As the branching factor in the sliding tile puzzle is approximately 2, it takes longer than linear time to process the Closed nodes.

In the case of $SA^*$, we see the behavior of the processing times for the Closed nodes in $SA^*$ is close to linear. Each Closed node in $SA^*$ is of course a BDD. While the branching factor in $SA^*$ and $A^*$ is of course the same, state space expansion based on BDD represen-
tation is different from the explicit state representation in $A^*$. When expanding the new states for the search tree, $SA^*$ can compute all successors with just one BDD operation (see line 9 in Figure 3.6). As well, due to the clustering effect of $SA^*$, the number of $Closed$ nodes is often less than those in $A^*$. Hence, the linear-like behavior of $SA^*$ shows the advantage of combining sets of states and processing them together. This effect is most pronounced in the case of a weak heuristic when a large part of the state space must be searched.

For the weakest heuristics ($h_1$ and $h_2$), there can be a performance difference of 2 orders of magnitude. In the case where either a solution is very close to the start configuration or the heuristic employed is very strong, however, $A^*$ outperforms $SA^*$ as depicted by the cross-overs in Figure 3.12, 3.13 and 3.14. So, in the case of a weak heuristic, provided the goal is very close to the start configuration, $A^*$ outperforms $SA^*$. The reason is that $SA^*$ needs to manage the operation on current states, transition relation and heuristic partitions $^6$, which all involve possibly complicated BDDs. If the heuristic employed is very strong, $SA^*$ has almost no advantage over $A^*$ although for a few tests $SA^*$ runs faster than $A^*$. This behaviour is direct consequence of the fact that only a small part of the state space is searched.

**Memory Consumption**

We have compared the memory consumption of $A^*$ and $SA^*$ for each of the 3 heuristics. The results are shown in Figure 3.15(a), (b) and (c). Note that we have opted to keep the scales the same for each of the 3 heuristics to illustrate the disparity in the proportion of state space that is explored. Each graph does however contain 500 data points.

One could expect that $SA^*$ should consume less memory than $A^*$ due to the compact data structure (BDDs) employed. This is not the case however. As shown in the 3 graphs, $A^*$ always consumes less memory than $SA^*$, by a factor of about one order of magnitude. The reason for this difference is not clear. While a BDD node in $SA^*$ can be represented by fewer bytes than an explicit node in $A^*$, there are many more BDD nodes than explicit nodes. Furthermore, $A^*$ computes the successors and heuristic values on-the-fly whereas $SA^*$ needs to keep a transition-relation BDD and a number of heuristic-partition BDDs in memory for the purpose of expanding the search space. Note that it seems that there is no obvious way to avoid storing these BDDs in memory because once a state is encoded in a BDD, its relationship with the successors cannot be seen without the transition relationship.

---

$^6$The time taken to compute the transition relation BDD and heuristic partition BDDs has not been taken into account since they are computed $a$ priori.
Figure 3.15. Memory consumption of both the algorithms directed by the 3 heuristics

The memory usage curves for $A^*$ in Figures 3.15(a), (b) and (c) show that there is a linear relationship between the memory consumption and the proportion of the explored state space. This result is expected because one state means one node in $A^*$. In the case of $SA^*$, however, we observe that for all the heuristics, the memory consumption increases rapidly at the start but tapers off as the search paths get longer. Of course, as we noted earlier, in the case of the strongest heuristics the search paths never get long.

In Figure 3.15 (a), we observe “steps”, which indicate that the memory consumption remains constant as states are added. This occurs when the states are similar and have the same cost, and this is characteristic of a weak heuristic giving rise to more efficient clustering. If the heuristic is strong, new (successor) states are generally quite different from those in the existing Closed list, or have a different cost, so each new state will impact on the BDD size. In Figure 3.15(b) there is still evidence of “steps” but it is less pronounced, and in Figure 3.15(c) the “steps” disappear.

3.5 Related Work

The first study of BDD-based $A^*$ algorithm was called BDDA* [ER98]. Another two variants, ADDA* and setA* respectively, have been proposed in [HZF02, JBV02]. Although
they are all variants of the conventional A* algorithm representing state space using decision diagrams, they use different Open set implementation and heuristic representation. Surprisingly, BDDA* and ADDA* do not seem to include mechanisms to record path information, therefore they can only be used to find the optimal search length of the goal given the heuristic is admissible. While these algorithms were tested, no performance comparison was made between the explicit-state A* and BDD-based A* algorithms when directed by different heuristics. We believe that the nature of the heuristic is crucial to the performance of the (BDD-based) algorithm, but in the above work, this aspect is generally ignored.

3.6 Summary

In this chapter, we have presented a symbolic, BDD-based A* algorithm called SA* and described its application to simple puzzles. We have also compared the performance of this algorithm to a conventional A* algorithm, using heuristics that have varying degrees of informedness.

The experimental results show the ability of SA* to cluster sets of states. This dramatically reduces the number of iterations needed by the algorithm to find an optimal path. We observe however that the benefit to be gained from clustering depends on the informedness of the heuristic: the more informed a heuristic is, the less the impact of clustering. In general, one can expect symmetries in state configurations will result in more clustering when the heuristic is poorly informed. We also note that the longer the search path the greater the benefit from clustering.

The performance comparison between A* and SA* based on our experiments have revealed the following facts:

- If only a weak heuristic is available, then SA* is faster than A*.
- If a strong heuristic is available, or the goal happens to be very close to the start configuration, A* is faster than SA*.
- A* uses substantially less memory than SA*, irrespective of the heuristic.
- In the case of a very weak heuristic the memory usage of the two algorithms converge, and clustering leads to a step-like behavior in the memory usage.

This chapter has shown that for problems with relative small state spaces and strong heuristics, explicit A* is a better choice. If only a weak heuristic exists, using SA* can
speed up the search, but at the cost of more memory. Our results also indicate that the SA*
algorithm tends to be of more benefit when the heuristic is weak. Unlike the AI domain,
where strong heuristics do exist, heuristics for model checking are usually very difficult to
find. This is due primarily to the high complexity of the system being model checked. In
the rest of this thesis we consider only symbolic model checking.
Chapter 4

Abstraction Databases

Finding a heuristic for a directed model checker is an ill-defined process. As we saw in the previous chapter, this heuristic needs to be well-informed to be effective. In this chapter, we describe both the method and practice of how to generate an effective heuristic automatically using what are called abstraction databases.

This chapter is based on two publications. In [QN04b] we describe the role that abstraction databases can play in the formation of heuristics. We extended this work in [QNS05b] to multiple abstraction databases, and we carried a series of experiments. Both of these works are reproduced here with minor modifications to the introductions and conclusions.

4.1 Introduction

Model checking has become the formal method of choice to verify large reactive systems [CGP99]. Using model checkers to detect errors in a design offers an important alternative for simulation based verification in practice. Directed model checking [YD98, RE99, San03] enhances the error-detection capability of a model checker by applying heuristic search algorithms. However, effective and intuitive heuristics are hard to find in directed model checking because of the high complexity of the system being studied.

CEGAR framework [Kur94, CGJ+00] is an approach that attempts to verify the correctness of a system. It uses abstraction to reduce the size the of the concrete model. It is crucial in CEGAR that the abstraction is chosen wisely. A poor abstraction will lead to
an excessive number of refinements being required to locate an error state. Specifically, it will lead to spurious counter-examples being generated, which in effect mask the real error state. The process of refinement to find the real counter-example is provably hard.

In artificial intelligence, an application-independent heuristic, called pattern databases, has been successfully employed to solve hard search problems such as the N-puzzle and Rubik’s cube [CS96b, Kor97]. In [HMZM96, HH00], Holte et al. extend the notion of “pattern” to homomorphic abstractions in an attempt to automatically create application-independent heuristics. Generally, we can abstract a system using two methods: homomorphic abstraction and embedded abstraction. A homomorphic abstraction can be achieved by merging groups of states, with the result that not all transitions within the group are observable in the abstract system. An embedded abstraction can be achieved by adding more transitions to the system.

Unlike CEGAR, heuristic search using pattern databases does not require the abstraction to be very accurate. The abstraction in this situation only serves as a guide to the search algorithm. In general, even a weak heuristic can lead to a significant reduction in the size of the search space that is traversed. In this chapter, we integrate pattern databases and abstractions into a single heuristic mechanism. Following [ELL04], we call this heuristic an abstraction database.

We first investigate directed model checking that is based on a single abstraction. We then extend this to multiple abstraction databases. Note that in this chapter we only study invariant properties that can be expressed in CTL by $\text{AG} \varphi$, where $\varphi$ is a Boolean expression of atomic propositions.

### 4.2 Abstractions in Model Checking

We define a finite-state transition system in the following way.

**Definition 4.1 (Finite State Transition System)** A finite state transition system is a 4-tuple $M = (S, S_0, R, G)$, where

- $S$ is a finite set of states
- $S_0 \subseteq S$ is a set of initial states
- $R \subseteq S \times S$ is a transition relation (or operator) that determines a set of successors for a given state $s \in S$

- $G \subseteq S$ is the set of goal states

**Definition 4.2 (Solution Path)** A path in a finite transition system $(S, S_0, R, G)$, denoted by $\pi$, is a sequence of states $s_0, s_1, \ldots, s_n$ and for all $0 \leq i < n$, $s_i \in S \land (s_i, s_{i+1}) \in R$. If $s_0 \in S_0$ and $s_n \in G$, then $\pi$ is a solution path. The length of $\pi$, written $L(\pi)$, is just the number of states in the path, i.e. $L(\pi) = n + 1$.

In verification, a solution path is called a counter-example, or alternatively an error trace, as it demonstrates why the property that is being verified is not true.

The set of states of a transition system can be described by a non-empty set of state variables $X = (x_0, x_1, \ldots, x_n)$, where each variable $x_i$ ranges over a finite domain $D_i$. A homomorphic abstraction of a transition system is denoted by a set of surjections $H = (h_1, h_2, \ldots, h_n)$, where each $h_i$ maps a finite domain $D_i$ to another finite domain $\hat{D}_i$ with $|\hat{D}_i| \leq |D_i|$. If we apply $H$ to all states of a transition system, denoted by $H(S)$, we will generate an abstract version of the original, concrete system.

**Definition 4.3 (Abstraction)** Given a transition system $M = (S, S_0, R, G)$ and a set of surjective mapping functions $H = (h_1, h_2, \ldots, h_n)$, a homomorphic abstraction of $M$ is also a transition system and denoted $\hat{M} = (\hat{S}, \hat{S}_0, \hat{R}, \hat{G})$, where

- $\hat{S} = H(S)$ is a set of states with $|\hat{S}| \leq |S|$

- $\hat{T} \subseteq \hat{S} \times \hat{S}$ is a transition relation, where $(\hat{s}_1, \hat{s}_2) \in \hat{T}$ iff $\hat{s}_1 = H(s_1) \land \hat{s}_2 = H(s_2) \land \exists s_1 \exists s_2 (s_1, s_2) \in R$

- $\hat{S}_0 = \{ \hat{s} | \hat{s} \in \hat{S} \land \hat{s} = H(s) \land s \in S_0 \}$

- $\hat{G} = \{ \hat{g} | \hat{g} \in \hat{S} \land \hat{g} = H(g) \land s \in G \}$

Intuitively, a homomorphic abstraction is a kind of relaxation of the concrete transition system in the sense that we deliberately eliminate some information by merging groups
of states. In [CGL94], Clarke et al. prove that a homomorphic abstraction preserves a class of temporal properties (ACTL∗): in other words, the concrete and abstract system satisfy the same set of formulae. Note however that if the abstract system violates the property, we cannot conclude that the concrete system also violates the property: the model needs to be further refined and model-checked again. Refinements however can be computationally expensive. Instead, we use the homomorphic abstraction to construct a pattern database that guides the search towards an error state (in the concrete system).

4.3 The Symbolic Abstraction Databases

4.3.1 The symbolic pattern databases

In directed model checking, one must provide a heuristic to direct the model-checking algorithm. In practice most heuristics are application dependent, so a systematic way of deriving a heuristic directly from the system is desirable. Pattern databases provides a solution to this problem. In essence, a pattern is a subgoal of the goal state. Before formally defining the basic concepts of pattern databases, we first use a simple example to explain pattern databases as well as the symbolic representation (i.e. BDDs) that we use.

Figure 4.1. A homomorphic abstraction of a transition system

In Figure 4.1 we show a transition system and a homomorphic abstraction that restricts the right-most bit to a single-element domain. By applying this abstraction to every state of the system, we generate the abstract system shown in Figure 4.1 as well. A
pattern database can be constructed based on this abstract system. An item in the pattern database is a 2-tuple \((\hat{s}, n)\), where \(\hat{s} \in \hat{S}\) is an abstract state and \(n\) is the number of transitions required to reach \(\hat{s}\) from the abstract goal. A simple example of a pattern database for the abstract system in Figure 4.1 is shown in Table 4.1.

A pattern database is based on a backward breadth-first traversal of the abstract system \(\hat{M}\). While traversing backwards, we label each abstract state \(\hat{s}\) with the number of transitions starting at state \(\hat{G}\) to a state \(\hat{s}\), but we keep the shortest one. (When the breadth-first traversal encounters a state that has been labeled before, it is simply ignored.) Since we are interested in finite systems, the breadth-first traversal will eventually expand all states backward-reachable from \(\hat{G}\) in \(\hat{M}\). This is called the fixed point. Once we reach this fixed point, we collect all states (with labels) and put them into a table similar to Table 4.1. This is the pattern database of the system \(M\) with respect to the homomorphic abstraction \(H\). Note that many states will be labeled with the same number because they are the same distance from the abstract goal state.

In symbolic model checking, sets of states can be encoded as binary decision diagrams (BDDs) [Bry86, McM93]. Since our approach is also based on symbolic model checking, we represent the entire pattern database using BDDs as well. Note that symbolic pattern databases have been used in planning problems and can represent very large pattern databases, and often uses relatively less memory than explicit representations [Ede02]. This is mainly because of the compact nature of BDDs [Bry86]. We represent those states in explicit pattern databases that have the same label (the same number of transitions starting at \(\hat{G}\)) with a single BDD, denoted \(b_i\), where \(i\) is the label of that set of states. Because the pattern database is derived using the backward breadth-first traversal in the abstract system, \(i\) should range over \(\{0, 1, 2, \ldots, N\}\) where \(N\) is the maximum depth of the traversal. In the worst case, \(N\) would be \(|\hat{S}| - 1\).

In Figure 4.2 we show the BDDs representing the symbolic pattern database of the example in Table 4.1. We use a Boolean vector \(X = (x_0, x_1, x_2)\) to represent a state in the concrete system, so an abstract state can be represented by \(\hat{X} = (x_0, x_1)\). The symbolic

<table>
<thead>
<tr>
<th>abstract state ((\hat{s}))</th>
<th>transitions from abstract goal state ((n))</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>00</td>
<td>2</td>
</tr>
<tr>
<td>01</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 4.1. An example of pattern database
pattern database contains 4 BDDs representing the characteristic Boolean functions of the corresponding abstract states: \( b_0 = x_0 \land x_1, b_1 = x_0 \land \overline{x_1}, b_2 = \overline{x_0} \land \overline{x_1}, b_3 = \overline{x_0} \land x_1 \).

### 4.3.2 Symbolic abstraction databases

In essence, a pattern is a type of abstraction. As our representation of the problem is based on BDDs, following [Ede02, ELL04], we refer to these heuristics as *symbolic abstraction databases* (SADBs).

Since we are only interested in symbolic heuristic search in this work, we encode \( M \) using Boolean expressions. Given a transition system \( M = (S, S_0, R, G) \), we use a set of Boolean variables \( X = \{x_1, x_2, \ldots, x_k\} \) to model the state space of \( M \). A state can be represented by a truth assignment vector of \( X \) and all possible truth assignment vectors comprise the state space \( S \). The Boolean functions \( S_0(x_1, x_2, \ldots, x_k) \) and \( G(x_1, x_2, \ldots, x_k) \) are characteristic functions that represent the states in \( S_0 \) and \( G \) respectively. To encode \( R \) we need another set of Boolean variables \( X' = (x'_1, x'_2, \ldots, x'_k) \) to represent the next state of a state \( s \). Likewise, \( R(x_1, x_2, \ldots, x_k, x'_1, x'_2, \ldots, x'_k) \) is the characteristic function for \( R \). In the discussion henceforth, we use \( \mathcal{M} = (S, S_0, R, G) \) to refer to the Boolean encoding of a transition system \( M \).

If a system is modeled using a set of Boolean variables \( X = \{x_1, x_2, \ldots, x_k\} \), we call \( X_p \subset X \) a *pattern set*, and call those variables that are not in the pattern set \( X_p \). A
minterm, \( x_0 \land x_1 \land ... \land x_p \) is a conjunction of positive form of Boolean variables, where each of variable in \( X_p \) appears exactly once. Given a pattern set and Boolean encoding of a transition system \( \mathcal{M} = (\mathcal{S}, S_0, \mathcal{R}, \mathcal{G}) \), we can abstract the transition system as follows.

**Definition 4.4 (Existential Abstraction)** The abstraction of \( \mathcal{M} = (\mathcal{S}, S_0, \mathcal{R}, \mathcal{G}) \) w.r.t a pattern set \( X_p \) is also a transition system \( \hat{\mathcal{M}} = (\hat{\mathcal{S}}, \hat{S}_0, \hat{\mathcal{R}}, \hat{\mathcal{G}}) \) represented by its Boolean encodings, where

- \( \hat{\mathcal{S}} \) is a disjunction of all minterms of variables in \( X_p \)
- \( \hat{\mathcal{S}}_0 \equiv \exists X_p S_0(x_1, x_2, \ldots, x_k) \)
- \( \hat{\mathcal{R}} \equiv \exists X_p X'_p \mathcal{R}(x_1, x_2, \ldots, x_k, x'_1, x'_2, \ldots, x'_k) \)
- \( \hat{\mathcal{G}} \equiv \exists X_p \mathcal{G}(x_1, x_2, \ldots, x_k) \)

**Definition 4.5 (Symbolic Abstraction Databases)** Given a Boolean encoding \( \hat{\mathcal{M}} = (\hat{\mathcal{S}}, \hat{\mathcal{S}}_0, \hat{\mathcal{R}}, \hat{\mathcal{G}}) \) of a transition system, we call a set \( \sigma = \{(B_0, 0), (B_1, 1), \ldots, (B_n, n)\} \) symbolic abstraction database such that:

- \((B_i, i)\), where \( B_i \) is a Boolean characteristic function and \( i \geq 0 \)
- \( B_0 \equiv \hat{\mathcal{G}} \) and \( B_i \equiv \exists X'_p (B_{i-1}[X_p/X'_p] \land \hat{\mathcal{R}}) \) for all \( 0 < i \leq n \)
- \( B_n \land \hat{\mathcal{S}}_0 \neq \text{False} \) and \( B_i \land \hat{\mathcal{S}}_0 \equiv \text{False} \) for all \( 0 \leq i < n \)
- \( B_i \land B_j \equiv \text{False} \) for all \( 0 \leq i \neq j \leq n \)
- \( B_0 \land B_1 \land \ldots \land B_n \equiv \text{False} \) for all \( 0 \leq i \leq n \)

The length of a symbolic abstraction database is \(|\sigma|\).

Intuitively, because the homomorphic abstraction clusters a set of states of the concrete system as a single state of the abstract system, paths in the concrete system can be short-circuited. In the extreme case, 2 states that are not reachable from each other in the
concrete system can become reachable when mapping them to the abstract system. For a transition system \( M = (S, T, S_0, G) \), we define the cost of any state \( s \) to be the minimum number of transitions using backward breadth-first traversal starting at \( G \), denoted \( c_s \). The following lemma shows that the cost of the state \( \hat{s} \) in the abstract system is a lower bound of the cost of its corresponding concrete state \( s \).

**Lemma 4.1** Given 2 transition systems \( M = (S, S_0, R, G) \) and \( \hat{M} = (\hat{S}, \hat{S}_0, \hat{R}, \hat{G}) \), if \( \hat{M} \) is a homomorphic abstraction of \( M \), i.e. \( \hat{M} = H(M) \), then for any state \( s \in S \) and \( \hat{s} \in \hat{S} \),\n\[ c_s \geq c_\hat{s} \text{ where } \hat{s} = H(s). \]

**Proof.** Suppose \( \pi = s_1, s_2, s_3, \ldots, s_n \) is an arbitrary path in \( M \) with \( s_n = G \) and all states along the path are distinct, then following the definition of the cost function we immediately have \( c_{s_1} = n \). If all the states \( H(s_1), H(s_2), \ldots, H(s_n) \) are also distinct, then following the definition of a homomorphic abstraction, there exists a path \( \hat{\pi} = H(s_1), H(s_2), \ldots, H(s_n) \). Thus, \( c_{s_1} = c_{s_1} \). If all states in \( H(s_1), H(s_2), \ldots, H(s_n) \) are not distinct, say \( H(s_i) = H(s_j), i < j \), then all states between them will be short-circuited in the abstract path, since a traversal from \( H(s_{i-1}) \) to \( H(s_{j+1}) \) only needs 2 transitions. Thus, \( c_{s_1} > c_{s_1} \).

An important issue in using A*-like search is that the heuristic function needs to be monotonic w.r.t. the real cost. This requirement makes the algorithm work more efficiently and easier to implement. If the heuristic function is not monotonic, the search must revisit the closed states and this can result in exponential run-time. The lemma below guarantees SADBs are monotonic.

**Lemma 4.2** Let \( M \) and \( \hat{M} \) be two transition systems where \( \hat{M} = H(M) \) for a given abstraction \( H \). Let \( s_1, s_2 \) be two states from \( M \) and \( \hat{s}_1, \hat{s}_2 \) from \( \hat{M} \). Then \( c_{\hat{s}_1} \geq c_{\hat{s}_2} \) if \( c_{s_1} \geq c_{s_2} \), i.e. for any abstract state \( \hat{s} \), the cost \( c_{\hat{s}} \) is monotonic.

**Proof.** Let us assume there is one transition from \( s_1 \) to \( s_2 \) and \( c_{s_1} - c_{s_2} = 1 \). By the definition of the abstraction, \( \hat{s}_1 \) and \( \hat{s}_2 \) are either the same abstract state or two abstract states that have a transition from \( \hat{s}_1 \) to \( \hat{s}_2 \). Let \( \sigma = \{(B_0, 0), (B_1, 1), \ldots, (B_n, n)\} \) be the SADB induced by the abstraction \( H \). By definition we have \( B_0 \land B_1 \land \ldots \land B_n \equiv False \) for all \( 0 \leq i \leq n \). This essentially means \( \hat{s}_1 \) must appear one transition before \( \hat{s}_s \). Hence, \( c_{\hat{s}_1} - c_{\hat{s}_2} \geq 1 \) and then \( c_{\hat{s}_1} \geq c_{\hat{s}_2} \).
4.4 Abstraction-Directed Verification

In Figure 1.1 we depicted the essence of our approach. In this figure, the initial module transforms the model. In this chapter we are only interested in invariant properties, so we do not need to carry out a transformation. We show the simplification in Figure 4.3.

![Diagram of Abstraction-Directed Symbolic Model Checking](image)

Figure 4.3. Abstraction-directed symbolic model checking

The process starts with the design, which we refer to as the concrete model. In the first step we generate a (homomorphic) data abstraction of the concrete model. Note that in this step, we can generate more than one abstraction of the concrete model. We refer to a single abstraction as SA and multiple abstractions as MA in Figure 4.3. The abstract model(s) are taken as input by a symbolic model checker. If the model checker verifies the abstract model(s), we terminate as the data abstraction guarantees the soundness of the properties we are interested in. If the abstract model(s) fail the verification, we construct abstraction heuristic(s) using the abstract model(s). The directed model-checking algorithm is then invoked to check the concrete system using this heuristic as guide. The outcome of the heuristic model checker is either that the concrete model is verified, or a counter-example (CX in the figure) that will reveal the defect in the design (assuming the algorithm terminates of course). Note that unlike other research in model checking, we
use the same abstraction to (1) reduce the size of the model and (2) to guide the heuristic search algorithm.

In the rest of this chapter we illustrate directed model checking that is based on single and multiple abstraction databases.

4.5 Single Abstraction Database

4.5.1 The BFS algorithm

In computational tree logic (CTL), an invariant of a transition system can be expressed as $AG\varphi$. In symbolic model checking, we can use two methods, namely pre-image and image computation, to check the correctness of this class of properties. Given a set of states $F$ and transition relation $T$, pre-image and image are computed as follows:

$$\text{pre-image}(T, F) = \{ s \mid (r \in F) \land ((s, r) \in T) \}$$

and

$$\text{image}(T, F) = \{ s \mid (r \in F) \land ((r, s) \in T) \}.$$

The pre-image invariant checking is based on a greatest fixed point calculation algorithm [McM93, CGP99] as characterised by $AG\varphi = \nu Z.\varphi \land AXZ$, where $\nu Z$ is the greatest fixed point operator. In practice, this algorithm corresponds to a backward breadth-first search and may be inefficient because some states generated by the backward search may not be reachable from the initial state of the system (and hence need not have been computed).

Another method, image computation, is based on forward breadth-first search. This algorithm requires two inputs: the transition system $(S, T, S_0)$ and the error state $\overline{\varphi}$, where $\varphi$ is the invariant that holds in any state along all paths. For convenience, we use the transition system with a goal $G = \overline{\varphi}$ as an input, where $G \not\in S_0$. The algorithm is shown in Figure 4.4. In each iteration, a set of new reachable states $R_{new}$ and the difference $F$ (frontier) between $R_{old}$ and $R_{new}$ are computed. To check whether the invariant has been violated, $F$ is intersected with the error state in each iteration as well.

If the intersection is not empty, we terminate the algorithm and report the error. If the intersection remains empty and the set of reachable states does not change (a fixed point), then we can claim that invariant $\varphi$ is never violated in this model. Note that we test the existence of an error state in each iteration (on-the-fly), whereas some model checkers compute the entire set of reachable states and then intersect it with error states. For large transition systems, computing all reachable states may not be feasible as the BDD representing the reachable states is too large. In this work, we use an on-the-fly technique to test for error states.
Procedure InvarCheck \((S, T, S_0, G)\)

1. \(R_{old} \leftarrow \text{False}\)
2. \(R_{new} \leftarrow S_0\)
3. \(\text{while}(R_{old} \neq R_{new})\)
   4. \(F \leftarrow R_{new} \land \overline{R_{old}}\)
   5. \(R_{old} \leftarrow R_{new}\)
   6. \(F \leftarrow \text{Image}(T, F)\)
   7. \(\text{if } (F \land G \neq \text{False})\)
      \(\text{return ErrorFound}\)
   8. \(R_{new} \leftarrow R_{old} \lor F\)
9. \(\text{return NoErrorFound}\)

Figure 4.4. Symbolic invariant checking algorithm

4.5.2 The directed algorithm: an example

Before formally introducing our directed model checking algorithm, we illustrate the technique on the example transition system that we saw in Figure 4.1. We use a Boolean vector \(X = (x_0, x_1, x_2)\) to represent a state, and the invariant we are interested in is that all three Boolean variables cannot be true simultaneously in any state. This property can be expressed in CTL as \(\text{AG}(\overline{\alpha})\), where \(\alpha = \overline{x_0} \lor \overline{x_1} \lor \overline{x_2}\). Hence, our search goal (error state) is the complement of the property \(\alpha\). To construct the pattern database, we define a homomorphic abstraction \(H\) that abstract the third Boolean variable, \(x_2\), to a single-element domain. So the abstract system can be constructed as shown in Figure 4.1. We then apply standard model checking (the InvarCheck algorithm shown in Figure 4.4) to the abstract system. In this example, InvarCheck will report an error because there is a path leading from \(\hat{S}_0 = (0, 0)\) to \(\hat{G} = (1, 1)\) in the abstract system.

At this point, instead of refining the abstract system and checking it again, we use the abstract system to construct a symbolic pattern database as shown in Table 4.1 and Figure 4.2.

The essence of the directed algorithm is that each state (set of states) is associated with an estimated distance to the goal as well as the actual distance from the initial state. We use the symbolic pattern database constructed from the homomorphic abstraction to assign to each state an estimated value (Figure 4.1 and Table 4.1). We map a state in
the concrete system to an abstract state and look up its estimated value in the database. For example, for the state \( (0, 1, 1) \), the corresponding abstract state is \( (0, 1, x) \) and the estimated value is 3 (see Table 4.1). In a symbolic pattern database, each item is a BDD representing a set of abstract states, so the database look-up can be accomplished by calculating the conjunction of two BDDs. For example, to look for the estimated distance for state \( (1, 0, 0) \), we iteratively compute \( (x_0 \land \overline{x_1} \land \overline{x_2}) \land b_i \) for \( i \) from \( N \) to 0. If the resulting BDD is not constant \( False \), we assign \( i \) to \( (1, 0, 0) \) as the estimated distance. In this particular case, \( (x_0 \land \overline{x_1} \land \overline{x_2}) \land b_1 \) is not constant \( False \), so we assign the estimated value 1 to that state. Thus, the symbolic pattern database will partition a set of states according to their estimated value. Our invariant model checking algorithm is therefore directed by the symbolic pattern database in its search for an error in the concrete system. In Figure 4.5, we show the difference in the search tree with and without heuristic guiding. In this figure we label the states in the directed algorithm by a pair consisting of the number of actual and estimated transitions (resp.).

Figure 4.5. The search trees generated by the (a) **InvarCheck**, and (b) **Direc
dInvarCheck** algorithms for the example in Section 4.5.2
4.5.3 The directed invariant verification algorithm

The directed algorithm is shown in Figure 4.6. In contrast to the standard algorithm InvarCheck, the directed algorithm takes a homomorphic abstraction function $H$ as input in addition to the concrete transition system.

Procedure DirectedInvarCheck $((S, T, S_0, G), H)$
1. if (InvarCheck($(H(S), H(T), H(S_0), H(G))$) = NoErrorFound)
2. return NoErrorFound
3. $\sigma \leftarrow \text{construct}(H(S), H(T), H(S_0), H(G))$
4. SearchQueue $\leftarrow (0, 0, S_0)$
5. Closed $\leftarrow$ False
6. while (SearchQueue $\neq \emptyset$)
7. $(g, h, F) \leftarrow \text{SearchQueue.pop}()$
8. if $(F \land G \neq \text{False})$
9. return ErrorFound
10. Closed $\leftarrow$ Closed $\lor$ F
11. $F \leftarrow \text{Image}(T, F) \land \overline{\text{Closed}}$
12. QueueImage($\sigma, F, g + 1$)
13. return NoErrorFound

Figure 4.6. The directed invariant checking algorithm that uses an SADB.

In line 1, an abstract model is constructed using the abstraction function $H$ and standard InvarCheck is called to prove the invariant. If this succeeds, then the invariant is true in the concrete system (as discussed in Section 4.2). If this fails, the algorithm then constructs a symbolic pattern database (line 3) according to the abstract function $H$ (as discussed in Section 4.3). SearchQueue in line 4 is a priority queue used to determine which state(s) should be explored first. The element of the queue is a 3-tuple, $(g, h, S)$ where $g$ is the actual number of transitions to $S$ from the initial state, $h$ is estimated number of transitions to a goal (error) state and $S$ is a BDD representing a set of states. When determining which element should be dequeued for further exploration, SearchQueue considers $f = g + h$ as the priority key and pops the element with minimum $f$. In lines 5-13, the heuristic search algorithm $A^*$ [Pea84, NQ03] is adapted to symbolically explore the state space in the concrete model.
Procedure QueueImage (σ, Img, Cost)

1. \( n \leftarrow |\sigma| \)
2. while \( (n > 0) \)
3. \( I \leftarrow \text{Simplify}(B_n, \text{Img}) \)
4. if \( (I \neq 0) \)
5. \( \text{SearchQueue} \leftarrow (\text{Cost}, n, I) \)
6. \( \text{Img} \leftarrow \text{Img} \land I \)
7. if \( (\text{Img} = 0) \) return
8. \( n \leftarrow n - 1 \)
9. \( \text{SearchQueue} \leftarrow (\text{Cost}, \infty, I) \)

Figure 4.7. The partition subroutine for a single SADB

The difference between the directed algorithm and InvarCheck is that whenever the image of the frontier, \( F \), is computed, we employ the symbolic pattern database to partition this image and assign each sub-image with a heuristic evaluation before we push it back to the search queue. This is shown in procedure QueueImage in Figure 4.7. Given a set of states, \( \text{Img} \), this procedure iterates through every item \( b_i \) in \( P \), and checks whether there exists a subset \( I \) of \( \text{Img} \), such that \( H(G) \) can be reached from \( H(I) \) in the abstract system (line 3 of procedure QueueImage). Note that if the \( \text{Img} \) cannot be partitioned by \( P \), we simply push it back to the search queue with heuristic evaluation \( \infty \) (line 9).

We prove that the directed algorithm DirectedInvarCheck is both correct and optimal.

Theorem 4.3 (Correctness) Given a transition system \( M = (S, T, S_0, G) \) and a homomorphic abstraction \( H \), and let \( R_g \) and \( R_i \) be the indication returned from DirectedInvarCheck((\( S, T, S_0, G \), \( H \)) and InvarCheck(\( S, T, S_0, G \)) respectively. Then \( R_g \Leftrightarrow R_i \).

Proof. As InvarCheck and DirectedInvarCheck both use state space exploration, if DirectedInvarCheck detects an error, so will InvarCheck, and vice versa. If there is no error in the system, InvarCheck will explore all reachable states (the fixed point) and report NoErrorFound. In this case, we have to prove DirectedInvarCheck also explores
all reachable states. This is detected by the Closed set that stores all states explored by DirectedInvarCheck. When all reachable states have been explored, Image(T, F) \land \overline{\text{Closed}} \text{ (line 11) will be an empty set, so nothing is pushed into the search queue. Hence, the search queue will eventually become empty and NoErrorFound will be returned. Thus, in all cases the two algorithms will return the same result, i.e. } R_g \Leftrightarrow R_i.

An important outcome of the model checking technique is that it can provide the counter-examples (or witnesses in the case of existential properties), showing why the property is violated. A counter-example is sometimes also referred as an error trace. Generally, the shorter the length of the error trace, the easier it is for human beings to interpret it. Because InvarCheck corresponds to a forward breadth-first search, it will determine the minimum-length error trace. The following theorem ensures that DirectedInvarCheck detects the minimum error trace as well.

**Theorem 4.4 (Optimality)** Let \( M = (S, T, S_0, G) \) be a transition system and \( H \) be a homomorphic abstraction. Let \( \pi_i \) and \( \pi_g \) be the error traces detected by InvarCheck and DirectedInvarCheck respectively. Then \( L(\pi_g) = L(\pi_i) \).

**Proof.** The proof of the theorem can be established by proving that DirectedInvarCheck detects the shortest path from the initial state \( s_0 \) to a goal state \( s_g = G \). Note that the state space exploration algorithm in DirectedInvarCheck is adapted from the heuristic search algorithm \( A^* \). If the lower bound heuristic is used, the algorithm guarantees the path is shortest (minimum cost) [Pea84, NQ03]. So we need to prove the symbolic pattern database heuristic is a lower bound. According to lemma 4.1, for any path \( \pi = s_0, s_1, \ldots, s_n \) in the concrete system and its corresponding path \( \hat{\pi} = \hat{s}_0, \hat{s}_1, \ldots, \hat{s}_n \) in the abstract system, \( c_{s_0} \geq c_{\hat{s}_0} \). Thus the symbolic pattern database heuristic is a lower bound and \( L(\pi_g) = L(\pi_i) \).

4.5.4 Experimental evaluation

To determine the effectiveness of guiding, we have implemented our algorithm in our tool, GOLFER, which will described in detail in Chapter 8. For the purpose of comparison, we use the original invariant model checking algorithm in NuSMV to test for the existence of an error, as shown in Figure 4.4. Experiments are carried out on an Intel 933Hz machine, with 512MB RAM and running the Linux operating system.
In this work, we did not use any particular variable ordering. The abstraction method we use is to make some Boolean variables invisible. Note that because our approach does not involve any refinements, we require that the abstraction not be too coarse. In general, the criteria to select an abstraction granularity is that it should be feasible to construct the symbolic pattern database in a small amount of time with maximum depth. The relation between the granularity of the abstraction and the accuracy of the resulting heuristic has been studied by Prieditis and Davis in 1995 [PD95]. In this work, we set the threshold for the construction to be 60 seconds. If the construction cannot finish within 60 seconds, we have to abandon the abstraction and choose another more abstract system.

**Experimental results**

The two benchmark circuits we use in the experiment were published in David L. Dill’s thesis [Dil88]. Since we focus on error detection, we use two “buggy” designs for our evaluation. The first circuit family is called a tree arbiter circuit, which is used to enforce mutual exclusion among users accessing shared resources. The basic element of this circuit family is the arbiter cell, which implements the mutual exclusion between two users. For more than two users, cells can be connected to form a tree-like structure. The bottom level cells are connected to users and the top cell to the shared resource. The request signal from a user propagates upwards from the bottom level of the tree, and if the shared resource is granted to the user, the acknowledgment propagates downwards only to the user who requested it. The second circuit family, a distributed mutual exclusion ring (DME), is used to implement mutual exclusion as well. Instead of forming tree-like structures, DME cells are connect as a ring. Mutual exclusion is implemented by passing a token in the cell ring. In this work, we construct interleaving finite models for these circuits and check the invariant that no two users receive an acknowledgement simultaneously. The results of our experiment are shown in Table 4.2.

Note that all error traces detected by our method have exactly the same length as detected by standard InvarCheck in NuSMV. For each circuit, we experimented with a few invariants and only report those errors with depth more than 20, because short error traces can be easily detected by both algorithms regardless of model size. We also report the number of BDD variables used to encode each model to reflect the size of the system. The memory use of the two algorithms is reflected in the total nodes allocated by the BDD engine that is contained in NuSMV. The hyphens in the table are for those experiments that did not terminate within (a randomly chosen) 6 hours.

For the tree arbiter circuits, DirectedInvarCheck can easily handle up to 19 cells, whereas InvarCheck cannot handle more than 15 cells in less than 6 hours. Note that
Table 4.2. Experimental Results for the tree arbiter and DME

<table>
<thead>
<tr>
<th>Circuits</th>
<th>Depth</th>
<th># vars</th>
<th># nodes</th>
<th>CPU time (s)</th>
<th># nodes</th>
<th>CPU time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>tarb7</td>
<td>20</td>
<td>50</td>
<td>322,452</td>
<td>1.340</td>
<td>47,778</td>
<td>2.357</td>
</tr>
<tr>
<td>tarb9</td>
<td>20</td>
<td>64</td>
<td>483,286</td>
<td>2.300</td>
<td>634,783</td>
<td>3.120</td>
</tr>
<tr>
<td>tarb11</td>
<td>21</td>
<td>78</td>
<td>1,249,438</td>
<td>6.360</td>
<td>560,439</td>
<td>2.440</td>
</tr>
<tr>
<td>tarb13</td>
<td>21</td>
<td>92</td>
<td>5,593,886</td>
<td>13.590</td>
<td>450,156</td>
<td>2.520</td>
</tr>
<tr>
<td>tarb15</td>
<td>24</td>
<td>106</td>
<td>161,297,839</td>
<td>4759.000</td>
<td>4,262,998</td>
<td>19.290</td>
</tr>
<tr>
<td>tarb17</td>
<td>24</td>
<td>120</td>
<td>—</td>
<td>&gt; 6 hours</td>
<td>7,323,386</td>
<td>35.260</td>
</tr>
<tr>
<td>tarb19</td>
<td>24</td>
<td>134</td>
<td>—</td>
<td>&gt; 6 hours</td>
<td>7,922,396</td>
<td>34.930</td>
</tr>
<tr>
<td>dme06</td>
<td>26</td>
<td>114</td>
<td>—</td>
<td>&gt; 6 hours</td>
<td>3,316,858</td>
<td>18.240</td>
</tr>
<tr>
<td>dme08</td>
<td>30</td>
<td>152</td>
<td>—</td>
<td>&gt; 6 hours</td>
<td>93,794,232</td>
<td>1137.000</td>
</tr>
</tbody>
</table>

there is a time-line cross-over in the table between 9 and 11 cells. For smaller circuits, InvarCheck can detect errors faster than DirectedInvarCheck. For larger circuits, the performance of InvarCheck deteriorates rapidly, while the time taken by DirectedInvarCheck remains relatively constant. This occurs because in systems with a low level of concurrency, BDD-based breadth-first search is more efficient than directed search. As described in Section 4.3, DirectedInvarCheck also needs to partition the image (BDD slicing) and this introduces some overhead as well. For systems with a high level of concurrency, BDD-based breadth-first search is dominated by huge BDD computations, whereas directed search partitions large BDDs into smaller ones, and only explores promising states, thereby avoiding the exponential growth in the size of the BDDs.

We only experimented with 2 of the circuits in the DME family as InvarCheck could not cope with circuits that had more than 6 cells in the available time. DirectedInvarCheck could however handle up to 8 cells with an error depth of 30. For larger circuits, we need to resort to manipulating the variable ordering to improve the performance.

The experimental results indicate that the directed approach can outperform standard model checking by several orders of magnitude, in both time and required memory. As expected, DirectedInvarCheck not only detects the errors much quicker than InvarCheck, but also found the shortest error traces (which has the same length as returned by InvarCheck).
4.6 Multiple Abstraction Databases

Incorporating multiple abstractions in our directed verification framework involves a simple substitution of the subroutine \texttt{QueueImage} in Figure 4.7. The main routine \texttt{DirectedInvarCheck} remains the same as in Figure 4.6. In essence, \texttt{QueueImage} partitions each BDD, representing a set of states, into several (usually smaller) BDDs, where each of them corresponds to a different heuristic value. \texttt{QueueImage} uses a single SADB to partition the BDD that passed by the main routine. When using multiple SADBs, the partition subroutine is similar to \texttt{QueueImage} except that it must handle the multiple heuristic values for the same concrete state. This is because a concrete state has multiple counterparts in every abstraction. A query from each SADB returns a heuristic value for the concrete state. We use 2 strategies to handle this problem. The partition algorithm that uses multiple abstractions is depicted in Figure 4.8.

Procedure \texttt{QueueImageMult} ($\Phi, Img, Cost, m, \text{strategy}$)

\begin{verbatim}
1   result $\leftarrow \{(Img, 0)\}$
2   for $i$ in 1..m do
3       temp $\leftarrow$ result
4       result $\leftarrow \{\}$
5       for each $(d, h) \in$ temp do
6           for each $(B_j, j) \in \sigma_i$ do
7               $I \leftarrow$ Simplify($B_j, d$)
8               $d \leftarrow d \land T$
9           if ($I \neq \phi$ & $m, \text{strategy} = \text{add}$)
10              result $\leftarrow$ result $\cup \{(I, j + h)\}$
11           if ($I \neq \phi$ & $m, \text{strategy} = \text{max}$)
12              result $\leftarrow$ result $\cup \{(I, \text{max}(j, h))\}$
13           if ($d \neq \phi$)
14              result $\leftarrow$ result $\cup \{(d, |\Phi|)\}$
15       for each element $(D, k)$ in result do
16           SearchQueue $\leftarrow$ (Cost, k, D)
\end{verbatim}

Figure 4.8. The partition subroutine for multiple SADBs
is a BDD representing a set of states of \( M \) and \( \Phi = \{\sigma_1, \ldots, \sigma_m\} \) is a set of SADBs. The algorithm below splits the BDD \( \text{Img} \) and assigns each sub-BDD an estimator according to the merge strategy of the SADBs. In lines 15 – 16, all partitions are pushed back to the search queue along with a real cost and corresponding heuristic estimations. The input \( m_{\text{strategy}} \) is the merge strategy used on the SADBs in the symbolic heuristic search. The only possible values are \textit{add} and \textit{max}.

We now formally define the notion of disjoint abstraction databases.

**Definition 4.6 (Disjoint SADBs)** Two symbolic abstraction databases \( \sigma_1 \) and \( \sigma_2 \) are disjoint if their corresponding pattern sets \( X_{p1} \) and \( X_{p2} \) are disjoint, i.e. \( X_{p1} \cap X_{p2} = \phi \).

Given a state \( s \) and characteristic function \( \mathcal{F_s} \) in the concrete model and a symbolic abstraction database \( \sigma = \{(B_0, 0), (B_1, 1), \ldots, (B_n, n)\} \), if \( \mathcal{F_s} \land B_j \neq \text{False} \) for some pair \((B_j, j)\), then we call the value \( j \) an estimator and denote it as \( \sigma(s) \). Given a symbolic abstraction database, this estimator is unique as all \( B_i \) are disjoint.

**Theorem 4.5 (Additiveness)** Let \( L(\pi_s) \) be the path length of \( s \) in \( M \), and \( \sigma_1 \) and \( \sigma_2 \) be two disjoint symbolic abstraction databases. Let \( \sigma_1(s) \) and \( \sigma_2(s) \) be the estimator of \( s \) in each of them. Then \( \sigma_1(s) + \sigma_2(s) \leq L(\pi_s) \).

**Proof.** Suppose a solution path for \( s \) in the concrete model is \( \pi = s_0s_1s_3\ldots s_k \) where \( s_0 = s \) and \( s_k \in G \). Let \( S_0S_1\ldots S_l \) be the abstract path induced by \( \sigma_1 \) and \( P_0P_1\ldots P_m \) be the abstract path induced by \( \sigma_2 \). Let \( \gamma \) be the concretization function that maps each abstract state to a set of concrete states. Since \( \sigma_1 \) and \( \sigma_2 \) are disjoint, \( \gamma(S_i) \nsubseteq \gamma(P_j) \) for all \( i \leq l \) and \( j \leq m \). This means the 2 abstractions actually abstracts the different parts of the concrete path. In other words, the total number of transitions for both abstractions will be less than the number of transitions of the original concrete path. Hence, \( \sigma_1(s) + \sigma_2(s) \leq L(\pi_s) \). □

This theorem guarantees that disjoint SADBs can be “added” together and the result will still be an admissible heuristic. In this work we in fact also consider pattern sets that are not disjoint. In verification, admissibility is less of an issue as we are more interested in finding just a good path to a defect in the system, not necessarily the shortest path. Our formal discussion about the disjoint SADBs actually shares many aspects with the disjoint pattern database heuristics in AI research [Ede02, KF02, FKH04].

In the next section, we conduct experiments in heuristic- and BDD-based symbolic search in model checking for models from various domains. We seek to understand the
effect that symbolic state enumeration has on heuristic search, particularly with respect to each of the domains. We do this for both single-heuristic and multiple-heuristic search strategies.

4.6.1 Experiment set-up

In explicit-state heuristic search, the number of states (or nodes) that are generated by the search algorithm can be used to evaluate the effectiveness of the heuristic. In BDD-based heuristic search, however, we cannot use the number of states that are generated by the algorithm as states are symbolically represented by Boolean functions. We note that the number of nodes in a BDD is not related to the number of states it represents. In fact, the effort that a symbolic search algorithm must make to solve a problem is largely determined by the internal operations of the BDD engine, not the number of states in the system.

The following attributes will be used to capture the search effort in our experiments.

**IM** The number of BDD image computations. These computations determine the successor states of the search. It is called the relational product and involves quantifier elimination, which is an expensive computation in symbolic model checking. (IM is related to the size of the closed set in the explicit-state A* algorithm.)

**SP** The number of splitting operations. The splitting operation involves the ‘restrict’ operation on BDDs. It is also expensive. (SP is related to the size of the open set in explicit-state A* algorithm.)

**ND** The total number of BDD nodes allocated. While the number of BDD nodes is not directly related to the number of states, it still reflects the memory usage of the algorithm and hence is the major memory measurement.

**AS** The average size of all BDDs. Reducing the size of BDDs is important because some BDD operations have exponential time complexity in terms of the BDD size.

**TM** The CPU time consumed by the search algorithm. In general the CPU time is strongly related to the values of IM and SP.

We have used 5 models in our experiments: two of them are puzzles from the AI heuristic search domain, and the other 3 are real-world design models of concurrent systems. Note that all models have at least one goal state that is reachable from the initial state set.
To construct SADBs we need to define the pattern set that is to be used for the abstraction. For each puzzle model we use 4 different pattern sets that are commonly used in AI heuristic search literature. For each of the verification models, we use a data dependency analysis to also generate 4 pattern sets. The pattern sets for each model are not necessarily disjoint as the optimality of the solution path is not a primary concern in our work.

We ran the model checker for each of the (single) SADBs generated by the pattern sets. We also used multiple SADBs that were constructed by merging 3 of the 4 SADBs for each model. We in fact constructed multiple SADBs by all of the $C_4^3$ combinations of SADBs and reported the best performance. As well, both the add and max merge strategies were studied.

**Caveat:** At this point, we should point out that symbolic search algorithms are not really suitable for solving the puzzle-like problems that often occur in AI heuristic search, where good heuristics are known. The advantage of the BDDs in manipulating sets of states in single operations is often outweighed by the computational complexity of these BDD operations [NQ03]. In essence, only when the BDDs represent large sets of states does their use pay off. In general, for these types of problems, explicit-state searches are often faster and require less memory. We use these models in this work, however, for comparison purposes.

### 4.6.2 Heuristic distribution experiment

The aim of this experiment was to study the distribution of the heuristic over the state space. In Figure 4.9, we show the number of states in the SADB for different heuristic values. In our symbolic approach, all states with the same heuristic value is represented by a single BDD. To compute these results, we needed to calculate how many (abstract) states a BDD can represent (which is of course different to the number of nodes in the BDD). Note the logarithm scale on the axis for the number of states. The diagram on the left is taken from randomly chosen SADBs for the 8-puzzle model, and on the right, for the DME circuit model.
In the figure we observe that different abstractions show similar behavior for each model. Comparing AI and verification, however, we observe that the number of states increases as the heuristic value increases in the case of the 8-puzzle, but decreases for the DME model.

We conjecture that this phenomenon is caused by fundamental differences in the nature of the state spaces in these domains. Puzzles, for example, often have few goal states, whereas safety properties in verification can be violated in many states, and hence there are many goal states. Having many goal states means that the value of the heuristic tends to remain small. When we construct SADBs, we use a backward breath-first traversal of the abstract model’s state space. In the case of puzzles, the resulting search tree grows exponentially. In verification, the size of the search tree decreases as many states share the same predecessor. A conclusion one could draw from the behavior we observe is that heuristic search using SADBs has less to gain in verification models than in AI models (in other words, the improvement over blind search will be less in verification).

### 4.6.3 Mean heuristic value experiment

In this experiment we wished to study whether or not the mean heuristic value [KRE01, Ede02], is a good predictor of the effort needed by the symbolic heuristic search. In explicit-state heuristic search, it is well known to be a good predictor of the search effort. The intuition is that it may not be in a symbolic setting because it does not factor in the added computational overhead that BDDs have.

The mean heuristic value (MHV) \( \bar{h} \) of a SADB \( \sigma \) determines the overall distribution
of heuristic values and is defined as follows.

\[
\bar{h} = \frac{1}{|\sigma|} \sum_{i=1}^{|\sigma|} i \times (|\{s \in \sigma | \sigma(s) = i\}|/|\{s \in \sigma\}|)
\]  

(4.1)

Equation 4.1 actually computes the weighted mean heuristic value of \(\sigma\). A high value for \(\bar{h}\) indicates that a larger proportion of states have large heuristic values than have low values, and therefore the level of “informedness” is high. Ideally, the value of \(\bar{h}\) for a SADB \(\sigma\) should be as close as possible to \(|\sigma|\), and in that case it is said to be “well-informed”.

We chose three models, and for each model we use 4 different SADBs. The results are shown below.

<table>
<thead>
<tr>
<th>SADBs</th>
<th>MHV</th>
<th>IM</th>
<th>SP</th>
<th>ND</th>
<th>AS</th>
<th>TM</th>
</tr>
</thead>
<tbody>
<tr>
<td>puzz-1</td>
<td>11.90</td>
<td>144</td>
<td>305</td>
<td>268421</td>
<td>361</td>
<td>1.050</td>
</tr>
<tr>
<td>puzz-2</td>
<td>12.30</td>
<td>679</td>
<td>1581</td>
<td>547046</td>
<td>404</td>
<td>3.390</td>
</tr>
<tr>
<td>puzz-3</td>
<td>9.03</td>
<td>495</td>
<td>1043</td>
<td>453495</td>
<td>455</td>
<td>2.970</td>
</tr>
<tr>
<td>puzz-4</td>
<td>8.30</td>
<td>1353</td>
<td>2769</td>
<td>400189</td>
<td>429</td>
<td>6.900</td>
</tr>
<tr>
<td>dme-1</td>
<td>2.73</td>
<td>616</td>
<td>1121</td>
<td>193004</td>
<td>2921</td>
<td>74.110</td>
</tr>
<tr>
<td>dme-2</td>
<td><strong>3.72</strong></td>
<td><strong>26</strong></td>
<td><strong>26</strong></td>
<td><strong>280683</strong></td>
<td><strong>3232</strong></td>
<td><strong>3.220</strong></td>
</tr>
<tr>
<td>dme-3</td>
<td>3.68</td>
<td>26</td>
<td>26</td>
<td>289808</td>
<td>6268</td>
<td>11.090</td>
</tr>
<tr>
<td>dme-4</td>
<td>3.57</td>
<td>26</td>
<td>26</td>
<td>524103</td>
<td>12394</td>
<td>45.540</td>
</tr>
<tr>
<td>ns-1</td>
<td>8.13</td>
<td>28</td>
<td>106</td>
<td>92126</td>
<td>815</td>
<td>6.840</td>
</tr>
<tr>
<td>ns-2</td>
<td>5.37</td>
<td>421</td>
<td>1693</td>
<td>371040</td>
<td>964</td>
<td>122.420</td>
</tr>
<tr>
<td>ns-3</td>
<td>6.16</td>
<td>1110</td>
<td>6361</td>
<td>629380</td>
<td>1843</td>
<td>389.670</td>
</tr>
<tr>
<td>ns-4</td>
<td>5.34</td>
<td>475</td>
<td>2073</td>
<td>519868</td>
<td>1874</td>
<td>228.980</td>
</tr>
</tbody>
</table>

The statistics that concern the BDDs come from the BDD engine of the model checker. Other statistics are generated from profilers. For each of the three models we give the results for each of the 4 SADBs. The value of MHV is calculated using equation 4.1. For each model, we bold the row that has the shortest run-time (TM).

For the puzzle model, the bolded row has the best performance in terms of every attribute. While the MHV for puzz-1 is not the highest one, the MHV is nevertheless a good predictor of performance.

For the two verification models, the SADB with the shortest run-time has the highest value of MHV. It is hence a very good predictor in the symbolic setting, contrary to our intuition.

Unrelated to the MHV, we observe that dme-2, dme-3 and dme-4 use exactly the same number of image computations and splitting operations (IM and SP). However, in spite of the fact that dme-2 uses more BDD nodes (ND) than dme-1, and will hence use more memory, it is still the best performer.
4.6.4 Multiple SADB experiment

This experiment concerns the main focus of this work, and that is compare the performance of single and multiple SADBs. Holte et al. [HNF+04] found that heuristic search that is based on multiple SADBs out-performs search based on single SADBs (for the same amount of memory). But Holte et al’s work is based on an explicit-state search. So does it apply to a symbolic heuristic search as well?

We run each model with each of the 4 single SADBs. From these 4 SADBs, we created 10 multiple SADBs \( C_2^4 \) from combinations of 2 SADBs plus \( C_3^4 \) from 3 SADBs. We ran the symbolic search on each model with each of these 10 heuristics. We show the results for just two of the models in the following table.

<table>
<thead>
<tr>
<th>SADB(s)</th>
<th>IM</th>
<th>SP</th>
<th>ND</th>
<th>AD</th>
<th>TM</th>
</tr>
</thead>
<tbody>
<tr>
<td>puzz-sgl</td>
<td>144</td>
<td>305</td>
<td>268421</td>
<td>361</td>
<td>1.050</td>
</tr>
<tr>
<td>puzz-mpl-1</td>
<td>133</td>
<td>284</td>
<td>353075</td>
<td>341</td>
<td>1.360</td>
</tr>
<tr>
<td>puzz-mpl-2</td>
<td>203</td>
<td>440</td>
<td>409425</td>
<td>343</td>
<td>1.650</td>
</tr>
<tr>
<td>puzz-mpl-3</td>
<td>160</td>
<td>349</td>
<td>299479</td>
<td>361</td>
<td>1.200</td>
</tr>
<tr>
<td>peter-sgl</td>
<td>40</td>
<td>97</td>
<td>109795</td>
<td>1332</td>
<td>4.414</td>
</tr>
<tr>
<td>peter-mpl-1</td>
<td>230</td>
<td>591</td>
<td>379365</td>
<td>816</td>
<td>14.273</td>
</tr>
<tr>
<td>peter-mpl-2</td>
<td>40</td>
<td>105</td>
<td>134202</td>
<td>822</td>
<td>16.442</td>
</tr>
<tr>
<td>peter-mpl-3</td>
<td>40</td>
<td>100</td>
<td>126754</td>
<td>716</td>
<td>16.357</td>
</tr>
</tbody>
</table>

In the table, the results for the single SADB (denoted by the .sgl suffix) are the ones that had the best performance. For the 10 multiple SADBs, we show just the best performing 3, and these are indicated by the .mpl suffix. We used both the add and max strategies to merge the SADBs.

Contrary to Holte et al’s findings, the results in the table above show that multiple heuristics perform worse than single heuristics. The first two rows in the table, puzz-sgl and puzz-mpl-1, are particularly interesting as they reveal that even when the multiple SADB search uses less image computations and splitting operations (which we noted earlier are the primary determinants of the computational complexity), it performs worse than the single SADB search. We note the multiple SADB models use more BDDs nodes, and the average size of the BDDs is smaller, in both models.

These results are quite surprising. Multiple pattern databases in explicit-state heuristic search are effective because they improve the overall heuristic distribution and hence result in smaller search trees. In symbolic heuristic search, the heuristic is (also) used to split the frontier BDDs, and one conjectures, it is this computation that causes the problem. Thus, much of the effort of symbolic heuristic search is spent on splitting the BDDs, offsetting any gains that may be had from the higher-quality, multiple heuristic.
4.6.5 Merge strategy experiment

In our final experiment we compared the performance of the add and max merge strategies for multiple SADBs. We carried out this experiment by using both strategies to merge both 3 and 4 single SADBs into a multiple SADB. We note that the add strategy is not admissible so it can generate non-optimal paths.

Unlike our earlier experiments, this time we present tables for each of the problem domains separately. The domains are AI puzzles, electronic circuits and communication/security protocols. We do this because we found that the choice of merge strategy effected the performance in a different way for each of these domains. The only change in the table format to our earlier experiments is the addition of the attribute LE, which indicates the solution length returned by the search. You can see from this column when a search generated a non-optimal path.

**AI puzzles**

In the table below we see the results for the AI puzzles.

<table>
<thead>
<tr>
<th>SADBs</th>
<th>IM</th>
<th>SP</th>
<th>ND</th>
<th>TM</th>
<th>LE</th>
</tr>
</thead>
<tbody>
<tr>
<td>puz-3-add</td>
<td>133</td>
<td>284</td>
<td>368552</td>
<td>1.450</td>
<td>24</td>
</tr>
<tr>
<td>puz-3-max</td>
<td>203</td>
<td>440</td>
<td>409425</td>
<td>1.650</td>
<td>24</td>
</tr>
<tr>
<td>puz-4-add</td>
<td>168</td>
<td>357</td>
<td>353075</td>
<td>1.360</td>
<td>24</td>
</tr>
<tr>
<td>puz-4-max</td>
<td>169</td>
<td>391</td>
<td>433784</td>
<td>1.880</td>
<td>24</td>
</tr>
<tr>
<td>perm-3-add</td>
<td>27</td>
<td>106</td>
<td>93224</td>
<td>5.720</td>
<td>13</td>
</tr>
<tr>
<td>perm-3-max</td>
<td>769</td>
<td>3446</td>
<td>1199309</td>
<td>294.340</td>
<td>11</td>
</tr>
<tr>
<td>perm-4-add</td>
<td>289</td>
<td>1833</td>
<td>237584</td>
<td>39.510</td>
<td>15</td>
</tr>
<tr>
<td>perm-4-max</td>
<td>2619</td>
<td>14427</td>
<td>859840</td>
<td>511.550</td>
<td>11</td>
</tr>
</tbody>
</table>

We observe that, while the add merge strategy can result in a non-optimal path, the resulting search is faster than that produced by the (optimal) max merge strategy. In fact, in the case of perm, it is one or two orders of magnitude faster. Note that there is almost the same difference between the max and add strategies in the number of image computations and partition operations, so the result is not surprising. There is a trade-off here: speed comes at the cost of optimality.

**Electronic circuit**

This model has been constructed from a real electronic circuit design and has been a widely used benchmark for symbolic model checking.
We observe that the *add* strategy clearly results in a faster model checker than *max* but at the cost of a much longer path to a goal state.

### Communication/security protocols

The two communication protocol models generate quite different results.

<table>
<thead>
<tr>
<th>SADBs</th>
<th>IM</th>
<th>SP</th>
<th>ND</th>
<th>TM</th>
<th>LE</th>
</tr>
</thead>
<tbody>
<tr>
<td>dme-3-add</td>
<td>169</td>
<td>232</td>
<td>142574</td>
<td>10.520</td>
<td>37</td>
</tr>
<tr>
<td>dme-3-max</td>
<td>616</td>
<td>1122</td>
<td>186500</td>
<td>77.380</td>
<td>27</td>
</tr>
<tr>
<td>dme-4-add</td>
<td>169</td>
<td>232</td>
<td>171070</td>
<td>10.260</td>
<td>41</td>
</tr>
<tr>
<td>dme-4-max</td>
<td>326</td>
<td>546</td>
<td>198282</td>
<td>16.220</td>
<td>27</td>
</tr>
</tbody>
</table>

Quite the opposite of the previous results, the *add* strategy for these models results in a model checker that takes a lot longer to find a longer, non-optimal path to a goal state. Clearly an unsatisfactory heuristic for this class of model.

In summary, the inadmissibility of the *add* merge strategy may lead to (very) sub-optimal paths, and a substantial speed-up in the search in some models, but a worsening in others.

### 4.6.6 Evaluation

Predicting how and when BDD-based heuristic search algorithms will perform better than explicit-state algorithms is extremely difficult. It is well known, for example, that finding an optimal variable ordering for BDDs is an NP-hard problem [BW96]. We have not considered the variable ordering in this work yet (but have in earlier work [QNS05a]). BDDs can be ‘exponentially’ efficient in representing very large sets of states, and because of this, can be vastly superior to explicit-state search algorithms. However, when the sizes of the sets they represent are not large, the computational overhead of manipulating
BDDs can result in very poor performance indeed. The problem of predicting performance is compounded when you add heuristics, and compounded again when you allow multiple heuristics. So the problem we are addressing is indeed very difficult.

In AI, finding the shortest path to the goal state is often paramount. In verification, finding a ‘reasonably short’ path is usually sufficient. More important is the time it takes to find this path. The reason for this is that the model checker is being used as a debugger, and hence we need to know quickly whether there is an error in the specification or not. In verification therefore, we are often prepared to sacrifice optimality for speed.

While we have tried to be comprehensive in the experiments, we do of course:

- have only a small sample of models,
- have just a few abstractions (derived automatically for the verification models)
- have just 2 merge strategies: one admissible, one non-admissible.

On the positive side, we have attempted to bridge disparate fields, AI and verification, by understanding the behavior of a technology, symbolic heuristic search, that is common to both. We can summaries the results of our experiments in the following way:

- The distribution of the heuristic over the state space is different for AI models than verification models. This difference could be caused by different characteristics of the state space: for example, there are typically more goal states in verification than in AI problems, and verification state spaces are less tree-like.
- The MHZ still makes a good predictor of effort in symbolic heuristic search.
- Contrary to Hole et al. [HNF+04], we found that multiple symbolic heuristics performed worse than single symbolic heuristics. We conjecture that this is caused by the overhead of splitting the BDDs. Note that in some cases splitting a BDD results in larger BDDs than the original. This is an unfortunate side-effect of this method that cannot easily be avoided or predicted.
- If you have a naturally good heuristic distribution, as AI problems tend to have, then an ‘aggressive’, non-optimal merge strategy like add will result in multiple SADBs that perform much better than single SADBs; albeit at the possible cost of optimality.
- Verification problems that have poorly, or narrowly distributed heuristics should not use non-optimal merge strategies.
AI puzzles and electronic circuits typically have very dense state spaces, while protocol models have relatively sparse state spaces. Intuitively, dense state spaces will contain a larger number of solution paths than sparse state spaces. This could be the cause of the behavior we observe in the merge-strategy experiment. A non-optimal strategy like add enables heuristic search algorithms like A* and IDA* to guide aggressively during the search because it increases the proportion of states that have larger heuristic values, and penetrates deeply into the state space. Consequently, however, the search may miss shallow solutions and fruitlessly pursue dead-end paths. Note that AI puzzles and circuits have relatively fewer goal states than protocol models. We conjectured in Section 4.6.2 that this was the cause of the behavior that we observed in the heuristic-distribution experiment. The topology of the state space is therefore potentially very important in determining the performance of the symbolic search.

4.7 Summary

In this chapter we have presented a symbolic model checking algorithm that combines homomorphic abstraction and directed search techniques. We introduce a mechanism called symbolic abstraction databases to provide a heuristic to guide the model checker. The abstraction databases represent the relaxed system and associate each state of the system with a heuristic value (i.e., the estimated number of transitions to an error state). This is required by the underlying heuristic search algorithm to partition the states and guide the search. The directed search is of course only used when an error is detected in the abstract system. In essence we double-dip on the abstraction: the abstraction reduces the size of the state space directly, but also indirectly as a result of the directed search.

The single abstraction database is quite effective in reducing the search space when there is an error in the model. This translates to fewer BDD nodes during the verification and shorter CPU time for most models on which we experimented. Experiments of multiple abstraction databases confirmed a simple criteria, the mean heuristic value, for choosing an effective abstraction. Unlike benchmarks in AI domain, using multiple abstraction databases in system designs reveals no significant improvement over single abstraction database.

It is important to note that directed model checking algorithms like ours are designed for debugging, and not proof of correctness. For systems that have no errors, the directed approach does not have any conceptual advantage over conventional model checking algorithms. However, the directed method does slice large BDDs, which reduces the sizes of the BDDs substantially. Although abstraction databases are memory-based heuristics,
the use of BDDs to represent them also helps to counter any potential size problem, and make a seamless integration with symbolic model checking possible.

The abstractions used in this chapter are manually determined by the user. While experienced users such as professional designers may not have a problem with crafting abstractions by hand, novice users certainly require a push-button technology. To generate an abstraction we need to extract domain information from the model. We call this process ‘synthesis’, and this is the subject of the next chapter.
Chapter 5

Heuristic Synthesis

One of the major barrier for formal methods to be accepted by practitioners is that it needs heavy expertise to use the tool. Our directed model checking improves the bug-hunting capability of a model checker, but introduces extra work for a user to derive abstractions. To alleviate user from this burden, we introduce a synthesis process in this chapter to automatically derive an abstraction (and hence the heuristic) from a concrete model. As well, we describe another guided algorithm that has more depth-first search nature.

This chapter is based on the work [QNS05a] presented in the conference on Formal Techniques for Networked and Distributed Systems (FORTE) in 2005.

5.1 Introduction

Model checking [CGP99] is often used in preference to theorem proving for the verification of properties in finite-state systems because of its high level of automation as well as its ability to produce counter-examples when a given property is found not to hold. The safety properties of a system can often be captured by one or more system invariants that characterize the set of states within which the system must reside. This process of checking invariants is also called a reachability analysis. The aim of a reachability analysis is to detect error states, where the paths leading to these states determine counter-examples to the invariant. Counter-examples provide valuable information to system developers about potential design errors in a system. In this work we are more focused on falsification of invariants than verification.
Because a BDD-based, depth-first search (BDD-DFS) strategy is not naturally layered like BFS, it requires a special mechanism to partition each BDD frontier during the search. The integration of BDD-DFS with heuristic search provides this mechanism: the heuristic values of states are not only used to estimate the distances to the goal, they are also used to partition the frontier into sub-frontiers. Each sub-frontier, represented by a single BDD, is treated by BDD-DFS as a single node in the search graph.

Our integration of heuristics and symbolic data is yet another development in the growing field of guided model checking [ELL01, QN04b, San03, SJL04] whose aim is to apply ‘smart’ technology to model verification. In Chapter 4, we integrated the $A^*$ search algorithm into a symbolic model checker. In this paper, we instead use a ‘more efficient’ version of $A^*$ called IDA$^*$ (iterative deepening $A^*$), designed to minimize memory usage. IDA$^*$ is in essence a mixture of BFS and DFS. If the heuristic is informative, the search is more like DFS, otherwise, with poor direction, the search works on a broad front. The tendency to mimic BFS when poorly informed means that $A^*$ can have exponential memory requirements. In 1985, Korf [Kor85] devised IDA$^*$ that is basically DFS, but has some BFS characteristics. He found this algorithm was often better than $A^*$ in solving hard AI problems [Kor93].

The new, integrated algorithm we develop is called BDD-IDA$^*$. The advantages of using BDD-IDA$^*$ are:

1. The iterative and bounded DFS strategy in $A^*$ detects so-called shallow and corner bugs that are difficult to detect by unbounded DFS.

2. In BFS, the frontier is layered. In BDD-IDA$^*$, the frontier is typically ‘onion shaped’ because of the action of the heuristic (biasing the search towards a particular path that leads to a goal state). Pruning of the search space in fact occurs before the bound is reached, so the frontier is more pointed to the goal for each iteration.

3. BDD-IDA$^*$ has the same linear (in the search depth) space complexity as DFS. (But note, as we use BDDs to represent sets of states, the actual space requirement can be exponential in the number of states.)

A second important feature of our work is that we have extended the idea presented in Chapter 4 of using an abstract version of the concrete model as a heuristic (the so-called pattern database). In that work we did not address the problem of how to obtain the abstract model. In this work, we address this issue and present an automatic method to generate abstract models that is based on a data-dependency analysis of the concrete model specification. In this analysis, we determine the strength of each variable. This information
is used by the heuristic generator to eliminate those variables that are considered less relevant (or weak), and thereby reduce the size of the model. We refer to this technique as *heuristic synthesis*. Being able to automatically determine a heuristic frees the system designer/verifier from this task, and makes the guided model checker fully automatic.

In summary, a number of model-checking and artificial intelligence approaches have been combined to produce an integrated, fully automatic framework that allows more efficient property falsification than alternative approaches.

### 5.2 DFS-based Heuristic Search

This work is based on the symbolic model checking approach [McM93] in which symbolic data structures called binary decision diagrams (BDDs) [Bry86] are used to represent a finite-state space. Invariant checking in symbolic model checking is usually done by either BDD-based or SAT-based algorithms. The BDD-based algorithm, BDD-BFS, conducts a breadth-first search on the system state space and records all reachable states. The goal for the search algorithm usually captures those states in which the invariants are violated. Being ‘blind’, BDD-BFS is an inefficient way to find error states as typically many regions of the state space will needlessly be searched. A BFS strategy is more suitable to correct models and is wasteful of space as generally all reachable states need to be stored. As well, for large systems, the sizes of BDDs in BDD-BFS can grow exponentially, making state space exploration almost impossible in realistic cases.

An alternative symbolic model checking approach is called bounded model checking [BCCZ99]. Bounded model checking translates the bounded semantics of the invariant into Boolean expressions and uses SAT procedures to determine their satisfiability. By incrementally increasing the bound, the algorithm iteratively deepens the state space exploration. If an error state is encountered at some level, the algorithm will terminate and report a counter-example. In general, SAT-based algorithms tend to detect counter-examples quicker than BDD-BFS due to the inherent DFS search strategy that SAT solvers use [CGKS02] if the counter-examples are short. SAT-based approaches, however, can be handicapped by a huge number of clauses that need to be input to the SAT solver. In our work, we combine aspects of both techniques by using BDDs to represent the state space and a heuristic DFS strategy to locate error states.
5.2.1 BDD-IDA* algorithm

The algorithm in Figure 5.1, called BDD-IDA*, is based on the explicit IDA* algorithm. The algorithm takes four inputs. The inputs $S_0$ and $G$ of the algorithm are BDDs representing the initial set of states and the goal, i.e. set of “bad” states. The input BDD $R$ is the transition relation. Note that we denote the calculation of the image of a given set of states $S$ by $R(S)$. The input $\sigma$ is a heuristic that will be illustrated in next section. Finally, the input $Bound$ determines the search depth. We use an explicit stack where each element in the stack is of the form $(g, h, S)$. The integer $g$ indicates the actual number of transitions from $S_0$ to $S$ and $h$ the heuristic estimation of number of transitions from $S$ to $G$. In line 10, the algorithm calls the procedure $\text{SplitAndPush}$. This procedure uses the heuristic $\sigma$ to partition a set of states $S$ (that constitute the frontier of the IDA* search) into subsets, and together with their associated costs $g$ and $h$, pushes each subset onto the stack. We show this procedure in the next session (Figure 5.2).

\begin{verbatim}
Procedure BDD-IDA* ($S_0$, $R$, $G$, $\sigma$, $Bound$)
1     stack.push() ← (0, 0, $S_0$)
2     counter ← 1
3     while (counter ≤ $Bound$) do
4         while (stack ≠ φ) do
5             $(g, h, S)$ ← stack.pop()
6             if ($S \land G ≠ φ$)
7                 return $Bound$
8             if ($h + g < counter$)
9                 $S ← R(S)$
10                $\text{SplitAndPush}(g, S, \sigma)$
11                counter ← counter + 1
12     stack.push() ← (0, 0, $S_0$)
13    return $\text{NoErrorInBound}$
\end{verbatim}

Figure 5.1. The BDD-IDA* algorithm

Note that we do not memorise the set of reachable states in the algorithm as we are only interested in the falsification of invariant properties.
5.3 The 3-Phase Heuristic Synthesis Algorithm

In this section we will outline how the heuristic $\sigma$ is synthesized. It is a three-phase process. (1) Abstraction: a data dependency analysis is used to automatically define an abstraction function for the concrete model. (2) Approximation: an approximation of the abstract model is then computed in order to avoid the computational penalty for exact abstraction. (3) Heuristic Construction: a standard BDD-BFS algorithm is used to compute all reachable frontiers in the approximate model. The result of this synthesis is a set of BDDs $\sigma = \{B_1, B_2, \ldots, B_n\}$ where each $B_i$ represents a set of states in the abstract model with the same heuristic value.

5.3.1 Automated abstraction

Let $M = (S, R, S_0)$ denote a concrete model where $S$ is a set of states, $S_0 \subseteq S$ is a set of initial states and $R$ is the transition relation. Let $H : S \rightarrow \hat{S}$ be a subjection that maps the concrete state space onto an abstract space $\hat{S}$ (i.e. $|\hat{S}| \leq |S|$). $H$ therefore induces an abstract model that is defined as follows.

Definition 5.1 (Abstraction) The abstraction of $M$ w.r.t. $H$ is denoted by $\hat{M} = (\hat{S}, \hat{R}, \hat{S}_0)$, where

- $\hat{S} = \{\hat{s} | s \in S \land \hat{s} = H(s)\}$
- $\hat{S}_0 = \{\hat{s} | s \in S \land \hat{s} = H(s) \land s \in S_0\}$
- $\hat{R} \subseteq \hat{S} \times \hat{S}$ is a transition relation, where $(\hat{s}_1, \hat{s}_2) \in \hat{R}$ iff $\hat{s}_1 = H(s_1) \land \hat{s}_2 = H(s_2) \land \exists s_1 \exists s_2 (s_1, s_2) \in R$

In symbolic model checking, $S_0$ and $R$ of $M$ are usually represented by two first-order formulas, $F_0(x_1, x_2, \ldots, x_n)$ and $F_R(x_1, x_2, \ldots, x_n, x'_1, x'_2, \ldots, x'_n)$, where $\{x_1, x_2, \ldots, x_n\}$ and $\{x'_1, x'_2, \ldots, x'_n\}$ are variables that represent the current state and next state of the model. Without loss of generality we assume all variables range over same domain $D$, hence the state set of $M$ is $S = D \times D \times \ldots \times D$. Let $\{\hat{x}_1, \hat{x}_2, \ldots, \hat{x}_n\}$ and $\{\hat{x'}_1, \hat{x'}_2, \ldots, \hat{x'}_n\}$ be variables that represent the current state and next state of $\hat{M}$. We denote $H(x_i) = \hat{x}_i$ iff $H$ maps every value of $x_i$ to an abstract value of $\hat{x}_i$. Let $\hat{F}_0$ and $\hat{F}_R$ denote the formulas that represent $\hat{S}_0$ and $\hat{R}$ respectively. Using the quantification on $F_0$ and $F_R$, we construct $\hat{S}_0$ and $\hat{R}$ by evaluating the following formulas.
1. \[ F_0 = \exists x_1 \ldots \exists x_n(H(x_1) = \hat{x}_1 \land \ldots \land H(x_n) = \hat{x}_n \land S_0(x_1, \ldots, x_n)) \]

2. \[ F_R = \exists x_1 \ldots \exists x_n \exists x'_1 \ldots \exists x'_n(H(x_1) = \hat{x}_1 \land \ldots \land H(x_n) = \hat{x}_n \land H(x'_1) = \hat{x}'_1 \land \ldots \land H(x'_n) = \hat{x}'_n \land R(x_1, \ldots, x_n, x'_1, \ldots, x'_n)) \]

To build the abstraction from a concrete model, we first need to define \( H \). In BDD-based symbolic model checking, every variable of the concrete model is encoded using a set of Boolean variables. Let \( V \) be a set of Boolean variables that encode all the variables in \( M \). Following [CGKS02] and Chapter refch4, we define \( H \) by restricting a subset \( V_{inv} \subset V \) to a single-element domain, i.e. \( H(F) = \exists v_0 \ldots \exists v_m F \) for all \( v_i \in V_{inv} \), where \( F \) is formula representing a set of concrete states. This abstraction essentially makes the variables in \( V_{inv} \) invisible. Note that in Chapter 4, the user had to provide \( V_{inv} \) to build the abstract model; possibly a difficult task. In this paper we describe an automated method to generate \( V_{inv} \) that is based on a data dependency analysis.

**The Data dependency analysis**

The aim of this analysis is to estimate the strength of each variable \( v_i \) in \( V \), and remove weak variables to form an abstract model. The analysis is based on the cone of influence (COI) abstraction techniques [CGP99]. Let \( V_p \subseteq V \) be a set of variables that appears in the specification \( \varphi \). The COI of \( V_p \), denoted by \( C \), is the minimal set of variables such that:

- \( V_p \subseteq C \)
- if for some \( v_i \in C \) its next\((v_i) \) depends on \( v_j \), then \( v_j \in C \)

If \( |C| < |V| \), then we construct a reduced model \( M' \) that only contains variables in \( C \). It is proved in [CGP99] that the reduced and original models form a bi-simulation relation w.r.t. all CTL specifications that only have variables from \( V_p \), i.e. \( M' \models \varphi \iff M \models \varphi \). As a result, model checking can be performed on the reduced model. Of course, every variable in \( C \) must be included, otherwise the reduced model is not bi-similar.

We use abstraction only to synthesize the heuristics that guide the model checker of the concrete model, so we do not need to restrict ourself to removing only the variables outside of \( C \) (unlike [CGP99]). Although all variables in \( C \) can influence the variables in \( \varphi \), the degree of influence will not normally be the same. Some variables in \( V_p \) are more strongly influenced by variables in \( C - V_p \) than others. To determine the subset of variables of \( V \) on which the truth of \( \varphi \) is heavily reliant, we build a dependency tree. Let \( D(v) \) be the positive integer denoting the distance from \( v \) to the root of the dependency tree. The smaller \( D(v) \) is, the stronger the influence of \( v \) on \( \varphi \). The following algorithm computes \( D(v) \) for all \( v \in C \).
initialize $i := 0$, $C := V_p$ and $V_t := V_p$;

**while** $C$ changes **do**

$$i := i + 1;$$

**for** each $v_i \in V_t$, compute all its dependable variables;

put those who are not in $C$ into $V_{tt}$;

assign $D(v) := i$ for all $v \in V_{tt}$;

assign $C := C \cup V_{tt}$ and $V_t := V_{tt}$;

To determine $V_{inv}$ we need to set the threshold $d$ for $D(v)$, and compute $V_{inv} := (V - C) \cup \{v|v \in C \land D(v) \geq d\}$, where $(V - C)$ are all variables that are outside of the COI. (In our tool the value of $d$ is a run-time option.)

### 5.3.2 Approximation of abstraction

In this Chapter, we describe details about the implementation abstracting the model in terms of BDD operations. Having defined $H$, we need to compute $\hat{F}_0$ and $\hat{F}_R$ in order to get BDDs for $\hat{S}_0$ and $\hat{R}$ of $\hat{M}$. We could compute them directly, i.e. through quantifier elimination. For asynchronous systems, $F_R$ is usually made up of a disjunction of Boolean formulae (transition blocks) and existential quantification can be pushed inside them. Synchronous models however consist of conjunctions of small transition blocks and existential quantifiers cannot be pushed inside. This means that we need to build a monolithic BDD for the formulae $F_0$ and $F_R$ and then perform quantifier elimination on them. This is computationally very expensive, especially in the case of $\hat{F}_R$.

This problem can be avoided if we allow the existential quantifiers to be pushed inside conjunctions. In other words, we want to push the quantification operation to formulae blocks. This computation can be relatively easy because the blocks are often small. This is essentially a compromise between the precision of the abstraction and the effort that it takes to compute that abstraction. The resulting model from the compromise is not $\hat{M}$ anymore, but an *approximation* of it, which we denote $\hat{M}_{app}$. We note at this point that although approximation and abstraction are used interchangeably occasionally in the literature, our use of these two is clearly different. In this thesis, we see abstraction is an relaxation of the original model, and approximation is a further relaxation of the abstraction. It is proved in [CGL94] that this approximation does not cause the loss of any initial states and transitions, i.e. $\hat{M}_{app}$ simulates $\hat{M}$ (see [CGL94] or Chapter 4 for the
definition of a simulation relation for Kripke structures). By transitivity of the simulation relation, $\hat{M}_{\text{app}}$ simulates $M$ because $\hat{M}_{\text{app}}$ simulates $\hat{M}$ and $\hat{M}$ simulates $M$ (Chapter 4). In order to show the correctness of the mechanism, we need to show that $\hat{M}_{\text{app}}$ contains the information that we can use to estimate the length of counter-examples of $M$. This we do in the following lemma.

**Lemma 5.1** If a state $s \in S$ is reachable in $M$ from a state in $S_0$, then its abstract counterpart $\hat{s}$ is also reachable in $\hat{M}_{\text{app}}$ from a state in $\hat{S}_0$.

The proof of this lemma is omitted. In essence, this lemma implies that if a counter-example is present in the original model, it must be manifest in the abstract model $\hat{M}$ as well as in the approximation $\hat{M}_{\text{app}}$. Note that both the abstraction and approximation do not ‘lose’ any transitions of the original system, although internal transitions with one abstract state are not visible in $\hat{M}$. Thus, the admissibility of the approach will therefore not be affected (see Chapter 4). This guarantees the resulting counter-example will be the shortest.

### 5.3.3 Constructing abstraction database

The purpose of a heuristic is to estimate the number of transitions from each concrete state to a goal state (or error state). The heuristic value for each state $s \in S$ in $M$ is simply the number of transitions from $\hat{s} = H(s)$ to the abstract goal state in $\hat{M}_{\text{app}}$. This type of heuristic is usually referred to as a pattern database [CS96b, ELL04, QN04b], where pattern is another term for abstraction. The term database means the heuristic is a memory-based heuristic that can be handled by a hash table, where the indices represent abstract states and the entries are heuristic values. As in Chapter 4, we use a set of BDDs to store the heuristic, called symbolic pattern databases (SADB) and denoted by $\sigma = \{B_1, B_2, ..., B_n\}$. Note that the set of states represented by these BDDs are disjoint, i.e. $B_i \cap B_j = \emptyset$ for $i \neq j$. Each $B_i$ represents a set of abstract states that have the same heuristic value, and hence have the same number of transitions to the abstract goal state. A SADB can be constructed using both backward and forward blind BFS search in $\hat{M}_{\text{app}}$:

**Backward construction** Use a BDD-based BFS strategy to explore $\hat{M}_{\text{app}}$, and start at the abstract goal. Put each frontier-BDD into the heuristic hash table with the number of iterations as the entry in SADB. Terminate if an abstract initial state is encountered.
**Forward construction** Instead, start at an abstract initial state, and store each frontier temporarily. If the search detects the abstract goal, then extract the path backward from the goal to the initial state. The set of BDDs that comprises the path is a forward SADB. This process is the same as the counter-example extraction in standard invariant checking. The paths generated here are a subset of the paths extracted by backward SADB.

It is of course possible that the heuristic synthesis procedure cannot find a trace in the approximation $\hat{M}_{app}$. In that case the original model $M$ does not have a counter-example (by Lemma 5.1).

**Splitting the BDD** Let $\sigma = \{B_1, B_2, \ldots, B_n\}$ be the heuristic (SADB) that is synthesized by the 3-phase process described above. The following algorithm splits a BDD into several BDDs that are the disjoint subsets of the original set of states. In order to contain the BDD size after splitting, we use the restrict operator on BDDs, denoted by $\downarrow$. Note the subscript $i$ of each $B_i$ indicates the number of transitions $B_i$ to the error state in the abstract model. The heuristic of a concrete state is simply the value of $i$ of its corresponding abstract state and is used by BDD-IDA* to prioritize the state space search and hence for efficient error detection.

**Procedure SplitAndPush** $(Cost, S, \sigma)$

```plaintext
for each $B_i \in \sigma$ do
    $I \leftarrow S \downarrow B_i$
    if ($I \neq \emptyset$)
        stack.push() $\leftarrow (Cost + 1, i, I)$
    $S \leftarrow S \land \bar{I}$
    if ($S \neq \emptyset$)
        stack.push() $\leftarrow (Cost + 1, \infty, S)$
```

Figure 5.2. The splitting procedure for BDD-IDA*
5.4 Experimental Results

The algorithms described above have been implemented in an model checker called GOLFER. The details of GOLFER can be found later in Chapter 8. The tool GOLFER incorporates the heuristic search algorithms A*, IDA* and weighted A*, and will construct a SADB as part of heuristic synthesis using the abstraction/approximation techniques described above. As well as the automatic abstraction construction that uses the data dependency analysis, GOLFER allows the user to input the abstraction \( H \) as a file of variable strength values.

To evaluate the ideas presented in this paper we have experimentally compared the performance of BDD-IDA* in GOLFER and the standard algorithms in NuSMV (namely BDD-BFS and a SAT-based bounded model checking algorithm). Note that in our experiment both BDD-IDA* and BDD-BFS use the same transition partition method of NuSMV for each model. We compare the run-time and memory usage for these methods. These approaches all operate in a fully automated manner, without any user interference except that we need to set up a bound for our algorithm and the SAT-based algorithm. In the experiment we set the bound to 50 and use the zChaff solver for the SAT algorithm. We first compare the performance of GOLFER and NuSMV on a simple game. We then follow with more realistic models. In the experiment we used known good BDD variable orderings for both BDD-IDA* and BDD-BFS when they were available. We also tried random orderings and found both algorithm has similar sensitivity to the same ordering. All experiments were carried out using shell scripts. The timeout operation (set to 2 hours) was implemented by a perl script. All experiments were conducted in a shared Intel X86 machine (CPU P4 933MHz) running Linux with 6G RAM.

5.4.1 Small models

The sliding-tile puzzle

consists of a board of \( n \times n \) squares occupied by \( n^2 - 1 \) tiles. Each tile exactly fits on one square and is labelled by a number ranging from 1 to \( n^2 - 1 \). Starting in some given initial configuration of the tiles on the board, the aim of the game is to move the tiles one at a time by utilizing the empty square until some given goal configuration is reached. Each state of the puzzle will have between 2 and 4 successors, hence the branching factor for the search graph is small. In the experiment, we use a \( 3 \times 3 \) board and 8 tiles. We encode the puzzle in the SMV input language and randomly generate 500 solvable initial
Figure 5.3. Run-time and memory usage for IDA* and BFS for the sliding-tile puzzle configurations. We show the results for our algorithm BDD-IDA* and the standard BDD-BFS approach in NuSMV in Figure 5.3. (The SAT-based approach is not included at this stage as it is not competitive on small models.)

In Figure 5.3 we group, average and order the data for the runs that result in the same solution depth. Generally, but not always, the shorter the solution depth, the faster the model checker finds the solution. Most solutions for the 500 starting configurations were in the range 17 to 27. Within this solution range BDD-IDA* outperforms BDD-BFS in both time and memory. For configurations with shorter solutions, BDD-BFS is generally faster than BDD-IDA* because of the overhead of the abstraction process in BDD-IDA*, which dominates when the goal configuration is just a few transitions away. For configurations with the longest solutions, BDD-IDA* and BDD-BFS perform similarly. It is not clear why this is the case. It is true that there are few long solutions, so the sparseness of data may be contributing to this behavior. However, we have observed the heuristic is quite poor for these configurations. If we manually generate a better heuristic for these configurations, we found that BDD-IDA* performed much better than BDD-BFS. We therefore feel that the data dependency analysis may be responsible, and conjecture that abstracting the system by eliminating supposedly weak variables loses validity in the longest runs. This may be an artifact of the particular data-dependency analysis that we have used.

The permutation puzzle

consists of a row of \(n\) distinct objects. Given some initial order of the objects, the aim of the game is to search for the row sequence that leads to some goal order. In the game, the order of the first \(k\) objects, where \(1 < k \leq n\), can be reversed, hence each state can have \(n-1\) successors, which is also the branching factor. We use two versions of this game, \(n = 8\) and \(n = 9\), and as before we randomly generated 500 different starting configurations. We omit memory-usage this time as it is very similar to the time performance. To improve
the clarity of the experimental results we partitioned the 500 runs into 25 instance groups, where for each group we compute the average run-time for IDA*, BFS and SAT. The results are shown in Figure 5.4.

The left diagram in Figure 5.4 depicts the average run-times for \( n = 8 \). All the methods solved the puzzle within our arbitrary time-limit. BDD-IDA* is clearly the best, but interestingly, the SAT method deteriorated surprisingly quickly at 20 showing that SAT is not always superior to BFS in finding counter-examples. The right diagram shows the run-times for just BDD-IDA* and BDD-BFS for \( n = 9 \). (The SAT algorithm failed to terminate within the allotted time.) As in the \( n = 8 \) case, BDD-IDA* performs consistently better than BDD-BFS.

While this data provides an interesting comparison between the 3 methods, one needs to remember that these methods are certainly not the best way to solve this kind of puzzle. An explicit-state model checker for example could be made to solve these puzzles faster than any of the above methods.

### 5.4.2 Real-world models

We applied the BDD-IDA* and BDD-BFS to the 8 models listed in Table 5.1. In this table, we show the type of model and the size of the SMV specification in each case. Some of the models can be found from Bwolen Yang’s collection of SMV models. If a model is parameterized, the value of the parameter is indicated by a numerical suffix in the name of the model. We sometimes also used the same model with different invariants. These models contain a parenthesized ‘p’ suffix in the model name.

The experimental results for comparing BDD-IDA* and BDD-BFS are shown in Table 5.2. For each model, we show the number of Boolean variables (\( \#\text{Vars} \)) that are used.
<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Type</th>
<th>Lines SMV code</th>
</tr>
</thead>
<tbody>
<tr>
<td>dme</td>
<td>distributed mutual exclusive ring</td>
<td>circuit</td>
<td>102</td>
</tr>
<tr>
<td>leader</td>
<td>concurrent leader election</td>
<td>protocol</td>
<td>129</td>
</tr>
<tr>
<td>mutex</td>
<td>mutual exclusion</td>
<td>protocol</td>
<td>116</td>
</tr>
<tr>
<td>ns</td>
<td>Needham-Schroeder public key protocol</td>
<td>protocol</td>
<td>319</td>
</tr>
<tr>
<td>peter</td>
<td>Peterson’s mutual exclusion algorithm</td>
<td>protocol</td>
<td>126</td>
</tr>
<tr>
<td>sr</td>
<td>sender receiver protocol</td>
<td>protocol</td>
<td>106</td>
</tr>
<tr>
<td>tarb</td>
<td>tree arbiter</td>
<td>circuit</td>
<td>142</td>
</tr>
<tr>
<td>tcas</td>
<td>traffic collision avoiding system</td>
<td>controller</td>
<td>3269</td>
</tr>
</tbody>
</table>

Table 5.1. Model used in the experimentation

To encode the model, the length of the counter-examples (CX), and the run-time and the number of BDD nodes for each of the methods (when possible). The table shows that BDD-IDA* consistently outperforms BDD-BFS in all but one case, *peter-3*. Note that the run-times for BDD-IDA* include the time for heuristic synthesis. We believe that the poor performance in the case of *peter-3* is an artifact of its smallness: the run-time is short and the automatic heuristic synthesis is an overhead that BDD-BFS does not have. In the cases *mutex* and *tcas*, BDD-IDA* can be up to 2 orders of magnitude faster. In most cases, less BDD nodes are used, sometimes an order of magnitude less. In the few cases where more BDD nodes were used, it was the same order of magnitude.

We cannot see from this data how much of the improved run-times comes from the ‘falsification superiority’ of DFS over BFS (note the very different BDD-partitioning schemes used in both strategies clouds this issue somewhat as well), and how much is a result of the guided search. We have used the same run-time options in all cases. In a few cases, we did notice that by changing certain run-time options such as the threshold of partition size or partition heuristics, we could improve the performance for BDD-BFS. However, we could never make it perform better than BDD-IDA*. We have not tried to fine-tune the partitioning scheme used in BDD-IDA*. Placing this work in context we should note that all the models contain at most a few invariant properties, and we know these properties are false. The experimental context is hence somewhat artificial and BDD-IDA* may not produce such performance improvements when used to verify models containing many properties. Furthermore, BDD-IDA* detects counter-examples. If the algorithm does not return before the timeout then we cannot say whether a counter-example exists or not.

We also compared the run-time of BDD-IDA*, BDD-BFS and SAT methods and the results are shown in Table 5.3. Note that we only do so if the SAT has better performance than BDD-BFS. Of 6 models we have experimented, BDD-IDA* detects error faster than
<table>
<thead>
<tr>
<th>Model</th>
<th># Vars</th>
<th>CX</th>
<th>Run-time</th>
<th># BDD Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>dme6</td>
<td>240</td>
<td>28</td>
<td>78.46</td>
<td>655163</td>
</tr>
<tr>
<td>dme8</td>
<td>320</td>
<td>30</td>
<td>7033.99</td>
<td>4499580</td>
</tr>
<tr>
<td>leader-3</td>
<td>85</td>
<td>16</td>
<td>1.01</td>
<td>305231</td>
</tr>
<tr>
<td>leader-4</td>
<td>128</td>
<td>18</td>
<td>11.07</td>
<td>228193</td>
</tr>
<tr>
<td>leader-5</td>
<td>168</td>
<td>20</td>
<td>111.62</td>
<td>981095</td>
</tr>
<tr>
<td>leader-6</td>
<td>200</td>
<td>22</td>
<td>1123.79</td>
<td>3945215</td>
</tr>
<tr>
<td>mutex-16</td>
<td>141</td>
<td>10</td>
<td>0.66</td>
<td>208967</td>
</tr>
<tr>
<td>mutex-20</td>
<td>175</td>
<td>10</td>
<td>1.01</td>
<td>320392</td>
</tr>
<tr>
<td>mutex-24</td>
<td>207</td>
<td>10</td>
<td>1.56</td>
<td>461404</td>
</tr>
<tr>
<td>mutex-28</td>
<td>239</td>
<td>10</td>
<td>2.30</td>
<td>158186</td>
</tr>
<tr>
<td>ns (p1)</td>
<td>87</td>
<td>14</td>
<td>0.90</td>
<td>183538</td>
</tr>
<tr>
<td>ns (p2)</td>
<td>133</td>
<td>14</td>
<td>7.43</td>
<td>103938</td>
</tr>
<tr>
<td>peter-3</td>
<td>72</td>
<td>26</td>
<td>0.59</td>
<td>151819</td>
</tr>
<tr>
<td>peter-4</td>
<td>103</td>
<td>42</td>
<td>10.06</td>
<td>278411</td>
</tr>
<tr>
<td>sr-11</td>
<td>273</td>
<td>14</td>
<td>0.44</td>
<td>177872</td>
</tr>
<tr>
<td>sr-12</td>
<td>297</td>
<td>14</td>
<td>0.70</td>
<td>211911</td>
</tr>
<tr>
<td>tarb-15</td>
<td>244</td>
<td>24</td>
<td>15.29</td>
<td>258489</td>
</tr>
<tr>
<td>tarb-17</td>
<td>276</td>
<td>24</td>
<td>32.11</td>
<td>464059</td>
</tr>
<tr>
<td>tarb-19</td>
<td>308</td>
<td>24</td>
<td>28.46</td>
<td>626627</td>
</tr>
<tr>
<td>tcas (p1)</td>
<td>292</td>
<td>12</td>
<td>4.44</td>
<td>190876</td>
</tr>
<tr>
<td>tcas (p2)</td>
<td>292</td>
<td>16</td>
<td>3.43</td>
<td>536990</td>
</tr>
<tr>
<td>tcas (p3)</td>
<td>292</td>
<td>24</td>
<td>9.45</td>
<td>328944</td>
</tr>
<tr>
<td>tcas (p4)</td>
<td>292</td>
<td>18</td>
<td>116.37</td>
<td>1745925</td>
</tr>
<tr>
<td>tcast (p1)</td>
<td>292</td>
<td>12</td>
<td>5.34</td>
<td>230336</td>
</tr>
<tr>
<td>tcast (p2)</td>
<td>292</td>
<td>18</td>
<td>38.05</td>
<td>623307</td>
</tr>
<tr>
<td>tcast (p3)</td>
<td>292</td>
<td>16</td>
<td>4.37</td>
<td>562949</td>
</tr>
<tr>
<td>tcast (p4)</td>
<td>292</td>
<td>16</td>
<td>4.97</td>
<td>552857</td>
</tr>
</tbody>
</table>

Table 5.2. Experimental results for BDD-IDA*, BDD-BFS
Table 5.3. Experimental results for BDD-IDA*, BDD-BFS and SAT

<table>
<thead>
<tr>
<th>Model</th>
<th>BDD-IDA*</th>
<th>BDD-BFS</th>
<th>SAT</th>
</tr>
</thead>
<tbody>
<tr>
<td>dme6</td>
<td>78.46</td>
<td>-</td>
<td>176.37</td>
</tr>
<tr>
<td>dme8</td>
<td>7033.99</td>
<td>-</td>
<td>521.77</td>
</tr>
<tr>
<td>ns (p2)</td>
<td>7.43</td>
<td>341.80</td>
<td>192.80</td>
</tr>
<tr>
<td>tarb-15</td>
<td>15.29</td>
<td>431.64</td>
<td>121.61</td>
</tr>
<tr>
<td>tarb-17</td>
<td>32.11</td>
<td>712.31</td>
<td>156.45</td>
</tr>
<tr>
<td>tarb-19</td>
<td>28.46</td>
<td>561.77</td>
<td>195.11</td>
</tr>
</tbody>
</table>

Figure 5.5. Run-times for 30 random different variable orderings

SAT in 5 models. For model “dme8”, our BDD-IDA* runs much slower than SAT for some unknown reason.

Variable Orderings

BDDs are well-known to be sensitive to the variable ordering [Bry86]. In the experiment we used a good variable ordering for both BDD-IDA* and BDD-BFS if such an ordering was available. We also tried other random variable orderings. The results are depicted in Figure 5.5, where you see the average run-time of BDD-IDA* and BDD-BFS for a given model using 30 different orderings (we actually tried hundreds of orderings and a number of the models but chose this set as representative). Generally, when BDD-IDA* performed well, BDD-BFS also performed well. There were exceptions: for example ordering 1 was better than ordering 2 for BDD-IDA*, but worse for BDD-BFS. These are probably the result of other factors like BDD partitioning working better for some orders than others in one or other of the schemes.
Optimality

The performance of BDD-IDA* is dependent on the quality of the heuristic. Suppose the heuristic cost for a BDD is $h^*$ and the exact cost $h$, then the quality is determined by the difference of $|h^* - h|$. We in fact do not care whether $h^*$ over- or under-estimates the real cost, but if it does over-estimate the cost, then we cannot guarantee that the algorithm will find the shortest counter-example. In model checking this is not normally an issue, but in general, and in particular in artificial intelligence, it can be a very serious issue. In fact, the heuristic synthesis procedure used in this work always results in a heuristic that under-estimates the cost because it is based on homomorphic abstractions (see Chapter 4). To improve the effectiveness of the heuristic, we could instead use the heuristic cost $a \times h^*$ instead of $h^*$, where $a > 1$ is a constant factor that can be tuned for specific applications. We could go a step further and use the total cost formula $f^* = b \times g + a \times h^*$ where $a, b$ are constant values and $g$ is the exact cost from the initial BDD (or state) to the current BDD (state). This may speed up the search dramatically, but optimality can no longer be guaranteed.

5.5 Related Work

The main work in BDD-based guided model checking uses prioritized traversal techniques. In [CNQ02] Cabodi et. al. proposed a mixed forward-backward prioritised traversal algorithm that checks invariant properties. This work is closely related to our approach as both share the idea of using prioritised traversal as well as abstractions and approximations. However, Cabodi et. al. differ in the way they combine these aspects:

- They construct an approximation of the concrete model and use it to approximate the forward reachable state set, which is then used to priorities the backward traversal of the state space. They use a best-first search algorithm. We use approximation for the computationally-intensive abstraction process. The heuristic synthesis in our approach is goal-oriented and BDD-IDA* also takes the real cost into account.

- They study only forward-backward traversal orders, which are more suited to circuits than communication protocols (for example) because traversals in both directions may not be possible even in the abstract model due to high branching factors. In our work we do not restrict ourselves in this way. During the approximation phase, we do not have to traverse from the target to the initial states. Because our approx-
imation is target-oriented, partial approximation traversal also serves as a heuristic for estimating the length of partial counter-examples.

In [RS95, FKZ+00] prioritized traversal algorithms are proposed based on BDD partitioning (or subsetting). A so-called “high-density” reachability analysis is used as a BDD optimization technique to traverse the state space of the system, and the density of a BDD is defined to be the ratio of states the BDD can represent over its size. The BDD with higher density will gain higher priority in the state space traversal. This technique only aims at optimizing the size of BDDs and offers no guidance to the model checking algorithm in the search for error states. In our work, heuristics are synthesized from the abstract model, and provide direct information about potential error states in the model. The above approaches use the VIS language and we use SMV. It would be possible combine our work with the above approaches, but there would be difficulties.

Most work in guided model checking is based on the explicit-state representation [ELL01, SJL04, TAC+04]. The first work on guided model checking in [YD98] applies prioritised state space exploration to model checking and proposes practical heuristics to guide the search. Heuristic search algorithms such as A* and IDA* have also been used in explicit-state model checking in HFS-SPIN [ELL01], recently Hopper (implemented on top of Murϕ) [SJL04] and FLAVERS [TAC+04]. All this work shows that the heuristic search algorithm can enhance the model checker’s ability to detect counter-examples. The role of BDDs, particularly in combination with IDA*, is an important aspect of our work of course, as is the use of heuristic synthesis, which none of these authors above have addressed.

5.6 Summary

In this chapter we have presented a fully automatic, symbolic, abstraction-guided model checker that builds its own abstract model, and uses this model as heuristic to guide the model checker. The main contribution of the work is its integration of:

- a data dependency analysis that is used to build an abstract model
- IDA* with heuristic synthesized from the abstract model, and
- a BDD partitioning algorithm in BDD-IDA* based on the heuristic.

The heuristic, which plays a vital role in guided search, is ‘double-dipped’: it not only guides the IDA* search strategy, it also provides a mechanism to partition the search space
BDDs). While it is true that the internal BDD operations are more complex than standard BDD-BFS, this is hidden from the user.

The notion of homomorphic abstraction allows us to hide certain details of a system model. The key idea of abstraction-directed verification is to use abstract hints to direct the search on a less abstraction model. Chapter 4 and 5 applies this idea on two levels, i.e. abstract and concrete model. In next Chapter, we generalize this idea and apply it to multiple levels of the abstraction. The abstraction-directed algorithm will be iteratively applied through all abstraction levels. The main advantage gained from this is that we are able to locate errors on each abstract level in a so-called region. The regions are then refined and reduced at each abstraction level. When this process gets to the concrete level, the region is relatively small and easy to search with.
Chapter 6

Ordered Abstractions

This chapter is based on the work [QN04a] presented in the Symposium on Automated Technology for Verification and Analysis (ATVA) in 2004. The following has been added:

- A section that describes choosing the abstraction orders.
- An experimental evaluation.

6.1 Introduction

Chapters 4 and 5 illustrate how abstractions are used in the framework of directed model checking. In these chapters, abstractions are used as a way to derive heuristic functions. The mechanism used to accomplish this is called abstraction databases. If multiple abstractions are available, multiple abstraction databases can also be derived, and used in different ways.

In this chapter, we describe a novel way to use multiple abstractions. The basic idea is that abstractions are ordered, from coarsest to finest abstraction, where the finest abstraction is the concrete model. Unlike Chapter 4, where we used multiple abstractions to derive multiple heuristics, in the work we present in this chapter, each abstraction is used to derive a new finer-grain abstraction that circulate the region of the state space around error states. The approach has the following characteristics:
• multiple abstractions are statically determined and ordered;
• each potential error trace is refined through the coarsest abstraction to the concrete model; and
• a coarser error trace can be used to refine a more precise one using a guided algorithm.

This approach allocates *error regions* in an abstract model. These regions are then refined along the abstraction in order to ‘hone in’ on a real error (if any) using guided search algorithms. In doing so, we ignore the portion of the state space that is outside of error regions. In effect, during the refinement of each region, the state space is being pruned.

### 6.2 Ordered Abstractions

We continue to use the finite state-transition system introduced in Chapter 4 as the semantic model to formalize our discussion. We denote \( M = (S, I, R, G) \) a finite state transition system (FSTS), where \( S \) is a set of states, \( I \) and \( G \) is a set of initial states and goal states respectively and \( R \) is a transition relation on \( S \times S \).

**Definition 6.1 (Simulation)** Let \( M_i \) and \( M_j \) be two FSTSs. We say \( M_j \) simulates \( M_i \), denoted \( M_i \preceq M_j \), if there exists a homomorphic abstraction \( H_{ij} \) that abstracts \( M_i \) to \( M_j \).

**Lemma 6.1** Let \( M_i, M_j, M_k \) be three FSTSs. If \( M_i \preceq M_j \) and \( M_j \preceq M_k \), then \( M_i \preceq M_k \), i.e. simulation is transitive.

**Proof.** Proof of this lemma is straightforward. Since \( M_i \preceq M_j \) and \( M_j \preceq M_k \), let \( H_{ij} \) and \( H_{jk} \) be two homomorphic abstractions such that \( M_j = H_{ij}(M_i) \) and \( M_k = H_{jk}(M_j) \). Suppose \( H_{ik} \) is abstraction that is composed by \( H_{jk}(H_{ij}(\cdot)) \). By the definition of homomorphic abstraction, \( H_{ij} \) is also homomorphic. Obviously, \( M_k = H_{jk}(H_{ij}(M_i)) = H_{ik}(M_i) \).

Let \( \mathcal{M} \) denote a sequence of abstractions \( M_1, M_2, \ldots, M_n \). Let \( \mathcal{H} = \{H_1, H_2, \ldots, H_{n-1}\} \) be a set of homomorphic abstractions such that \( M_2 = H_1(M_1), M_3 = H_2(M_2), \ldots, M_n = H_{n-1}(M_{n-1}) \). We call \( \mathcal{M} \) an abstraction order w.r.t. \( \mathcal{H} \). By the above lemma, we have \( M_1 \preceq M_2 \preceq \ldots \preceq M_n \).
In [CGL94], Clarke et al prove that for all properties in ACTL*, if the homomorphic abstraction of the concrete model satisfies the property, then so does the concrete model. The converse of the inverse of the theorem can be stated as follows. If the concrete model violates the property, the homomorphic abstraction of it should also violate the property.

**Definition 6.2 (Concretization)** Let $M_1 = (S_1, I_1, R_1, G_1)$ and $M_2 = (S_2, I_2, R_2, G_2)$ be two FSTSs and $M_2 = H(M_1)$ where $H$ be homomorphic abstraction. The inverse of $H$, denote $H^{-1}$, is called the concretization function. If $s \in S_2$, then $H^{-1}(s) = \{x | H(x) = s \wedge x \in S_1\}$. For the convenience, we also denote $H^{-1}(\Sigma)$ the concretization of a set of states $\Sigma$.

We now prove the theorem that forms the theoretical basis of the algorithm in this chapter. The theorem below essentially says that a error trace in a less abstract model is contained in the concretization of its counterpart in the abstract model. This theorem allows us to safely ignore the proportion of the state space outside of the abstract error trace.

**Theorem 6.2** Let $M_1$ and $M_2$ be two FSTSs such that $M_2 = H(M_1)$. If there is a solution path $\pi_1$ in $M_1$, then there must exist a solution path $\pi_2$ in $M_2$ such that for $s_i \in \pi_1, s_j \in \pi_2, s_j = H(s_i)$. Let $\Sigma(\pi)$ denote the set of states that constitute $\pi$. Then $\Sigma(\pi_1) \subseteq H^{-1}(\Sigma(\pi_2))$.

**Proof.** We use refutation to prove this theorem. Suppose there is a state $s \in \Sigma(\pi_1)$ and $s \not\in H^{-1}(\Sigma(\pi_2))$. Hence, $H(s) \not\in \Sigma_2$. By definition, we know every state of $\pi_1$ can be mapped to a state of $\pi_2$. This is contradicted to $H(s) \not\in \Sigma(\pi_2)$. Therefore, $\Sigma(\pi_1) \subseteq H^{-1}(\Sigma(\pi_2))$.

From the definition of the abstraction order and the above theorem, it is easy to have the following corollary.

**Corollary 6.3** Let $\mathcal{M} = M_1 \preceq M_2 \preceq \ldots \preceq M_n$ be an abstraction order. Let $P = \{p_1, p_2, \ldots, p_k\}$ be the set of error paths in $M_1$ and $Q = \{q_1, q_2, \ldots, q_l\}$ be the set of error paths in $M_n$. $H$ is the abstraction from $M_1$ to $M_n$, i.e. $M_n = H(M_1)$. Then (1) $|Q| \leq |P|$; (2) For each $0 \leq i \leq k$ there exists $0 \leq j \leq l$ such that $\Sigma(p_i) \subseteq H^{-1}(\Sigma(q_j))$. 104
6.3 Verification Guided by Ordered Abstractions

6.3.1 Static abstraction-guided verification

This approach shares common features with the well-known abstraction-refinement approach that we reviewed in Chapter 2. Both approaches use a sequence of abstractions (that can be ordered by a relation). The abstraction refinement starts with an initial abstraction and checks whether the abstraction satisfies the given property. If the abstraction does not verify the property, it dynamically refines the initial abstraction and generates another abstraction. The idea behind the refinement is to eliminate irrelevant information from the abstraction and add useful information. The result of the refinement is that a new abstract model is generated. The process iterates until the property is shown to be true or false. The CEGAR algorithm in Figure 2.1 is one of implementations of the abstraction-refinement framework of model checking. Note that the abstractions used in this scheme are dynamically determined.

In our approach, abstractions are statically determined. Instead of refining the abstraction dynamically, we refine the error paths that are found in the abstractions to determine whether they are spurious or not. The new approach, which we refer to as the Static Abstraction Guided (SAG) model checking algorithm is depicted in Figure 6.1. Notice that we have a sequence of abstractions on the left-hand side of the figure. We start with the initial abstraction $M_n$ (the coarsest) and use a standard forward model checking
algorithm [IN96] to compute all the paths through the state space that lead to an error state. The union of all the states in these error paths we call the *error region* as it must contain the error state(s), if there are any. It is feasible to determine all of the error paths because the size of the initial abstract model is relatively small. These error paths are abstract counterexamples, and are identical to those computed in the abstraction-refinement approach. However, in the abstraction-refinement approach, an error path is (only) used to provide assistance in refining the abstraction, and usually a single counterexample is computed.

In SAG, if we cannot find any error path in $M_n$, then we may conclude the model satisfies the safety property (we only consider property-preserving abstractions in this work). We then proceed to each of the finer levels of abstraction $M_{n-1}...M_1$. At each level, some (or even all) of the error paths may prove to be spurious. If at some level of abstraction we find that no error paths remain, then we have verified the complete concrete model, irrespective of the abstraction level that we are currently at. Alternatively, if there are still error paths remaining when we reach the concrete model, then we have found ‘real’, concrete counterexamples.

The process of eliminating error paths is conducted using the heuristic search algorithm $A^*$. As we step through the abstractions, we use the previous abstraction as a heuristic for $A^*$. In our scheme then, the abstraction and directed search approaches are closely entwined. In general terms, we differ from the abstraction-refinement approach in the following ways:

1. Our approach uses a predefined set of abstractions.
2. We compute all error paths in the (coarsest) abstract model, not just one.
3. We do not refine the abstract model during model checking, but the error paths themselves, and eliminate those that are spurious.
4. The heuristic search algorithm $A^*$, which uses the courser abstract model as heuristic at each level, is used to speed up the search as well as to optimally compute the shortest error path.

### 6.3.2 Example

Before formally describing the algorithm, we illustrate how it works using a trivial example. Since our aim in this dissertation is to detect errors in a model, the example we show here contains an safety error.
Consider the abstraction order shown in Figure 6.2. The order contains 3 models with the order $M_1 \preceq M_2 \preceq M_3$. Hence, $M_1$ is the concrete model and $M_3$ is the most abstract one. We represent a concrete system $M_1$ using a 4-bit vector $(x_0, x_1, x_2, x_3)$ where $x_i$ is a Boolean variable. Let $(1101)$ be the (error) state that violates the safety property (double-circled state). The task is to detect the error path originating from $(0000)$ that leads to the error state in $M$. We define a 2-level abstraction sequence as shown in the figure. At the coarsest abstract level, $M_1$, the third and fourth bits of the vector are made invisible, resulting in 4 states in total. We omit a state’s transition to itself as it need not contribute to finite error paths. This is safe in our methodology because removing self-loops does not affect the reachability in the abstract system, thus the heuristic value computed is not affected. As can be seen, the abstract error path, indicated by the dotted-line oval, is detected by exhaustive search in $M_3$. Next we refine this initial abstract initial error path to a less abstract level, $M_2$. In $M_2$, only the fourth bit is made invisible and the abstract system consists of 8 states. In searching for a more refined error path in $M_2$, we make use of the abstract error path in $M_1$ as a heuristic for the distance from a state to the abstract target state $(110x)$. More precisely, our algorithm begins with $(000x)$ in $M_2$, and only explores the successor whose corresponding state in $M_3$ has the minimum distance to
the final state of error path in $M_3$. Because the heuristic search algorithm $A^*$ is used, we only need to process the states in the dotted-line circle in $M_2$.\footnote{In the worst case, the algorithm is of course equivalent to breadth-first search.} Corresponding heuristic value ($h$) and actual cost ($g$) are given with format $(h, g)$ near every node on the path in $M_2$ and $M_1$. Once the abstract error in $M_2$ has been determined, we use the same method to refine it to the concrete level (the dotted-line circle in $M_1$).

### 6.3.3 The SAG algorithm

The SAG algorithm consists of three procedures. The main procedure is shown in Figure 6.5 and two sub-procedures are in Figure 6.3 and 6.4.

Detecting all error regions

Initially, we abstract away information from the concrete system to produce an initial abstraction that has fewer states, and identify the potential regions where the error state may reside, while keeping the model size manageable. In essence, we isolate that part of the state space of the initial abstraction that may contain error paths. An procedure to find all error regions in an abstraction $M_n$ is shown in Figure 6.3.

**Procedure FindAll** ($M_n \leftarrow (S, I, R, G)$)

1. $reach \leftarrow I$; $old \leftarrow \emptyset$; $EP \leftarrow \emptyset$
2. while $(reach \neq S \lor reach \neq old)$ do
3.    if $(reach \cap G \neq \emptyset)$ then
4.        $EP \leftarrow \text{extractEP}()$
5.    $old \leftarrow reach$
6.    $reach \leftarrow R(reach)$
7.    if $EP \neq \emptyset$ then
8.        return $EP$
9.    else
10.       return $Safe$

![Figure 6.3. Detect all error paths in $M_n$](image)
This algorithm computes the set of error paths $EP$ by doing a reachability analysis of $M_n$. The set $EP$ constitutes the error region. The significant difference between this algorithm and standard fixed-point reachability algorithms is this algorithm detects and collects all error paths. This is essential because when entering the next, finer level of abstraction we do not want to examine the entire state space but only that part that contains an error path (by Corollary 6.3). If we missed an abstract error path this may result in concrete counterexamples being missed. In the CEGAR (Figure 2.1) approach, all abstract error paths $M_n$ are not required because only a single error path is used to guide the refinement, and blind search of the entire state space is used.

**Refining an region**

If the algorithm **FindAll** returns a set of abstract error paths $EP = \{\pi_1, \pi_2, \ldots, \pi_k\}$, we need to refine them to see which paths are spurious. The refinement utilizes the pre-defined abstraction order. The essence of the refinement is that the set $EP$ of error paths is reduced by mapping the set to the next finer abstraction level and checking for spuriousness. The algorithm for refining a single abstract error path is shown in Figure 6.4.

When we refine the (initial) error region, we produce a set of error paths at a finer level of abstraction. The refinement essentially prunes the paths that are infeasible at the finer level (but feasible at abstract level). This new set of paths are the shortest representative paths that correspond to the previous (coarser) paths. The essence of path refinement in the algorithm **Refine** is that we map a path in the abstract model $M_i = (S_i, I_i, R_i, G_i)$ to a path in $M_{i-1} = (S_{i-1}, I_{i-1}, R_{i-1}, G_{i-1})$. The states in this path are mapped to a set region, which must contain the error path if it is not spurious.

To determine the spuriousness we apply the A* algorithm (see Line 3-12 in the algorithm) where the heuristic value for a state is calculated by the function **distance** (Line 10). Given a state $s$ and an abstract error path $\pi$, **distance** calculates the number of transitions from the $\hat{s}$ to the goal state of $\pi$, where $\hat{s}$ is the abstraction of $s$. Note that this is similar to the method we use to compute the heuristic value in Chapter 4 and 5. Note that if we discover a state $s$ that lies outside of the error region, then this state is assigned a cost of $\infty$.

**The main procedure**

The complete algorithm that verifies a model by using a statically determined abstraction order is shown in Figure 6.5.
Procedure Refine \((\pi, (S_i, I_i, R_i, G_i), (S_{i-1}, I_{i-1}, R_{i-1}, G_{i-1}), H, \text{region}))\)

1. \( \text{old, open} \leftarrow \emptyset; \text{rank} \leftarrow 0; \)
2. \( \text{states} \leftarrow I_{i-1} \bigcap \text{region}; \text{reach} \leftarrow \text{states}; \text{open} \leftarrow (\text{rank, states}) \)
3. \( \text{while} (\text{reach} = \text{old}) \text{ do} \)
4. \( \text{old} \leftarrow \text{reach} \)
5. \( (r, s) \leftarrow \text{open.pop()} \)
6. \( \text{if} (s \bigcap G_{i-1} \neq \emptyset) \text{ then} \)
7. \( \quad \text{return extractEP()} \)
8. \( s \leftarrow R_{i-1}(s) \bigcap \text{region} \)
9. \( \text{reach} \leftarrow \text{reach} \bigcup s \)
10. \( d \leftarrow \text{distance}(\pi, s) \)
11. \( r \leftarrow r + 1 + d \)
12. \( \text{open} \leftarrow (r, s) \)
13. \( \text{return Safe} \)

Figure 6.4. Refine \(\pi\) in \(M_{i-1}\) guided by \(M_i\)

The SAG algorithm first determines the error paths at the coarsest level of abstraction, \(M_n\), by calling FindAll. It then repeatedly refines each error path in every abstract model of the order using Refine. Note that in the version of the algorithm above, we descend the abstraction sequence for each error path in all. This is a depth-first lazy approach, but we could of course also have used a breadth-first approach. If no error paths are found in the last model in the abstraction sequence \(M_1\), which corresponds to the original, concrete model \(M\), then safe is returned, otherwise a set of error paths. These error paths are concrete counterexamples.

### 6.4 Choosing an Abstraction Order

A list of abstractions must be statically determined prior to the verification process. The technique of determining the abstraction should satisfy following criteria:

- it imposes the simulation relation on the abstraction easily; and
Procedure SAG \((M_1, M_2, \ldots, M_n)\)
\begin{algorithmic}[1]
    \State \texttt{all} \leftarrow \emptyset; \texttt{err} \leftarrow \emptyset
    \State \texttt{all} \leftarrow \text{FindAll}(M_n)
    \If {\texttt{all} \neq \textit{Safe}}
        \For {\texttt{each} \pi \in \texttt{all}}
            \State \texttt{region} \leftarrow \Sigma(\pi)
            \For {i = n \text{ to } i = 2}
                \If {\text{Refine}(\pi, M_i, M_{i-1}, H_i, \texttt{region}) = \textit{Safe}}
                    \State \text{break}
                \Else
                    \State \pi \leftarrow \text{Refine}(\pi, M_i, M_{i-1}, H_i)
                    \If {i = 2}
                        \State \texttt{err} \leftarrow \pi
                        \If {\texttt{err} \neq \emptyset}
                            \State \Return \texttt{err}
                        \EndIf
                    \EndIf
            \EndFor
        \EndFor
    \EndIf
\EndAlgorithmic
\end{algorithmic}

Figure 6.5. The main procedure of SAG algorithm

- it should be computationally feasible to construct.

There are many methods available to derive a homomorphic abstraction of a model. For example, to drive an abstraction database, existential abstraction was used in Chapter 4. For efficiency reasons, approximation techniques are usually needed as the exact existential abstraction is very expensive (see Chapter 5). In this section, we describe a abstraction technique that is compositional and computationally effective.

Consider a simple method to specify a model as follows. Let \(M\) be the finite state transition system specified by several small fragments called \textit{modules}. Practically \(M\) is often specified hierarchically by reusing a certain module many times. For convenience, we assume the hierarchy of \(M\) is flattened to a set of modules, denoted \(\{m_1, m_2, \ldots, m_n\}\). Each \(m_i\) \((1 \leq i \leq n)\) consists of a set of variables \(V_i\) and assignment statements \(A_i\). Each element in \(A_i\) is one of the two forms:

\[
\text{init}(v) := C
\]
where \( v \in V_i \) and \( C \) is a constant; or

\[
\text{next}(v) := \text{expr}
\]

where \( v \in V_i \) and \( \text{expr} \) is a expression consists of a syntactic composition of the constants and variables. Note that the \( \text{expr} \) can involve variables that are in other modules. Each \( \text{init}(v) \) statement specifies an initial value of \( v \) and \( \text{next}(v) \) specifies the transition relation of \( v \). Hence, all variables in \( M \) is \( V = \bigcup V_i \). A set of statements \( A = \bigcup A_i \) essentially specifies the model \( M \). The transition of \( M \) is constructed by computing a \textit{synchronous product} of the \textit{next} states of each module. During the construction of \( M \), if we parameterize the compiler to only include a subset of modules, then the result, \( \hat{M} \), is homomorphic abstraction of \( M \). Let \( \hat{V} \subseteq V \) and \( \hat{A} \subseteq A \) be the variable and statement set for \( \hat{M} \). During this construction, it is possible that a statement involves a variable that is not in \( \hat{V} \). If this is the case, the compiler will existentially quantify it out. So in essence, the method is same to the approximation method we use in Chapter 5.

Given a finite set of modules \( M = \{m_1, m_2, \ldots, m_n\} \), a system model can be constructed by compiling in any number of modules. Hence, the number of models that we can construct is \( 2^{|M|} - 1 \). The one with all modules is called the concrete model. Those with fewer modules are abstractions that simulate the concrete model. Simulation relation between two abstractions is defined by the set inclusion of the modules that comprise them, i.e. \( M_i \preceq M_j \) iff \( M_j \subseteq M_i \) or \( |M_i| = |M_j| \). Let \( Q \) denote an abstraction that simulates all possible models that are constructed from \( M \). Together with \( Q \), all possible models forms
The directed edges in the diagram indicates the simulation relationship between two models. Hence, a path from $Q$ to $\{m_1, m_2, m_3, m_4\}$ is a list of abstractions that is ordered by the simulation relation. We call such list a policy for choosing an abstraction order. For example, 4 shadowed nodes constitute legal abstraction order. We refer it to as the fringe policy. There are of course many policies that can be chosen statically for the input of the SAG algorithm. In theory, all possible options should be equally valid. In practice, the policies for choosing some of the options can be heuristically determined.

### 6.5 Experiments

The SAG algorithm has been incorporated into our tool GOLFER. We use the two parametric model family to evaluate the effectiveness of the guiding heuristic. The first model family mutex is a mutual exclusion protocol that can be parameterized by the number of processes. The other model family sender is a packet transmission protocol parameterized by the number of agents. We only evaluated the buggy version of these protocols to determine the effectiveness of the SAG algorithm. All the experiments are carried out in a Linux machine with Pentium 3 CPU and 1 GB RAM.

We compared SAG with two other algorithms: the first is the standard BFS algorithm implemented in most model checkers, and the other BF-SAG is a brute-force version of the SAG algorithm. BF-SAG works in a similar manner to SAG except that all guiding mechanisms are removed, which means that at every abstraction level, the refinement of the abstract error paths are done by BFS instead of $A^*$. 

<table>
<thead>
<tr>
<th>model</th>
<th>BF-SAG</th>
<th>SAG</th>
<th>standard BFS</th>
</tr>
</thead>
<tbody>
<tr>
<td>mutex 16</td>
<td>8.0</td>
<td>1.7</td>
<td>6.5</td>
</tr>
<tr>
<td>mutex 20</td>
<td>30.0</td>
<td>7.6</td>
<td>21.5</td>
</tr>
<tr>
<td>mutex 24</td>
<td>73.0</td>
<td>17.8</td>
<td>55.5</td>
</tr>
<tr>
<td>mutex 28</td>
<td>156.0</td>
<td>30.0</td>
<td>124.4</td>
</tr>
<tr>
<td>sender 11</td>
<td>76.0</td>
<td>33.5</td>
<td>43.8</td>
</tr>
<tr>
<td>sender 12</td>
<td>90.0</td>
<td>36.8</td>
<td>56.3</td>
</tr>
</tbody>
</table>

Table 6.1. Run-time comparison for SAG algorithm
We use 4 levels of abstraction (including the concrete model) in the experiment. The abstraction order is hand-crafted using the fringe policy. Table 6.1 shows the run-time of the 3 algorithms. For both model families, SAG is consistently fastest. The BF-BFS algorithm uses the most time for all models. The SAG algorithm can save as much as 75% in the run-time needed by the standard BFS algorithm. During the experimentation, we also observed that the BF-BFS consumes the most time in refining the error path at the concrete level, and that the run-time is evenly distributed amongst the abstraction levels. The reason for this discrepancy is that the guiding technique used in the procedure Refine (see Figure 6.4). During this refinement process, the guided search directs the algorithm to the goal very quickly without examining all possible states. The result is that less time is needed for the final refinement of the concrete model.

6.6 Summary

In this chapter we have presented a model checking framework that is based on both abstraction and heuristic search. This work extends the work in previous chapters that demonstrated that an abstract model can be used as a heuristic in a model checking algorithm. Instead of using a single abstract model, the SAG algorithm uses a set of abstract models. These models must be ordered by a simulation relation.

Unlike CEGAR, which refines the model dynamically, the SAG algorithm uses a static strategy to determine a set of abstractions for detecting the error in a model. Once a potential error is located in an abstract model, SAG refines only the error path in that abstract model. Hence, the expensive refinement of the entire model can be avoided. The inaccuracy caused by the static abstraction order is compensated by using the guided search at each abstract level. The experimental results shows an improvement of SAG over the standard BFS search and the brute-force SAG algorithm.

All the guided techniques that we have presented have concerned invariant model checking. In the next chapter, we present a mechanism to translate CTL model checking into invariant model checking. This will enable all the guided techniques that we have presented to be used on CTL.
Chapter 7

Dealing with Full CTL

In previous chapters we have considered only invariants. In this chapter we apply abstraction-guided verification to general properties. We do this by transforming the general CTL model checking into invariant checking. The work is based on the publication [QN05] presented in the Workshop on Model Checking and Artificial Intelligence (MoChArt) in 2005, which was co-located with the International Conference on Concurrency (CONCUR).

7.1 Introduction

In BDD-based model checking, the property to be verified is often specified by a branching-time logic, usually CTL. While branching-time model checking can be solved efficiently by using the fixed-point characterization of temporal logic formulae, those algorithms are still based on brute-force search on the state space. The abstraction-guided techniques described in previous chapters combines heuristic search algorithms and the BDD-based state-space search, making the search more goal-oriented. These techniques are most effective for a reachability analysis where only a single fixed-point is calculated. For general CTL model checking, it is not clear how these techniques can be applied.

Schuppan and Biere [SB04] proposed a technique called state recording that transforms LTL model checking into a reachability analysis. In so doing, liveness properties are translated into safety properties. The satisfiability of the liveness properties in the original model and the safety properties in the transformed model are equivalent. One of the
goals in this chapter is to generalize this technique and provide a framework to transform branching-time model checking into a reachability analysis.

Alternating tree automata and AND/OR graphs provide elegant formalisms that enable branching-time logics to be verified in linear time. The seminal work of Kupferman et al. [KVW00] showed that 1) branching-time model checking is reducible to the language non-emptiness checking of the product of two alternating automata representing the model and property under verification, and 2) the non-emptiness problem can be solved by performing a search on an AND/OR graph representing this product. Their algorithm, however, can only be implemented in an explicit-state model checker because it needs stacks to detect accepting and rejecting runs.

In this chapter, we propose a BDD-based approach to check the language non-emptiness of the product automaton. We use state recording to emulate the stack mechanism from explicit-state model checking. This technique allows us to transform the product automaton into a well-defined AND/OR graph. We develop a BDD-based reachability algorithm to efficiently determine whether a solution graph for the AND/OR graph exists and thereby solve the model-checking problem. While state recording increases the size of the state space, the advantage of our approach lies in the memory saving BDDs can offer. Moreover, this transformation is part of our abstraction-guided verification framework. It makes our framework applicable for the entire class of CTL and CTL*. The key contributions of the work in this chapter are:

- Reduce a branching-time model-checking problem to a reachability analysis by using alternating tree automata and state recording.
- Devise a BDD-based AND/OR-graph search algorithm.
- Generalize the work of Schuppan and Biere [SB04] by dealing with any branching-time properties.
- Modify the approach of Kupferman et. al. [KVW00] to work in a symbolic setting.

7.2 Alternating Tree Automata

The motivation for using an alternating tree automaton (ATA) is that it can be constructed in linear-time for a branching-time logic. Here we briefly describe the formal concepts. For a more comprehensive view the readers should refer to [Tho90, KVW00].
A digraph $T = (V_T, E_T)$ is a tree if every node $x \in V_T$ has exactly one incoming edge except for the root $x_0$, which has no incoming edges. We denote $(x, y) \in E_T$ a directed edge from $x$ to $y$ and call $y$ a successor of $x$. The degree of a node $x \in V_T$ is defined to be the number of successors of $x$, denoted $d(x)$. A leaf is a node without successors. A path $\pi$ of a tree $T$ is a set of nodes $\pi \subseteq V_T$ such that the root $x_0 \in \pi$ and for very node $x \in \pi$, $x$ is either a leaf or there exists a unique node $y$ where $(x, y) \in E_T$. Note that we allow infinite trees and infinite paths. Let $\Sigma$ be an alphabet and $W : V_T \rightarrow \Sigma$ be a labeling function that maps each node of a tree to a letter in $\Sigma$. The pair $(T, W)$ is called a $\Sigma$-labelled tree. Given a Kripke structure $K = (S, R, s_0)$ where $S$ is a set of states, $R : S \rightarrow S$ is a transition relation and $s_0 \in S$ is an initial state, one can unwind $K$ from $s_0$ and obtain a possibly infinite ($\Sigma$-labelled) tree where all the labels come from $S$. This tree is called the computation tree of $K$.

Let $A = (\Sigma, Q, \delta, Q_0, F)$ be an automaton where $\Sigma$ is an alphabet, $Q$ is a finite set of states, $\delta$ is a transition relation, $Q_0 \subseteq Q$ is an initial state and $F$ the acceptance conditions. In a word automaton, given a state $q$ and input letter $\sigma$, the transition relation $\delta(q, \sigma) : Q \times \Sigma \rightarrow Q$ maps the pair $(q, \sigma)$ to a set $\{q_0, q_1, \ldots, q_k\} \subseteq Q$. The automaton is deterministic if the set is a singleton, and non-deterministic otherwise (assuming the transition is total). A run of a word automaton corresponds to an infinite sequence of states $\rho = q_0 q_1 q_2 \ldots \ldots$ such that $q_0 \in Q_0$ and $\exists \sigma(q_{i+1} \in \delta(q_i, \sigma))$.

In a tree automaton, the transition $\delta$ maps each state and input letter to a set $\{{B_0, B_1, \ldots, B_k}\}$ where $B_i \in P(Q)$. For example, the transition $\delta(q, \sigma) = \{{q_0, q_1, q_2}\}, \{q_3, q_4\}\}$ causes a given state $q$ of the automaton to change, simultaneously and non-deterministically, to one of the two sets. Hence, unlike word automata, a run of a tree automaton will generate an (infinite) computation tree. In general, the nodes of the tree have varying branching degrees. If all degrees of any node is contained in a set $\mathcal{D} \subset \mathbb{N}$, then we call the tree a $\mathcal{D}$-tree. Note, if $\mathcal{D}$ is a singleton, and the element of the singleton is 1, then the tree automaton is simply a word automaton.

An alternating tree automaton generalizes a non-deterministic tree automaton defined above by allowing both universal and existential choices for automata state transitions [MS87, Tho90]. Consider again a non-deterministic tree automaton with transition $\delta(q, \sigma) = \{{q_0, q_1, q_2}\}, \{q_3, q_4\}\}$. If we express the right-hand side as a Boolean formula, e.g., $\delta(q, \sigma) = q_0 \land q_1 \land q_2 \lor q_3 \lor q_4$, we can express both 1) transitions within one group and 2) non-deterministic choice between groups, simultaneously. In fact, this Boolean formula is the characteristic function of the two sets. To accommodate the branching degrees of a tree, we add an extra argument to the transition $\delta$, and rewrite the above transition as $\delta(q, \sigma, 2) = (0, q_0) \land (1, q_1) \land (2, q_2) \lor (0, q_3) \land (1, q_4)$.
Automata-theoretic model checking allows a range of acceptance conditions to be used to express different properties [VW86]. In this paper, we restrict our discussion to a Büchi acceptance condition. If we have an infinite word automaton \((\Sigma, Q, \delta, Q_0, F)\), then \(F \subseteq Q\). If we have some infinite run \(\rho\), then, because \(Q\) is finite, some \(Q_\rho \subseteq Q\) must appear infinitely often in \(\rho\). A run \(\rho\) is a (Büchi) accepting run iff \(Q_\rho \cap F \neq \emptyset\). Because a run in an ATA is actually a (computation) tree, the acceptance condition is defined over a path in the run. Paths in an ATA correspond to runs in word automata. If all paths are accepting paths, the run of the automaton is an accepting run.

### 7.3 The State Recording Mechanism

Checking liveness properties involves checking for the existence of a cycle in a transition system. In explicit-state model checking, an explicit stack is usually used to implement the cycle detection. In symbolic model checking, sets of states are represented by BDD and therefore individual path information is lost. Implementing an explicit stack is generally considered to be infeasible. Schuppan and Biere [SB04] describe a method that translates the cycle detection into a reachability problem. We illustrate this approach using a trivial example below. For a detailed description, readers are referred to [SB04].

Figure 7.1 (a) shows a transition system with 3 states. There is a cyclic transition path in the diagram, namely from \(s_1\) to \(s_2\) and goes back to \(s_1\). A state of a system is an assignment to a set of state variables, also called a *state vector*. For this example, since there are three states, we use variable \(s\) to represent the state space, i.e. \(s \in \{s_0, s_1, s_2\}\). The state vector of the system is shown in Figure 7.1 (b). The key of state recording is that the individual path information must be embedded in the single state. The existence of a cycle is therefore detected by a reachability analysis that only checks whether a particular state is reachable. A variable \(s'\) is used to record the state at the beginning of the cycle. During the reachability analysis, a cycle loops back to its beginning state iff \(s = s'\).

To implement state recording, two issues need to be addressed. First, the beginning of a cycle is usually not known in advance. Second, there are two situations where \(s = s'\), (1) at the beginning of the recording; and (2) at the state that a cycle returns to. A Boolean variable \(b\) is used to guess the beginning of a cycle. At every state, \(b\) non-deterministically has two values 0 and 1. At the beginning of a cycle it is assumed \(b = 1\). Another Boolean variable \(c\) is used to indicate whether recording has already happened. If recording has already happened along a path then \(c = 1\). In state recording, the state vector is shown in Figure 7.1 (c). Hence, the existence of a cycle in the system can be judged only by the state value, namely \(s = s' \land b = 1 \land c = 1\) iff a cycle is present. The process of the
Figure 7.1. Cycle detection via reachability analysis

reachability analysis after state recording is illustrated in Figure 7.1 (d). Note that ‘x’ is a special character for irrelevant value. The shaded node is the state where the cycle is detected.

7.4 Symbolic AND/OR Graphs

AND/OR graphs are commonly used in AI to model problem reduction schemes [Nil80]. To solve a non-trivial problem, one decomposes the problem to a number of (smaller) subproblems. Successfully solving the subproblems will produce a final solution to the original problem according to the decomposing conditions. A simple example of such problem reduction is shown in Figure 7.2.

To play tennis we must have two conjunctive conditions (1) good weather and (2) court available. The problem “good weather” is considered to be an atomic subproblem here so we can treat it as a proposition. The other condition to play tennis is to have a court available and in this case we may have to choose between a public court and a private
court. Again, a public court is atomic. A private court can be decomposed into making a booking and paying the deposit. This is essentially similar to the process of natural deduction. Thus, we can say in order to solve “play tennis”, we need to solve the problems “good weather” and “have a court” and so on. Eventually we reach the terminal nodes that have the value true or false.

Formally, an AND/OR graph is a digraph $G = (V, E)$ defined as follows:

- A designated node $n_0 \in V$ is called the root.
- There exists a function $\zeta$ that maps each node in $V$ to a unique label in $\{\land, \lor, \top, \bot\}$
- Nodes with $\zeta(n) = \top$ or $\zeta(n) = \bot$ are called terminal nodes and have no outgoing edges to other nodes except themselves.

Given an AND/OR graph $G = (V, E)$, a solution graph $G_s = (V_s, E_s)$ is a digraph where:

- $V_s \subseteq V$, $E_s \subseteq E$ and $n_0 \in V_s$
- For each node $n \in V_s$, if $\zeta(n) = \lor$, then only one successor of $n$ is in $V_s$.
- For each node $n \in V_s$, if $\zeta(n) = \land$, then all successors of $n$ are in $V_s$.
- All finite paths must end with a $\top$-node and all infinite path must have an infinite number of $\top$-nodes as suffix.
The height $H(G_s)$ of a solution graph $G_s$ is the number of states of the longest prefix before the $\top$-node. In Figure 7.2 the AND/OR graph has two solution graphs, with heights 2 and 3 respectively, indicated by the dotted and dot-dashed polygons, and they correspond to the two possible solutions of the problem represented by the root node. The existence of the solution graph ideally captures the satisfiability of the property in the model. AND/OR graphs can be coupled with heuristic search algorithms to produce an efficient mechanism to solve the model checking problem [San03].

A symbolic algorithm to detect solution graphs

Algorithms for searching for a solution graph in an AND/OR graph has been studied at great length in AI [Nil80, MB85]. In general, these algorithms can be categorized as either top-down or bottom-up. Top-down algorithms construct the partial solution graph from the root and use bottom-up propagation to confirm the existence of the solution graph. Bottom-up algorithms construct the solution graph from the terminal nodes and the existence of the solution graph can be determined by checking the reachability of the root. Both these approaches are of course based on explicit-state representations. One of our goals is however to determine a solution graph of an AND/OR graph that is represented with BDDs. When the transition relation is represented by BDDs, it is convenient to compute the predecessor of a given node by computing its pre-image. Thus, our approach is in fact bottom-up, i.e. it constructs the graph and checks the reachability of the root in a single run.

Let $G = (S, s_0, R, L)$ be a Kripke structure that induces an AND/OR graph. Here $S$ is set of states, $s_0$ is the root, $R \subseteq S \times S$ is a transition relation and $L \subseteq S \times \{\land, \lor, \top, \bot\}$ is a labelling function. We require $L$ to be total, so all states in $S$ are labelled. We denote the set of states with label $\land$ by $S_\land$. Similarly for $\lor, \top$ and $\bot$. To compute the AND/OR graph symbolically, we need to compute the set of states $S_{sol} \subseteq S$ for all possible solution graphs of $G$, and then check whether $s_0 \in S_{sol}$. The set $S_{sol}$ is characterized by a fixed-point formula $S_{sol} = \mu Z . (S_\top \cup (EX(Z) \cap S_\lor) \cup (AX(Z) \cap S_\land))$.

**Theorem 7.1** $S_{sol}$ contains all states from all solution graphs induced by $G$.

**Proof.** The proof follows from the semantics of a fixed-point and the definition of a solution graph. The right-hand side of the fixed-point formula computes all states of any possible solution graph from the bottom up. It is easy to see the function of the fixed-point is monotonic. The computation starts with the set of states $S_\top$ that are true terminals.
The component \((EX(Z) \cap S_{\lor})\) adds all \(\lor\)-nodes into \(S_{sol}\) and \((AX(Z) \cap S_{\land})\) adds all \(\land\)-nodes into \(S_{sol}\). The convergence of the fixed-point guarantees all nodes of all possible solution graphs are computed.

One feature of our work is that the model-checking problem is transformed to a reachability problem. Our approach also generalizes the work in [SB04] and can deal with any type of property. Since the above formula characterizes a single fixed-point, the checking of the existence of the solution can be performed on-the-fly: i.e. in every iteration of the calculation we check whether \(s_0 \in Z\) twice after \((EX(Z) \cap S_{\lor})\) and \((AX(Z) \cap S_{\land})\). Of course, checking on-the-fly means that we can avoid searching the entire state space before we detect the minimum-height solution graph. In practice, if this optimal graph is a trace, it usually correspond to the shortest counter-example, which is highly desirable for diagnosis purposes of course.

7.5 From CTL Model Checking to a Reachability Analysis

In this work, we use the automata-theoretic framework to verify CTL formulae.\(^1\) The model-checking problem involves determining whether \(M, s_0 \models \varphi\), where \(M = (S, R, s_0)\) is a Kripke structure and \(\varphi\) a CTL property. In the automata-theoretic framework proposed in [KVW00], this problem is reduced to a language-emptiness checking problem. We also use this approach, but need to transform the language-emptiness checking to an AND/OR graph reachability analysis.

Let \(K = (S, R, s_0, L, AP)\) be a labeled Kripke structure where \(S\) is a set of states, \(R \subseteq S \times S\) is a transition relation, \(s_0 \in S\) is an initial state (in general a structure may have a set of initial states) and \(L \subseteq S \times AP\) is a labeling function. Let \((T_K, W_K)\) be the computation tree of \(K\), where \(W_K\) labels each node of the tree with a letter from \(2^{AP}\). Let \(\varphi\) be a CTL formula in positive normal form so that negations are pushed inside and placed before atomic propositions by De Morgan’s laws. For convenience, we use semantic equivalences during the normalization of the formulae to reduce the number of operators. Given a formula \(\varphi\), the closure of \(\phi\), written cl(\(\varphi\)), comprises of formulae that can be defined inductively as follows:

\(^1\)Our approach is equally applicable to other branching-time logics such as CTL\(^*\), alternation-free \(\mu\)-Calculus and full \(\mu\)-Calculus.
• \( \varphi \in \text{cl}(\varphi) \), true \( \not\in \text{cl}(\varphi) \) and false \( \not\in \text{cl}(\varphi) \).

• if \( \varphi_1 \land \varphi_2 \in \text{cl}(\varphi) \) or \( \varphi_1 \lor \varphi_2 \in \text{cl}(\varphi) \), then \( \varphi_1 \in \text{cl}(\varphi) \) and \( \varphi_2 \in \text{cl}(\varphi) \).

• if \( AX \varphi \in \text{cl}(\varphi) \) or \( EX \varphi \in \text{cl}(\varphi) \), then \( \varphi \in \text{cl}(\varphi) \).

• if \( A(\varphi_1 U \varphi_2) \in \text{cl}(\varphi) \), then \( \varphi_1, \varphi_2, \varphi_1 \land AX(A(\varphi_1 U \varphi_2)) \in \text{cl}(\varphi) \).

• if \( E(\varphi_1 U \varphi_2) \in \text{cl}(\varphi) \), then \( \varphi_1, \varphi_2, \varphi_1 \land EX(E(\varphi_1 U \varphi_2)) \in \text{cl}(\varphi) \).

• if \( A(\varphi_1 R \varphi_2) \in \text{cl}(\varphi) \), then \( \varphi_1, \varphi_2, \varphi_1 \lor AX(A(\varphi_1 R \varphi_2)) \in \text{cl}(\varphi) \).

• if \( E(\varphi_1 R \varphi_2) \in \text{cl}(\varphi) \), then \( \varphi_1, \varphi_2, \varphi_1 \lor EX(E(\varphi_1 R \varphi_2)) \in \text{cl}(\varphi) \).

The formulae in \( \text{cl}(\varphi) \) constitute the states of the alternating automaton \( \mathcal{A}_\varphi \) for the CTL formula \( \varphi \), and the initial state is \( \varphi \). Following convention, we denote the outermost temporal operator * as a *-formula. For example, \( A(false R (E(true U (¬p)))) \) is a R-formula and \( EXP \) is an X-formula. All R-formulae in \( \text{cl}(\varphi) \) are accepting states.

Let \( \mathcal{A}_\varphi = (2^{AP}, Q, q_0, \delta_\varphi, \mathcal{L}_\varphi, F) \) be an ATA for a CTL formula \( \varphi \). We now define the transition relation \( \delta_\varphi \). Note that our construction is different from the one in [KVW00] as we use a labeling function \( \mathcal{L}_\varphi \) to label each state of the ATA with a Boolean connective from the set \{\&, \lor, \top, \bot\}. This allows us to omit connectives in the labels of the transitions. We also use the assumption that if \( p \not\in \sigma \) then \( ¬p \in \sigma \). The transition relation \( \delta_\varphi \) and its corresponding labelling function \( \mathcal{L}_\varphi \) are defined as follows:

<table>
<thead>
<tr>
<th>( \psi )</th>
<th>( \delta_\varphi(\psi, \sigma, k) )</th>
<th>( \mathcal{L}_\varphi(\psi) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( p \in \sigma )</td>
<td>true</td>
<td>\top</td>
</tr>
<tr>
<td>( p \not\in \sigma )</td>
<td>false</td>
<td>\bot</td>
</tr>
<tr>
<td>( \varphi_1 \land \varphi_2 )</td>
<td>{\delta_\varphi(\varphi_1, \sigma, k), \delta_\varphi(\varphi_2, \sigma, k)}</td>
<td>&amp;</td>
</tr>
<tr>
<td>( \varphi_1 \lor \varphi_2 )</td>
<td>{\delta_\varphi(\varphi_1, \sigma, k), \delta_\varphi(\varphi_2, \sigma, k)}</td>
<td>\lor</td>
</tr>
<tr>
<td>( AX \varphi )</td>
<td>{(0, \varphi), (1, \varphi), \ldots, (k - 1, \varphi)}</td>
<td>&amp;</td>
</tr>
<tr>
<td>( EX \varphi )</td>
<td>{(0, \varphi), (1, \varphi), \ldots, (k - 1, \varphi)}</td>
<td>\lor</td>
</tr>
<tr>
<td>( A(\varphi_1 U \varphi_2) )</td>
<td>{\delta_\varphi(\varphi_2, \sigma, k), \delta_\varphi(\varphi_1 \land AX(\varphi), \sigma, k)}</td>
<td>\lor</td>
</tr>
<tr>
<td>( E(\varphi_1 U \varphi_2) )</td>
<td>{\delta_\varphi(\varphi_2, \sigma, k), \delta_\varphi(\varphi_1 \land EX(\varphi), \sigma, k)}</td>
<td>\lor</td>
</tr>
<tr>
<td>( A(\varphi_1 R \varphi_2) )</td>
<td>{\delta_\varphi(\varphi_2, \sigma, k), \delta_\varphi(\varphi_1 \lor AX(\varphi), \sigma, k)}</td>
<td>\lor</td>
</tr>
</tbody>
</table>

Following [KVW00], the product \( K \times \mathcal{A}_\varphi \) of the labelled Kripke structure \( K = (S, R, s_0, L, AP) \) and property \( \mathcal{A}_\varphi = (2^{AP}, Q, q_0, \delta_\varphi, \mathcal{L}_\varphi, F) \) is defined as a 1-letter alternating word automaton \( \mathcal{A}_{K, \varphi} = (\{a\}, S \times Q, (s_0, q_0), \delta, \mathcal{L}, S \times F) \). If the property transition \( \delta_\varphi(q, L(s), k) = \theta \) and the Kripke transition \( R(s) = t_0, \ldots, t_k \) then the transition relation
and labeling function of the word automaton are \( \delta((s, q), a) = \theta([c, q]/\langle t_c, q' \rangle) \) for each \( c \) and \( L((s, q)) = L_\varphi(q) \).

Having defined the product automaton, we now introduce the implementation of state recording mechanism formally. We use a 3-state automaton to guess the beginning of the cycle. We also use a 2-state automaton to indicate whether the state has been recorded. The essence of state recording is that accepting (rejecting) cycles are flattened, resulting in a product automaton that is a well-defined AND/OR graph. Thus, the model-checking problem is transformed to the problem of determining the existence of a solution graph in an AND/OR graph. The reachability algorithm of Theorem 7.1 can be used to determine the existence of a solution graph and thereby solve the model-checking problem.

Let \( S_r = S \) be a set of states that we use to save a copy of a state in \( K \) when necessary. Two small Kripke structures \( K_\varsigma \) and \( K_\chi \) are used to determine when the state recording is necessary, as follows:

\[
\begin{array}{c|c|c|c|c}
K_\varsigma = (S_\varsigma, s^0_\varsigma, R_\varsigma) & S & s^0 & R \\
\hline
\{T, L, R\} & T & R_\varsigma(T) = \{L, R\}, R_\varsigma(L) = L, R_\varsigma(R) = T \\
\{0, 1\} & 0 & R_\chi(0) = 1, R_\chi(1) = 1
\end{array}
\]

The product of \( A_{K,\varphi}, K_\varsigma \) and \( K_\chi \) after translation is an automaton \( A = (\{a\}, S, \langle s_0, s_0, q_0, T, 0 \rangle, R, L) \) where \( S = S \times S_r \times Q \times S_\varsigma \times S_\chi \). The transition relation \( R \) is defined as follows:

- \( R((s, s_r, q, s_\varsigma, s_\chi), a) = R_\varsigma(s_\varsigma) \times \langle s, s_r, q, s_\chi \rangle \) if \( s_\varsigma = T \), \( s_\chi = 0 \) and \( q \) is an accepting state (guess the beginning of a cycle), or
- \( R((s, s_r, q, L, 0), a) = \langle s, s, q, L, 1 \rangle \) (state recording), or
- \( R((s, s_r, q, s_\varsigma, s_\chi), a) = R(s) \times \langle s_r \rangle \times \delta(q, a) \times R_\varsigma(s_\varsigma) \times R_\chi(s_\chi) \).

The corresponding labeling function \( L \) is defined as:

- \( L((s, s_r, q, T, 0)) = \top \) if \( q \) is an accepting state, or
- \( L((s, s, q, L, 1)) = \top \) if \( q \) is an accepting state, or
- \( L((s, s_r, q, s_\varsigma, s_\chi)) = L_\varphi(q) \).

The labeling of states in \( A \) follows the labeling of states in \( A_{K,\varphi} \) except that when we need to guess the beginning of an accepting cycle or we have found an accepting cycle.

**Lemma 7.2** The graph \( G_A \) induced by \( A \) is a well-defined AND/OR graph.
Proof. It follows directly from the definition of transition relation that $A_\varphi$ is a well-defined AND/OR graph. The problem is that the product $A_{K,\varphi}$ will not be well-defined due to the presence of accepting (and rejecting) runs. When the state recording automata $K_\chi$ and $K_\chi$ are used, the accepting (rejecting) runs are in essence flattened, changing from cycles to finite traces. The beginning of an accepting cycle in $A$ is a $\lor$-node and another copy, which indicates the existence of the cycle, is a $\top$-node. Thus, the product $A$ induces a well-defined AND/OR graph.

\textbf{Theorem 7.3} $G_A$ has a solution graph iff the language of $A_{K,\varphi}$ is non-empty, i.e $K \models \varphi$.

Proof. The key to the proof is that our state recording automata are capable of catching all possible accepting (rejecting) cycles. Since we do not know in advance which state will be at the beginning of a cycle, we use in $K_\chi$ a non-deterministic transition $R_\chi(T) = \{L, R\}$ for every accepting (rejecting) state to guess the starting point. Using our construction rules, this state is labeled as a $\lor$-node. This consideration ensures every possible accepting (rejecting) cycle will be caught. Hence, the existence of a solution graph in $G_A$ confirms non-emptiness of the language of $A_{K,\varphi}$ and therefore $K \models \varphi$.

\section{7.6 Our Approach}

In essence, our approach, illustrated in Figure 7.3, uses BDD-based algorithms in an automata-theoretic framework to solve the language non-emptiness problem for branching-time logic. The process begins with the model, represented by a Kripke structure, and the property to be verified, expressed as a CTL formula. Unlike conventional model checking, we also require a state recording automaton as input. In the first step, the CTL formula is translated to a (weak) alternating tree automata. Then a product automaton for the three inputs is constructed and an AND/OR graph is generated. To determine whether the AND/OR graph has a solution graph, we use a special reachability analysis algorithm. If this algorithm finds a solution graph, the property is verified and the solution graph is a witness of the property. Otherwise, the property is violated. To determine a counter-example, one needs to determine a witness of the negation of the property. The reachability analysis in the AND/OR graph uses BDDs. We encode the system model, property automaton and state recording automaton as BDDs, and compute the product automaton and AND/OR graph as BDDs.
From an ATA to AND/OR graphs using BDDs

The key to automata-theoretic branching-time model checking is to determine the language non-emptiness of the product alternating automaton. Kupferman et. al. [KVW00] reduce this problem to a 1-letter automaton non-emptiness checking. While their algorithm uses AND/OR graphs as underlying data structures, the run of the 1-letter automaton does not correspond to an AND/OR graph as we defined above. The serious problem concerns cycle detection. When their algorithm first encounters an accepting (or rejecting) state, it may label the state a $\land$-node or $\lor$-node. If subsequently a cycle that goes back to an accepting (or rejecting) state is found, then the same node may be labeled $\top$ or $\bot$. If this occurs, the state does not have a unique label. In a explicit-state model checker, this problem can be avoided by using a stack to memorize the accepting cycle and if the cycle does exist, the algorithm just relabels the state. This does not work in a BDD-based model checker because states are represented implicitly and path information cannot be recorded by using a structure like a stack.

Example

Consider the Kripke structure and CTL property in Figure 7.4. The structure has 4 states and $s_3$ is labelled by the only atomic proposition $p$ (assume $\overline{p}$ labels all other states). The property is obviously true in this model.

To use the algorithm in [KVW00] to verify the property, we first construct the alternating automaton of $AG(\text{EF} p)$ as follows:
where q is the state of the property automaton, the state $AG(EFp)$ is an accepting state, and the other state is a rejecting state. The accepting state of the product automaton is determined by the property automaton. Note that if a transition has the value true, this means the path leading to that state does not have a successor in a run of the tree automaton. This corresponds to $\top$-node in AND/OR graph. If the transition has the value false, the run can never be accepted, i.e. the solution graph can never have a $\bot$-node.

We omit the formal description here of the product automaton and instead depict in Figure 7.5 the runs that the algorithm in [KVW00] uses to check the non-emptiness of the language. Each node in the graph is a state of the product automaton. The initial state of this automaton contains both the initial states from the Kripke structure and the property automaton. The algorithm begins with the initial state #1, which states that $AG(EFp)$ is true at state $s_0$. This means that $EFp$ must be true at $s_0$, which results in node #2, and $AG(EFp)$ must be true at all successors of $s_0$, namely $s_1$, which results in node #3. Node #1 is hence a $\land$-node. Other states are expanded in a similar manner. The transition true results in node #7, which is a $\top$-node. At this point we need to back-propagate the label according to the definition of the solution graph of an AND/OR graph. Since node #4 is a $\lor$-node and one of its children is a $\top$-node, node #4 will be labeled as a $\top$-node as well. The language is non-empty iff the initial state (#1) is labeled a $\top$-node. The problematic state in this example is node #6 because one of its successors loops back to a state that was expanded before. In this situation, the label of node #1 will be recursively dependent on itself (because of the cycle of node #1, #3 and #6). This problem occurs when an automaton has an accepting or rejecting run. In fact, it is easy to see the cycle here is an accepting cycle as #1 is an accepting state. To avoid this problem, if during the back-propagation of the labels a successor of a node leads to an accepting cycle, we consider the node has a $\top$-node as successor. Thus, node #6 is a $\top$-node as #7 is a $\top$-node and node #1 leads to an accepting cycle. Alternatively, if a successor of a node leads to a rejecting cycle, we consider the node has a $\bot$-node as successor. It is easy to see #1
will eventually be labeled as a $\top$-node and the language of the product automaton is not empty.

In an explicit-state model checker, the non-emptiness algorithm above detects accepting and rejecting cycles by maintaining two stacks. Our aim is to use BDD-based algorithms to check for language non-emptiness, and to represent the model under verification and property using BDDs. In general, BDD-based state-space search does not use a stack to record which states have been visited, so the above approach cannot be used directly. Instead, state recording has been used. The key idea is to use another set of state variables to record the state that possibly lies in a cycle. The existence of the cycle can be determined by checking whether the value of original state variable is equal to the recorded variable. In effect, state recording emulates a stack, and cycles can be detected by reachability algorithms.

We use this approach to detect both accepting cycles and rejecting cycles. Figure 7.6 depicts the search graph that uses this technique. In Figure 7.5 the state of the product automaton has two components, namely a state of the Kripke structure and a state of the property automaton. After recording the state, the state of the product automaton has 5 components: namely $(s, t, \varphi, save, saved)$, where $s$ is the state of the Kripke structure, $t$ is the state that copies $s$ when necessary, $\varphi$ is still the state of property automaton, and finally $save$ and $saved$ are two variables that are used to determine when to record a state and whether the state has already been recorded. To detect an accepting cycle, it is sufficient to record the state at an accepting state. For example, in Figure 7.6, node #1 is an accepting state and hence $save$ will non-deterministically split into two branches. The value for $save$ becomes $L$ in #2 and $R$ in #3. If $save = L$, we will copy $s$ to $t$ in the next transition and set the flag $saved$ to be 1. Note that the non-deterministic branch leaves

Figure 7.5. Automata-theoretic CTL model checking
open the question whether that state is the beginning of an accepting cycle, so the label of this ‘split’ node is \( \lor \). While the graphs in Figure 7.5 and Figure 7.6 are similar, the latter contains more nodes. In particular, the shaded node #11 suggests that there is an accepting cycle. Node #11 turns out to be a \( \top \)-node.

By applying state recording, the product automaton becomes an AND/OR graph in which the label on each state will only depend on its own position. Using our symbolic reachability algorithm, we can determine whether there exists a solution graph and thereby determine the emptiness of the language of the product automaton. The SMV code for this example can be found in Appendix B.

### 7.7 Experiments

We have implemented this approach in the symbolic model checker NuSMV [CCGR99]. The reachability algorithm of the fixed-point formula has been implemented in a separate module and uses both weak pre-image and strong pre-image routines from NuSMV for the temporal operators \( EX \) and \( AX \). We employ most of the features of BDD-based model checking, such as input variable ordering and a partitioned transition relation.

The model under verification is expressed using the SMV input language. We require the input model to our algorithm to be a translated automaton that induces an AND/OR graph. This requires a source-to-source translation from the original model and property to a new input model. To produce the experimental results below we needed to do this translation by hand, but we are developing a tool that can perform this automatically.
The model we experimented on is a gigamax cache consistency protocol. The property we verified is $AG(\text{EF}p_0.\text{readable})$, which states that no matter what state the system is in, there is a future state from which process $p_0$ is readable. The property can be verified by both our automata-based approach and the CTL model checking algorithm based on a fixed-point characterization. A few preliminary results are shown in the following table.

<table>
<thead>
<tr>
<th>algorithm</th>
<th># BDD vars</th>
<th>run-time (s)</th>
<th># BDD nodes</th>
<th>iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td>our approach</td>
<td>188</td>
<td>1.370</td>
<td>425748</td>
<td>4</td>
</tr>
<tr>
<td>standard</td>
<td>88</td>
<td>0.250</td>
<td>65947</td>
<td>11</td>
</tr>
</tbody>
</table>

As we need to use a set of variables to save a copy of the original states, our approach uses more than twice the number of Boolean variables than the standard algorithm. Specifically, we use another 88 variables to copy the state, and 12 variables to encode the property automaton as well as the state recording automata. The net result is a longer run-time and more BDD nodes (and hence memory). The last column shows the number of iterations of the fixed-point calculation. This is a measure of how many times the algorithm calls the pre-image (or image) computation routine, which is normally an expensive operation in symbolic model checking. In this example we call only 4 pre-image operations, which is substantially lower than the number called by the standard algorithm.

### 7.8 Summary

In this chapter we have transformed a branching-time logic problem into a reachability problem on AND/OR graphs and placed it in a symbolic setting. The big advantage of this transformation is that we are now in a position to apply symbolic heuristic-search algorithms during the reachability analysis to reduce the size of the state space. We already have extensive experience with heuristic search [QN04b, QN04a] and feel that the combination of symbolic and heuristic search techniques offers much potential. This work, together with the work of Kupferman et. al. [KVW00], has highlighted the need for really efficient (heuristic) search algorithms to solve what is a quint-essentially AI problem representation, AND/OR graphs.

It is true that the above approach will lead to worse performance because of the extra load of the state recording. Note that this is consistent with Schuppan and Biere’s results in [SB04]. However, in the longer term, the advantage is that, having transformed branching-time model checking into a reachability analysis, an arsenal of optimisation techniques can be applied to improve performance. While we do not explore these techniques in this work, this is future work.

130
While we have focused on CTL model checking, the approach used in this paper is applicable to other branching-time logics such as CTL* and \( \mu \)-Calculus. In our algorithms we have used Büchi acceptance conditions. In fact, the automata constructed from CTL satisfy ‘weak’ acceptance conditions and hence these automata are often referred to as weak alternating automata (WAA). In their automata-theoretic approach, Kupferman et. al. [KVW00] construct hesitant alternating automata (HAA) for CTL* and \( \mu \)-Calculus.
Chapter 8

The GOLFER Tool

8.1 Overview

GOLFER is a symbolic model checker that implements the heuristic model checking algorithms that were described in Chapter 3-7. The aim of these algorithms is to:

• avoid exhaustive model checking algorithms and find alternatives that are more scalable to large models,
• automatically generate heuristics for directed model checking, and improve the usability of model checking, and
• make directed model checking applicable to both safety and liveness properties.

The infrastructure of GOLFER for manipulating BDDs, FSMs and I/O is based on NuSMV [CCGR99]. The tool is implemented as a separate package that is interfaced to the main procedures of NuSMV. The functionalities of NuSMV are therefore not affected and can be invoked just as before. The package is invoked by a command-line option of the executable command. A user-crafted abstraction is read via an input file called an abstraction file. This file usually specifies a list of variables (or bits of a Boolean encoding). The variable ordering of the BDDs works in the same way as in NuSMV. The modeling language is the SMV language as defined in [McM93]. Other NuSMV features such as dynamic variable ordering, cone of influence reduction and transition partition also works for GOLFER.
8.2 Architecture

The diagram in Figure 8.1 depicts the architecture of GOLFER. The GOLFER package is indicated by a module that includes 6 smaller modules. The module at the very bottom represents the NuSMV package and the double-headed arrow between them represents the interface between the two packages. The interface between GOLFER and NuSMV is used when GOLFER needs support for managing BDD operations, FSM manipulations and I/O operations. GOLFER accepts two input files and produces one output as indicated in the diagram by three document-shaped boxes and signal-headed arrows. This architecture reflects our aims in this thesis and implements our approach depicted in Figure 1.1 in Chapter 1.

![Figure 8.1. The architecture of GOLFER](image)

Inside GOLFER there are 6 modules. Between these modules, a number of interfaces are indicated. A brief description of each module and a reference to the algorithm it implements are given below.

**Model transformation unit** is a source-to-source transformer that takes an arbitrary CTL property as input and generates a new model with a reachability property. The principle of the transformation is based on the state-recording mechanism [SB04]. The method was illustrated in Figure 7.1 and 7.6. The theoretical work and related algorithms were described in Section 7.4 and 7.5. Note that if the property specified in the input model is a reachability property, the tool actually bypasses this module.
Data dependency analysis is responsible for creating an abstraction function. The function used to derive an abstract model can be either an input from the user (see e.g. in Chapter 4) or generated automatically by using an algorithm described in Section 5.3. The data dependency analysis module takes the model and analyze the the variable dependency in the model and outputs an abstraction function. The main idea is based on a notion that we call variable strength. In the analysis, the ‘weak’ variables are identified and output to the abstraction file.

Abstractor computes the abstract model according to the abstraction function that is either input by the user or the output from the data dependency analysis module described above. The abstractor can build the exact abstraction that is defined in Definition 4.3 of Chapter 4. When the exact abstraction of a model is expensive to build, the abstractor can choose to approximate the abstraction. The approximation algorithm was described in Section 5.3.2 of Chapter 5.

Heuristic synthesizer is the module responsible for constructing the abstraction database heuristic for our directed model checking algorithms. The basic idea for constructing the heuristic is to use a brute-force algorithm to explore the abstract model and store all abstract states and their corresponding distance (number of transitions) to the abstract error state. Two methods were described in Section 5.3.3 of Chapter 5 to construct the heuristic.

Abstract model checking implements the standard BFS-based invariant model checking algorithm, InvarCheck, in Figure 4.4 of Chapter 4. This module can output a positive indication to the user if the abstract model satisfied the property. The verification process stops if the abstract model satisfies the property as our abstraction method preserves the property. The verification of the abstract model induces the verification of concrete model.

Directed model checking is the core module of GOLFER. It includes the implementation of BDD-based heuristic algorithm in Figure 3.6, the DirectedInvarCheck algorithm in Figure 4.6, the BDD-IDA* algorithm in Figure 5.1 and the SAG algorithm in Figure 6.5. The positive output indication from this module has the same meaning as the output from the abstract model checking, meaning that the concrete model has been verified. The negative output indication suggests that the concrete model cannot be verified. To help the user diagnose the cause, this module also outputs the counter-example.

The current implementation of GOLFER includes most algorithms described above except for the model transformation algorithm. Automatic model transformation for an
arbitrary CTL property involves intensive coding and engineering work. Hence, the implementation of the transformation algorithm is out of the goal of this thesis. We endeavor to include the implementation in the future version of GOLFER.

8.3 Usage

8.3.1 Property specification

GOLFER uses the SMV [McM93, CCGR99] language as the model specification language. Readers who are not familiar with this language are referred to the NuSMV user manual. To illustrate the usage of GOLFER, we consider the following simple model along with two properties. We assume the model is stored in a file called example.smv.

```plaintext
MODULE main
VAR
  s : {s0, s1, s2, s3};
ASSIGN
  init(s) := s0;
  next(s) := case
    s=s0 : {s1};
    s=s1 : {s2, s3};
    s=s2 : {s3};
    s=s3 : {s0};
  esac;
SPEC
  AG !p   -- NuSMV property
  AG(EFp) -- NuSMV property
DEFINE
  p := (s=s3)
```

This model specifies a simple 4-state transition system depicted in Figure 8.2. The first property, $AG \neg p$, is a safety property stating that the proposition $p$ is never true in all reachable states. The second property, $AG(EFp)$ is a liveness property stating that whatever state the system is in, it should be able to get to a state where $p$ is true. The property specification above is the standard NuSMV format. The format used by GOLFER is slightly different.

1http://nusmv.irst.itc.it/NuSMV/userman/index-v2.html
For safety properties, we always use the format \texttt{INVARSPEC (proposition)}. Therefore, the safety property in the example would be specified as:

\texttt{INVARSPEC !p -- GOLFER property}

For liveness properties, we first need to transform the CTL property to an automaton and specify the property as a reachability property in an AND/OR graph. The reachability analysis algorithm in Section 7.4 requires three sets of states, $\top$, $\lor$- and $\land$-states. These sets are labeled by our transformation that is described in detail in Appendix B. In \texttt{GOLFER}, the liveness property is therefore expressed by listing these three sets. We use the keyword \texttt{AO} to indicate that the property is a \texttt{GOLFER} liveness property:

\texttt{AO \{true\_states & or\_states & and\_states\} -- GOLFER property}

where \texttt{true\_states}, \texttt{and\_states} and \texttt{or\_states} correspond to $\top$, $\lor$- and $\land$-states.

### 8.3.2 Input abstraction

The function that \texttt{GOLFER} uses for computing the abstract model can be input to the abstractor module via an explicit file. We call this file an \textit{abstraction file} and it has the extension \texttt{.abst}. The abstraction file basically specifies those variables that are not (or less) relevant to the property. Remember that this does not need to be accurate as we only use that abstraction to direct the heuristic algorithm. The variables specified in the abstraction file can be Boolean variables, scalar variables and Boolean bits of a scalar variable. The syntax of specifying the bits of a scalar variable is similar to specify
an input variable ordering in NuSMV. Take for example the model in `example.smv`. If we want to abstract the model to be only two states, we could craft an abstraction file named `example.abst` that consists of a single line (note that lines commencing with `--` are comments):

```
-- example.abst
-- abstraction file for example.smv
--
s.0
```

Since there is only one variable in the model, we can abstract bits of only that variable away. The variable `s` has 4 states and hence is encoded using only 2 bits. In the abstraction file, `s.0` means the abstractor will abstract away the first bit of the variable, and therefore the abstract model has only 2 states. We could of course use `s.1` to abstract away the second bit. If no bits are specified, the variable will be abstracted away completely.

### 8.3.3 Automatic abstraction

In GOLFER the user has an option to use the built-in automatic abstraction algorithm (described in Section 5.3). In this case, GOLFER will construct a variable dependency graph. The dependency graph is an acyclic directed graph (DAG) with the roots being the variables in the specified property. Figure 8.3 depicts a dependency graph for a model that contains 11 variables. Using the command-line option `autogen` will generate a file to the standard output that represents the dependency graph. The file for the graph in Figure 8.3 is as follows:

```
-- root layer
s2
s5
--
-- first layer
s0
s3
s7
s10
--
-- second layer
s1
```
The user needs to redirect the standard output to a file. It is up to the user to decide how to form the abstraction file. The general practice is to remove all layers except the last one as the bottom layer of variables are considered to be ‘weak’. Hence, we could create an abstraction file as follows.

------------------------------------
-- example2.abst
-- abstraction file for example2.smv
------------------------------------
-- third layer
s6
s9
--

![Dependency Graph](image)

Figure 8.3. An example of dependency graph
8.3.4 User commands

The executable is still called NuSMV and GOLFER is invoked by the command-line option \(-golfer\) to NuSMV. The user can still specify the standard NuSMV options. However, all parameters that occur after the \(-golfer\) option are considered to be options for GOLFER. A series of examples are presented first in this section and followed by the synopsis and a full description of the commands. In each example, the command-line option is followed by an explanation of the meanings of each option. Options that are repeated in more than one example are only explained once.

Example 8.3.1 Command:

\[
\text{#NuSMV } -\text{golfer a 50 s ex.abst ex.smv}
\]

- \(-\text{golfer}\) the option to invoke the GOLFER tool
- \(a\) to invoke the DirectedInvarCheck algorithm
- \(50\) the depth bound for the DirectedInvarCheck
- \(s\) using single abstraction for directed model checking
- \(\text{ex.abst}\) input abstraction file
- \(\text{ex.smv}\) input file of the model and property

Example 8.3.2 Command:

\[
\text{#NuSMV } -\text{golfer a 50 m 3 ex1.abst ex2.abst ex3.abst ex.smv}
\]

- \(m\) using multiple abstractions for directed model checking
- \(3\) to specify the number of abstractions
- \(\text{ex1-3.abst}\) a list of input abstraction files

Example 8.3.3 Command:

\[
\text{#NuSMV } -\text{golfer autogen ex.smv } > \text{ ex.abst}
\]

- \(\text{autogen}\) automatic abstraction using data dependency analysis
- \(\text{ex.abst}\) redirected output from the automatic abstraction

Example 8.3.4 Command:

\[
\text{#NuSMV } -\text{golfer sag 3 ex1.abst ex2.abst ex3.abst ex.smv}
\]
• sag to invoke the static abstraction guided (SAG) algorithm
• 3 to specify the number of ordered abstractions
• ex1-3.abst a list of ordered abstraction files

Example 8.3.5 Command:

```
#NuSMV -golfer ctl ex.smv
```

• ctl to invoke the AND/OR reachability algorithm

Below we reproduce the manual entry for the GOLFER interface to NuSMV.

SYNOPSIS

```
#NuSMV [nusmv_options] -golfer golfer_options [input.abst ...] input.smv
```

golfer_options: [ctl] [autogen] [a num/ida num/sag num] [s/m num]
[mta/ata] [fadb/badb] [heu u w]

DESCRIPTION

nusmv_options
the user can specify any standard NuSMV option, such as partitioned transition relation and input variable ordering

ctl
switch to model transformation mode to check a general CTL property;
if not specified, the tool only accepts invariant property;
the algorithm was described in the section titled "From CTL Model Checking to A Reachability Analysis" in Chapter 7

autogen
automatically generates an variable dependency graph; the algorithm was shown in section ‘‘Automatic Abstraction’’ in Chapter 6

a num
invokes the DirectedInvarCheck algorithm in Chapter 4; num specifies the depth bound for the algorithm

ida num
invokes the BDD-IDA* algorithm introduced in Chapter 5; num specifies the depth bound for the algorithm
sag num
invokes the SAG algorithm described in Chapter 6; num specifies
the number of ordered options; note the abstractions should be
ordered with the most abstract one being the first and the concrete
model the last.

s
indicates only a single abstraction is used, so only one .abst file
is followed

m num
indicates multiple abstractions are used; the num should be equal
to the number of .abst files followed

mta
exact abstraction is computed according to the abstraction file

ata
approximation of the abstraction is computed; the algorithm for
approximation can be found in the section ‘‘Approximation of
Abstraction’’ of Chapter 6.

fadb
invokes the forward abstraction database algorithm

badb
invokes the backward abstraction database algorithm; this is the
default

h u w
this is used when the user prefers to use weighted heuristic
search; the estimation function of the weighted algorithm is
f = u*g + w*h where u and w are weights for the real cost g and
heuristic estimation h; the default value for both is set to 1
Chapter 9

Conclusion

9.1 Summary

Formal verification has attracted much attention from both academia and industry over recent years. Formal verification is a field in verification engineering that is more thorough and rigorous than simulation-based methods. The work in this thesis contributes a novel formal-verification methodology that combines the power of several existing techniques, and develops them further to produce a new verification methodology that we call the abstraction-directed verification framework. This framework consists of three related aspects that we summarize as follows.

Directed error detection

Directed error detection in formal verification is concerned with using heuristic search algorithms for detecting flaws in a model. In Chapter 3 we have developed a heuristic search algorithm that is based on a symbolic data structure. This algorithm, SA*, shown in Figure 3.6, re-implements the conventional explicit-state A* algorithm using BDDs. Unlike other symbolic implementation of A*, SA* uses a set of BDDs to represent the heuristic function. The set of BDDs not only provides a distance metric for a set of states, but also partitions the frontier BDD which usually has a large size. This is essential in model checking as it avoids computations involving very large BDDs. Our results have shown that SA* algorithm has the ability to cluster a set of states and speed up the search when the heuristic is weak (see e.g. Figure 3.12). The DirectedInvarCheck algorithm in
Figure 4.6 is a modified version of SA* and can handle multiple heuristic functions. The BDD-IDA* algorithm in Figure 5.1 is a BDD-based IDA* algorithm. When coupled with heuristics, these algorithms are able to effectively locate flaws in models of systems (see e.g. Table 4.2 and Table 5.2).

**Abstraction-based heuristics**

For AI search problems, deriving a heuristic remains an ad hoc process. In directed error detection, we need a systematic way to derive these heuristics as an ad hoc process reduces the usability of model checking. In this thesis, a connection between abstractions and heuristics has been established through a mechanism called abstraction databases that were described in Chapter 4. A hand-crafted abstraction function is input to the model checker and a heuristic function is generated automatically. Our results show that for many models that contain an error, a single abstraction is enough to speed up the heuristic search and find the error in a shorter time than the naive algorithm. The improvement is up to 2 orders of magnitude (see Figure 4.2). Multiple abstraction databases can also be used for error detection in model checking. The gain achieved by using multiple abstractions varies according to the class of problem. AI search problems tend to benefit more from using multiple abstractions as they tend to have a better distribution (i.e. variation) in heuristic values (see the tables in Section 4.6.5).

To preserve the push-button usability of model checking, we further developed an automated heuristic synthesis process that generates an abstraction function. By applying this technique, our directed model checking behaves just like standard model checking. There is no need for the user to craft the abstraction function manually. The results in Table 5.2 have shown that the automatically generated abstraction is almost as good as hand-crafted ones in Table 4.2.

A generalization for applying abstraction-based heuristics on multiple level of abstractions has been discussed in Chapter 6. Multiple abstractions are ordered by a relation called simulation. The abstraction-guided search is carried out iteratively from the most abstract model to the concrete model. Each error in an abstract model represents a region where a real error may reside. This region gets smaller and smaller when search is conducted along the abstraction order. The major benefit of this technique is that the “safe” portion of the state space (that will not contain an error state) is determined at the abstract level and therefore need not to be searched. This could turn many intractable problems into solvable problems.
Model transformation

A major limitation of directed error detection is that it is effective for checking reachability of state-based properties only. This is because heuristic search algorithms such as A∗ are not suitable for detecting violations of transition-based properties. For example, liveness properties in model checking are usually transition-based properties. The violation of liveness properties can be captured by a cycle in the state space. In this situation, the property and model need to be transformed into a model that has a reachability property. We have described a transformation algorithm in Chapter 7 that converts branching-time model checking into a reachability analysis. Our technique is based on state recording and a symbolic AND/OR search algorithm. While the conversion involves more BDD variables and increases the state space, our technique uses fewer iterations (see the tables in Section 7.7). This indicates that, although the transformed model has a larger state space, the error can be detected in fewer iterations.

9.2 Conclusion

A model checker can be used in practice as a debugger to detect errors. In contrast to simulation techniques, model checkers explore the state space of a model in a systematic way. In this work, heuristic search techniques have been applied to symbolic model checking. As well as the savings made by using BDDs, heuristic search further reduces the search space when detecting an error. Homomorphic abstractions are used as a relaxation of the concrete system. Hence, the heuristic function can be built on top of these abstractions. The abstraction mechanism used in this work demonstrates a systematic way to create search heuristics for model checking. More importantly, our automated heuristic synthesis removes the burden of manual abstraction from users. This maximizes the usability of the abstraction-directed technique for error detection. The model transformation is able to convert full CTL model checking to a reachability analysis. This is an important step towards applying similar abstraction-directed technique to AND/OR graph search. In principle, this makes the abstraction-directed technique capable of handling full CTL formulas.

9.3 Future Work

Three possible directions of future work are:
• New techniques are needed to come up with abstraction that contains more accurate information, e.g. SAT-based abstraction techniques.

• Our abstraction-based framework and the CEGAR framework can be integrated.

• The GOLFER tool needs further development.

We explain these direction below.

9.3.1 SAT-based automated abstraction

The automated heuristic synthesis in Chapter 5 uses a data dependency analysis. In this analysis, less dependent variables are considered weak variables that can be removed from the abstraction. This technique does not work in a situation where all variables have similar dependencies. In such cases, an alternative technique is required. McMillan and Amla [MA03] proposed a technique called proof-based abstraction. It uses a SAT procedure to determine the unsatisfiable core, which is an abstraction. The process of generating this core is in fact a process that eliminates the irrelevant information from the original model. Hence, this method could also be used in our abstraction-guided error detection. It could determine a more accurate abstraction and this will result in a more precise heuristic function (possibly in general).

9.3.2 Integration with CEGAR

While our technique is different to CEGAR in many respects, the two frameworks can be integrated for their mutual benefit. For example, heuristic search algorithms can be used during the refinement process to eliminate spurious counter-examples. As well, the refinement process can be called when the abstraction is too coarse. When an abstraction is too coarse, the heuristic function created from it would not be effective. Refining the model to another abstract level may help create a more precise heuristic and hence speed up the error detection as well.

9.3.3 Tool development

The GOLFER tool remains an experimental tool for evaluation purposes. To make the tool more useful, the following issues need to be addressed.
1. The model transformation needs to be automated in the tool.

2. Abstraction-directed heuristic search needs to be applied to AND/OR graph search for checking general CTL properties.

3. The interaction with the tool needs to be user-friendly (and not command-line driven). A GUI is preferable for novice users.
Bibliography


Appendix A

List of Publications


Appendix B

Model Transformation: An Example

Original model in SMV language before transformation is as follows.

MODULE main
VAR
    s : {s0, s1, s2, s3};
ASSIGN
    init(s) := s0;
    next(s) := case
        s=s0 : {s1};
        s=s1 : {s2, s3};
        s=s2 : {s3};
        s=s3 : {s0};
    esac;
SPEC
    AG(EF s=s3) -- NuSMV property

The transformed model is illustrated below. Note that it has more lines of code than the original one. This is mainly because of the alternating automaton for AG(EF s=s3). In practice, the code for the automaton is usually much less than the model. Hence, the transformed model may not look much different to the original.

MODULE main
VAR
    s : {s0, s1, s2, s3};
ASSIGN
    init(s) := s0;
    next(s) := case
        s=s0 : {s1};
        s=s1 : {s2, s3};
        s=s2 : {s3};
        s=s3 : {s0};
    esac;

case
    s=s0  : {s1};
    s=s1  : {s2, s3};
    s=s2  : {s3};
    s=s3  : {s0};
esac;

-- alternating tree automaton for AG(EF s=s3)
VAR
    p: {ag_ef, xag_ef, ef, xef, atom_p};
ASSIGN
    init(p) := ag_ef;
    next(p) := case
        split | true_states : p;
        p=ag_ef : {ef, xag_ef};
        p=xag_ef : ag_ef;
        p=ef : {atom_p, xef};
        p=xef : ef;
        1 : p;
esac;

VAR
    save: {rt, l, r};
    saved : boolean;
ASSIGN
    init(save) := rt;
    next(save) := case
        split : {l, r};
        save=r : rt;
        1 : saved;
esac;

VAR
    s_s : {s0, s1, s2, s3};
ASSIGN
    init(s_s) := s0;
    next(s_s) := case
        save=l & !saved : s;
        1 : s_s;
esac;
DEFINE
-- state transition of original model
move := p=xag_ef | p=xef;
split := p=ag_ef & save=rt;
-- define the AND/OR graph
true_states := (p=ag_ef & s=s_s & saved) | (p=atom_p & s=s3);
or_states := (p=ag_ef & save=rt & saved=0) | p=xef | p=ef;
and_states := (p=xag_ef | (p=ag_ef & !save=rt)) & !true_states;
SPEC
AO (true_states & or_states & and_states) -- GOLFER property