Guided Random-Walk Based Model Checking

BUI HOAI THANG

A thesis submitted in fulfilment
of the requirements for the degree of
Doctor of Philosophy

from

School of Computer Science and Engineering
THE UNIVERSITY OF
NEW SOUTH WALES

SYDNEY · AUSTRALIA

March 2009

To my wife, Kieu Oanh, our son, Nhat Huy, and daughter, Nhat Vy.

ORIGINALITY STATEMENT

'I hereby declare that this submission is my own work and to the best of my knowledge it contains no materials previously published or written by another person, or substantial proportions of material which have been accepted for the award of any other degree or diploma at UNSW or any other educational institution, except where due acknowledgement is made in the thesis. Any contribution made to the research by others, with whom I have worked at UNSW or elsewhere, is explicitly acknowledged in the thesis. I also declare that the intellectual content of this thesis is the product of my own work, except to the extent that assistance from others in the project's design and conception or in style, presentation and linguistic expression is acknowledged.'

Signed: ..... **Bui Hoai Thang**

Date : .....

Abstract

The ever increasing use of computer systems in society brings new challenges to companies and system designers. The reliability of software and hardware can be financially critical, and lives can depend on it. The growth in size and complexity of software, and increasing concurrency, compounds the problem. The potential for errors is greater than ever before, and the stakes are higher than ever before. Formal methods, particularly *model checking*, is an approach that attempts to prove mathematically that a model of the behaviour of a product is correct with respect to certain properties. Certain errors can therefore be proved never to occur in the model. This approach has tremendous potential in system development to provide guarantees of correctness. Unfortunately, in practice, model checking cannot handle the enormous sizes of the models of real-world systems. The reason is that the approach requires an exhaustive search of the model to be conducted. While there are exceptions, in general model checkers are said not to scale well.

In this thesis, we deal with this scaling issue by using a guiding technique that avoids searching areas of the model, which are unlikely to contain errors. This technique is based on a process of *model abstraction* in which a new, much smaller model is generated that retains certain important model information but discards the rest. This new model is called a *heuristic*. While model checking using a heuristic as a guide can be extremely effective, in the worst case (when the guide is of no help), it performs the same as exhaustive search, and hence it also does not scale well in all cases.

A second technique is employed to deal with the scaling issue. This technique is based on the concept of *random walks*. A random walk is simply a 'walk' through the model of the system, carried out by selecting states in the model randomly. Such a walk may encounter an error, or it may not. It

is a non-exhaustive technique in the sense that only a manageable number of walks are carried out before the search is terminated. This technique cannot replace the conventional model checking as it can never guarantee the correctness of a model. It can however, be a very useful debugging tool because it scales well. From this point of view, it relieves the system designer from the difficult task of dealing with the problem of size in model checking. Using random walks, the effort goes instead into looking for errors.

The effectiveness of model checking can be greatly enhanced if the above two techniques are combined: a random walk is used to search for errors, but the walk is guided by a heuristic. This in a nutshell is the focus of this work. We should emphasise that the random walk approach uses the same formal model as model checking. Furthermore, the same heuristic technique is used to guide the random walk as a guided model checker. Together, guidance and random walks are shown in this work to result in vastly improved performance over conventional model checking. Verification has been sacrificed of course, but the new technique is able to find errors far more quickly, and deal with much larger models.

Acknowledgements

At that time I would like to express my gratitude to those who supports me in my long and interesting journey.

Firstly, I would like to give my thanks to the Governor of the Socialist Republic of Vietnam, especially the Ministry of Education and Training (MOET), for awarding me a scholarship (*Vietnam Overseas Scholarship Program - Project 322*) to carry out my PhD research at the University of New South Wales (UNSW), Australia. Without the scholarship, I could not have come to Australia for my PhD degree. I also thank to UNSW for an additional living stipend scholarship called UNSW-MOET Scholarship, the School of Computer Science and Engineering at UNSW, for the tutorship and resources I have received in doing the research. Without those supports, I could have not concentrated to the research.

Secondly and importantly, I would like to thank my supervisor, Associate Professor Albert Nymeyer, for his great guidance, helpful advice, and patient discussion. I was very fresh in the Model Checking research field when I first came here. Under his excellent research supports over years, my knowledge has been extended, my confidence has been increased, and my research skills and writing have also been improved much. I wish him the best.

Another deeply thanks must go to the person, whose supports are invaluable, my wife Kieu Oanh. She, herself, has been always taking care of my family when I was away. My son, Nhat Huy, who was very little when I start my study, and my little girl, Nhat Vy, who was brought into the world in the middle of my journey, have been grown up well-educated under her excellent parenthood. Her strong supports to my family and her sharing in weal or woe encourage me to keep focus on this research. This thesis is for you all, Kieu Oanh and Nhat Huy and Nhat Vy.

Last but not least, I would like to thank to all my colleagues and review panel members, who gave me advice, suggestions in my research. Special thanks to Nguyen Huu Hai, my colleague in the Faculty of Computer Science and Engineering, Ho Chi Minh City University of Technology, Vietnam, who informed me a paper on random walk model checking, that drove me into the new paradigm, where I combined the heuristic guidance and the random walks in model checking, that forms the kernel of this thesis.

Of course, there are still many people around to give thanks to. To you all, even though your name is not there, you are in my heart.

I love you all.

Contents

List of Figures

List of Tables

This page intentionally left blank.

Chapter 1

Introduction

1.1 Overview

As the size and complexity of computer systems increases, the possibility that the system contains serious and difficult to find errors increases. To develop robust, "error-free" systems[1], the use of formal methods would seem imperative. This however is not the case in industry. The vast majority of the software and hardware development industry ensures *correctness* by using some testing regime instead of formal methods. Formal methods, or more particularly formal verification, are generally used to verify the system satisfies given properties, and to do so provably and completely. Testing in contrast consists of carrying out simulations using test cases derived from either the system code or the requirements. The success of testing depends very much on the quality of the test cases, and by itself, testing cannot be used to (completely) verify a system, as pointed out by the world famous computer scientist Edsgar Dijkstra [26]: *program testing can be a very effec-*

[1] One could argue that every system contains errors, and all we can hope to achieve is minimise the number and severity of those that occur in normal operation.

tive way to show the presence of bugs, but is hopelessly inadequate for showing their absence. With this in mind, in this research we have endeavoured to find a middle road between these two techniques. We have abandoned the notion of verification, and chosen to develop a simulator, but not one that uses test cases, instead it uses randomly-generated 'walks' between states in a formal model, which is identical to those used in formal methods such as model checking [20, 77, 23].

One of the reasons for the success of testing and simulation in industry is *scalability.* It is always possible to think of new test cases that can be used to exercise the system. While testing is a 'never-ending' pursuit, it increases the confidence that the system is 'correct' with every successful test run. Not so formal verification, which tends to be 'all or nothing': either a model (property) is verified, or it is not. This is a serious drawback because most real-world problems require models that are extremely large. In fact, and in general, most models corresponding to real-world systems are intractable to model checking. There are many techniques of course that can be used to alleviate this situation, such as abstraction, more efficient data structures, reduction techniques, but the performance of model checkers is well known to quickly and severely degrade as the degree of concurrency (for example) increases. While the difficulty of constructing formal models, and dealing with formal logic also contribute to its poor 'take-up' by industry, poor scalability is generally acknowledged to the main reason why model checking is not generally used in mainstream system development.

The 'middle road' that this research takes has the scalability of simulation, but it is still formal. Instead of verification, the focus of this research is searching for errors that are defined by properties provided by the system engineer. To improve the 'error-hunting' ability of our technique, we have added

a *guidance* mechanism to the random-walk based algorithm. This mechanism is borrowed from other, so-called guided, model-checking research, where the guide is a *heuristic* derived from the model itself. This has allowed an interesting comparison to be made between the error detection capability of a guided model checker and our guided, random-walk based, formal simulator. In essence, this thesis attempts to show how effective random-walks can be in finding particular states in a formal model, particularly if assisted by a heuristic.

Model Checking: Two well-known model checking tools are SPIN [49] and NUSMV [18]. SPIN is an explicit-state model-checking tool in which a state is represented explicitly (as an array of bits). Given a model of a system and a specification of a property, SPIN builds the state space on-the-fly and searches for a state that violates the property using a (nested) depth-first search algorithm. In contrast, NUSMV is a symbolic model-checking tool in which sets of states are represented symbolically using a data structure called a *Binary Decision Diagram* (BDD) [7]. To check an (invariant) property, NUSMV uses a breath-first search algorithm to locate any (goal) states that violate the property. Model checkers have the desirable feature of being able to report how an error occurred in the model. In other words, they can report along what path of states the error state can be reached. This path is usually referred to as a *counter-example* to the property, or a *witness* of its violation. Unlike SPIN, NUSMV has the even more desirable property that it will report the very shortest path, which makes it an optimal technique.

Note that the search algorithms used in both SPIN and NUSMV are *exhaustive* because they are designed to traverse the entire search space. Furthermore, they do so in a 'blind' fashion, taking no account of what states are likely or most unlikely to violate the property. Indeed, they waste time

and memory searching areas in the state space that are (logically) unlikely to contain the desired state. This can be addressed by using domain information as a heuristic to focus on some particular areas of the state space where the error is likely to be located. There has been much research on how to add domain information to a model checker. Such a model checker is generally called a *guided* model checker [31, 73, 75, 76, 74, 50, 40, 38, 53].

One method of adding this capability to a model checker is to use a search algorithm based on A* [43], a heuristic guided search that can finds the least-cost path from an initial state to the goals. Earlier work by Qian and Nymeyer [73, 75, 76, 74] using this technique has resulted in a tool called GOLFER. The tool GOLFER uses a heuristic generated from a static analysis of the model. The static analysis produces a data abstraction of the system, which is used to compute heuristic costs for each state of the system. The static analysis builds an abstract data model by computing the data dependencies in the concrete model and then removing so-call 'weak' variables [75], which the other variables in the model are least depend on. This method is based on the so-called *cone of influence* abstraction technique [22]. If the abstraction is well-informed (i.e. it has 'correct' and useful domain knowledge), then the speed-up over conventional model checking can be dramatic [62]. In the worst case, if the heuristic has no effect, then the guided model checker performs exactly like a conventional model checker (and will hence search the entire state space). Computing the heuristic at every visited state is of course an overhead that conventional search does not have, but this will not affect the complexity except by a constant amount (assuming the computation itself is constant of course, which is generally easy to show).

Another method (of applying heuristic search to model checking) is to use the so-called *meta-heuristic* approaches, which are not only used to search for

a (near) optimal solution, but also to increase the coverage of the search in exploring a (large) state space using limited resources. Examples are *Genetic Algorithms* with a simple domain-independent heuristic [37], and *Ant Colony Optimisation* with a formula-based heuristic [17] that can be used to search for errors in a very large system.

In the last decade, a new paradigm based on random walks has appeared [44, 59, 86, 83, 64, 65, 42]. This approach has been demonstrated to be effective in finding errors. For example, Owen *et al.* [63, 64, 65] implemented a random-search algorithm in a tool called LURCH, and their experimental results showed that LURCH found 90% of the errors of NUSMV, in orders of magnitude less time and space [65]. Random-walk algorithms require the user to consider how long to allow a given random walk to proceed, and how many random walks to permit. Random walks may continue indefinitely, and an indefinite number of random walks may be required to find a particular error state. There have been many attempts to understand when to terminate a random-walk algorithm, and how many walks to use. There is research that suggests a walk can be bounded in length arbitrarily [44, 64, 65]. In other research, Owen *et al.* [64, 65] set the maximum number of random walks arbitrarily. Alternatively they track how many 'new' states are being discovered relative to 'old' states until some *saturation point* is reached. Grosu and Smolka [42], proposes the application of a Monte-Carlo method. In that research, the state space is sampled by random walks that end when the first loop is encountered. These walks are called *lassos*. If no property violation is found, the search is terminated after $N = ln(\delta)/ln(1-\epsilon)$ walks where δ and ϵ are very small input numbers. In that case, the probability that a state that violates the property still exists in the state space can be calculated, but it is not clear how practical or effective or even appropriate

the probability argument is in limiting the search.

1.2 Scope

The main focus of this thesis is on applying heuristic guidance to random-walk search in model checking. Each random walk through the state space is biased towards an area of the state space that, according to a heuristic, is likely to contain an error state. Heuristics are computed using abstractions, which involves removing data from the model and creating a new, abstract model. This model contains (generally) far fewer states, which enables it to be analysed, and a distance heuristic to be generated. The abstraction method consists of a data-dependency analysis of the original system specification. The process of computing an abstract model requires data to be ranked in 'importance'. Removing variables in the model corresponding to 'unimportant' data simplifies the model. Just one method of ranking data is used in this work. It is based on a *data dependency analysis*. Many other forms of abstraction are conceivable: for example, variables can be ranked in a different way, the control flow can be abstracted, and a combination of both the above can be used. The relative merits of different kinds of abstraction, and how well they guide the search for error states is an interesting research topic in itself, but it is not in the scope of this thesis. For the sake of experimentation and comparison, we have used the same abstraction method throughout. Clearly, the research that we have conducted could be repeated using different methods of generating abstract models to provide the heuristics. Other heuristic methods, including the meta-heuristic, which could be used in conjunction with the guided random-walk search present in this research, are out of the scope of this thesis.

1.3 Aims

The aim of this work is to develop a new approach to formal system analysis. If rigour is required in system development, then instead of using a standard model checker to find property errors, a formal model 'simulator' is proposed that is based on the same formal model, but replaces exhaustive search by a guided random walk. The advantage of doing this is that this improves the scalability, which means that properties can be studied in a more focused way. In essence, it is proposed that a model random-walk simulator should be used in the early stages of system development as a debugger, instead of a model checker, which is a 'blunt' tool that should be used only when a system development has reached the verification stage.

Specifically, the primary aims in this thesis are to:

1. Develop a random-walk search algorithm, where random walks can be bounded (in length), or unbounded, and can start from an initial state or a random state from a previous random walk.

2. Develop a heuristic guided random-walk algorithm that biases random walks. Heuristics are generated automatically from an abstract model.

3. Develop prototype tools that implement the guided/unguided random-walk search algorithms in both symbolic and explicit-state contexts.

4. Carry out experiments to compare model checking and random walks in both symbolic and explicit-state contexts and the effect of guidance on the random walks in both contexts.

5. Investigate the effectiveness of (random-walk) guidance for different models and properties in both contexts, and consider under what circumstances a guide is most effective.

1.4 The Approach in A Nutshell

Given a formal model of a system and a specification of a property, a guided random-walk based framework has been developed that will search for states in the model that violate the property. The framework is implemented in both symbolic and explicit-state contexts, in tools called GRANGOLFER and GRANSPIN, respectively. If the property is violated, then our tools report this fact together with an indication how the error occurred, just like other model checkers. If an error state is not found, then there is no conclusive result. The user may choose to continue the search by increasing the number or, if bounded, the length of random walks, or changing the guide. Alternatively, he may choose to use a model checker to verify that the property is never violated. The same model and property specification can be used in both cases.

A brief overview of the approach is depicted in Figure 1.1. In this figure we denote a model by M and a property by φ. These comprise the input to the tools. On the left in the figure we depict the *Heuristic Builder*, whose output is a so-called abstract database, called *Abstract DB*. This 'database' is a heuristic and is input to the main algorithm. Given these inputs, the stages of the main algorithm are as follows:

Abstraction abstracts away so-called 'weak' variables in the input model M and generates an abstract model $\hat{M}$.

Heuristic Generator given the abstract model $\hat{M}$, builds an abstract database (*Abstract DB*), which consists of all abstract states in $\hat{M}$ together with their distances (i.e. number of abstract transitions) to the abstract goal state.

isTermination? checks whether some *termination* condition is satisfied. If

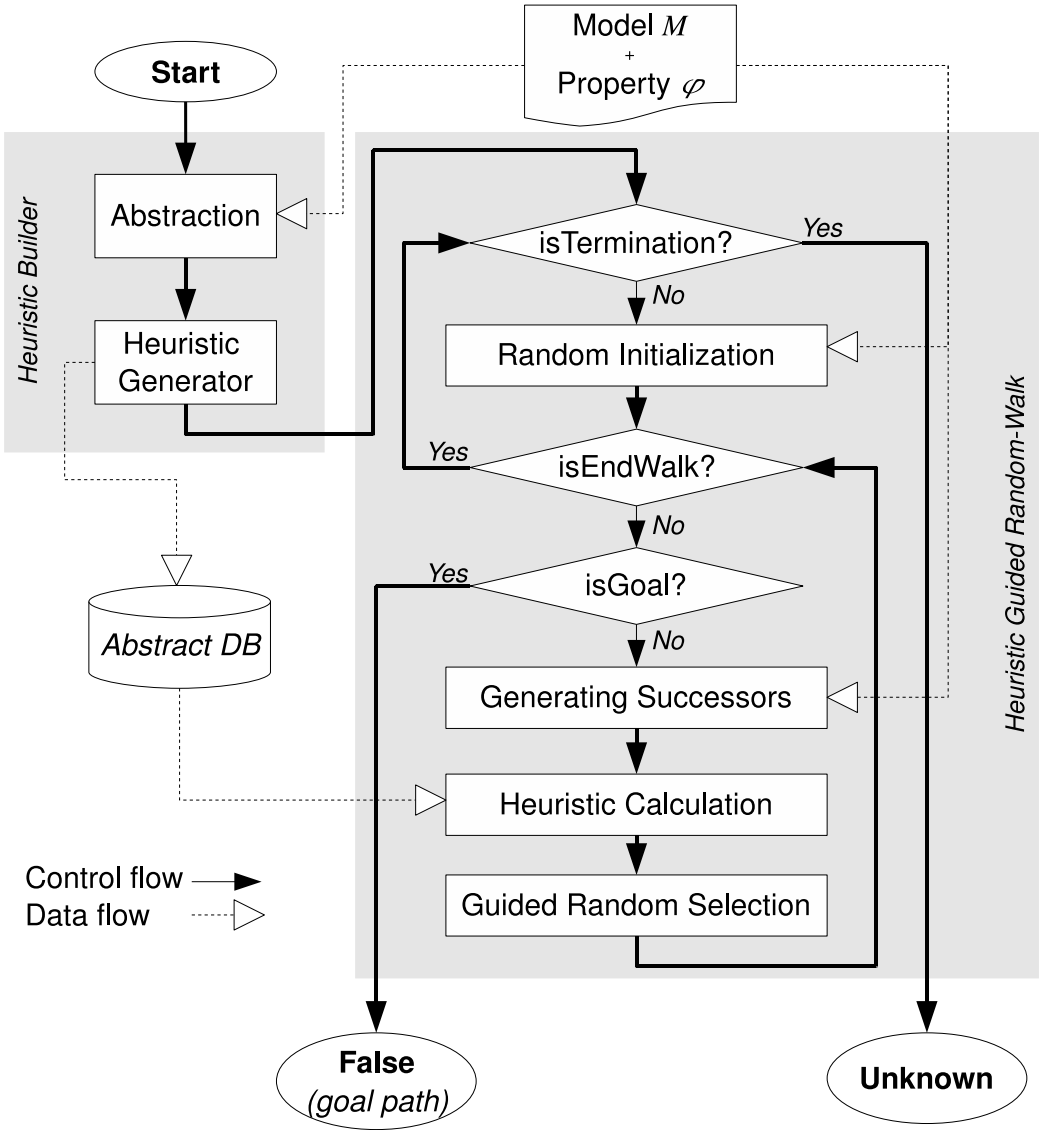


Figure 1.1: The approach

it is satisfied, the search is terminated, and in effect there is no result. While we have worked with many different termination conditions, we generally use a maximum number of walks N. If the condition is not satisfied, the search continues.

Random Initialization takes a state randomly from the set of initial states of M (this is the starting point of a walk).

isEndWalk? checks whether some *end-walk* condition is satisfied. Examples of conditions are the maximum length of the walk, the current state has been visited earlier (hence we have found a loop), and the current state is a leaf. If the condition is satisfied, the current walk is ended.

isGoal? checks whether a path, which leads to a (goal) state in which the property φ is violated, has been found. If *isGoal?* is true then the path represents a counter-example to the property.

Generating Successors generates all successor states of the current state (in the model M).

Heuristic Calculation matches each successor state to the patterns in the input pattern database *Abstract DB* and assigns those that match states the corresponding heuristic cost.

Guided Random Selection uses a biased random strategy to select one of the successor states of the current state. In this strategy, successor states that have smaller heuristic costs are selected more often than states that have larger heuristic costs during the walk.

1.5 Contribution

A new formal, heuristically-guided, random-walk based model simulation framework has been developed. It uses abstract-model based heuristics to bias random walks, which are used to search for error states in the state

space of a formal model of a system. These error states are defined by properties that are expressed in temporal logic, just as in model checking. The methodology scales because it is simulation, and is more efficient than standard search because of heuristic guidance. More specific contributions made by this thesis are:

1. We have developed a formal random-walk based search algorithm and implemented it in a new tool called ranSPIN, which is derived from SPIN by replacing the depth-first search algorithm. We have conducted experiments using ranSPIN and SPIN, and compared their run-time, memory usage and the length of the goal path. We have also compared the robustness of the tools.

2. We have added a heuristic mechanism to the random-walk based search algorithm. This was implemented, resulting in a new tool granSPIN. We compared the performance of all 3 tools: SPIN, ranSPIN and granSPIN. We also studied the effects of informed and mis-informed heuristics on the guided search.

3. We have developed a symbolic, formal guided random-walk based search algorithm and implemented it in the existing tool GOLFER [73, 75, 76, 74], resulting in the tool granGOLFER. We compared the performance of GOLFER and granGOLFER.

4. In this research, we have proposed the use of random trails. A random trail is a random walk in which no state is re-visited. We have proved that, using only maximal random trails, one can apply the Monte-Carlo theory proposed by Grosu and Smolka [42]. Using random trails instead of random walks (typically) improves the performance of the search.

5. We have proposed a new model metric called *Process Error Participation* (PEP) factor. The PEP factor measures the relative level of concurrency in the model with respect to the error state. We use the metric to interpret and classify the behaviour of the guided search algorithms. While the mechanism is rudimentary, it provides some understanding why the different search algorithms perform as they do.

6. We have proposed a classification that depicts which tool is best to analyse models when properties are specified using LTL (the explicit-state SPIN tools) or when the property is simply an invariant (the symbolic GOLFER tools). We observe that the PEP factor affects the performance in the symbolic invariant models but not in the explicit-state LTL models.

7. The guided and unguided random-walk search algorithms have been developed in both symbolic and explicit-state contexts. The symbolic version has been implemented in the tool GRANGOLFER. The tool is flexible, allowing either random walks or random trails to be used, in guided or in unguided mode, and in bounded or in unbounded form. The explicit-state version has been implemented in GRANSPIN. Again, the random walks can be either guided or unguided, but also be (re-)started at a random state taken from the previous walk (instead of the initial state).

1.6 Thesis Organisation

The related work is discussed in Chapter 2 and the basic technical definitions are presented in Chapter 3.

In Chapters 4, 5, 6, 7 the main body of research is described. These chapters are more or less chronological in the sense that they present the work in the same order as the research was conducted. While they also approximately correspond to the publications that have been generated in this research, the publications themselves have not necessarily appeared in this order (because of different lead times and review processes). Each chapter is prefaced by a reference to the publication(s), and any changes made to publications to bring them up to date, to expand the work, and simply to integrate into the thesis. In the last section of each chapter there is a section summarising the research presented in that chapter, as well as the motivation of the next chapter. Both the preface and summary sections appear in italics for clarity. More specifically, these chapters present the work in the following way:

Chapter 4 introduces the random-walk model-checking algorithm and describes its implementation in the explicit-state model-checking tool ranSPIN. The performance of ranSPIN and SPIN are compared.

Chapter 5 studies the application of random-walk search in symbolic model checking. Guided random walks that use an abstract model as heuristic are also developed in this chapter. Experiments are carried out to compare the performance of guided and unguided random-walk based model checking, and standard guided and unguided model checking. The new metric, the *Process Error Participation* factor, is proposed in this chapter.

Chapter 6 revisits ranSPIN and adds the guided random-walk based search algorithm to the tool, resulting in the new tool granSPIN. The use of abstract-model based heuristics as guide is discussed in this chapter,

as well as the consequences of using a heuristic that is well-informed or mis-informed. Selection strategies that describe how states are chosen during a random walk are also investigated.

Chapter 7 integrates all the guided random-walk based algorithms (namely the symbolic and explicit-state) into a single framework. The complexity of these algorithms is analysed. A model classification is proposed that involves the *Process Error Participation*.

The tools produced in this research are briefly described in Chapter 8, and in Chapter 9 we summarise the research, discuss conclusions and propose further research.

Chapter 2

Related Work

The focus of this thesis is using heuristics to guide a random-walk algorithm in model checking. Related work on applying heuristics in model checking, on random-walk search, and on the combination of the two paradigms are discussed in Section 2.1, 2.2 and 2.3 respectively. Related work on model characterization and classification is presented in Section 2.4.

2.1 Heuristic Guided Search Based Model Checking

In model checking, there are a variety of uses of heuristics to reduce the impact of the well-known 'state-space explosion' problem and to improve the efficiency of the verification method, the memory usage and the length of the counter-example. In *directed model checking* [31], heuristics are used to rank states in terms of their closeness to a possible error state. A guided search algorithm, A*, at each step then chooses and examines a state that is estimated to lie on the shortest path to an error state.

While guided search algorithms can be considered conceptually equiva-

lent, the method of generating a heuristic are generally different. We classify these algorithms as *abstraction-based heuristics* [73, 32, 54, 53] if heuristics are generated from an abstract version of the concrete model; *structure-based heuristics* [31, 81, 40, 38] if they are generated based on the structure of the properties [33, 81, 38] or the structure of the models/programs [40]; and less commonly *other heuristics* to include user-defined heuristics, which are input from the user using source code annotations, and method-based heuristics, which are used in *AI planning* [29, 54].

2.1.1 Abstraction-Based Heuristics

Abstraction is a kind of relaxation to reduce the complexity of the model checking [19]. Tools such as BLAST [45] and SLAM [5] use it to reduce the size of the state space so that the resulting abstract model can be verified with limited resources (time and memory). The advantage of using the abstraction method is that if the abstract model is verified (i.e. error-free), then provably so is the concrete model. Unfortunately, if the verification of the abstract model fails, there is no guarantee that the verification will also fail on the concrete model. (If it does not fail it is called *spurious*.) In the application of the abstraction method to heuristic guided model checking, Qian and Nymeyer [73] are the first to use the abstraction also to generate a heuristic, which is then used to guide the search. That work is implemented in a model checker called GOLFER [75, 76, 74], which is based on NUSMV.

In Qian and Nymeyer [73], an abstract model is built by abstracting away a certain number of 'weak' variables from the concrete model. The 'weak' variables are chosen after statically analysing the dependency of variables in the concrete model. The abstract model is then used to build a so-called *pattern database* [25], also called an *abstract database*, to store all abstract

states with their distances to the abstract goal states. In [73], (abstract) states that have the same distance are stored in a single BDD (Binary Decision Diagram), so the pattern database comprises a set of BDDs and the corresponding distances. The pattern database is later used as a heuristic function in the guided search during model checking of the concrete system.

The abstraction method is also applied by Edelkamp *et al.* in [32]. They use a tool called αSPIN [35] to abstract an input PROMELA model, and use the abstract model to build heuristics. The heuristics are used in the explicit-state (heuristically guided) model checker HSF-SPIN. The work of Edelkamp *et al.* and Qian and Nymeyer are different: Edelkamp *et al.* uses αSPIN to abstract the semantics of variables, whereas Qian and Nymeyer abstract the set of variables by omitting 'weak' variables.

The abstract-distance approach has also been used in [82] to guide the random simulation. However, the algorithm used in that research is actually a randomised best-first search. In the search, a number of short random walks are carried out starting at the current state. Each random walk selects a successor state that is not on the priority queue (and hence has not been visited yet), and that has a smaller heuristic distance (to the possible goal state). These successor states are put onto the priority queue for the next iteration. At each interaction, the 'best' state on the queue is selected. While the best-first search is hence randomised, the technique cannot be used in verification as it may miss some state in expanding successor states.

Related work on abstracting the variable domains has been carried out in AI planning, for example in a tool called UPPAAL [54]. This tool is an explicit-state model checker for timed automata. In this approach, a so-called 'monotonicity' abstraction is used that assumes that each state variable keeps its value forever once it has obtained that value. Under this assumption,

a so-called 'abstract transition' graph, which is a layered graph encoding reachability information, is built and used to calculate the heuristic distances (of states) on-the-fly. Later, in [55], the authors reuse this calculation on the abstract model of a concrete system to build a pattern database similar to the work of Qian and Nymeyer.

Another type of abstraction based on automata theory has also been proposed [27]. In this technique, the abstraction function is automatically computed in such a way to preserve the error distance. It differs from the approaches outlined above because the abstraction functions need to be provided beforehand. In this method, a model consisting of multiple processes is considered as the product of the processes. The abstract model is computed incrementally by repeatedly replacing pairs of processes (from the set of all processes) by an abstraction of their product, with a limit of N states in each abstraction. States with the same error distance are merged first, followed by states with high error distance. This approach is also developed in UPPAAL/DMC [53], an extension of UPPAAL.

Predicate abstraction [5] is also used to generate a heuristic function in UPPAAL [46]. In that work, the state space is divided into equivalence classes with respect to a list of logical expressions (predicates). During the search, each concrete state is mapped into its corresponding abstract state, and the abstract distance (to the abstract goal state) is derived as the heuristic cost of the (concrete) state, and stored in a pattern database which is used as the heuristic function. This method is different to other methods that use a pattern database as it merges states based on the truth table of the (predefined) predicates, instead of abstracting away state variables or reducing the semantics of variables.

We should note that the coarser the abstraction, the more informed [61]

the generated heuristic as less information is removed from the concrete model. Of course, the more informed the heuristic is the more quickly the guided search is in finding a goal state. The major disadvantage of course abstractions is that exploring a coarse abstract model costs more time than exploring a finer abstract model, mainly because the former is more complex than the latter. Choosing coarse/fine abstractions is not well studied in the literature.

2.1.2 Structure-Based Heuristics

Edelkamp *et al.* in 2001 presented a tool called HSF-SPIN [31], which is derived from SPIN. The heuristic used in the tool is based on the type of property (e.g. livelock and deadlock) as well as the (Boolean) property formulas. In searching for errors in multi-process protocols, the tool uses the so-called *Local State Distance Table*, which presents all distances between any two local states in a single process within the protocol, together with a calculation that estimates the number of steps from the current state to a goal state in which the property is violated. Their experimental results showed that the number of expanded states and the length of the path to the goal state (i.e. the goal path) are significantly reduced compared to SPIN. Unfortunately, the running time of the tool is generally slower than SPIN, partially because it must generate all the successor states of a given state at each step, and maintain those states, whereas SPIN only executes a single transition (at each step) in a depth-first search algorithm.

This method of calculating the heuristic using property formulas was later extended in [81] using Bayesian reasoning to improve the expected behaviours of the guided search. It is also called *property-specific heuristics*. In related and parallel work, Gradara *et al.* in the tool called DELFIN$^+$ [38] used the

structure of the verified formulas expressed in the selective Hennessy-Milner logic to partially construct the input transition system, which is presented in the Calculus of Communicating Systems. This method allows the tool to reduce the number of generated states in order to prove the correctness of the formulas.

In addition to property-specific heuristics, *structural-heuristics* can be used to explore a model or program [40, 24]. For example, Groce and Visser in [40] use *code-coverage*, *thread interleaving* and *thread preference* heuristics to model check Java programs using a tool called JPF. This kind of heuristic is also studied using a tool called FLAVERS [24]. This tool analyses an input model expressed as a control-flow graph. We mention finally that while Groce and Visser's approach [40] has similarities to our work, there are substantial differences. We discuss these differences in more detail in Section 2.4.

2.1.3 Other Heuristics

User-defined heuristics involve using knowledge from the user (presumably a system engineer) to guide the search. This requires input from the user (in the form of source code annotation, for example), which cannot be done automatically, and is hence not considered further in this work.

There are other methods of generating heuristics (different to abstraction-based, structure-based and user-defined heuristics), which we place in a class called *method-based* heuristics. In [29], for example, Edelkamp translated a (subset of a) PROMELA model (used in SPIN) into an AI planning model, which is then used to produce an estimate of a heuristic. This kind of approach is also studied by Kupferschmid *et al.* [54]. Another method of guiding the search is developed by Kuehlmann *et al.* [52], who rank states in the priority queue based on the probability that they will reach the target.

The probability of a state is determined by using a set of random simulation runs. States with highest probability are first chosen to expand during the search.

In other research on method-based heuristics, a tool called HOPPER [50] has been developed that uses a *randomised* guided search. The method is actually a modified version of the best-first search algorithm. In each search step, instead of selecting the first element in the priority queue, one element from the few first elements in the queue is selected. No new heuristic generating method is proposed in this work however. The work is extended later in [78] to generate counter-examples.

Recent work in the area of meta-heuristics in model checking has been carried out by [37, 81, 17]. In [81], meta-heuristics are used to adjust the heuristic estimation using Bayesian theory, but the underlying algorithm is guided search. The approaches used in [37, 17] apply *Genetic Algorithms* (GAs) and *Ant Colony Optimisations* (ACOs) to model checking. Meta-heuristics are general frameworks for heuristics in GAs. Ant Colony Optimisation is also an example of a meta-heuristic. These techniques are very different to the approach taken in this thesis.

Genetic algorithms (GAs) are heuristic-search methods that are based on the principle of 'survival of the fittest'. They differ from conventional search in that they start from a "population" of solutions, or so-called "chromosomes". The process of searching involves removing weak solutions from the population. Solutions with the highest fitness can be "recombined" or "cross-overed" with other solutions, or they can be "mutated" by modifying an element of the solution, and these are used to generate new and better solutions. GAs can use some random function to mutate or cross-over the chromosomes. The search itself in GAs is heuristic, in contrast to the (guided

random) search technique in this thesis.

An application of GAs by [37] encodes paths from the initial state to some state up to a maximum depth in chromosomes. These are maintained as a population. In evaluating the fitness of a chromosome, the algorithm executes the path that is encoded by the chromosome and a model checker is invoked during the execution. A simple domain-independent heuristic, such as the sum of the number of enabled transitions at each state along the path (for deadlock detection), is used in the fitness evaluation. In GAs, the length of a chromosome (the number of bits used to encode a chromosome) is fixed before invoking the algorithm. Consequently, the paths encoded in GA model checking are bounded in length. That limits the work in [37] to searching for errors only. Alba and Troya [3] applied a GA to detect errors such as deadlocks, useless states and useless transitions in communication protocols. Later, [2] used GAs for detecting deadlock in (large) Java programs.

Recently, Alba and Chicano [17, 1] used the ACO algorithm to detect errors in large concurrent systems. An ACO requires initial non-optimal solutions in an attempt to generate a (near) optimal solution. Its application to model checking requires a number of initial goal paths. As the search for these goal paths may exhaust resources, Alba and Chicano bound the paths during the search. They use multiple-optimisation stages in which the first stage starts the search at the initial states of the state space and up to the bound. The last state in the "best" path is used for starting the next stage. In this way, the search tries to go further in the state space, and continues until the goal state is reached. Alba and Chicano use Edelkamp *et al.*'s heuristics (from the HSF-SPIN framework), which we outlined earlier in this chapter. We note that just like our abstraction heuristics, the fitness function used in GAs/ACOs needs domain information (hence "well-informedness" is

necessary) to perform well.

2.1.4 Discussion

Although guided search can improve the efficiency of model checking by minimizing the amount of search that is carried out in 'fruitless' sections of the state space, its (worst-case) complexity is equivalent to breadth-first search. When the size of the state space gets truly enormous, guided search suffers the same scalability problems as conventional search. You could say that (conventional) heuristic guidance just delays the onset of state explosion. To overcome this short-coming, we propose in this thesis applying heuristics to random-walk based search.

In this chapter we have surveyed heuristic methods that are related to this approach. In particular, we survey abstraction-based, structure-based and other methods used to build heuristics. In our symbolic algorithm, we use the approach of Qian and Nymeyer to eliminate so-called 'weak' variables from a concrete model to build an abstract model. In our explicit-state algorithm, two abstraction methods are used: (1) we abstract multiple processes into a single process, and use a 'Local State Distance' table to estimate the heuristic cost of a concrete state; and (2) we abstract by reducing the number of processes, and use the goal path in the resulting abstract model to approximate the heuristic distance of a concrete state.

2.2 Random-Walk Based Model Checking

The application of random walks in model checking was first proposed in 1986 by West [88] to check protocols. The technique has been studied more recently by [59, 44, 86, 63, 42]. In [44], Haslum proposed that one needs to

walk randomly at least $\frac{1}{\epsilon}|V||E|$ steps in a graph, where V is the set of states and E is the set of edges, to ensure that it covers the whole state space, given the probability $1 - \epsilon$. Unfortunately, the number (of steps) is too large if ϵ is chosen to be small, given the sizes of V and E are also large. Moreover, the technique can be, or should be, applied only to closed strongly connected graphs.

In order to trade space with time, Tronci *et al.* [86] proposed a randomised breadth-first search, which, when the memory is exhausted, the verification does not stop, but just slows down. In particular, they replace the hash table used in the standard BFS algorithm by a cache memory (with no collision detection), and they bound the queue to a fixed size. In the implementation (which is based on Murφ), the authors claim a saving of 30% of memory at the cost of an average 100% increase in time.

Very extensive work on random-based methods has been carried out by Owen *et al.* [63, 64, 65], who implemented a random-search algorithm in a tool called LURCH. This tool searches for a goal state in an AND-OR graph that represents the state space of the system. Their random-search strategy involved maintaining a queue of successor states (which are just nodes in the graph) in random order, and at each step in the search, popping the next state from the queue. Their experimental results showed that LURCH is very effective (in both time and space) in finding errors, at least if compared to NUSMV.

Another approach called *deep random search*, which ensures the full coverage up to a predefined cut-off depth, is proposed by Grosu *et al.* [41]. In this approach, at each step in a walk, after choosing the next state among all successor states of the current state, the remaining successor states (of the current state) are stored in a so-called *fringe*. If the current walk, which is

bounded by the cut-off depth, is not a goal path, the next random walk is started at a state taken from the fringe. The fringe is kept up to date and the search is terminated if it is empty. Regardless of the running time (and space), the search is guaranteed to cover the state space up to the cut-off depth. Although the search can be repeated many times with the incremental cut-off depth, there is no guarantee that the approach will cover the whole state space however.

Pelánek [66] is the first to study the coverage of the search algorithm using random walks over a state space. He concluded that random search cannot cover the state space in most of the models studied, and that the coverage is dependent on the structure or topology of the state space. Later, in the [68], Pelánek *et al.* extends that work, and finds that random walks can be enhanced by combining them with other methods such as re-initialization, local exhaustive search, caching, parallel, traces and guiding. Note that there is no heuristic guidance in their research. Interestingly, they argue that the choice of the best method of search is model dependent. Pelánek further extended that work in [67] a produce a model classification. This is discussed in more detail in Section 2.4.

Since one of the advantages of using random walks is that every walk is independent of every other walk, random walks can be executed in parallel [84, 83, 28]. In [84], random walks are carried out using a wide range of distributed memory multiprocessors and networks of workstations in the parallel version of the tool called Murφ. The results showed that the speed-up is nearly linear (in the number of processors). The tool is later extended in [83] to combining a local exhaustive-search algorithm and a random-walk algorithm, in various configurations: for example, an initial bounded Breadth-First Search (bBFS), followed by random walks combined with bBFS; or

initial random walks combined with bBFS, followed by a final random walk. More recently, the parallelization of random walks has been studied by Dwyer *et al.* in [28], who manage to speed-up the search 1000 times using just a small number of parallel processors.

In research on determining the bounds of the number of random walks needed to search a state space, Grosu and Smolka [42] were the first to present a random-walk based *Monte Carlo* algorithm to model checking. In that research, given a system model S and a Linear Temporal Logic (LTL) property φ, the Büchi automaton $B = B_S \times B_{\neg\varphi}$ is built and checked whether the language $L(B)$ of the Büchi automaton B is empty. If a random walk that ends in a circle (referred to by the authors as a *lasso*) contains an accepting state, the language $L(B)$ is not empty, and the walk (called the *accepting lasso*) is a 'witness'. If no accepting lasso is found, Grosu and Smolka determine that after $N = \ln(\delta)/\ln(1-\epsilon)$ random walks, the probability of sampling the state-space further and finding an accepting walk is less than δ, given that the probability of an accepting walk is greater than ϵ. This gives the user a measure of when to stop the Monte Carlo algorithm. While this research is certainly interesting, it is not clear how practical or effective or even appropriate the probability argument is in limiting the search.

In this thesis, our pure random-walk search is related to the random search used in Owen *et al.* [63, 64, 65], but also to the approach used in Grosu and Smolka [42] as sampling is halted after a pre-determined number of walks. The main differences are the implementation and the context of the algorithms (our work is carried out using SPIN and NUSMV to allow for comparison of the explicit-state and symbolic approaches), and of course the fact that we use *guided* random walks, which are not studied by any other author. Also note that our random-walk search approach is different to "ran-

domised search". For example, a randomised depth-first search implemented in [78] is a depth-first search, in which successor states at every step are shuffled before choosing one. This is very different to random-walk search, where walks are carried out completely randomly and independently from the initial state until termination.

2.2.1 Discussion

The main advantage of random-walk based search is its scalability. Its performance degrades at the same rate as the size of the state space increases. Its problem is, in a nutshell, its 'randomness': a random search can be inefficient as it may spend all its time searching areas of the state space that contain no errors. A random-walk algorithm, by definition, lacks 'control': the user cannot direct the search, apart from determining when it should stop.

2.3 Guided Random-Walk Based Model Checking

The integration of guided search and random-walk based search seeks to overcome the drawback of each approach. We search randomly to make the search scalable, but we allow the random search to be guided, thereby improving efficiency. We are the first to propose this in our earlier work [9] in the tool called GRANSPIN, which is based on SPIN - an explicit-state model checking tool. In that work, we used a heuristic to rank successor states at every step in a walk. This biases or guides the random walk. In a biased random walk, successor states that have smaller heuristic costs are selected

more often than states that have larger heuristic costs during the walk. The heuristic is derived (semi-) automatically by a special form of abstraction. We reduce the concurrency of the system by reducing the number of processes. This reduced model is an abstraction of the original system, and we refer to the reduced model as an abstract model. The heuristic for the original model is simply the distances from each of the states in the abstract model to the goal (or error) states in the abstract model. This same principle was used in other research by the authors on symbolic model checking [73]. The work is later extended in [10, 13] to study successor-state selection strategies and the effectiveness of heuristics in guided random-walk based search.

At the same time, we applied the same mechanism of guiding the random walks to the symbolic approach [11], and built a new tool called GRANGOLFER that implemented the technique. This tool is based on GOLFER, developed by Qian and Nymeyer, and extended to include an abstraction guided random-walk mechanism. In fact, the same abstraction mechanism is used in both the guided search and guided random-walk based search algorithms. The comparison of the two implementations is carried out in [14], where a classification of models that is based on the number of processes involved in error states is also proposed. That latter aspect will be discussed in more detail in Section 2.4.

Note that although our work uses a (guided) random-walk based technique, it still corresponds to model checking because it still checks the temporal properties of models. In contrast, conventional simulation that uses 'pure' random walks only simulates the system (generally using input from the user). Our method therefore inherits advantages from both fields: guided search (from verification) and random-walk based search (from simulation).

Other work [85] has combined simulation and verification. In that work,

a simulator is used to generate a prefix of a computation of a model and a verifier is used to extend that prefix. This is a very different application of the approach. In yet other work [82], abstract distances are used to guide the random simulation. However, that work is different to our work because we carry out a sequence of random walks, where each walk starts from the initial state (or a state from the last random walk), whereas they use the random simulation to enlarge the boundary of the visited states (towards the targets).

Note again that our guided random-walk search approach is different to the randomised guided search carried out in [50, 78]. Our approach is a random-walk search, in which a random successor state is selected in a biassed manner, whereas the latter approach is a guided search, in which some best states in the priority open queue are shuffled before choosing one.

2.4 Model Characterization and Classifications

There are many different approaches to model checking. Which approach is best suited to which problem, or more precisely, to which model of a problem, has been rarely studied. There are studies in related areas, for example [79] who investigates the use of software-configuration management techniques for large and complex systems; [80] who samples the state space to decide which BDD-based reachability technique is the best for a given model; [60] who uses an expert system method to find an efficient proof strategy; and [34] who creates an open framework that enables the combination of techniques in a multi-threading domain.

In our work [11, 14], a static variable-dependency analysis technique (see [73]) is used to characterize models in terms of the number of processes

that 'participate' in an interaction that results in an error state. We call this characterization the *Process Error Participation* (PEP) factor. A low PEP factor means that few of the existing processes in the model are involved; a high PEP factor means that most or all are involved.

Avrunin *et al.* in [4] laments the lack of studies that compare verifiers. They remark that generally only anecdotal accounts of success are reported, but little empirical data comparing different verifiers is available. This is particularly problematical for the transfer of the technology from the research world to industry, where predictability and reliability are key issues.

In seemingly related work [40], Groce and Visser study the so-called 'Thread Preference Heuristic'. However, that work is aimed at finding heuristics for multi-threaded Java programs, whereas the PEP factor is used to classify and predict model behaviour. Our classification is also very different to that of Pelánek in [67] who focusses on the shape of the state space, irrespective of errors, whereas we focus on multi-process models and the involvement of processes in particular goal (error) states, For example, the Peterson protocol (for N-processes sharing a critical section) in our work can be either low PEP or high PEP model depending on the property that is checked, but [67] characterises the protocol simply as a member of the 'Mutual exclusion algorithm' class.

Holzmann in [48] points out that the performance of different model checkers can be affected by the options selected. For example, partial-order reduction can substantially improve the performance by reducing the size of the state space. In the experiments carried out in this research, only default options of SPIN and NUSMV are used. The study of which is the best option for each case study is out of scope in this research. The variability introduced by selecting options that optimise performance for each particular

case study makes comparisons and drawing conclusions harder, and is hence not attempted. It is certainly an avenue that could be explored in further research.

2.5 Summary

In this chapter, we have surveyed model checking, heuristic guided search and random walks, which together form the essence of the technique presented in this thesis. Model checking is used to guarantee the correctness of a given model with respect to a temporal logic property. In directed model checking, heuristics are applied to improve the efficiency of the search. Its applicability to industry where models tend to be very large is limited because of the well-known 'state-space explosion' problem. In contrast, simulation based on random walks, which is widely used in industry because of its simplicity and scalability, is just a 'blind' search and lacks guidance. Bridging the gap between these two paradigms is one of the aims of this thesis.

Interestingly, while heuristics have been applied to model checking, they have not been applied to simulation. One possible reason may be the extra work required to rank the successor states (using the heuristic). In later chapters in this thesis, when we apply heuristic mechanisms to random walks, we analyse the complexity and compare the performances of our new approach to the original random-walk approach, to guided search, as well as the conventional model-checking algorithm. The performance metrics considered are execution time, memory usage and length of goal path.

Related work on model characterization and classifications is also considered in this chapter. The most recent work in this area is that of Pelánek in [67], who characterise the models in term of the shape or structure of the

state space, is very different to the error-dependent classification proposed in this thesis. This classification is presented in later chapters.

In the next chapter, we will present the technical background used in this thesis.

Chapter 3

Background

In software development, formal verification and simulation are seen as complimentary paradigms. As mentioned earlier, while model checking is a largely automatic technique that can guarantee the correctness of a system, its applicability is limited because of the well-known 'state explosion' problem. Testing and simulation in contrast scale well, and consequently these techniques are popular in industry, but they are basically 'hit and miss' techniques that cannot guarantee the absence of errors.

In the work of Qian and Nymeyer [62, 73, 72, 75, 76, 74], a mechanism of statically analysing a model has been used to build a dependency graph of the variables (in the model). This graph is used to build an abstraction of the original model, which in turn is used to guide a heuristic search in a guided model checker. We extend that work and apply the same technique to build a heuristically driven random-walk model checker, which we call *guided random-walk based model checking*. This work sits at the intersection of a number of research areas: model checking, random walks, heuristic search and simulation. Particularly novel is the use of a heuristic mechanism to guide the random walk towards states of the model that possibly violate user-

defined properties, and the use of an automatic abstraction scheme to build the heuristic. This contrasts to conventional simulation techniques, which require the user to provide input, and does not check temporal properties.

3.1 Technical Overview

In model checking, there are two general approaches to represent systems: *temporal-logic* [20, 77] and *automata-theoretic* [87] based. In the first approach, a system is modelled as a finite state transition system M, which consists of a finite set of states and a transition relation among states, and a property φ is represented in a *temporal logic* [70] expression. If there is no reachable state of the model M, in which the property φ is violated, M is verified as a *model* of φ, $M \models \varphi$. In the automata-theoretic approach, any model and property can be represented as Büchi automata for determining whether the model behaviour conforms to that of the property. It is accomplished by checking the *emptiness* of the language $L(B)$ of the product Büchi automaton $B = B_S \times B_{\neg\varphi}$, where B_S is the automaton of the checking system S, and $B_{\neg\varphi}$ is the automaton of the negation of the property φ. If a sequence of steps (called a *run*) in which a defined *final state* is repeated infinitely often is founded in B, then the language $L(B)$ is not empty and the run is a *witness* to the property violation. In this case, φ does not hold in S. Actually, the two approaches are related when a finite state transition system and a temporal logic expression can be both translated into automata [87]. In both approaches, exhaustive search algorithms are usually used in finding witnesses (of property violation).

In the implementation, data in model checking can be represented *explicitly* or *symbolically*. Explicit-state model checking manages states individu-

ally. When the number of states is increased as the size of the system gets more complex or large, there are several methods to reduce space, including *partial order reduction* [69], *state compression* [39], and to increase the accessing time (to stored states) such as *Hashing* [89]. On the other hand, in symbolic model checking, a large set of states can be represented in a possibly small data structure called a *Binary Decision Diagram* (BDD) [7].[1] By using BDDs, symbolic model checkers have been used to verify enormous systems, containing more than 10^{120} states [15]. However, the effectiveness of manipulating BDDs (in symbolic model checking) depends on the order of the variables used in building the BDDs [7].

The most well-known explicit-state model checking tool is SPIN [49]. The main algorithm in SPIN is a (nested) depth-first search that traverses the state space searching for a violation of the desired property [2]. SPIN is also automata-theoretic based model checking. Given a set of concurrent processes represented in the PROMELA language, SPIN interprets the union of all processes as an automaton. The (negation of the) desired property is also given as a so-called *never-claim* process in PROMELA language. In the search, if the current path (starting at the initial state) is accepted, it is returned by SPIN as a counter-example to indicate that the property is violated for the given system. If SPIN completes the search without finding any violation, it guarantees that the property holds (for that system). In practice, when there is no guarantee that SPIN can search through the whole state space of large systems, it is limit.[3]

[1] Another approach in symbolic model checking is to use *satisfiability* (SAT) [6] procedures (and the so-called *Bounded Model Checking* technique).

[2] The complete functionality of SPIN is described in the SPIN manual [49].

[3] For a large system, the users can bound the maximum depth of the (depth-first) search of SPIN. If SPIN completes the search within some limit of d steps, one can conclude that

In symbolic model checking, NUSMV [18] is a well-known tool. It is an extension of SMV [56]. The kernel of NUSMV is a breadth-first search algorithm that initially takes the set of initial states as the current set of states and repeatedly advances from the current set of states (as a BDD) into the set of successor states (a new BDD) until it reaches a goal state. Advancing is carried out by applying BDD operations to the current BDD and the relation transition, which is represented as another BDD. Using BDDs in NUSMV increases the size of state spaces (of models) that can be handled in limited storage. However, performance depends mainly on the performance of manipulating the BDDs, which in turn depends on the order of the variables in BDDs.

The lack of scalability in model checking can be handled at different levels. The use of better data structures (such as using BDDs) is one possibility, and the use of better algorithms (such as partial-order reduction techniques) is another. Adding guidance to the underlying search algorithm is yet another possibility. The guide comes in the form of a heuristic. Heuristics have been used for decades of course to direct search algorithms; consider the search algorithm A* for example. In A*, heuristics are used to rank states in terms of their distance to a possible error state. A state is always chosen that is estimated to lie on the shortest path to an error state. In model checking, this approach is called *guided* model checkers, and examples are: (IO-)HSF-SPIN [31, 30], GOLFER [75, 76, 74], HOPPER [50], JPF [40], DELFIN$^+$ [38] and an extension of UPPAAL [53]. Importantly, the worst case is when the heuristic is 'useless' (i.e. has no domain knowledge), and the guided search

the property holds for the model in the area of state space bounded by d steps from the initial state. This can increase the confidence but not guarantee the absence of an error. The default maximum depth in SPIN is 9999.

acts like a conventional exhaustive search.

Research in tackling the scalability issue has recently seen the advent of a new model-checking paradigm based on random walks [44, 59, 86, 83, 64, 65]. The advantage of using random walks is that it is always possible to carry out a random walk though the state space of a system, no matter how large and complex the system is. In practice, the random-walk approach has been demonstrated to be effective in finding errors [65]. Note however, like simulation, in practice, it is difficult if not impossible to conduct enough random walks to verify correctness. Note as well that a random walk is as 'blind' as exhaustive search: it does not take into account any domain knowledge that it may have as to the whereabouts of possible errors in the state space.

In this thesis, we focus on using the strengths of both guided search and the random-walk paradigm to tackle scalability. In particular, we apply the abstraction heuristic technique proposed by Qian and Nymeyer to random walks. Technical definitions of our model, abstraction heuristic and random walks are given in the next section.

3.2 Technical Definitions

In model checking, given a model and a property, a model checker attempts to answer the question whether the model is correct with respect to the property. The answer will be yes, or a counter-example if it is no. We sometimes refer to the counter-example as the *goal path* in this thesis. Formally, a model with respect to a property is defined as a *transition system*. Its definition and other formal definitions used in this thesis are given in this section.

3.2.1 Transition System and Abstraction Database

A finite-state transition system is formalized in the following way.

Definition 3.2.1 [Transition systems] A finite state transition system is a tuple $M = (S, T, S_0, G)$, where

- S is a finite set of states,
- $T \subseteq S \times S$ is a transition relation,
- $S_0 \subseteq S$ is a set of initial states and
- $G \subseteq S$ is a set of goal states.

When the set of states of a finite state transition system is finite, that set 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, we will generate an abstract version of the original, concrete system.

Definition 3.2.2 [Homomorphic abstraction] Given a transition system $M = (S, T, S_0, G)$ 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 $\hat{M} = (\hat{S}, \hat{T}, \hat{S}_0, \hat{G})$, where

- $\hat{S} = \{\hat{s} \mid \hat{s} = H(s) \wedge s \in S\}$
- $(\hat{s_1}, \hat{s_2}) \in \hat{T}$ iff $\hat{s_1} = H(s_1) \wedge \hat{s_2} = H(s_2) \wedge (s_1, s_2) \in T$

- $\hat{S}_0 = \{\hat{s} \mid \hat{s} = H(s) \wedge s \in S_0\}$
- $\hat{G} = \{\hat{s} \mid \hat{s} = H(s) \wedge s \in G\}$.

A homomorphic abstraction is a kind of relaxation of the concrete transition system in which certain groups of states are merged. Clarke *et al.* [19] show that a homomorphic abstraction preserves a class of temporal properties (namely ACTL*). In other words, the concrete system and the abstract system satisfy the same set of formulae. If the abstract system does not satisfy some given property, we cannot conclude that the concrete system violates the property as the abstract system may generate *false negatives*. A false negative is a counter-example to the formulae that exists in the abstract system, but not in the concrete system. If such a counter-example is found, the model is further refined and again verified, and the process is repeated. This is called the *Counter-Example Guided Abstraction Refinement* scheme [21], often referred to as CEGAR.

In [75], the authors presented *Abstraction-Guided Model Checking*, which uses the abstraction only as a guide, or heuristic. In practice, an abstraction is represented by a so-called *pattern database*, which contains the shortest distance between each abstract state and the abstract goal state. It is proved in [73] that the distance between any state $\hat{s}$ and a goal state $\hat{g}$ in the abstract system is a lower bound to the distance between its corresponding concrete states s and g. The pattern database, therefore, can act as an *admissible* cost function to guide an A^*-based search towards the goal state in the concrete system. If the search algorithm finds the goal state then the admissibility property means that the path it has found is guaranteed optimal.

3.2.2 Symbolic Representation

In symbolic model checking, sets of states can be encoded as binary decision diagrams (BDDs) [7]. A BDD is a graphical representation of a *Boolean expression*, which is constructed by *Boolean variables* that range over two constants 0 and 1; *Boolean connectives* such as *conjunction* $\wedge$, *disjunction* $\vee$, *negation* $\neg$, *implication* $\rightarrow$, and *bi-implication* $\leftrightarrow$; and the two constants, 0 and 1. By definition, a Boolean variable x and the two constants 0 and 1 are Boolean expressions, and if t_1 and t_2 are Boolean expressions, so are $\neg t_1$, $t_1 \wedge t_2$, $t_1 \vee t_2$, $t_1 \rightarrow t_2$, and $t_1 \leftrightarrow t_2$. We can denote a Boolean expression f with n variables by $f(x_0, x_1, \ldots, x_{n-1})$, and a list of assignments (of value 0 or 1) to each of the variables x_i (truth assignment) by a Boolean vector. When assigning a Boolean vector, a Boolean expression can be evaluated to a value of 0 or 1. Graphically, a BDD consists of the set of nodes, which are Boolean variables and the two constants 0 and 1, and the set of "0" and "1" edges connecting nodes. For example, the Boolean expression $f(x_0, x_1, x_2) = x_0 \wedge x_1 \vee x_2$ can be presented as a BDD in Figure 3.1. In that

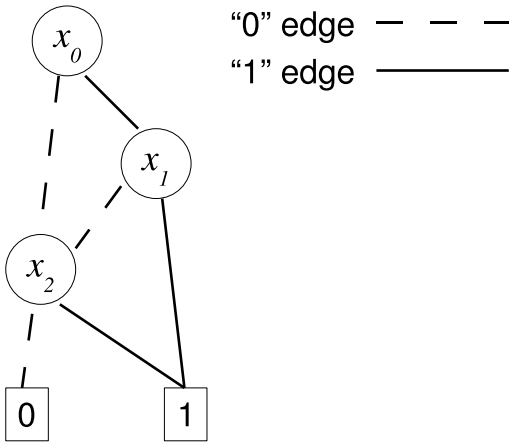


Figure 3.1: A BDD for $f(x_0, x_1, x_2) = x_0 \wedge x_1 \vee x_2$

figure, the expression is evaluated to 1 only when x_0 and x_1 are assigned to 1 ("1" edges connecting from node "x_0" to node "x_1" and to node "1"), or when x_2 is assigned to 1 ("1" edge connecting from node "x_2" to node "1"). In other cases, it is evaluated to 0. Therefore, for the given Boolean vector $(0, 1, 0)$, it is easy to see that f is evaluated to 0. A Boolean vector $(0, 1, 0)$ for the Boolean expression $f(x_0, x_1, x_2)$ can be also denoted by $(\overline{x_0}, x_1, \overline{x_2})$.

In model checking, a transition system M can be represented symbolically using BDDs as follows: (1) let n be the number of Boolean variables needed to represent the entire state space (which means that 2^n is the smallest number greater than the number of states in M); (2) every state in M can be represented as a Boolean vector over n Boolean variables; (3) let x_i' be the corresponding variable in the successor states s' of variable x in a state s, the transition $s \rightarrow s'$ in M can be expressed as $(x_0, x_1, \ldots, x_{n-1}) \rightarrow (x_0', x_1', \ldots, x_{n-1}')$. For example, the transition system on the left of Figure 3.2 with 8 states can be expressed as a BDD consisting of 3 Boolean variables x_0, x_1, and x_2. The transitions from state denoted 101 to states denoted 000,

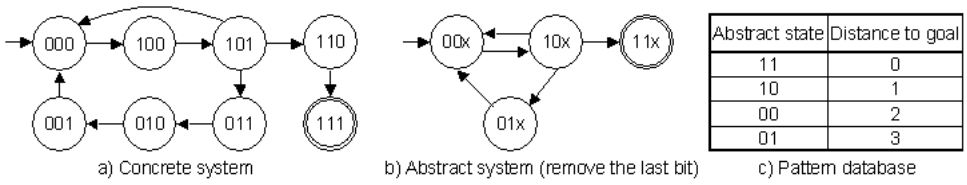


Figure 3.2: An example of an abstraction of a concrete system, and its pattern database

and 110 can be expressed as a transition relation $r = (x_0 \wedge \overline{x_1} \wedge x_2) \wedge ((\overline{x_0'} \wedge \overline{x_1'} \wedge \overline{x_2'}) \vee (x_0' \wedge x_1' \wedge \overline{x_2'}))$, or $r = (x_0 \wedge \overline{x_1} \wedge x_2) \wedge (((\overline{x_0'} \wedge \overline{x_1'}) \vee (x_0' \wedge x_1')) \wedge \overline{x_2'})$.

In the above example, the full transition relation of the model can be expressed as follows:

$$
\begin{aligned}
R = & (((\overline{x_0} \wedge \overline{x_1} \wedge \overline{x_2}) \wedge (x'_0 \wedge \overline{x'_1} \wedge \overline{x'_2})) \\
& \vee ((x_0 \wedge \overline{x_1} \wedge \overline{x_2}) \wedge (x'_0 \wedge \overline{x'_1} \wedge x'_2)) \\
& \vee ((x_0 \wedge \overline{x_1} \wedge x_2) \wedge ((\overline{x'_0} \wedge \overline{x'_1} \wedge \overline{x'_2}) \vee (\overline{x'_0} \wedge x'_1 \wedge x'_2)) \vee (x'_0 \wedge x'_1 \wedge \overline{x'_2})) \\
& \vee ((x_0 \wedge x_1 \wedge \overline{x_2}) \wedge (x'_0 \wedge x'_1 \wedge x'_2)) \\
& \vee ((\overline{x_0} \wedge x_1 \wedge x_2) \wedge (\overline{x'_0} \wedge x'_1 \wedge \overline{x'_2})) \\
& \vee ((\overline{x_0} \wedge x_1 \wedge \overline{x_2}) \wedge (\overline{x'_0} \wedge \overline{x'_1} \wedge x'_2)) \\
& \vee ((\overline{x_0} \wedge \overline{x_1} \wedge x_2) \wedge (\overline{x'_0} \wedge \overline{x'_1} \wedge \overline{x'_2})))
\end{aligned}
$$

For a given set of states $ss = \{010, 110\}$, the set of successor states ss' can be generated as follows:

$$
\begin{aligned}
ss &= \{010, 110\} \equiv (\overline{x_0} \wedge x_1 \wedge \overline{x_2}) \vee (x_0 \wedge x_1 \wedge \overline{x_2}) \\
T &= ss \wedge R \\
&= ((\overline{x_0} \wedge x_1 \wedge \overline{x_2}) \wedge (\overline{x'_0} \wedge \overline{x'_1} \wedge x'_2)) \\
&\quad \vee ((x_0 \wedge x_1 \wedge \overline{x_2}) \wedge (x'_0 \wedge x'_1 \wedge x'_2)) \\
ss' &= T(x'_0, x'_1, x'_2) = (\overline{x'_0} \wedge \overline{x'_1} \wedge x'_2) \vee (x'_0 \wedge x'_1 \wedge x'_2) \equiv \{001, 111\}.
\end{aligned}
$$

When the operations on BDDs have been shown effectiveness [7], and the number of states that can be stored during the checking process is large [16], the scalability of symbolic model checking is improved.

In GOLFER, an abstraction model $\hat{M}$ of a concrete model M (as described in Section 3.2.1) is generated by removing 'weak' variables[4]. For example, the concrete model on the left of Figure 3.2 can be abstracted into the model in the middle of that figure by ignoring variable x_2. The goal state and abstract goal state are shown as concentric circles. The abstract transition relation is then reduced by removing this variable. For example, the abstract transition relation of r above is $\hat{r} = (x_0 \wedge \overline{x_1}) \wedge ((\overline{x'_0} \wedge \overline{x'_1}) \vee (x'_0 \wedge x'_1))$. Actually, all transitions that transit into the same states are removed.

[4]Note again that those variables are deemed 'weak' based on a data dependency analysis when the number of variables depending to those variables is small [75].

In GOLFER, the heuristic function is represented as a *pattern database*, which records the distances between all abstract states and the abstract goal. For example, the pattern database of the abstract model in Figure 3.2 is shown on the right of that figure. We choose to store all abstract states equi-distant from the goal state into a single BDD. Note that the set of variables in the abstract model and that of the concrete model are the same. The only difference is that a few variables in the abstract model always have a special value, which is either 0 or 1 at the same time. General speaking, each abstract state represents a set of concrete states. For example, the abstract state denoted 01`x`, whereas `x` is the special value that can be set to 0 or 1 arbitrarily, represents the set of concrete states $\{010, 011\}$. It also means that a BDD of abstract states in the pattern database and a BDD of concrete states can be operated directly without a decoding. We denote a pattern database as $adb = \{b_i\}$, where each b_i contains a set of abstract states that have a distance i to the abstract goal state.

Given a pattern database $adb = \{b_i\}$, any set of states ss can be divided into partitions, each is a set of equi-distant states according to the pattern database adb. The partitioning operation simply involves conjugating abstract patterns b_i and sets of states ss. For example, consider the abstraction in Figure 3.2 again. Given the abstract state $b_1 = 10 \equiv (x_0 \wedge \overline{x_1})$ and the set of states $ss = \{100, 011, 010\} \equiv ((x_0 \wedge \overline{x_1} \wedge \overline{x_2}) \vee (\overline{x_0} \wedge x_1))$, we compute the set ss_1, that any state in which has a distance 1 to the goal states, by $ss_1 = b_1 \cap ss \equiv (x_0 \wedge \overline{x_1}) \wedge ((x_0 \wedge \overline{x_1} \wedge \overline{x_2}) \vee (\overline{x_0} \wedge x_1)) = x_0 \wedge \overline{x_1} \wedge \overline{x_2} \equiv \{100\}$.

In Abstraction Guided Model Checking (AGMC) scheme proposed by Qian and Nymeyer [73], given a pattern database, at each step, after generating the set of successor states ss, a partitioning method will be called to divide ss into partitions. The distance which each subset is from the abstract

goal state is its heuristic cost h.

3.2.3 Random-Walk

In this thesis, we focus on random walks in a transition system only.

Definition 3.2.3 [Random walk] A *random walk* in a transition system $M = (S, T, S_0, G)$ is a path $\rho = s_0 s_1 \dots s_i s_{i+1} \dots s_n$ in which the initial state $s_0 \in S_0$, $s_i \in S$ for $1 \leq i \leq n$, and the state s_{i+1} is chosen uniform randomly from the set of successors of s_i denoted $Img(s_i) = \{s \mid (s_i, s) \in T\}$. A random walk is *accepted* if $\exists\, 0 \leq k \leq n, s_k \in G$.

Assuming that the set of walks is measurable, the probability of a walk can be computed as follows:

Definition 3.2.4 [Probability of a walk] The probability of a walk $\mathbf{Pr}[\delta]$ is defined as $\mathbf{Pr}[s_0] = |S_0|^{-1}$ and $\mathbf{Pr}[s_0 s_1 \dots s_{n-1} s_n] = \mathbf{Pr}[s_0 s_1 \dots s_{n-1}] \times |Img(s_{n-1})|^{-1}$.

In checking a model, a random-walk algorithm traverses the state space of a given transition system using random walks, starting at some initial state. A random walk is accepted if it reaches a goal state. We refer to such a random walk as a *goal path*. Although some research proposed only one random walk with a length which is long enough to cover (most of) the state space [44], many recent algorithms use a number of random walks to detect the goal state. There are 2 particular issues that need to be resolved when using a random-walk algorithm: when do we terminate each random walk, and when do we terminate the algorithm? We refer to the former as the *end-walk*

condition, and to the latter as the *termination condition*.

End-walk condition: A random walk may never end, so given finite resources such as time and memory, we need to determine some condition to end walks prematurely. In the research of Grosu and Smolka [42], the end of a walk occurs at the first loop. In other research [44, 63, 64, 65], the maximum length of a walk is bounded arbitrarily.

Termination condition: Even if all walks can end, we also need to consider how many walks will need to be carried out in the search for the goal state. In the research of Owen *et al.* [63, 64, 65], the maximum number of random walks is bound arbitrarily. An alternative method involves tracking how many 'new' states are being discovered relative to 'old' states and terminating when some *saturation point* is reached [57].

Yet another termination condition, this time probabilistically determined, was defined by [42]. Given a system S, and a linear temporal logic (LTL) property φ, their work builds a Büchi automaton $B = B_S \times B_{\neg\varphi}$, and checks whether the language of this automaton $L(B)$ is empty. They refer to a random walk, which ends in a cycle, as a *lasso*, and state that a lasso, which contains an accepting state in its cycle, as an accepting lasso. The research involves sampling lassos, and if an accepting lasso is found then that is the witness of $L(B) \neq \emptyset$. Otherwise, to calculate the maximum number of samples that are needed, they associate a Bernoulli random variable Z, which takes value 1 with probability p_Z and value 0 with probability $1 - p_Z$, for each lasso.

Let X be a *geometric* random variable whose value is the number of independent samples required before arriving at one that is accepted. The cumulative distribution function of X for N independent samples of Z is

$F(N) = \mathbf{Pr}[X \leq N] = \sum_{n \leq N}(\mathbf{Pr}[X = n]) = \sum_{n \leq N}((1 - p_Z)^{n-1} p_Z) = 1-(1-p_Z)^N$. Given small numbers δ and ϵ, and requiring that $F(N) \geq 1-\delta$ and $p_Z \geq \epsilon$, then from the equation, $N \geq \ln(\delta)/\ln(1-\epsilon)$. Therefore, after sampling $N = \ln(\delta)/\ln(1-\epsilon)$ lassos (without finding an accepting one), they conclude $L(B) = \emptyset$. They further report that with $p_Z \geq \epsilon$, the probability of finding an accepting lasso by further sampling the state space (after N samples) is less than δ.

There is of course a difference between the probability that an error state is discovered by random search, and the probability the error occurs in real life, so it is not clear how useful this quantification is in practice. Nevertheless, the strength of this work is the formal underpinning it provides, and the fact that properties can be expressed in LTL. Furthermore, this probabilistic termination condition can be applied only if the search space covers the probability space.

3.3 Summary

In this chapter, we have given a technical overview of model checking from the perspective of guided and random-walk based search.

Having presenting the technical background of this thesis, in the following chapters, we will present our works in applying the guidance into the model checking using random walks.

Chapter 4

Random-Walk Based Model Checking

This chapter is based on the work [8] presented at the International Workshop on Advanced Computing and Application (ACMOP2008) in Ho Chi Minh City, Vietnam in March 2008.

The changes made are:

- *The algorithm is shown in more detail for the* ranMove *function, which randomly selects a successor state at each step in a walk.*
- *The case studies used in the experimentation are described in more detail.*

4.1 Introduction

In this chapter we change the core depth-first search algorithm in SPIN so that it does not verify a system but simulates the system using an algorithm

based on random walks.[1] The new tool is called ranSPIN. Note that, ranSPIN searches the model with respect to a property, hence it is a model checker. We also carry out a series of experiments on well-known problems and protocols with ranSPIN, and show how its performance compares to SPIN, particularly with respect to scalability and its ability to find errors.

Our work in this chapter is closely related to the work of Owen *et al.* [63, 64, 65], and Grosu and Smolka [42] in application of random-walk search into the model checking field. Although, in general, the structure and the (time and space) complexity of our algorithm are equivalent to those of the above two algorithms, our algorithm allows users to bound the length of walks arbitrarily (as in SPIN). As well, there is an option in our algorithm to start a new walk at a random state from the previous walk to avoid wasting time in the initial parts of the state space.

The rest of this chapter is organized as follows: Section 4.2 presents the random-walk algorithm implemented in SPIN, and Section 4.3 describes the experiments. At the end of this chapter, the summary and the motivation for the following chapter are presented.[2]

4.2 Random-Walk Based Model Checking

In this section we develop a random-walk algorithm and apply it to SPIN. We also study a new strategy called *random-start random walk*, in which a new random walk starts from a random state in the previous random walk (instead of always starting from the initial state).

[1]The reader should not confuse this with the simulator option provided by SPIN. The simulator can work randomly, but it does not play any role in formal verification, although it will check assertions.

[2]For the technical background, please refer to Chapter 3.

In SPIN, a model is presented in PROMELA language, a C-like programming language. It usually consists of a set of concurrency processes, and can be checked in a wide range of properties using SPIN: deadlock, livelock, assertion, or a linear temporal logic (LTL) expression expressed in a so-called *never-claim* process[3]. Given a model consisting of a set of processes P and a never-claim process *pnc*, and by considering all processes in P and the never-claim process as Büchi automata, SPIN generates an automata $A = (\bigcup P) \times pnc$, where $(\bigcup P)$ is the union of all processes in P. The non-emptiness of $L(A)$ is proved if a state is accepted in the never-claim process. The path from the initial state to that state is a witness showing that the property presented by the never-claim process is violated. To improve the efficiency, the state space of A is generated on-the-fly. A new state is generated in SPIN by making a move in a process in P and a move in the never-claim *pnc* always at the same time. A move is actually an execution of a transition of a process at the current state.

In Figure 4.1, we show our random-walk based model-checking algorithm. This algorithm replaces SPIN's depth-first search algorithm. The algorithm randomly and repeatedly traverses the state space of a given model (P with type ***set of processes***, and *pnc* with type ***never-claim process***) until some termination condition, called *terminationCond*, is satisfied. Each traversal is restricted in length by the end-walk condition, *endCond*. Just as in SPIN, in each traversal a move in a process (line 8) is followed by a move in the never-claim process (line 9). The subsequent state of the system is recorded in *path* (line 10). The algorithm will return the goal path if the current state s is accepted in *pnc* (lines 11 and 12). The algorithm will return

[3]Note that each of the properties can be expressed in a never-claim process. See [47] for more detail.

RANSPIN algorithm

Input: ***set of processes*** P, ***never-claim process*** pnc,
boolean $ranStart$

Output: $goal\ path$ | **not found**

```
 1:  while (! terminationCond)
 2:    if (ranStart ∧ (path ≠ ∅))
 3:      s = random(path)
 4:    else
 5:      s = init()
 6:    path = ∅
 7:    while (! endCond)
 8:      s = ranMove(P, s)
 9:      s = move(pnc)
10:      path = path ∪ {s}
11:      if (accepted(pnc, s))
12:        return path
13:  return not found
```

Figure 4.1: The random-walk model-checking algorithm in RANSPIN

not found if the termination condition is satisfied.

In the algorithm in Figure 4.1, a single step in a process is executed in the function *ranMove* (Figure 4.2), which first randomly selects an executable process from the set of processes P, and then randomly selects one of the successor states of the current state (of that process). To do that, a process p is first randomly selected from the set of all processes P (line 17), then the

function ranMove(s, P)

15: $s' = s, plist = P$

16: **while** $((s' == s) \wedge (plist \neq \emptyset))$

17: $p = random(plist)$

18: $plist = plist - \{p\}$

19: $tlist = allTrans(p, s)$

20: **while** $((s' == s) \wedge (tlist \neq \emptyset))$

21: $t = random(tlist)$

22: $tlist = tlist - \{t\}$

23: $s' = execute(t, s)$

25: **return** s'

Figure 4.2: The *ranMove* function in the random-walk algorithm RANSPIN

set of all transitions at the current state of that process *tlist* is generated by calling the function *allTrans* (line 19), and a transition is selected from *tlist* (line 21). That transition is then executed to generate a successor state s' (line 23). If it cannot be executed, then s' is unchanged (and equivalent to the current state s). In this case, another transition is then selected. If all transitions of the selected process p are non-executable, another process is selected and the work is repeated. The function *random* in lines 3, 17 and 21 is used to select an element from the input set randomly and uniformly. The resulting successor state of the function *ranMove* is the new current state.[4] Similarly, the function *move* (in line 9) is a random execution of the

[4]Note that the selection strategy in the *ranMove* function is not totally random. The strategy is sufficiently random however to provide an interesting comparison in research

never-claim process.

An option to start a random walk from some arbitrary point in the previous path is given by the Boolean variable *ranStart*. If *ranStart* is true, the function *random* (in line 3) selects a state, uniformly and randomly, from the previous *path* as a starting point for the new random walk (lines 2 and 3). The random-start option is useful to increase the penetration of the state space. Instead of 'wasting time' in the initial parts of the state space retracing earlier paths, the random walk can branch out from an earlier path deep in the state space. The expectation is that this will improve coverage and hence efficiency.

The space complexity of the algorithm is based on the length of the path. If D is the average length of all traversals, then on average, the space complexity of the algorithm is $O(D)$. The time complexity is the product of the number of traversals and the time for each traversal. On average, this is $O(N * D * \eta)$, whereas N is the number of traversals before the goal is found or the termination condition is satisfied, and η is the time of randomly selecting a successor state (at each step).

4.3 Experimentation

In a series of experiments, the performance of SPIN and RANSPIN were compared. The tool RANSPIN offers the user various simulation strategies. These include conventional random walks and modified random walks where the walk starts at a random point in an earlier random walk.

The tools SPIN and RANSPIN were applied to three well-known case studies: the dining-philosophers problem, the leader-election protocol, and the

that we will describe later.

mutual-exclusion protocol. As we are interested in finding error states, the error version of each of these case studies was used. The aim of course was to see how quickly SPIN and RANSPIN found this error state for each of the case studies. The maximum length of walks (the end-walk condition) and maximum number of random walks (termination condition) also needed to be considered. We experimented with both bounded and unbounded walks. The termination condition of RANSPIN was the limitation of number of trials of 2020[5]. If this limit was not reached, the time limit was set to 1 minute. As RANSPIN is based on SPIN, we used the same random-number generator, which is the *rand*() function in the C programming language.

The use of a random-number generator means that the experiments are statistical in nature and hence need to be repeated with different seeds. In the results that we present, each data point (generally representing the execution time) is the average of 50 repetitions of the experiment, each using a different seed in the random-number generator. The seeds are the 50 prime numbers starting from 23. We also calculated the standard deviation of the data. All the experiments were performed on a Pentium IV 3.0GHz with 1GB RAM running a Linux operating system called Ubuntu 6.06.

Note that SPIN is used in this work without any special option (see [49]). This means all default options. such as partial-order reduction [36], is employed in the search. This ensures that SPIN is fastest and consumes the least memory in general (as pointed out in [48]). We could of course change the options to see if it is possible to improve the performance of SPIN in particular cases, but this makes comparisons more complex, and optimising SPIN is not part of this research. SPIN, NUSMV and GOLFER, are used

[5]There is, by following MC^2 theory, the result of $N = \ln(\delta)/\ln(1-\epsilon) \approx 2020$, whereas δ is 4×10^{-5} and ϵ is 5×10^{-3}.

as "paradigm benchmarks" in this research that allow comparison with our (guided) random-walk paradigm.

4.3.1 Case Studies

Dining-philosophers problem The philosophers are connected in a ring and will dead-lock if they all wait on each other to pick up a free fork.

Leader-election protocol This protocol[6] is used to determine which process may use a single resource that is shared by processes connected in a ring. Messages are passed around the ring between neighbouring processes. A process is given access to the resource when it is released by another process. It also allows an arbitrary process to send its unique identifier around the ring after a timeout. When this identifier arrives back at the sender, the sender process is given access to the resource. This strategy allows an arbitrary process to gain access to the resource. However, it also may result in two processes accessing the resource at the same time, which is an error state.

Mutual-exclusion protocol This protocol is used by concurrent processes to claim a critical section of code. To enter the critical section, a process raises a flag and assigns its identifier to a system variable *lastID*. A process is granted the critical section if no other process's flag is raised or no other process has set *lastID*. In our version, we allow a process to first set *lastID* and then raise its flag. If another process sets *lastID* immediately after the first process sets that variable, then both processes may be in the critical section at the same time. For example,

[6]The specifications of this protocol (both the original version and error version) and the dining-philosophers problem can be found at `http://anna.fi.muni.cz/benchmark`

suppose that the set of actions in a process is $\{rf,id,cr\}$, which stand for 'raise flag', 'set *lastID*' and 'critical section' respectively. Two processes p and q may cause an error if the following occurs: $p.id$, $q.id$, $q.rf$, $q.cr$, $p.rf$, $p.cr$.

4.3.2 Results

In Figure 4.3, we show the results of applying SPIN and RANSPIN to the **dining-**

ph.	SPIN			RANSPIN				
	time	cxl	mem	time	std dev	N	cxl	mem
10	0.000	2298	2.622	0.005	0.009	2.28	243.84	2.428
20	0.020	6358	2.724	0.003	0.007	1.58	693.04	2.448
30	0.020	9998	2.929	0.008	0.010	1.62	1060.64	2.480
40	0.030	14478	6.116	0.012	0.010	1.72	1575.84	2.539
50	0.040	18538	6.526	0.015	0.013	1.56	1981.60	2.589
60	0.060	22598	7.038	0.014	0.010	1.44	2392.16	2.639
70	0.100	26658	7.550	0.030	0.024	1.64	3025.68	2.770
80	0.120	30718	8.164	0.029	0.019	1.38	2961.28	2.767
90	0.140	34778	8.881	0.037	0.022	1.52	3443.12	2.916
100	0.180	38838	9.598	0.038	0.021	1.34	4008.00	2.977
110	0.200	42898	10.622	0.051	0.030	1.52	4357.84	3.187
120	0.250	46958	11.646	0.056	0.046	1.66	4488.56	3.392
130	0.300	51018	12.670	0.061	0.041	1.38	5022.72	3.383
140	0.320	55078	13.694	0.058	0.029	1.38	5577.76	6.415
150	0.360	59138	14.923	0.096	0.068	1.66	5860.88	6.830

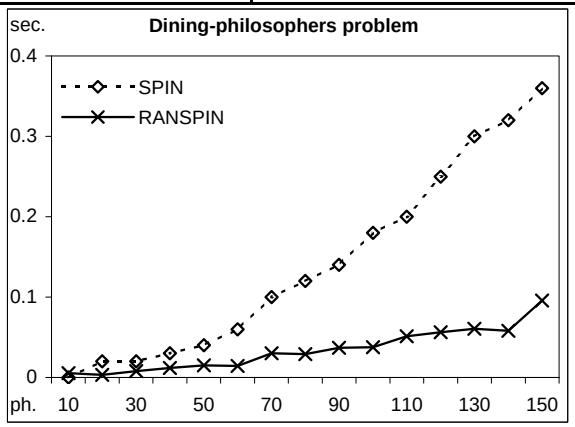


Figure 4.3: Comparing SPIN and RANSPIN for the dining-philosophers problem.

philosophers problem. The table on the top of this figure shows the

execution time **time** (in seconds), the amount of memory used **mem** (in megabytes) and the length of the goal path **cxl** for both algorithms. We also include the standard deviation **std dev** of the execution time, which can describe the variability of the execution time, and the average number of traversals **N** (before reaching the goal state) in the case of ranSPIN. The column **ph.** indicates the number of philosophers in the system. The chart on the bottom plots the execution times for up to 150 philosophers. As we can see, ranSPIN is faster than SPIN by a factor of between 2 and 4, and shows the same factor improvement in memory usage. The column **N** shows that ranSPIN on average took more than one random walk to find the error, but surprisingly, the length of the goal path discovered by ranSPIN is around 10 times shorter than that of SPIN.

To avoid the search algorithm going off 'forever' on a random walk (exhausting time or memory) we apply a bound to the length of random walks. However, setting this bound can be 'tricky' in practice as the bound must not be too short as this will mean the search algorithm never reaches the error state. Setting the bound is hence a trade-off between efficiency and reachability.

The comparison between SPIN and ranSPIN for the **leader-election protocol** is shown in Figure 4.4. The column headings on the table in this figure are the same as those in Figure 4.3, except that column **pr.** indicates the number of processes. We note that SPIN runs out of memory (denoted *oom*) with 9 processes. ranSPIN remains very fast independent of the number of processes, showing excellent scalability. In our version of this protocol, after a non-deterministic timeout, an arbitrary process may send a message around the ring and wait for it to arrive back to claim the resource. SPIN simulates the non-deterministic choice as a backtracking strategy, and may

pr.	SPIN			RANSPIN				
	time	cxl	mem	time	std dev	N	cxl	mem
3	0.010	7122	2.724	0.028	0.029	7.60	273.54	5.367
4	0.070	34100	6.219	0.019	0.017	5.98	288.18	5.371
5	0.310	136598	12.491	0.028	0.033	8.22	353.24	8.616
6	1.292	556796	52.324	0.022	0.025	7.18	349.04	2.553
7	6.168	2064834	399.102	0.037	0.040	10.28	384.80	5.504
8	21.389	7795092	648.958	0.037	0.034	10.42	375.84	5.547
9	*oom*			0.043	0.039	11.28	458.48	5.610

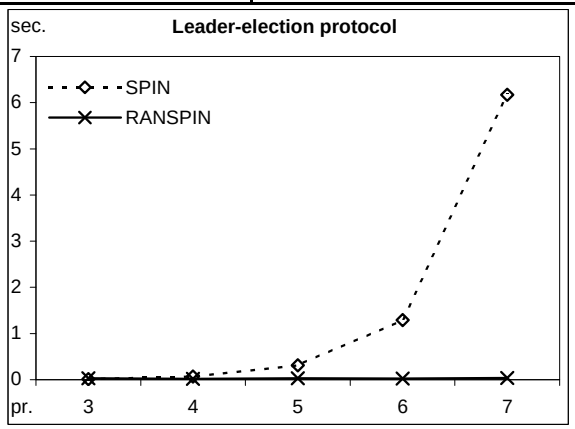


Figure 4.4: Comparing SPIN and RANSPIN for the leader-election protocol.

choose 'incorrectly' and head off in the direction away from the error state. There is no way for the search to correct itself. RANSPIN, in contrast, will start a new random walk when a walk terminates naturally by reaching a previously visited state. By taking new random walks, choices that do not lead to success can be corrected in RANSPIN. The result is a very fast, scalable algorithm, which uses orders of magnitude less memory, and generates counter-examples that are many orders of magnitude shorter.

The results for the **mutual-exclusion protocol** are shown in Figure 4.5. We see that RANSPIN is orders of magnitude faster, and uses far less memory. In the mutual-exclusion protocol, we are interested in whether any two processes find themselves in the critical section at the same time. If that occurs, we have an error state, and it does not matter in what state the other pro-

pr.	SPIN			RANSPIN				
	time	cxl	mem	time	std dev	N	cxl	mem
8	0.020	9999	2.827	0.003	0.007	1.96	318.48	2.427
12	0.040	9999	2.929	0.003	0.006	2.06	428.68	2.432
16	0.100	9999	3.544	0.003	0.006	1.96	577.12	2.439
20	0.190	9999	3.953	0.004	0.007	1.96	782.56	2.450
24	0.420	9999	5.387	0.004	0.008	1.96	853.44	2.458
28	0.810	9999	7.435	0.005	0.008	1.96	857.88	2.463
32	1.040	9999	9.688	0.005	0.007	1.96	1153.80	2.485
36	1.970	9999	15.729	0.005	0.007	1.96	1019.88	2.484
40	3.850	9999	21.976	0.006	0.007	1.96	1256.40	2.506

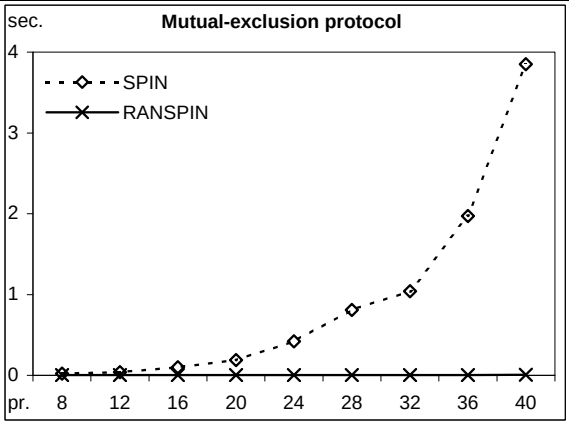


Figure 4.5: Comparing SPIN and RANSPIN for the mutual exclusion protocol.

cesses are. In this situation, SPIN performs badly if it does not choose the 'critical' processes to execute during the depth-first search. The column **cxl** of SPIN in the table of the figure indicates that the depth of the search is bounded to 10000. (Reducing SPIN's bound does improve the performance, but not significantly.)

The performance of SPIN in the leader-election and mutual-exclusion protocols (but not the dining-philosophers problem) degrades quickly when the number of processes in the models increases (see Figure 4.4 and Figure 4.5). Of course, one reason for this is the increasing size of the state space, but another reason is the location of the error state(s) (within the state space) and the fact that SPIN uses a depth-first search algorithm. Generally, a depth-

first search algorithm always selects the left-most or always the right-most branch at a state during the search, and performs backtracking if it reaches a dead-end. In the worst case, if the depth-first algorithm always chooses the left-branch for example, and the error state is on the extreme right of the state space, the search algorithm will traverse the entire state space before achieving success. Conversely, the best case is when they are both left or both right.

To study this problem more closely, we varied the location of the error state in the mutual-exclusion protocol with respect to the selection strategy. We then repeated the experiment to see the variation between the best-case and worst-case performance. The results can be seen in Figure 4.6. The

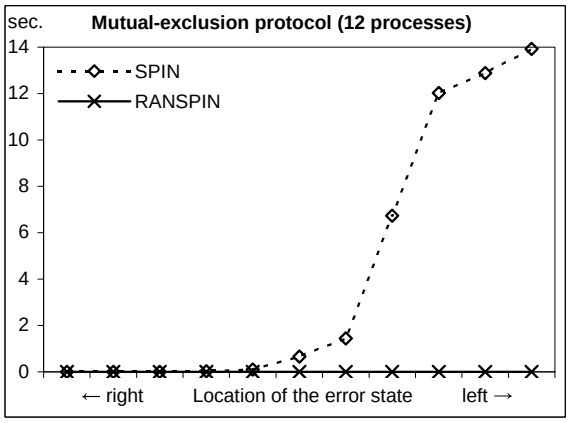


Figure 4.6: Comparing SPIN and RANSPIN for best-case and worst-case behaviour.

difference between best and worst case for SPIN is very large. In contrast, it had no effect for RANSPIN.

Finally, we conducted an experiment to see what effect a random start had on RANSPIN. (This does not apply to SPIN of course.) In Figure 4.7, we compare results for RANSPIN in normal mode, where each random walk starts

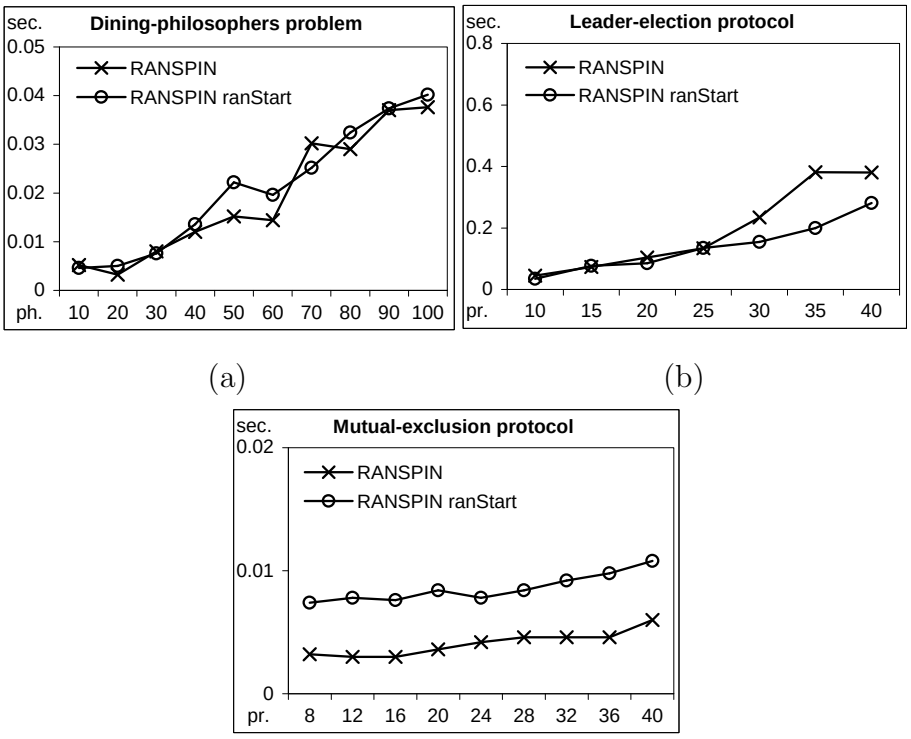


Figure 4.7: Comparing RANSPIN for random walks that start normally and start randomly.

at the initial state, and random-start mode, where each random walks starts from a random state selected from the previous path. We denote the latter by *ranStart*. We reran RANSPIN for each of the three case studies.

In the results for the dining-philosophers problem, shown in Figure 4.7(a), we see that there is no significant advantage in using *ranStart*. In the leader-election protocol results shown in Figure 4.7(b), there is a significant improvement in the *ranStart* version if the number of processes is large. However, for the mutual-exclusion protocol, we see in Figure 4.7(c) that *ranStart* consistently performs worse. At first, this was a puzzling result. We had expected

better performance for *ranStart* because there should be less 'path overlap' and there would be more randomness. However, if a random walk goes the 'wrong' direction and terminates, starting the next random walk from somewhere on this path is likely to be a poor strategy. It is then better to start from the initial state (and go in a completely different direction). This shows that selecting the best strategy can require quite deep model insight and understanding of the topology of the state space and the nature of the error state.

4.4 Summary

In this chapter we have developed a random-walk based model-checking algorithm, and modified the tool SPIN to implement the algorithm. The new tool, ranSPIN, has far superior performance than SPIN in speed, memory and length of counter-example in experiments on three case studies. As well as being faster and more efficient, ranSPIN is more robust and able to quickly find errors located in 'corners' of the state space. The underlying random walk search scales well in time and memory on our experiments.

It should be noted that neither SPIN nor ranSPIN is optimal: neither model checker can find the shortest counter-example that proves a property is violated by a model, although ranSPIN generally found shorter ones than SPIN. Of course, to place this work in context, if a model is tractable and no error is found, then SPIN is a verifier that can *guarantee* that a system satisfies a given property. ranSPIN can never do this: it is a simulator that is able to scour very large state spaces and find errors. As such, in software development, ranSPIN would be used early in the development to find errors, and SPIN later in the development to verify correctness.

In the next chapter we apply the random-walk approach to a symbolic model checker using an earlier developed tool GOLFER *as platform.* GOLFER *includes a heuristic mechanism to guide a search towards an error state. The aim is to integrate the random-walk approach and guidance mechanism in a single tool.*

Chapter 5

Symbolic Guided Random-Walk Based Model Checking

This chapter is based on the work [11] presented at the 7th International Conference on integrated Formal Methods (iFM'09) in Düsseldorf, Germany in February 2008.

The changes made are:

- *The abstraction pattern database is discussed in more detail.*
- *A proof of the theorem that random trails covers the state space is given.*

5.1 Introduction

As mentioned in previous chapters, the poor scalability of conventional model checking approaches limits its use. The two main approaches in tackling this problem are guided search [31, 75], which avoids exhaustively examining the

whole state space by directing the search in some particular area where the errors are seemly located, and random-walk search [88, 42], which randomly samples the state space in locating the errors. Regardless of the fact that in the worst-case, guided search is equivalent to exhaustive search, it is a verification technique that can guarantee the correctness of a model of a system with respect to a given temporal property. Moreover, given well-informed heuristics, it can speed-up the conventional model checking dramatically [62]. In contrast, a random-walk search can only partially verify correctness. Being only partial, it is really just confidence building because in general no amount of random walks will guarantee correctness. A major reason for the popularity of simulation in industry is that no matter how large and complex a system is, it is always possible to carry out a simulation (using random-walk), unlike conventional model checking (including guided search model checking), where users are (still) inevitably confronted by time/space limitations and intractability.

Pelánek *et al.* [68] studied how different characteristics of the state-space graph (taken from real-world case studies) effect the behaviour of the random walk; in particular the coverage achieved by the walk and the probability distribution of visiting states. Both [68, 83] noticed that the random walk could get trapped in small, dense 'clumps' of the state space. These clumps are characterised by a small subset of program variables that are critical to the behaviour of the program. Owen and Menzies [63] refer to this concept as *funnel theory*, and *clumping*: the effective state space of a program, which determines its critical behaviour, is relatively small when compared to all reachable states, and is dominated by a small number of key variables. In very recent work (2007) [58], these are called *collar* variables. A developer searching for errors only needs to sample the range of collar variables. Other

variables play only a minor role and can be safely ignored. In other work [76] that inspired this research, the 'critical' behaviour of programs was studied. In that research, a data-dependency analysis is used to rank variables, and those that were deemed 'weak' were abstracted away automatically. In essence, Qian and Nymeyer [76] do a *static* analysis of the critical behaviour, and Owen and Menzies a *dynamic* analysis. Whatever approach is used, research [65, 76] suggests that exhaustive search used in conventional model checking can be an overkill, and that a partial, random search can be more effective and cost much less.

In this work, we extend the work of Qian and Nymeyer on symbolic guided model checking and add the capability of (guided) random-walk based search. In their work, a so-called *pattern database*, is generated using a static-analysis method that abstracts the given model and generates a smaller model. The pattern database stores the heuristic (i.e. estimated) distance to the goal state for every state in the (abstract) state space. These distances are used to guide a search through the state space (i.e. A*). This same heuristic can also be used to guide a random walk through the state space of course. The random-walk based search was also implemented explicitly in the previous chapter in the tool RANSPIN. In our experiments, the tool RANSPIN outperforms SPIN in running time, memory usage and the length of the goal path (the path from an initial state to an error state).

The new algorithm uses the same symbolic heuristic as in the earlier research (i.e. the research of Qian and Nymeyer) to bias the random walks towards particular states that potentially violate user-defined properties. We also characterise models in terms of the number of processes that must 'participate' in an interaction that results in a state which violates a given property. We call this characterisation the *Process Error Participation* (PEP)

factor. A low PEP factor means that few processes in the model are involved; a high PEP factor means that most or all are involved. PEP depends on both the user-defined property and the model: a deadlock may be caused by just two processes claiming a resource, or it may require all processes to claim the resource and halt progress. Note that our work is different to the seemingly related work of Groce and Visser [40]. In that work, they study the so-called 'Thread Preference Heuristic' in generating the heuristics for multi-threaded Java programs. Our heuristics are built by model abstractions, and the PEP factor is used to predict when substantial performance gains can be expected from particular model-checking approaches.

Some technical background (i.e. the work of Qian and Nymeyer) is outlined in Section 5.2. Our algorithmic contribution is described in Section 5.3, where we define *random trails*, present an algorithm to compute random trails and additionally, modify the algorithm to allow walks and trails to be *guided*. The experimental work is described in Section 5.4, including the comparison of various random-walk and random-trail based algorithms, both guided and unguided. The summary and suggestions for future research are given in the last section.

5.2 Symbolic Abstraction-Guided Model Checking

Qian and Nymeyer present a symbolic, guided model-checking framework. The framework is implemented in a system called GOLFER, which in turn is based on NUSMV [18].

For comparison purposes, the AGMC algorithm described in that framework [73] has been slightly modified, such as instead of building the abstract

database in the algorithm, the modified version receives it as an input. This modification allows the abstract database to be reuse in other algorithms in this research once it was built. The modified version is renamed SGMC here. It is shown in Figure 5.1. In particular we have removed those constructs

---

SGMC algorithm

Input: ***transition system*** $M = (S, T, S_0, G)$, ***abstractDB*** adb

Output: *goal path* | **not found**

1: $openSet = \{(S_0, 0, 0)\}$
2: $visitedStates = \emptyset$
3: **while** $(openSet \neq \emptyset)$
4: $(ss, cost, h) = getMin(openSet)$
5: $visitedStates = visitedStates \cup ss$
6: **return** $buildPath(G, visitedStates)$ **when** $(ss \cap G \neq \emptyset)$
7: $ss' = \{s' \mid s \in ss, (s, s') \in T\} - visitedStates$
8: **for all** $b_i \in adb$
9: $ss'' = b_i \cap ss'$
10: $openSet = openSet \ \cup \{(ss'', cost + 1, i)\}$ **unless** $(ss'' == \emptyset)$
11: **return not found**

Figure 5.1: A Symbolic Abstraction-Guided Model-Checking Algorithm

in the AGMC algorithm that concern the symbolic implementation. We do this in all the symbolic algorithms shown in this chapter.

The SGMC algorithm maintains a set of current states *ss* called the *openSet* from which the algorithm must choose optimally. For each state in this set, the algorithm records the exact cost of reaching that state and the predicted

cost h of reaching the goal from that state, also called the heuristic value. The *getMin* function, which extracts a set of states that has a minimum cost ($cost + h$) from the input open set, is called in each iteration. If it reaches a goal, the algorithm will return a path from an initial state to a goal state, which is called a *goal path*. The function *buildPath* extracts the goal path from *visitedStates*, starting at the goal states and working backwards to an initial state. The set of successor states ss' is computed in line 7. Those states in ss' that have already been visited are also removed. The set is then partitioned (line 8 to 10). Each partition is a set of states that are equi-distant from the goal, as given by the abstract pattern database, *adb*. The cost of a successor state is set to the cost of the parent state plus 1, and the state is added to *openSet*. Remembering that the algorithm is symbolic, the reader should note that an element in *openSet* is a tuple consisting of a BDD, which encodes a set of states being equi-distant from the goal state, together with the corresponding current cost and heuristic cost. This process is repeated until the goal is reached.

In the actual symbolic version of the algorithm, the partitioning operation involves conjugating abstract patterns b_i and sets of states. For example, consider the abstraction in Figure 3.2 (on page 41) again. In that example, the concrete system, which is represented by the three Boolean variables x_0, x_1, x_2, is abstracted by omitting the last Boolean variable x_2. The abstract database $adb = \{(b_i, i)\}$ is shown in more detail in Table 5.1. The process of partitioning the set of states $ss' = \{100, 011, 010\}$, for example, is as follows:

- $ss'' = b_0 \cap ss' \equiv (x_0 \wedge x_1) \wedge ((x_0 \wedge \overline{x_1} \wedge \overline{x_2}) \vee (\overline{x_0} \wedge x_1)) = 0 \equiv \emptyset$.
- $ss'' = b_1 \cap ss' \equiv (x_0 \wedge \overline{x_1}) \wedge ((x_0 \wedge \overline{x_1} \wedge \overline{x_2}) \vee (\overline{x_0} \wedge x_1)) = x_0 \wedge \overline{x_1} \wedge \overline{x_2} \equiv \{100\}$.
- $ss'' = b_2 \cap ss' \equiv (\overline{x_0} \wedge \overline{x_1}) \wedge ((x_0 \wedge \overline{x_1} \wedge \overline{x_2}) \vee (\overline{x_0} \wedge x_1)) = 0 \equiv \emptyset$.

b_i	abstract state	Boolean expression
b_0	11	$x_0 \wedge x_1$
b_1	10	$x_0 \wedge \overline{x_1}$
b_2	00	$\overline{x_0} \wedge \overline{x_1}$
b_3	01	$\overline{x_0} \wedge x_1$

Table 5.1: An example of abstract database

- $ss'' = b_3 \cap ss' \equiv (\overline{x_0} \wedge x_1) \wedge ((x_0 \wedge \overline{x_1} \wedge \overline{x_2}) \vee (\overline{x_0} \wedge x_1)) = \overline{x_0} \wedge x_1 \equiv \{011, 010\}$.

That means, the set of state $\{100, 011, 010\}$ is divided into two subsets: $\{100\}$ with the heuristic cost of 1, and $\{011, 010\}$ with the heuristic cost of 3. Those subsets (with the corresponding costs $f = g + h$) are put back into the open set. Actually, the above processing in Boolean operations is done using the BDD operations (described in [7]) on the corresponding BDD structures represented ss' and b_i.

5.3 Random-Walk and Abstraction-Guided Algorithms

In this section we develop the random-walk and abstraction-guided algorithms that form the basis of this work. In Section 3.2.3 (Chapter 3), a random walk has been defined as a path in the state space. It is too general because of the presence of loops in a walk. In this Chapter, we define a subset of random walks called *random trails*.

Definition 5.3.1 [Random trail] A *random trail* in a transition system

$M = (S, T, S_0, G)$ is a random walk $\theta = s_0 s_1 \ldots s_n$ in which $\forall\, 0 \leq i \neq j \leq n$, $s_i \neq s_j$. A random trail is *accepted* if $\exists\, 0 \leq k \leq n, s_k \in G$.

Definition 5.3.2 [Probability of a random trail] The *probability of a trail* $\mathbf{Pr}[\theta]$ is defined as: $\mathbf{Pr}[s_0] = |S_0|^{-1}$ and $\mathbf{Pr}[s_0 s_1 \ldots s_{n-1} s_n] = \mathbf{Pr}[s_0 s_1 \ldots s_{n-1}] * |Img(s_{n-1}) - \{s_0, s_1, \ldots, s_{n-1}\}|^{-1}$.

Definition 5.3.3 [Maximal random trail] A *maximal* random trail is a random trail that ends at a leaf state or a state in which all of its children have already been visited.

Theorem 5.3.4. *Given a transition system $M = (S, T, S_0, G)$ and assume that all states in S are reachable from some state in S_0. The set of <u>all</u> maximal random trails in S:*

1. *covers the entire state space; and*
2. *together with their probabilities, covers the probability space.*

Proof. There are two points:

- Point (1) is trivial when all states in S are reachable from some state in S_0.
- Point (2) can be proved by induction on the number of states.

  It is true for the system with only one state. Suppose that, it is true for the system with n states. Given an n-state system M, add a new state s'_0 to make a new system M' with $(n+1)$ states. Without loss

of generality, suppose that the new state s'_0 is the only initial state of M' that has only transitions to the old initial states of M, called $s^1_0, s^2_0, \ldots, s^k_0$, and that there is no transition in M to s'_0.

In system M (which consists of n states), according to the theorem, the sum of all probabilities of maximal random trails in M is 1, $\Upsilon = 1$.

By definition,

$$\begin{aligned}\Upsilon &= \Sigma_{i\in[1,k]}(\mathbf{Pr}[s^i_0 s^i_j \ldots]) \\ &= \tfrac{1}{k}\Sigma(\mathbf{Pr}[s^1_j \ldots]) + \tfrac{1}{k}\Sigma(\mathbf{Pr}[s^2_j \ldots]) + \ldots + \tfrac{1}{k}\Sigma(\mathbf{Pr}[s^k_j \ldots])\end{aligned}$$

where $s^i_0 s^i_j \ldots$ is a maximal random trail that start at the initial state s^i_0. The result of this sum in M' is:

$$\begin{aligned}\Upsilon' &= \Sigma(\mathbf{Pr}[s'_0 \ldots]) \\ &= \Sigma_{i\in[1,k]}(\mathbf{Pr}[s'_0 s^i_0 s^i_j \ldots]) \\ &= (\Sigma_{i\in[1,k]}(\mathbf{Pr}[s'_0 s^i_0]\Sigma(\mathbf{Pr}[s^i_j \ldots]))) \\ &= \tfrac{1}{k}\Sigma(\mathbf{Pr}[s^1_j \ldots]) + \tfrac{1}{k}\Sigma(\mathbf{Pr}[s^2_j \ldots]) + \ldots + \tfrac{1}{k}\Sigma(\mathbf{Pr}[s^k_j \ldots]) \\ &= \Upsilon \\ &= 1.\end{aligned}$$

That means, the sum of all probabilities of maximal random trails in M' is 1. The proof is done.

□

Given input parameters δ and ϵ, and by considering maximal random trails only, we can terminate the random-search algorithm after executing $N = \ln(\delta)/\ln(1-\epsilon)$ random trails if no goal is found. This is the same result as that of Grosu and Smolka [42].

Given a transition system $M = (S, T, S_0, G)$, the maximum length of a maximal random trail is $|S|$. In that worst case, if the trail does not contain the goal state, the goal state cannot be reached from the initial state.

Some care is needed in the random-trail algorithm to record visited states because a trail is not allowed to revisit a state. In this research, states that have been visited in a trail are represented by a BDD, which is used to reduce (using BDD set operations) the set of potential successor states of the current state, which of course is also represented by a BDD. The actual successor state is (randomly) chosen from this set of states. Alternatively, in an explicit-state context, a hash table is used to store and retrieve visited states (as in SPIN [47]).

In a *guided* random walk, each state is assigned its minimum heuristic cost (i.e. distance) to a goal state. In this work, we use an abstraction of the system as a heuristic to assign a cost to each state, and use this cost to guide the supposedly random walk towards a possible goal. During a walk, states with smaller costs are chosen more often than states with larger costs. This abstraction heuristic mechanism is precisely the same approach used in AGMC [75]. In Figure 5.2, we show the new symbolic abstraction-guided random-walk model-checking algorithm SGRMC. Remember, as in SGMC, the algorithm is actually symbolic, and the details of the BDD data structures and operations have been left out. The algorithm handles both walks and trails and both guided and unguided random walks. If the variable *guided* is false, the algorithm is a "pure" (i.e. unguided) random-walk algorithm, which for experimental purposes we call SRMC. Otherwise the algorithm will be guided, and is referred to as SGRMC.

The algorithm SGRMC randomly and repeatedly traverses the state space of the given system until *terminationCond* is satisfied. Each traversal is restricted in length by *endCond*. In each traversal, all visited states are recorded (line 5) in *visitedStates*. The algorithm will return the goal path if a goal is located (lines 6). Otherwise, the algorithm will return **not found** when

SGRMC algorithm

Input: ***transition system*** $M = (S, T, S_0, G)$, ***abstractDB*** adb,
boolean $guided$, ***boolean*** $randTrail$

Output: $goal\ path$ | **not found**

```
1:  while (! terminationCond)
2:    s = random(S_0)
3:    visitedStates = ∅
4:    while (! endCond(s))
5:      visitedStates = visitedStates ∪ {s}
6:      return getPath(visitedStates) when (s ∈ G)
7:      ss = {s' | (s, s') ∈ T}
8:      ss = ss − visitedStates when (randTrail)
9:      if (guided)
10:       costSet p = ∅
11:       for all b_i ∈ adb
12:         ss'' = b_i ∩ ss
13:         p = p ∪ {(ss'', i)} unless (ss'' == ∅)
14:       ss = guidedRandom(p, |adb|)
15:     s = random(ss)
16: return not found
```

Figure 5.2: A Symbolic Abstraction-Guided Random-Walk Model Checking algorithm

the termination condition is satisfied. The function *getPath* examines the times the states were visited and extracts the goal path. The set of succes-

sor states *ss* of a state *s* is generated in line 7. The input Boolean variable *randTrail* enables us to carry out random walks or random trails. If *randTrail* is true then we remove the visited states whenever they occur in the set of successor states (lines 8). The function *random* chooses a next state, uniformly and randomly, from *ss* (lines 2 and 15).

Like the SGMC algorithm, the algorithm divides *ss* into partitions, which are stored in a variable p that has type ***costSet*** (line 10 to 13). Each partition is assigned a cost i, where i is the abstract distance to the goals of all the states in that partition (line 13). A guided random-selection function, called *guidedRandom*, is called to compute the set of successor states *ss* (line 14). This function takes as input $p = \{(ss_0, i_0), \ldots, (ss_n, i_n)\}$ and the size of the input abstract database $|adb|$, and selects one of the sets ss_k from p in a biased fashion. The smaller the cost i_k, the greater the probability that ss_k will be selected. To achieve this, each instance ss_k is repeated $|adb| - i_k$ times, and then a set is selected randomly. The set of successor states is then restricted to that partition only. Thereafter, a successor state is selected uniformly and randomly from the (restricted) set of successor states (line 15).

5.4 Experimentation

In this section, we apply the four algorithms SRMC, SGRMC, SGMC and NUSMV to four well-known problems: the dining-philosophers problem, leader-election protocol, send/receive protocol and mutual-exclusion protocol. Precisely the same (abstraction-based) heuristic is used in both guided algorithms SGMC and SGRMC. As we are interested in finding bugs, 'error' versions of each of the protocols that violate particular properties are used in this research.

In the first series of experiments (Section 5.4.3), we were interested in finding which of the random-walk algorithms SRMC$_w$, SRMC$_t$, SGRMC$_w$ and SGRMC$_t$ is 'fastest'. The subscripts 'w' and 't' in the names of those algorithms indicate that the corresponding algorithm uses random-walks or random-trails, respectively. In the second series of experiments (Section 5.4.4), we compare the performance of the 'fastest' random-walk algorithm with the guided-search algorithm SGMC and conventional NUSMV. Remember though that random-walk algorithms are a form of simulation, so in our experiment we are in fact comparing the performance of a simulator and two verifiers in finding an error in a model.

As in the previous chapter, each data point in the random-walk experiments is the average of executing the algorithm 50 times, where each execution uses a different seed in the random-number generator. Further note that, in this research, the conventional breadth first search in NUSMV is invoked by constructing the property of the model as an INVAR property, and calling NUSMV without options. All the experiments were also performed on a Pentium IV 3.0GHz with 1GB RAM running a Linux operating system called Ubuntu 6.06.

5.4.1 Case Studies

The dining-philosophers problem, the leader-election protocol, and the mutual-exclusion protocol are presented in Chapter 4. A simple instance of the dining-philosophers problem that consists of just two philosophers *p1* and *p2* is illustrated in Figure 5.3. In this system, the status of each philosopher is one of $\{t, lf, rf, e\}$, which stand for 'thinking', picking the 'left fork', picking the 'right fork' and 'eating' respectively. In the error state, each philosopher has a fork in the left hand.

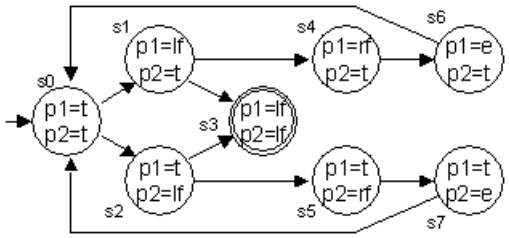


Figure 5.3: A simple dining-philosophers problem

A send/receive system is a communication system, in which a sender sends out data on a communication channel, and a receiver receives data from that channel. Synchronisation occurs by acknowledgement signals. An error state occurs when either the sender or the receiver times-out. Figure 5.4 depicts a simple instance of this protocol. In this system, after sending data,

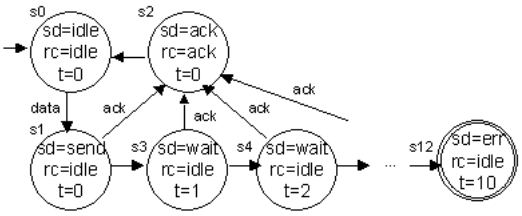


Figure 5.4: A simple send/receive protocol

the sender *sd* waits for an acknowledgement from the receiver *rc*. If no acknowledgement is received by the count of 10 (arbitrarily set), the sender signals an error.

5.4.2 Process Error Participation - PEP

We have characterised the above protocols in terms of the number of processes that 'participate' in an interaction that results in an error state. We call this characterisation the *Process Error Participation* (PEP) factor. A low PEP factor means that few of the existing processes in the system are involved;

a high PEP factor means that most or all are involved. For example, it is easy to see that in the dining-philosophers problem, all processes need to 'work together' to result in deadlock, so its PEP factor is high. The leader-election protocol also generally has a high PEP factor as all processes are involved when an error occurs[1]. In contrast, in the send/receive protocol, an interaction between just two processes may give rise to an error[2], so it normally has a low PEP factor. Similarly, in the mutual-exclusion protocol, we need just two processes to be in the critical section at the same time for an error to occur, so it also has a low PEP factor. We have considered the PEP factor in interpreting the results of each experiment.

We note that the PEP factor is dependent on the property that the user defines. So, for example, if the property in the mutual-exclusion protocol is that all processes cannot be pair-wise in the critical section at the same time, then the model is high PEP. We add that there is no relationship between the PEP factor and the tightness/looseness of the process coupling in the model.

5.4.3 Which is the 'fastest' random-walk based algorithm?

In this first series of experiments, we compare the 'unguided' random-walk SRMC$_w$, 'unguided' random-trail algorithm SRMC$_t$, guided random-walk algorithm SGRMC$_w$ and guided random-trail algorithm SGRMC$_t$. We first carry out the experiments with no bound on the length of the random walks and trails, and then repeat the experiments with a bound. We remind the reader

[1]They need to pass a token around the ring in order to vote for a leader.

[2]A sender process signals an error if it waited for an acknowledgement signal (from a receiver process) for a count of 10 after sending data.

that all the systems contain an error, so verification is not being performed.

Unbounded Random Walks and Trails

Dining Philosophers Problem: In Figure 5.5, we compare the experi-

ph.	SRMC$_w$				SRMC$_t$				SGRMC$_w$				SGRMC$_t$			
	time	std	N	len	time	std	N	len	time	std	N	len	time	std	N	len
10	0.32	0.3	1	361.6	0.30	0.3	1.12	285.2	0.19	0.1	1	215.7	0.17	0.1	1.06	155.1
12	0.89	0.7	1	602.8	0.83	0.5	1.04	531.3	0.66	0.4	1	467.0	0.43	0.3	1.02	284.9
14	1.98	1.6	1	968.8	1.65	1.2	1.06	783.5	0.96	0.7	1	471.7	0.92	0.7	1.06	375.3
16	5.78	5.0	1	2080.9	4.66	3.7	1.08	1532.6	2.75	1.9	1	975.6	2.02	1.5	1.16	564.4
18	12.34	12.0	1	3298.7	16.26	14.6	1.12	3870.6	6.20	4.8	1	1528.6	3.42	2.7	1.04	753.3

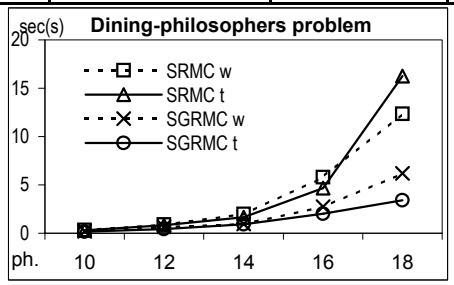


Figure 5.5: Performance of unbounded random walks on the dining-philosophers problem

mental results of all algorithms for the dining-philosophers problem. In the table we show the number of philosophers **ph.**, execution **time** (in seconds), standard deviation of the execution time **std**, number of traversals to reach the goal **N** and length of the goal path **len**. On the bottom of the table we plot the execution times. There are a number of observations that can be made:

1. The 'walk' algorithms reach the goal in just 1 traversal, while the 'trail' algorithms may need more than one. The reason is that trails must restart if all the successors of a state have been visited before.

2. The length of the goal path is much greater than optimal (note the minimum length is twice the number of philosophers).

3. On average, the fastest algorithm is the guided trail algorithm, SGRMC$_t$, which appears to scale well.

We also looked at the distribution of the data, and noting that the guided data had a lower standard deviation than the unguided data, found that the guided data was better behaved. In Figure 5.6 we show for example the time taken to find the deadlock for 16 dining philosophers using SRMC$_t$ and using SGRMC$_t$. It is easy to see that the time taken to find the deadlock for

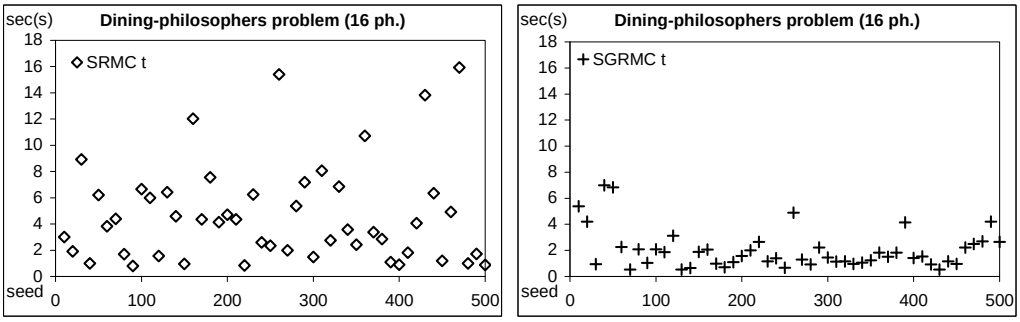


Figure 5.6: Distributions of execution time for 16 dining philosophers using SGMC$_t$ and SGRMC$_t$

16 dining philosophers using (unguided) random-walk search (SRMC$_t$) shows wilder fluctuations than that of guided random-walk search (SGRMC$_t$).

Leader-Election Protocol: The other high PEP factor system is the leader-election protocol, whose results are shown in Figure 5.7. The column **pr.** indicates the number of processes. The rest of the columns have the same meaning as before. We note the following:

1. The behaviour of the standard deviation (of the time) and the required number of traversals is similar to the dining-philosophers problem.

pr.	SRMC$_w$				SRMC$_t$				SGRMC$_w$				SGRMC$_t$			
	time	std	N	len	time	std	N	len	time	std	N	len	time	std	N	len
3	0.92	0.9	1	1584.3	0.81	0.7	1	936.8	0.42	0.4	1	685.2	0.33	0.3	1	333.9
4	4.67	4.1	1	4644.7	2.91	2.4	1	1566.7	3.02	2.8	1	3068.2	2.32	1.9	1	1206.1
5	29.85	31.0	1	19246.0	16.07	13.9	1	5262.1	13.61	13.2	1	8915.7	10.05	9.6	1	3025.8
6	176.94	217.1	1	69492.8	89.30	81.6	1	16664.5	106.75	96.6	1	42177.5	33.31	35.0	1	6126.3

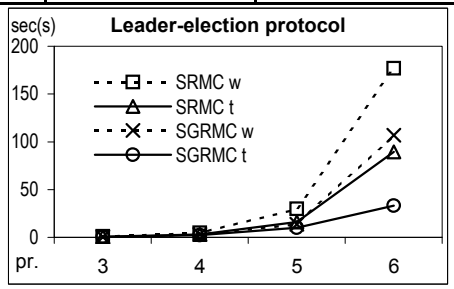


Figure 5.7: Random walks for the leader-election protocol

2. The use of trails seems more effective than walks.

3. The guided trail algorithm SGRMC$_t$ is fastest and scaled the best.

Send/Receive Protocol: Unlike the previous two protocols, this protocol has a low PEP factor. The experimental results are shown in Figure 5.8. The main observation here is that the guided walk and guided trail algorithms show similar behaviour, and are both much faster than the unguided algorithms. The conjecture is that the guide is more effective in a low PEP system because the costs of states associated with processes that play a role in the violation of the property are significantly lower than other costs of other states, so it is easier to generate a guide that is informed, and hence effective.

Mutual Exclusion Protocol: The results for this other low PEP factor system is presented in Figure 5.9. The main observation here is that all algorithms perform well, although the guided algorithms perform better, taking

pr.	SRMC$_w$				SRMC$_t$				SGRMC$_w$				SGRMC$_t$			
	time	std	N	len	time	std	N	len	time	std	N	len	time	std	N	len
3	56.84	65.2	1	95589.4	32.55	42.4	3.7	7070.1	0.06	0.01	1	124.9	0.07	0.02	1	119.9
4	119.51	139.1	1	89829.8	43.67	61.9	1.1	25032.2	0.12	0.03	1	132.4	0.13	0.03	1	125.5
5	244.79	338.1	1	105878.6	124.81	130.7	1.0	46591.5	0.15	0.06	1	125.0	0.15	0.05	1	121.7
6	343.32	413.2	1	104177.0	250.56	270.7	1.0	68416.9	0.20	0.07	1	120.3	0.21	0.06	1	119.4

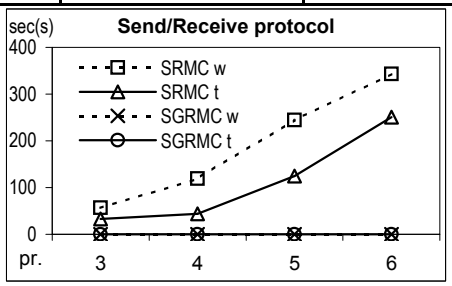


Figure 5.8: Random walks for the send/receive protocol

pr.	SRMC$_w$				SRMC$_t$				SGRMC$_w$				SGRMC$_t$			
	time	std	N	len	time	std	N	len	time	std	N	len	time	std	N	len
16	0.22	0.21	1	111.9	0.20	0.16	1	107.6	0.12	0.06	1	41.3	0.13	0.06	1	41.3
20	0.29	0.26	1	93.9	0.31	0.29	1	97.5	0.20	0.09	1	42.8	0.20	0.08	1	42.0
24	0.55	0.46	1	123.0	0.53	0.42	1	114.7	0.23	0.11	1	30.3	0.24	0.12	1	30.3
28	0.54	0.42	1	82.5	0.55	0.44	1	85.2	0.34	0.20	1	35.5	0.34	0.20	1	35.5

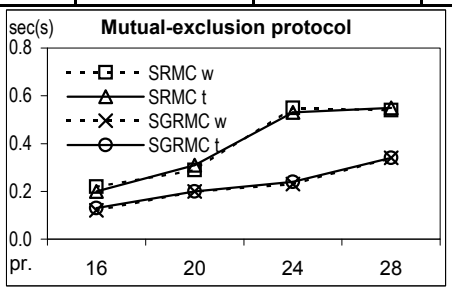


Figure 5.9: Random walks for the mutual-exclusion protocol

approximately half the time of the unguided algorithms. The reason for this general good performance is that there are many paths that lead to an error state, and the error state is close by, so all search algorithms, guided or not, can quickly find the error state.

Bounded Random Walks and Trails

It is easy to speculate that an *unbounded* random walk through a state space (as all the experiments above are) can waste time 'going in the wrong direction'. In such cases, terminating the walk prematurely, and restarting it, may improve its chances of finding the error. However, the downside to this strategy is that if the bound is too short, we may never walk far enough to find the error (it may always stop short). There is hence a trade-off between the length and number of walks. To understand this trade-off better, we arbitrarily apply a bound to the length of all random walks. This bound is 100 times the number of concurrent processes in the system. The experiments are repeated with this bound.

In Figure 5.10(a), we show the execution time for the **dining-philosophers**

ph.	SRMC$_w$	SRMC$_t$	SGRMC$_w$	SGRMC$_t$
10	0%	0%	0%	0%
12	-2%	-1%	2%	2%
14	7%	0%	6%	0%
16	22%	26%	14%	7%
18	41%	6%	-1%	4%

(a)

pr.	SRMC$_w$	SRMC$_t$	SGRMC$_w$	SGRMC$_t$
3	39%	28%	0%	-15%
4	186%	44%	96%	28%
5	640%	100%	522%	72%
6	182%	140%	623%	173%

(b)

pr.	SRMC$_w$	SRMC$_t$	SGRMC$_w$	SGRMC$_t$
3	-91%	-82%	0%	0%
4	-90%	-77%	-17%	-15%
5	-92%	-84%	13%	13%
6	-92%	-88%	5%	5%

(c)

pr.	SRMC$_w$	SRMC$_t$	SGRMC$_w$	SGRMC$_t$
16	0%	5%	0%	-8%
20	0%	-3%	0%	0%
24	0%	0%	0%	-4%
28	0%	0%	0%	0%

(d)

Figure 5.10: Percentage change in execution time in the bounded (a) dining-philosophers problem, (b) leader-election protocol, (c) send/receive protocol, and (d) mutual-exclusion protocol

problem, expressed as a percentage change of the previous time. For example, consider the case with 18 philosophers and the unguided algorithm

SRMC. The bound is set to 1800 steps, with the result that (on average) the search is 41% *slower* than the unbounded case. The main conclusion one can draw from these results is that arbitrarily bounding the walk/trail generally results in a worse execution time, but that the guided algorithms are less affected than unguided algorithms.

In Figure 5.10(b) we see in the results for the **leader-election protocol** that the degradation in performance is even more pronounced, but that the trail algorithms are less affected. In the results for the **send/receive protocol** shown in Figure 5.10(c), we see that the unguided walk algorithm's performance improves approximately 90% when we bound the walk (so e.g. the execution time for bound SRMC is approximately 27 seconds, which we note is still 100 times slower than SGRMC). Remembering from the unbounded experiment that the goal is only a short distance from the initial state, it is intuitively obvious that bounding the unguided algorithms is desirable to avoid long walks 'getting lost'. The guided algorithm already efficiently finds the error, so it gains little by being bound.

Finally we have the results for the **mutual-exclusion protocol** shown in Figure 5.10(d). In this figure we see that bounding the walks and trails has virtually no effect on the time. What is important to note here is that, while this protocol and the send/receive protocol both have error states at only a short distance from the initial state, the number of processes in this protocol is much higher than the send/receive protocol (28 versus 6). This means that the bound for this protocol is much larger (remember that the bound is set to 100 times the number of processes), and a large bound is in effect no bound.

Discussion

These experiments suggest that unguided algorithms are faster with a bound than without if the bound is appropriate (i.e. not too long). Clearly, the bound should not be a function of just the number of processes but should take into account the structure of the state space and closeness of the error. In certain circumstances it is advantageous to apply a bound to random search (e.g. if the error is 'close' and the guiding abstraction is weak). However, any bound carries a risk of preventing the search from finding errors that are 'too far away'.

Overall, the guided algorithms are significantly faster for high PEP systems, and orders of magnitude faster for low PEP systems. In particular, the guided trail algorithm SGRMC is almost always the fastest, and it finds the shortest path to the error state (but this path may still be longer than the optimal path, as we shall see in the next section). Comparing the unguided and guided random trail algorithms, we see that the guided random trail algorithm always results in a shorter path, which suggests that the guide is having a significant effect. It would be interesting to study the relationship between the size of this effect and the 'quality' of the heuristic. There did not seem to be any relationship between the size of the effect and the PEP factor.

5.4.4 How much faster is (guided) simulation than verification?

In a second series of experiments, we compare the fastest simulation algorithm (SGRMC$_t$) with two verifiers: the symbolic model checker NUSMV and the symbolic abstraction-guided model checker SGMC. For the purposes of

comparison, we consider just the unbounded version of SGRMC$_t$.

In Figure 5.11, we show the comparison for the **dining-philosophers**

ph.	NuSMV			SGMC			SGRMC$_t$			
	time	mem	len	time	mem	len	time	std	mem	len
4	0.01	0.76	8	0.01	0.87	8	0.01	0.00	0.88	31.80
6	0.06	1.70	12	0.12	2.47	12	0.03	0.01	1.25	68.60
8	0.56	8.82	16	1.78	11.46	16	0.07	0.02	1.82	91.90
10	11.91	11.41	20	54.26	15.15	20	0.17	0.07	3.29	155.10
12	106.75	40.58	24	168.50	37.49	24	0.43	0.27	6.65	284.90
14	882.94	203.69	28	15704.50	337.47	28	0.92	0.72	9.51	375.30
16	8693.00	886.83	32	> 10h			2.02	1.50	11.20	564.40
18	> 10h			> 10h			3.42	2.66	11.51	753.30
20	> 10h			> 10h			7.11	5.35	12.24	1038.10

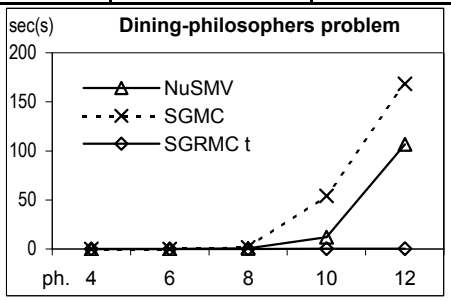


Figure 5.11: Comparing simulation and verification for the dining-philosophers problem

problem. The table on top of this figure shows the execution **time** (in seconds), the amount of memory used **mem** (in megabytes) and the length of the goal path **len** for each algorithm. We also include the standard deviation **std** in the case of SGRMC$_t$. The chart on the bottom plots the execution times for up to 12 philosophers. While non-optimal, the results show that SGRMC$_t$ is orders of magnitude faster. This is somewhat a surprising result as it is well known that the 'pathological' high level of symmetry in the dining-philosophers problem is problematic for guided verification. This does not extend to the guided simulation, even though it uses precisely the same heuristic. The reader may remember that even the unguided random-walk

algorithm SRMC did very well on this model, so clearly simulation is a better approach than verification in this case.

The poor performance of guided verification (SGMC) is well-known and caused by the symmetry in the model that makes it difficult to find a useful heuristic. In effect, in this model SGMC works breadth-first like NUSMV, but has extra overhead due to 'expensive' BDD (partitioning) operations that it must perform (and NUSMV does not). Also interesting is the efficient memory utilisation of SGRMC$_t$, remembering this algorithm is based on BDDs.

The graph of the performance of the other high PEP factor model, the **leader-election protocol**, is shown in Figure 5.12. The guided random-

pr.	NuSMV			SGMC			SGRMC$_t$			
	time	mem	len	time	mem	len	time	std	mem	len
3	0.74	10.00	18	0.48	6.86	18	0.33	0.30	4.36	333.86
4	20.37	22.93	19	12.56	11.58	19	2.32	1.87	10.16	1206.08
5	237.30	145.30	21	164.87	33.58	21	10.05	9.58	11.40	3025.82
6	2108.62	645.14	23	755.19	121.39	23	33.31	35.01	13.25	6126.34

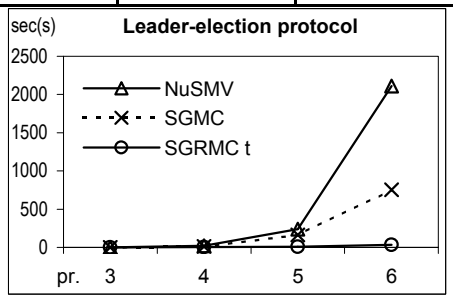


Figure 5.12: Comparing simulation and verification for the leader-election protocol

trail algorithm SGRMC$_t$ is again faster, and uses less memory, than the verification algorithms. However, comparing the verification algorithms, SGMC performed better than NUSMV. The reason for this is the system does not have as high a PEP factor as the dining-philosophers problem. While all processes in this system need to be involved in passing a message around the

ring, in our system, a process may unilaterally claim leadership. Because other processes will be unaware of this, an error can result. This lower PEP-factor behaviour allows the heuristic to be a more effective guide, resulting in significantly better performance.

In the random-walk experiments we saw that both guided random-walk and guided random-trail algorithms were particularly effective in low PEP factor systems. In Figure 5.13, we compare verification and simulation for the **send/receive protocol**. The graph in this figure shows that both guided

pr.	NuSMV			SGMC			SGRMC$_t$			
	time	mem	len	time	mem	len	time	std	mem	len
5	2.60	11.46	13	0.07	2.08	13	0.04	0.05	3.47	121.70
10	26.93	24.54	13	0.16	3.79	13	0.60	0.20	9.17	124.36
15	72.47	49.01	13	0.27	5.11	13	1.49	0.60	9.49	140.44
20	150.97	105.51	13	0.55	8.44	13	2.69	0.96	9.69	136.12
25	273.65	152.72	13	1.08	9.73	13	4.63	2.25	10.20	132.16
30	421.76	225.04	13	1.35	8.34	13	8.25	5.49	11.06	158.26

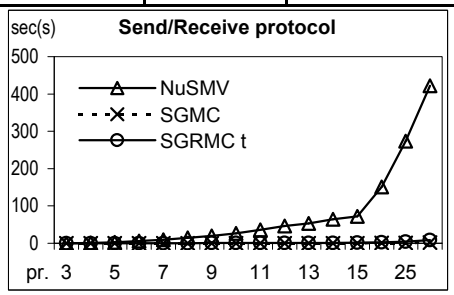


Figure 5.13: Comparing simulation and verification for the send/receive protocol

verification and simulation are very fast compared to NUSMV, and both require far less memory. It is conjectured that the asymmetry in the search space caused by the relatively few processes play a role in the error state makes guiding effective. Interestingly, guided verification has better performance than guided simulation; not only in execution time but also in memory usage. Contributing to the worse performance of simulation is undoubtedly

the long path it takes to reach the error state. As expected, both NUSMV and SGMC generate optimal goal path lengths, while SGRMC$_t$ does not by an order of magnitude.

The experimental results for the other low PEP factor system, the **mutual-exclusion protocol**, are shown in Figure 5.14. The same general behaviour

pr.	NuSMV			SGMC			SGRMC$_t$			
	time	mem	len	time	mem	len	time	std	mem	len
16	5.13	18.66	8	0.08	2.33	8	0.13	0.06	2.96	41.32
20	17.23	50.61	8	0.14	3.24	8	0.20	0.08	4.13	42.04
24	43.80	102.67	8	0.21	4.32	8	0.24	0.12	4.70	30.26
28	98.25	173.90	8	0.27	5.52	8	0.34	0.20	6.22	35.50
30	162.51	206.83	8	0.33	6.48	8	0.43	0.23	7.44	39.52
40	> 1hr			0.60	10.32	8	0.64	0.32	8.84	28.62
50	> 1hr			1.03	9.71	8	1.28	0.57	8.96	39.18
60	> 1hr			1.33	10.21	8	2.22	1.08	10.13	44.42
70	> 1hr			2.14	12.20	8	3.19	1.47	10.46	40.88

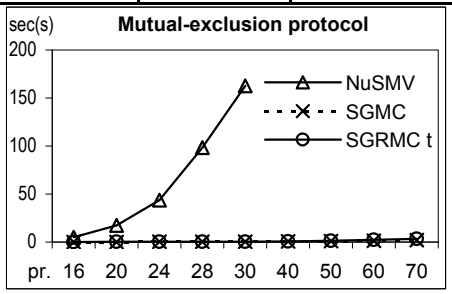


Figure 5.14: Comparing simulation and verification for the mutual exclusion protocol

described above is seen here: the guided algorithms SGMC and SGRMC$_t$ are both many orders of magnitude faster than (unguided) NUSMV, and require far less memory. However, the memory requirements of both algorithms are similar this time. As before, the verification algorithms are optimal, simulation is not.

Discussion

As expected, the guided random simulator SGRMC$_t$ is many orders of magnitude faster and uses less memory than the conventional verifier NUSMV in the series of experiments carried out here. In fact, SGRMC$_t$ can find errors in models that are intractable for NUSMV.

A comparison with the other verifier, SGMC, is quite different. Surprisingly, for low PEP factor systems, guided verification (i.e. SGMC) is faster than guided simulation (SGRMC). The reason for this is that, although precisely the same heuristic is used in guided verification and simulation, it plays a different role in each. In the former, there is no randomness. In the latter, the walk is still predominately random.

Of course, if successful, the guided verifier SGRMC generates an optimal goal path, and guided simulation does not. So, while a random-trail algorithm can find an error state very quickly, the path to the error may be long and circuitous, and one would assume unhelpful to the user unless the user had a tool, for example, that could prune the path.

5.5 Summary

In this work we have developed symbolic, guided random-walk and random-trail model-checking algorithms, applied these algorithms to four well-known problems, investigated the effect of bounding the length of walks and trails, and compared the fastest of these algorithms with conventional and guided symbolic model-checking algorithms. We will not repeat the discussions here of the experimental results from Section 5.4.3 and 5.4.4, except to say that the guided random-trail algorithm SGRMC$_t$ is generally the best at locating errors, and when it is not, the guided model checker SGMC is the best. Each

of the protocols studied in this chapter contained an error, and the error was always found. We did not consider what conclusions can be drawn if no error was found.

Noting that the guided random-walk approach uses precisely the same abstract model as heuristic as the guided model checker, a tool that allows the verification engineer to alternately and seamlessly carry out property-based simulation and verification is possible. What is necessary here is a 'feedback' loop that would integrate the two technologies and allow the verifier to benefit from the information obtained by the simulator.

Pelánek [68] has shown the structure, or shape, of state spaces for many specifications. The PEP characterisation that we have used is an attempt to relate these structures and properties. A low PEP factor can be viewed as indicating asymmetry in the state space. The more 'skew' the structure is with respect to the property, the easier it is to create an abstract model that will act as an effective heuristic. However, there is much work that needs to be done to understand the PEP factor. Can it be formalised for example? Does it need to take into account other aspects of the model or property?

Another aspect that needs to be considered is the method used to generate the abstract model. Currently this is done by using a 'crude' static analysis of the specification. While it ranks variables in terms of their dependencies, it does not, for example, rank variables with respect to the property under investigation, which could be important. Also not addressed in this chapter is the symbolic aspect. While all algorithms have been implemented using BDDs, the benefits (if any) of this approach have not been considered.

Finally, only small protocols have been studied in this chapter. The application of guided random walks to more realistic industrial case studies has yet to be carried out. The expectation however is that in real life, heuristics

can be very effective because small but identifiable parts of the specification considered with respect to the property under investigation generally dominate the behaviour and lead to states that violate the property.

The application of a heuristically driven guide to random walks has shown promising results in symbolic model checking, and provides the motivation to apply the same approach to the previously developed explicit-state random-walk based model-checking tool RANSPIN.

This page intentionally left blank.

Chapter 6

Explicit-state Guided Random-Walk Based Model Checking

In this chapter, the abstraction heuristic method used in the previous chapter is applied to explicit-state model checking. However, instead of sampling the state space purely randomly, the ranSPIN *algorithm in chapter 4 is modified to allow an abstraction heuristic to guide the random walk. The new algorithm (and tool) is called* granSPIN*. As well, various strategies that determine how successor states are 'randomly' selected are studied.*

This chapter is based on three publications [9, 10, 13]. In [9], the abstraction guidance approach is first described. In subsequent work [10, 13], the method of making heuristic is improved, and various random-walk successor-state selection strategies were studied. The efficiency of searches using 'good' (which steered the search towards the known error state) and so-called 'bad' (steered the search away from the error state) were compared in a series of experiments. These bad heuristics are referred to as 'mis-informed'.

6.1 Introduction

This chapter is an extension of Chapter 4 in this thesis, which is based on [8]. In that chapter, a tool called ranSPIN was first described. In that work, the depth-first search algorithm that SPIN [47] uses was replaced by a random-walk algorithm. ranSPIN simulates the model of the system (i.e. the property together with the system specification). The results of experiments comparing the performance of SPIN and ranSPIN on various protocols were presented. As expected, ranSPIN was found to be much faster (often orders of magnitude) and more space efficient in locating particular error states. Noteworthy was that SPIN could perform very well or very poorly, depending on the location of the error state relative to the route the depth-first search algorithm takes. In essence, the depth-first search algorithm sometimes 'gets lucky'. Sometimes it is 'unlucky' and needs to traverse almost the entire state space to find a particular error. ranSPIN in contrast was found to be very stable: given a heuristic, it performed equally well irrespective of the location of the error state.

In this chapter, the random-walk algorithm was modified by adding a guide. In guided random-walk based search, a successor state is selected 'non-uniform' randomly from the set of all successor states of the current state. The selection is based on an abstraction heuristic, which is generated from the original model. In essence, during selection, states that have smaller heuristic cost are selected more often than states that have larger heuristic cost. The issue on how the selection strategy should be implemented is also studied. In an unguided random-walk algorithm, a transition is uniform randomly selected and tested whether it is executable in the current state. If it is not, then another transition is selected. We name this strategy "try-a-child". In the guided random-walk algorithm, a different strategy may be

used. This strategy required all the successor states to be computed, and each state to be assigned a heuristic cost, and then the selection to be made. We name that strategy "all-children". The comparisons of the two strategies for both guided and unguided random-walk algorithms are then made.

We also study the effectiveness of coarse/fine abstraction heuristics in the guided algorithm. The abstraction of a concrete system that consists of multiple concurrent processes can be a single process or a small number of concurrent processes. In other work by the Qian and Nymeyer, an abstract model of the concrete system is derived (automatically) by using a static analysis that ranks the 'importance' of variables, and removes those variables that are deemed to be least important. The consequences of using a "mis-informed" heuristic that guides the search 'away from the goal' is also studied in this chapter.

In summary, the contributions of this chapter are the following:

- A guided random-walk based algorithm for an explicit-state model checker is proposed.
- This algorithm provides for two selection strategies:
    1. the all-children strategy in which all successor states are generated before selecting one
    2. the more efficient try-a-child strategy in which a single successor state is selected and tested for validity
- Experiments are carried out comparing SPIN, unguided RANSPIN and guided GRANSPIN using different selection strategies and heuristics.
- The 'informedness' of the heuristic, which says how much information the heuristic contains about the location of error states, is studied.

- The difference between well-informed and mis-informed heuristics is investigated and related to the nature of the error in the model.

In Section 6.2 we present the new guided, random-walk based algorithm. We have carried out experiments on some case studies and compared the performance of GRANSPIN, in guided and unguided form, and with the all-children and try-a-child strategies of selecting a successor state. The results are discussed in Section 6.3. In Section 6.4 we consider the metric, the *Process Error Participation* factor, which we have proposed in Chapter 5 for symbolic model checking. We study this metric for explicit-state model checking in this chapter. The summary and plans for the next chapter are given in the last section.

6.2 Guided Random-Walk Based Model Checking

In Figure 6.1, we show our guided random-walk based model-checking algorithm. The algorithm in Figure 6.1 randomly and repeatedly traverses the state space of a given model (with respect to the property) $P \times pnc$, where P is a set of processes, and *pnc* is a never-claim process, until some termination condition, called *terminationCond*, is satisfied. Each traversal is restricted in length by the end-walk condition, *endCond*. Just as in SPIN, in each traversal a move in a process (lines 5-8) is followed by a move in the never-claim process (line 9). The subsequent state of the system is recorded in *path* (line 10). The algorithm will return the goal path if the current state s is accepted in *pnc* (line 11). The algorithm will return **not found** if the termination condition is satisfied.

GRANSPIN algorithm

Input: ***set of processes*** P, ***never-claim process*** pnc,
Boolean $guided$, ***Boolean*** $trychild$

Output: $goal\ path$ | **not found**

```
 1:  while (! terminationCond)
 2:    s = init()
 3:    path = ∅
 4:    while (! endCond)
 5:      if (guided)
 6:        s = guidedRanMove(P, s, trychild)
 7:      else
 8:        s = unguidedRanMove(P, s, trychild)
 9:      s = move(pnc, s)
10:      path = path ∪ {s}
11:      return path  when (accepted(pnc, s))
12:  return not found
```

Figure 6.1: The guided random-walk algorithm GRANSPIN

There are two functions for randomly moving in a process: the function *unguidedRanMove* shown in Figure 6.2 is for the unguided random-walk algorithm, and *guidedRanMove* shown in Figure 6.3 is for the guided version. In the GRANSPIN algorithm, the value of the Boolean parameter *guided* determines which of these functions is called. There is a second Boolean parameter, *trychild*, which determines the selection strategy to be used. If *trychild* is true, the algorithm uses the try-a-child strategy to select a suc-

```
function unguidedRanMove(P, s, trychild)
15:  list = ∅, s'' = s, plist = P
16:  if (trychild)
17:    while ((s'' == s) ∧ (plist ≠ ∅))
18:      p = random(plist)
19:      plist = plist − {p}
20:      tlist = allTrans(p, s)
21:      while ((s'' == s) ∧ (tlist ≠ ∅))
22:        t = random(tlist)
23:        tlist = tlist − {t}
24:        s'' = execute(t, s)
25:  else
26:    for all p ∈ plist, t ∈ allTrans(p, s)
27:      s' = execute(t, s)
28:      list = list ∪ {t} when (s' ≠ s)
29:    t = random(list)
30:    s'' = execute(t, s)
31:  return s''
```

Figure 6.2: The function unguidedRanMove

cessor state, otherwise, it uses the all-children strategy. These two Boolean variables allow GRANSPIN to be executed in 4 variants, as described below:

RANSPIN$_T$: The random-walk algorithm is unguided and the try-a-child strategy is used (the subscript T denotes the strategy). This is equivalent

```
function guidedRanMove(P, s, trychild)
34:  list = ∅, s'' = s
35:  for all p ∈ P, t ∈ allTrans(p, s)
36:    if (trychild)
37:      s' = successor(t, s)
38:    else
39:      s' = execute(t, s)
40:    list = list ∪ {(t, heu(s'))} when (s' ≠ s)
41:  while ((s'' == s) ∧ (list ≠ ∅))
42:    (t, h) = guidedRandom(list)
43:    list = list − {(t, h)}
44:    s'' = execute(t, s)
45:  return s''
```

Figure 6.3: The function guidedRanMove

to the algorithm defined in Chapter 4. A transition is selected randomly from all the transitions in all the processes in P (lines 17-24 in Figure 6.2). A process is selected randomly from P, and from that process, a transition is selected randomly. If this transition is not valid, it is removed and another transition is randomly selected. If eventually no transition is found to be executable, another process is randomly selected, and we try to find an executable transition in that process.

RANSPINA : The random-walk algorithm is again unguided, and the all-children strategy is used (the subscript A denotes the strategy). In the function,

we start by executing *all* possible transitions in all processes from the current state (lines 26-28 in Figure 6.2). This generates *all* 'real' successor states of the current state. We actually do not keep the successor states in memory, but instead remember the corresponding executable transitions in the list. A transition is then selected uniform randomly from that list, and the state is generated by executing that transition again (lines 29-30). Of course, there is a trade-off between memory and time. Storing the set of executable transition costs less memory than storing the set of successor states, but requires a re-execution of the selected transition.

GRANSPINA : A guided random-walk algorithm is used, together with the all-children strategy. We generate *all* successor states of the current state (lines 35-39 in Figure 6.3). Each state is assigned a cost (the heuristic cost) that estimates its distance from the goal state. A successor state is then selected in such a way that states that have smaller costs are chosen more often than states that have larger costs (this biases the random selection). As with the unguided option, we keep the corresponding executable transitions (for successor states) in a list (line 40). The successor state is selected in lines 41-44.

GRANSPINT : A guided random-walk algorithm is used, together with the try-a-child strategy. Instead of executing all possible transitions, we simply put them all in a list. By assuming that all transitions are executable, the potential successor state is generated (line 37 in Figure 6.3). Actually, the destination state of the transition is used without checking whether the transition is executable. That state, together with its heuristic cost, is then put into a list (line 40). A try-a-child proce-

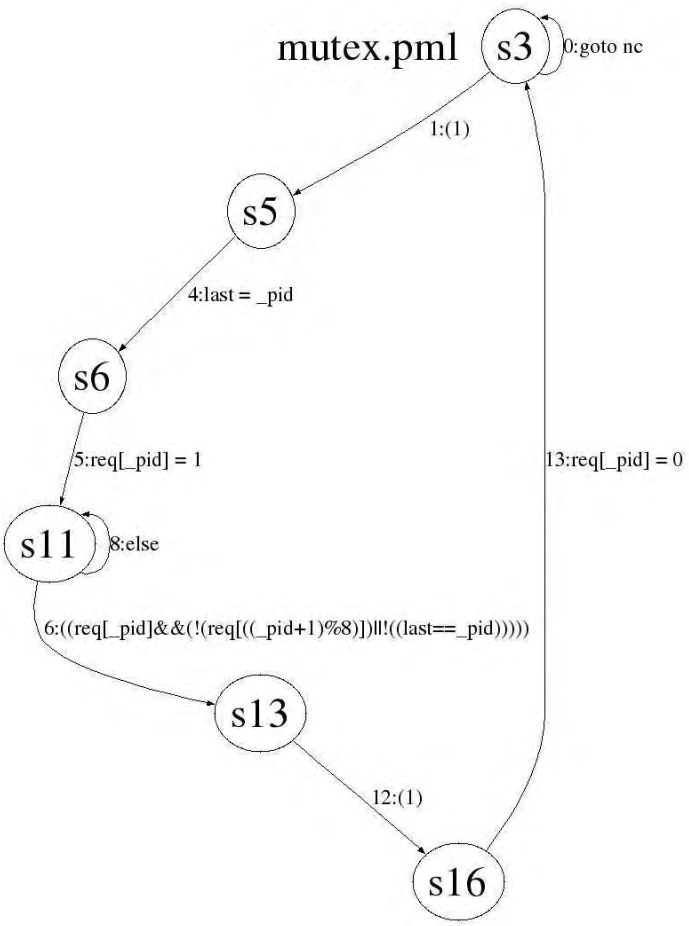
dure is used (lines 41-44) to select a 'real' successor state from amongst them.

Following the algorithm in Chapter 4, the function *random* (lines 18, 22 and 29 in Figure 6.2) uniform randomly selects one element in the input set (of processes or transitions). The function *guidedRandom* (line 42 in Figure 6.3) non-uniform randomly selects one transition from the given set of transition-cost pairs $\{(transition, cost)\}$. Non-uniform selection occurs by repeating each element in the set $(maxDist - cost)$ times, where *maxDist* is the maximum distance to the goal state, and then uniform randomly selecting one from the resulting set. This selection selects transitions that have smaller cost more often than transitions that have larger cost. The function *allTrans* (lines 20 and 26 in Figure 6.2, and line 35 in Figure 6.3) returns the set of transitions of a process starting at the current state. The function *execute* (lines 24 and 30 in Figure 6.2, and line 44 in Figure 6.3) checks whether the given transition can be carried out at the current state, and then makes a new state if necessary. The function *successor* (line 37 in Figure 6.3) simply returns the destination state of the given transition from the current state.

6.2.1 Heuristic function

In this section, we study two abstraction mechanisms of generating the heuristic. In the first mechanism, each process in the model (which consists of multiple concurrent processes) is analysed and a local-distance table is generated. The local-distance table contains a set of 'local' states (of the process) and their distances to the 'local' goal state ([31] also uses this technique). These local distances form the heuristic of the original, concrete model. Whenever we require a heuristic for a concrete state, we simply use the local distance of the corresponding local state (of the single process).

For example, the 'local' transition system presenting 'local' states of a single process and the corresponding local-distance table of the mutual-exclusion protocol are presented in Figure 6.4 (on the left and the right of the figure, respectively). In this case, we assume that the 'local' goal state is $s13$.



local state	distance
3	4
5	3
6	2
11	1
13	0
16	5

Figure 6.4: The 'local' transition system and the corresponding local-distance table of the mutual-exclusion protocol

Another method of abstraction has been used in earlier work by the authors on a symbolic model checker called GOLFER. In that work, "weak" variables are removed from the concrete system to build the abstract system.

Whatever method of abstraction is used, it can be proved that the existence of the goal state in the concrete system implies the existence of an abstract goal state in the abstract system. Note however that the reverse is not true.

In the second abstraction mechanism, we abstract a concurrent model into a smaller model by reducing the number of processes. That abstract model is analysed to generate the heuristic functions. Actually, a breadth-first search algorithm (in SPIN) is used on the abstract model to generate a shortest goal path to an error state. The goal path is then examined and a heuristic table is generated. A goal path and the corresponding heuristic function for the leader-election protocol are presented in Figure 6.5 (on the top and the bottom of the figure, respectively). In the function *heu* (see line 40 of Figure 6.3), the input state, which consists of all state variables defined by the model and their current values, is compared to the abstract states in the predefined pattern-database [73]. If it matches an abstract state in the pattern-database, the corresponding heuristic cost is returned. Otherwise, the maximum heuristic cost is returned. (Later, the input state is assigned the returned heuristic cost.) Note that a heuristic is generated just once, and can used to check the model for any (even large) number of processes. The cost of generating a heuristic is not included in our performance measures. The above technique is done manually, solely for the purposes of the experiment. It could be automated but this is out of scope of this work.

Complexity: The difference in behaviour between the all-children and try-a-child strategies in selecting a successor state has ramifications for the complexity, and hence the performance. Let D be the average length of all random walks, and N the number of random walks before the goal is found or the termination condition is satisfied. The time complexity of GRANSPIN is the product of N and the time complexity for each random walk. The time

```
 1. pi: _pid==0        prev_local_state==4
 2. p1: out!c1,_pid    proc==1, prev_local_state==40, local_state==39
 3. p1: state==1       proc==1, prev_local_state==39, local_state==40
 4. pi: in?c1,claim    proc>1, prev_local_state==40, local_state==36
...
16. p2: resource++     proc==2, prev_local_state==16, local_state==8
...
23. p1: if             proc==1, prev_local_state==36, local_state==33
24. p1: resource++     proc==1, prev_local_state==33, local_state==25

----------------------------
typedef struct abstract_state_struct {
 proc, prev_local_state, local_state;  //control variables
 resource, p, state;          //state variables defined by the model
};

static abstract_state_struct pattern_database[] = {
/* proc, prev_local_state, local_state, resource, p, state */
/* 1 */  {*,4,*,*,*,*},
/* 2 */  {1,40,39,0,2,0},
/* 3 */  {1,39,40,0,2,0},
/* 4 */  {>1,40,36,*,2,1},
...
/* 16 */ {2,16,8,*,*,*},
...
/* 23 */ {1,36,33,*,*,*},
/* 24 */ {1,33,25,*,*,*}
};

int get_state_heu(int proc, int prev_local_state, int local_state,
                  STATE_STRUCT *state) {
 int heu1;
 for (int i=0; i<len(pattern_database); i++) {
  if (compare_abstract_state(pattern_database[i],
              proc, pre_local_state, local_state, state) == 0)
   return ((len(patern_database)-i);         //matched a pattern
 } return MAX_HEU;
}
```

Figure 6.5: A (shortest) goal path and the corresponding heuristic function for the leader-election protocol

complexity of a random walk is D times some factor that accounts for the different selection schemes in the all-children and try-a-child strategies:

try-a-child If we assume that the average number of executable transitions at each step (over all processes in P) is α then the time complexity of try-a-child GRANSPIN is $O(N * D * \alpha)$, where $\alpha \geq 1$.

all-children Here we must execute all transitions in all the processes in P. If we assume that the average number of executable transitions *of a process* is β, then the time complexity of all-children GRANSPIN is $O(N * D * \beta * |P|)$, where $\beta \geq 1$.

Obviously, $1 \leq \alpha \leq \beta * |P|$. Hence we would expect the try-a-child strategy to be faster.[1]

The space complexity of the algorithm is $O(D)$ for all the cases.

6.3 Experimentation

We applied the 4 variants of the GRANSPIN tool to 4 well-known case studies: the dining-philosophers problem, the mutual-exclusion protocol (an extended version of mutual-exclusion for two processes), the leader-election protocol and the Peterson protocol (mutual-exclusion for N processes) [2]. As in previous chapters, only default options are used for SPIN.

6.3.1 GRANSPIN vs. SPIN

In this section, the heuristic is generated using the second abstraction mechanism discussed in the previous section.

In Figure 6.6, we show the results for the **dining-philosophers problem**. The table on the top of this figure shows the number of philosophers **ph.**, the execution time **t** (in milliseconds), the number of trials before the goal path **n**, the length of the goal path **cx**, and the memory usage in

[1] We actually also assume that D and N are the same in all cases.

[2] The specification of this protocol (both error and correct versions) can also be found at `http://anna.fi.muni.cz/benchmark`.

ph.	SPIN			ranSPIN t				ranSPIN a				granSPIN t				granSPIN a			
	t	cx	m	t	n	cx	m	t	n	cx	m	t	n	cx	m	t	n	cx	m
10	0	2298	2.6	0	1.0	336.7	2.4	0	1.0	407.0	2.4	0	1.0	40.0	2.4	0	1.0	40.0	2.4
20	20	6358	2.7	1	1.0	754.5	2.4	1	1.0	710.3	2.4	0	1.0	80.0	2.4	0	1.0	80.0	2.4
30	20	9998	2.9	2	1.0	1105.3	2.5	6	1.0	1132.4	2.5	0	1.0	120.0	2.4	0	1.0	120.7	2.4
40	30	14478	6.1	4	1.0	1608.2	2.5	13	1.0	1661.0	2.5	0	1.0	163.1	2.4	0	1.0	162.4	2.4
50	40	18538	6.5	7	1.0	1997.7	2.5	22	1.0	1842.6	2.5	0	1.0	203.4	2.4	0	1.0	206.0	2.4
60	60	22598	7.0	12	1.0	2519.4	2.6	37	1.0	2343.9	2.6	0	1.0	253.4	2.4	0	1.0	240.2	2.4
70	100	26658	7.6	15	1.0	2745.1	2.6	50	1.0	2725.9	2.6	0	1.0	285.3	2.4	0	1.0	291.5	2.4
80	120	30718	8.2	17	1.0	3196.0	2.7	54	1.0	3052.2	2.7	0	1.0	328.4	2.5	0	1.0	334.6	2.5
90	140	34778	8.9	24	1.0	3596.0	2.8	86	1.0	3674.7	2.8	1	1.0	376.3	2.5	1	1.0	381.6	2.5
100	180	38838	9.6	31	1.0	4240.3	2.9	115	1.0	4222.1	2.9	1	1.0	425.9	2.5	1	1.0	413.0	2.5
110	200	42898	10.6	35	1.0	4409.4	2.9	126	1.0	4214.6	2.9	2	1.0	463.3	2.5	2	1.0	455.4	2.5
120	250	46958	11.6	31	1.0	4806.6	3.0	118	1.0	4600.2	3.0	2	1.0	509.0	2.5	4	1.0	493.7	2.5
130	300	51018	12.7	47	1.0	5216.9	3.1	176	1.0	5097.4	3.1	5	1.0	550.2	2.5	8	1.0	551.7	2.5
140	320	55078	13.7	51	1.0	5552.3	3.2	205	1.0	5487.8	3.2	5	1.0	598.4	2.5	9	1.0	595.8	2.5
150	360	59138	14.9	57	1.0	6125.1	3.4	266	1.0	6263.1	3.4	11	1.0	648.2	2.5	14	1.0	628.8	2.5

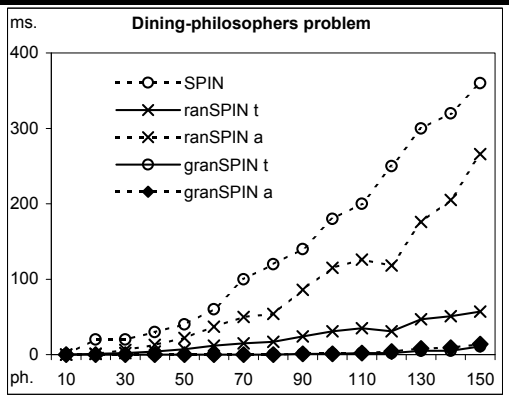


Figure 6.6: Comparing SPIN and (guided/unguided) GRANSPIN for the dining-philosophers problem

megabytes (**m**) for each variant of GRANSPIN. The graph on the bottom of the figure plots the execution times.

The results show (even) unguided RANSPIN is faster and more memory efficient than SPIN. The unguided 'try-and-test' RANSPIN$_T$ is faster than SPIN by a factor around 6, and uses less memory than SPIN by a factor of 4 (when the number of philosophers is large). Fundamentally, the reason for this is SPIN tries to find the error state exhaustively (using back-tracking), whereas (unguided) RANSPIN works randomly, restarting when a path is exhausted. Interestingly, the length of the goal path discovered by unguided RANSPIN is

around 10 times shorter than that of SPIN. There is no obvious explanation of the factor of 10, and we feel it is simply an artifact of the model. The unguided 'all-children' ranSPINa is a little bit slower than unguided 'try-a-child' ranSPINt. It is faster than SPIN just by a factor less than 2. A possible reason is that ranSPINa executes all transitions at each step to generate all 'real' successor states, that really takes time. Obviously, when the lengths of goal path in ranSPINa and ranSPINt are equivalent, they consume the same amount of memory.

As expected, *guided* ranSPIN are all even faster than the unguided versions: ranSPINt is faster than ranSPINt by a factor of 5, and that of ranSPINa to ranSPINa is 18 for the larger numbers of philosophers. As well, the length of the goal path discovered by ranSPIN is around 10 times shorter. The guide hence seems to be very effective in directing the random walk towards the error state. The amount of memory used by ranSPIN is also smaller, and interestingly, the percentage improvement increases as the number of philosophers increases.

In Figure 6.7, we show the results for the **mutual-exclusion protocol**. The columns in the table on the top of this figure are the same of those of the table in Figure 6.6, except **pr.** stands for the number of processes. As in dining-philosophers problem, these results show that both (guided) ranSPIN and (unguided) ranSPIN are faster than SPIN, and return shorter goal paths.

The complexity analysis from the previous section showed that the average number of executable transitions is an important factor in the performance. Since the try-a-child strategy reduces this number we would expect it to have shorter running times. The results in Figure 6.7 confirm this.

The results show that both versions of ranSPIN and of ranSPIN are faster than SPIN, and return shorter goal paths. ranSPIN is particularly fast, re-

pr.	SPIN			ranSPIN t				ranSPIN a				granSPIN t				granSPIN a			
	t	cx	m	t	n	cx	m	t	n	cx	m	t	n	cx	m	t	n	cx	m
8	20	9999	2.8	0	1.0	266.4	2.4	0	1.0	242.2	2.4	0	1.0	34.1	2.4	0	1.0	28.0	2.4
12	50	9999	2.9	0	1.0	444.0	2.4	1	1.0	403.9	2.4	0	1.0	36.2	2.4	0	1.0	35.4	2.4
16	80	9999	3.3	0	1.0	537.4	2.4	2	1.0	448.4	2.4	0	1.0	51.7	2.4	0	1.0	40.2	2.4
20	160	9999	4.0	0	1.0	550.5	2.4	6	1.0	671.7	2.4	0	1.0	77.1	2.4	0	1.0	57.3	2.4
24	360	9999	5.4	1	1.0	771.1	2.5	7	1.0	766.6	2.5	0	1.0	60.1	2.4	0	1.0	47.7	2.4
28	700	9999	7.4	1	1.0	742.5	2.5	8	1.0	693.4	2.5	0	1.0	53.8	2.4	0	1.0	58.9	2.4
32	1040	9999	9.7	1	1.0	1014.5	2.5	20	1.0	1279.2	2.5	0	1.0	65.7	2.4	0	1.0	64.4	2.4
36	2010	9999	15.7	3	1.0	1383.3	2.5	21	1.0	1216.0	2.5	0	1.0	77.7	2.4	0	1.0	88.6	2.4
40	3150	9999	22.0	3	1.0	1369.4	2.5	26	1.0	1276.4	2.5	0	1.0	63.3	2.4	0	1.0	79.1	2.4
44	4800	9999	30.8	3	1.0	1257.2	2.5	29	1.0	1312.3	2.5	0	1.0	103.1	2.4	0	1.0	103.0	2.4
48	8350	9999	49.7	3	1.0	1415.0	2.5	35	1.0	1408.6	2.5	0	1.0	105.2	2.4	0	1.0	123.9	2.4
52	11690	9999	66.4	4	1.0	1613.3	2.6	56	1.0	2016.6	2.6	0	1.0	109.3	2.4	0	1.0	107.4	2.4
56	16870	9999	88.8	4	1.0	1780.2	2.6	42	1.0	1392.4	2.5	0	1.0	99.2	2.4	0	1.0	81.3	2.4
60	22630	9999	116.0	6	1.0	1899.9	2.6	61	1.0	1846.9	2.6	0	1.0	115.8	2.4	0	1.0	129.2	2.4

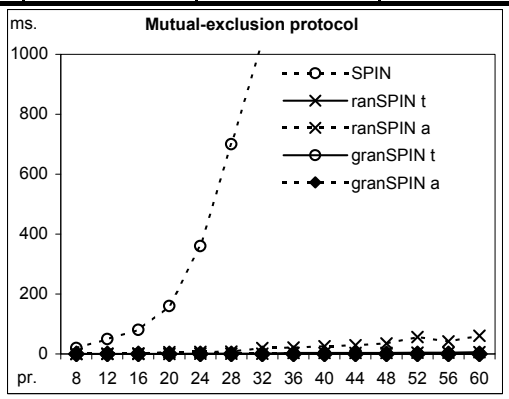


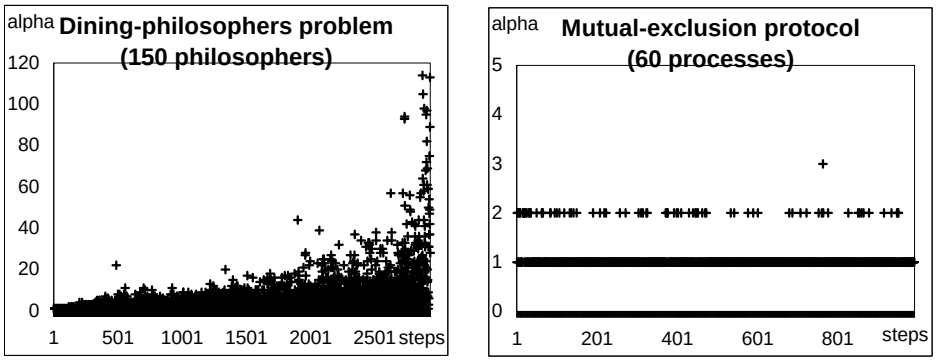
Figure 6.7: Comparing SPIN and (guided/unguided) GRANSPIN for the mutual-exclusion protocol

turning with the error state almost immediately (0 millisecond). The reason for this is that only two processes are involved in the error state (i.e. the two processes that find themselves in the critical section at the same time). It does not matter in what state the other processes are. The heuristic is able to guide the search to those processes and ignore others. In contrast, SPIN performs badly in these circumstances; if the depth-first traversal does not 'happen' to visit those processes, which are involved in an error, then it will waste time and space.

We saw in Section 6.2 that the memory usage for GRANSPIN is the memory

required to store all the states in the execution path. In practice, a state may be repeated many times in a walk, so the actual memory usage may be smaller. The reader can observe in Figure 6.7 that both versions of GRANSPIN return shorter goal paths, and use less memory. Of course, the memory usage for SPIN is different as it stores all the states in its depth-first, backtracking search. In the experiment, we let the maximum depth of the search be 10000 (this is default), so at this depth, the search algorithm is forced to backtrack. In Figure 6.7 we see the lengths of the goal paths are all 9999, but the amount of memory used still increases as the number of processes increases.

As observed in Figure 6.6 and Figure 6.7, the running time of RANSPINA is slower than that of RANSPINT of dining-philosophers problem about 5 times, whereas that of mutual-exclusion protocol is around 10 times. The reason for this behaviour is the different nature of the error in the two models. In the dining-philosophers problem, when we get close to the deadlock state, most of processes cannot move (no transition is executable) and hence α will be large, and close in value to $\beta * |P|$. In effect, the search algorithm 'bogs down' as we get close to the error. This does not happen in the mutual-exclusion protocol, where we only want to check if a pair of processes is in the critical section, regardless of the state of all the other processes. In such a situation, the search algorithm has plenty of executable transitions to choose from, and this corresponds to an α factor near its lowest bound (approximately 1). We illustrate the difference between the two models in Figure 6.8. In this figure, the x-axis corresponds to the steps in the random walk, and the y-axis to the value of α. These random walks were taken for a large number of processes (hence above the crossover point). We see in this figure that α becomes large as the walk gets close to the deadlock state in the dining-philosophers problem. In contrast, in the mutual-exclusion protocol, α remains more or



(a) (b)

Figure 6.8: The values of α for the (a) dining-philosophers problem and (b) mutual-exclusion protocol

less constant irrespective of the number of processes in the system.

The results for the **leader-election protocol** are shown in Figure 6.9. Although SPIN did quite well in this experiment, GRANSPIN is an order of magnitude faster, and produces far shorter goal paths. In this protocol, a token is passed around the ring (of processes), which a process may claim to assume leadership. At the same time, another process may proclaim itself to be the leader by sending its own identifier around the ring and waiting for that identifier to return. Other processes pass the token or the process identifier around. So the length of the goal path for GRANSPIN increases as the number of processes increases. In the experiment, the length of the path is an order of magnitude shorter in the guided case than the unguided case.

The results for the **Peterson protocol** are shown in Figure 6.10. The performance of SPIN is very bad in this experiment (in the figure, *oom* means 'out of memory' by the way). GRANSPIN was faster than RANSPIN by half an order of magnitude, and produced shorter goal paths. As you would expect, the reason for this is that RANSPIN attempts many more random walks before

pr.	SPIN			ranSPIN t				ranSPIN a				granSPIN t				granSPIN a			
	t	cx	m	t	n	cx	m	t	n	cx	m	t	n	cx	m	t	n	cx	m
4	10	3865	2.7	0	1.0	718.6	2.4	0	1.0	585.7	2.4	0	1.0	78.3	2.4	0	1.0	69.0	2.4
6	30	9999	3.0	1	1.0	1116.1	2.5	1	1.0	729.2	2.5	0	1.0	104.4	2.4	0	1.0	81.4	2.4
8	20	9983	3.2	2	1.0	1374.9	2.5	4	1.0	1273.4	2.5	0	1.0	131.5	2.4	0	1.0	97.5	2.4
10	20	9985	3.2	3	1.0	1672.4	2.5	8	1.0	1622.8	2.5	0	1.0	111.8	2.4	0	1.0	124.3	2.4
12	20	9993	3.4	7	1.0	2382.7	2.6	12	1.0	1986.2	2.6	0	1.0	164.9	2.4	0	1.0	138.1	2.4
14	40	9993	3.6	8	1.0	2572.8	2.7	16	1.0	2470.0	2.7	1	1.0	261.8	2.4	0	1.0	188.1	2.4
16	30	9999	3.6	14	1.0	3434.4	2.8	24	1.0	3012.9	2.8	1	1.0	281.0	2.5	2	1.0	328.6	2.5
18	20	9999	3.7	19	1.0	3898.8	2.9	35	1.0	3844.3	2.9	1	1.0	288.6	2.5	1	1.0	316.6	2.5
20	40	9991	3.9	27	1.0	4930.4	3.1	44	1.0	4728.8	3.0	2	1.0	445.2	2.5	3	1.0	268.4	2.5

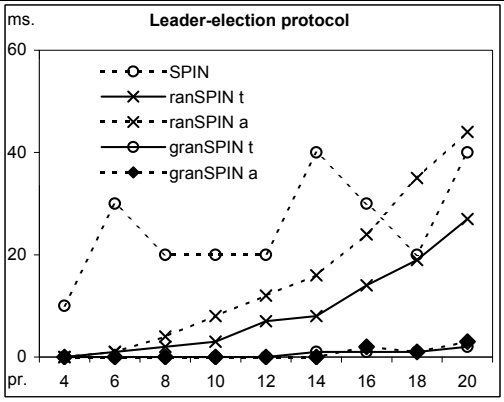


Figure 6.9: Comparing SPIN and (guided/unguided) GRANSPIN for the leader-election protocol

finding the goal path than GRANSPIN. (For example, for 11 processes RANSPIN$_T$ required 8.4 walks while GRANSPIN$_T$ required 1.9.)

6.3.2 What happens if the guide is mis-informed?

While the experiments above compare guided and unguided random walks, they do not tell us how good, or well-informed, the guide is relative to other guides, or indeed, how sensitive the search is to the quality of the guide. To address these issues, we re-ran the previous experiments with a 'bad', so-called *mis-informed* heuristic. We do this by reversing the supposedly 'good' heuristic that we used in the previous experiment (and assume the opposite of 'good' is 'bad'). In practice, whenever the 'good' heuristic produces a

pr.	SPIN			ranSPIN t				ranSPIN a				granSPIN t				granSPIN a			
	t	cx	m	t	n	cx	m	t	n	cx	m	t	n	cx	m	t	n	cx	m
3	0	6519	2.7	6	1.0	2631.1	2.5	10	1.1	2798.5	2.5	1	1.0	1083.4	2.4	1	1.0	1047.5	2.4
5	73020	9993	839.2	42	2.4	3319.1	2.7	72	2.4	3561.0	2.8	17	1.3	2936.8	2.5	15	1.2	2437.5	2.5
7	*oom*			96	4.7	3258.8	3.4	190	5.1	2705.2	3.5	25	1.4	3693.3	2.6	20	1.2	2998.3	2.6
9				189	7.2	4698.8	4.4	337	6.7	4565.9	4.3	45	1.8	4101.8	2.8	51	1.4	4989.4	2.7
11				222	8.4	6997.5	5.3	366	7.0	6920.2	4.9	52	1.9	4898.0	2.9	98	2.2	5873.8	3.1

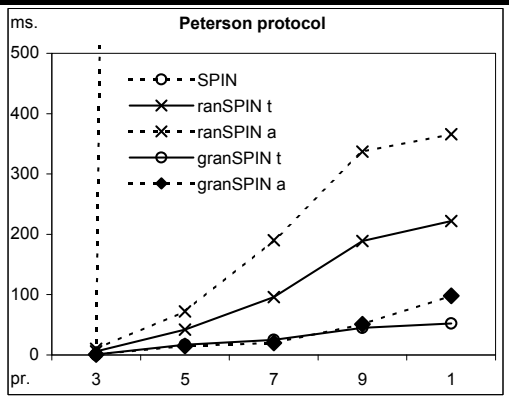


Figure 6.10: Comparing SPIN and (guided/unguided) GRANSPIN for the Peterson protocol

distance, or cost, h for a successor state, we replace that cost by $maxDist - h$. This means the algorithm biases the search away from goals, which should result in longer running times and greater memory usage.

To get even more understanding, we consider both heuristics discussed in Section 6.2.1. The heuristic from the previous section, which we call **h1**, abstracted the multi-process concrete system into an abstract system consisting of a small number of processes. The other abstraction reduced (the concrete system) to a system consisting of just a single process. We named this second heuristic **h0**.

So we now have 4 heuristics: **h0** and **h1**, and their reversals **h0 rev** and **h1 rev**, respectively. Note, importantly, that **h1** should be better informed than **h0** because **h1** is a less-coarse abstraction (i.e. it throws less information away). If a heuristic is well-informed, it can be expected to really guide the

search and lead to very good performance. If we reverse this heuristic, we should see very poor performance. The more dramatic this difference is, the 'better' the result. So, as **h1** is better informed than **h0**, the difference in performance between **h1** and **h1 rev** should be greater than **h0** and **h0 rev**.

The for each of the problem/protocols are presented in Figures 6.11 and 6.12. For better readability, we have labelled the experiments *un-*

ph.	unguided				guided h0				guided h0 rev				guided h1				guided h1 rev			
	t	n	cx	m	t	n	cx	m	t	n	cx	m	t	n	cx	m	t	n	cx	m
10	0	1.0	336.7	2.4	0	1.0	58.1	2.4	12059	356.1	4893.0	30.6	0	1.0	40.0	2.4	10	1.0	2687.2	2.5
20	1	1.0	754.5	2.4	0	1.0	127.4	2.4	*oom*				0	1.0	80.0	2.4	18	1.1	3611.6	2.5
30	2	1.0	1105.3	2.5	0	1.0	191.1	2.4					0	1.0	120.0	2.4	33	1.1	3943.2	2.6
40	4	1.0	1608.2	2.5	0	1.0	252.5	2.4					0	1.0	163.1	2.4	32	1.1	3797.9	2.6
50	7	1.0	1997.7	2.5	0	1.0	322.5	2.4					0	1.0	203.4	2.4	56	1.1	4482.3	2.7
60	12	1.0	2519.4	2.6	0	1.0	390.8	2.4					0	1.0	253.4	2.4	53	1.1	4551.1	2.8
70	15	1.0	2745.1	2.6	2	1.0	438.5	2.5					0	1.0	285.3	2.4	84	1.1	5109.8	2.9
80	17	1.0	3196.0	2.7	1	1.0	488.6	2.5					0	1.0	328.4	2.5	79	1.1	5471.1	3.0
90	24	1.0	3596.0	2.8	3	1.0	560.0	2.5					1	1.0	376.3	2.5	123	1.2	5549.5	3.1
100	31	1.0	4240.3	2.9	8	1.0	629.5	2.5					1	1.0	425.9	2.5	138	1.3	6174.2	3.4
110	35	1.0	4409.4	2.9	10	1.0	712.2	2.5					2	1.0	463.3	2.5	196	1.3	6323.8	3.5
120	31	1.0	4806.6	3.0	11	1.0	767.8	2.5					2	1.0	509.0	2.5	160	1.3	6656.2	3.6
130	47	1.0	5216.9	3.1	15	1.0	800.4	2.5					5	1.0	550.2	2.5	245	1.4	7050.6	3.8
140	51	1.0	5552.3	3.2	17	1.0	885.8	2.6					5	1.0	598.4	2.5	207	1.3	7532.0	3.9
150	57	1.0	6125.1	3.4	20	1.0	951.0	2.6					11	1.0	648.2	2.5	331	1.5	7651.0	4.4

(a) dining-philosophers problem

pr.	unguided				guided h0				guided h0 rev				guided h1				guided h1 rev			
	t	n	cx	m	t	n	cx	m	t	n	cx	m	t	n	cx	m	t	n	cx	m
12	0	1.0	444.0	2.4	0	1.0	120.5	2.4	8	1.0	2168.4	2.5	0	1.0	34.1	2.4	50	2.0	4586.7	2.7
20	0	1.0	550.5	2.4	0	1.0	168.0	2.4	24	1.1	3335.6	2.6	0	1.0	51.7	2.4	135	3.6	4907.0	3.2
28	1	1.0	742.5	2.5	0	1.0	224.2	2.4	51	1.4	3897.3	2.8	0	1.0	60.1	2.4	236	4.8	5787.7	4.1
36	3	1.0	1383.3	2.5	0	1.0	265.9	2.4	69	1.5	3461.7	2.9	0	1.0	65.7	2.4	438	7.3	5202.6	5.9
44	3	1.0	1257.2	2.5	1	1.0	346.1	2.4	115	1.8	4149.5	3.2	0	1.0	63.3	2.4	770	10.8	5176.0	8.9
52	4	1.0	1613.3	2.6	0	1.0	383.6	2.5	118	1.6	4867.5	3.3	0	1.0	105.2	2.4	1102	13.5	5602.6	12.1
60	6	1.0	1899.9	2.6	2	1.0	463.7	2.5	214	2.2	4591.9	4.0	0	1.0	99.2	2.4	1480	16.2	5943.4	16.1

(b) mutual-exclusion protocol

Figure 6.11: GRANSPIN using a mis-informed heuristic for dining-philosophers problem and mutual-exclusion protocol

guided, *guided h0* and *guided h0 rev*, and finally *guided h1* and *guided h1 rev*. The data headings are as before. We see in the case of the dining-philosophers problem in Figure 6.11(a) and the mutual-exclusion protocol

in Figure 6.11(b) that the degradation in the performance when we reverse the heuristic is dramatic. In fact, the problem is 'intractable' for the reversed (hence bad) heuristic **h0 rev** for the dining-philosophers problem and heuristic **h1 rev** for the mutual-exclusion protocol. This effect is far less pronounced for **h1** for the dining-philosophers problem and heuristic **h0** for the mutual-exclusion protocol where there is just an order of magnitude degradation in elapsed time on reversal.

Of course, as expected, the reversed heuristics slow down the algorithms. Surprisingly, different kinds of reversed heuristic made the problems intractable: **h0 rev** for the dining-philosophers problem, and **h1 rev** for the mutual-exclusion protocol. The possible reason for this is how we have constructed the heuristics and the different nature of the errors in the two models (as we discussed in Section 6.3.1). For the dining-philosophers problem, the heuristic **h0** allows each philosopher to run into a situation in which the philosopher is holding a left fork and waiting for the right fork. If all processes are in that situation, the deadlock occurs. The reversed heuristic **h0 rev** avoids that situation, and hence the deadlock rarely occurs. The heuristic **h1** is derived from a trail taken from a small system with only a few philosophers in which processes are running alternatively (in some order). The reversed heuristic **h1 rev** causes the processes to run in a different order to that of **h1**, or just another alternative order. Hence, it is still easy to cause a deadlock.

For the mutual-exclusion protocol, because we would like to check two predefined processes are in the critical section at the same time, those processes are in higher priority (to be executed). Therefore, the better the heuristic the worse the reversed heuristic. That explains the terrible results of the problem in Figure 6.11(b), when **h1** is more informed than **h0**.

The results for the leader-election protocol are shown in Figure 6.12(a), and for the Peterson protocol in Figure 6.12(b). In both these protocols, all

pr.	unguided				guided h0				guided h0 rev				guided h1				guided h1 rev			
	t	n	cx	m	t	n	cx	m	t	n	cx	m	t	n	cx	m	t	n	cx	m
4	0	1.0	718.6	2.4	0	1.0	433.0	2.4	1	1.0	793.4	2.4	0	1.0	78.3	2.4	1	1.0	814.0	2.5
8	2	1.0	1374.9	2.5	2	1.0	1088.3	2.5	3	1.0	1329.6	2.5	0	1.0	131.5	2.4	5	1.0	1515.5	2.5
12	7	1.0	2382.7	2.6	8	1.0	1752.0	2.6	12	1.0	2576.1	2.6	0	1.0	164.9	2.4	13	1.0	2096.1	2.6
16	14	1.0	3434.4	2.8	20	1.0	2918.7	2.7	21	1.0	3310.2	2.8	1	1.0	281.0	2.5	22	1.0	3087.8	2.7
20	27	1.0	4930.4	3.1	38	1.0	4534.1	3.0	42	1.0	4844.1	3.1	2	1.0	445.2	2.5	44	1.0	4855.1	3.0

(a) leader-election protocol

pr.	unguided				guided h0				guided h0 rev				guided h1				guided h1 rev			
	t	n	cx	m	t	n	cx	m	t	n	cx	m	t	n	cx	m	t	n	cx	m
3	6	1.0	2631.1	2.5	5	1.0	2149.1	2.5	11	1.1	3845.6	2.5	1	1.0	1083.4	2.4	0	1.0	389.6	2.4
5	42	2.4	3319.1	2.7	56	2.4	3458.0	2.7	93	3.4	3981.8	2.9	17	1.3	2936.8	2.5	5	1.0	2135.8	2.5
7	96	4.7	3258.8	3.4	117	4.1	3802.5	3.3	185	6.4	3162.1	3.8	25	1.4	3693.3	2.6	20	1.1	5175.2	2.6
9	189	7.2	4698.8	4.4	249	6.6	4277.3	4.2	419	10.8	4293.2	5.4	45	1.8	4101.8	2.8	285	7.5	5266.4	4.5
11	222	8.4	6997.5	5.3	342	8.5	6937.6	5.3	459	11.5	6866.3	6.4	52	1.9	4898.0	2.9	409	10.6	6191.4	6.1

(b) Peterson protocol

Figure 6.12: GRANSPIN using a mis-informed heuristic for the leader-election and Peterson protocols

processes are involved in causing an error. Hence the quality, i.e. informedness, of the heuristic is expected to be less (because the abstractions **h0** and **h1** have thrown away 'vital' information). This is reflected in the results: the difference in performance of the heuristic, and its reverse, is much less pronounced than for mutual exclusion (and the dining-philosophers problem). As well, this effect is even less pronounced for less well-informed **h0** than for **h1**.

Nevertheless, even though the heuristics are reversed or mis-informed, the guided random search is still better than SPIN in execution time, memory usage and length of goal path. This is to be expected, intuitively, because even a guided random search still has a degree of randomness, and this allows the search to 'stumble' on the error, well before SPIN's exhaustive search finds

it. In general terms, we can say that the guided random-walk based search can be effective with even weak levels of heuristic informedness.

6.4 Characterisation

There is a strong suggestion that the number of processes, which are involved in an error state, plays an important role in what heuristic to use, and in how effective a guide the heuristic can be expected to be. We have noted this phenomenon in Chapter 5, where we referred to it as the *Process Error Participation* factor, or PEP factor for short. The PEP factor measures the proportion of all the processes that are involved (i.e. participate) in the error. A low PEP factor means a small number of processes participate: a PEP factor equal to 1 means all processes participate. The mutual-exclusion protocol is an example of a low PEP system because just two (pre-determined) processes that are in the critical section at the same time is needed for an error to occur. The other two protocols studied here are high PEP systems because all the processes must 'work together' to cause an error. The hypothesis is that low PEP systems can be expected to profit more from guidance than high PEP systems. While this aspect has not been directly addressed in chapter, the result of these experiments is consistent with this hypothesis.

Note that the topology of the state space can affect the degree of coverage that can be expected from random traversals [66]. In turn, this of course can affect the effectiveness of using random walks in finding errors. However, the topology does not play a significant role in the computation of the (abstraction-based) heuristic, and this is important in determining the effectiveness of the guided walk. In Pelánek *et al.* [67], systems are classified into different domains based on their topologies, and ignores errors (there are

none). In our work, systems are classified according to the interaction between the error and the processes (PEP), so a given Pelánek 'topology' may correspond to models with different PEP factors, depending on the error.

6.5 Summary

In this chapter we have presented formal model simulation using a guided random-walk based model-checking algorithm. The algorithm is implemented by modifying SPIN. The resulting tool, GRANSPIN, is a (formal) model simulator, unlike SPIN, which is a verifier. GRANSPIN has been demonstrated to be faster and more memory efficient than SPIN in locating errors. Furthermore, GRANSPIN scales well, and is guided.

In a first series of experiments, we have compared SPIN with (unguided) RANSPIN and (guided) GRANSPIN. Studying time, memory and length of witness for 4 different protocols, GRANSPIN performed better than SPIN or RANSPIN. In a second series of experiments, we studied the heuristic behaviour of GRANSPIN. We tried two kinds of heuristic strategies, called 'try-a-child' and 'all-children'. We also tried two 'levels' of abstraction in the heuristic: **h0** which is course-grained, and **h1** which is finer-grained. The aim here is to study the sensitivity of the guided random-walk algorithm to the informedness of the heuristic. The expectation was that a mis-informed heuristic should have worse performance than a heuristic that is well-informed. Intuitively, we expect the speed-up that can be expected from using guided search to be directly related to the informedness of the heuristic.

The mechanism and cost of generating a heuristic has not been addressed in this chapter. Nor have we considered how we can automate the manual techniques that we have used in this chapter. Our earlier work did use a

static-analysis technique to automatically generate the heuristic. However, it is still an open question what kind of analysis of the specification can be expected to generate the best-informed heuristic. Computing a heuristic cost must add to the execution time of a guided search of course. However, generally the saving in the number of states searched in a guided search is much greater than the extra time it takes to compute the heuristic. In conventional search, the degree of informedness of the heuristic can be important, and there is an obvious trade-off: the less informed the heuristic, the less worthwhile it is taking time to compute it. In a random setting, as we have seen, it appears a heuristic guide can only really help the search, hence this trade-off appears not to apply.

In the previous chapter, we presented a hypothesis that low PEP systems (like the mutual-exclusion protocol) can be expected to profit more from guidance than high PEP systems (like the other two protocols studied here). This has been confirmed in this research on explicit-state models. It should be emphasised that PEP is a function of the error (as specified by the user), and it is not simply a characterisation of the topology.

We have extended the symbolic model checker GOLFER *with a random-walk search strategy.* GOLFER *hence now comes in two versions: one based on heuristic search, the other based on random walks, both of which can be guided and unguided. In principle at least, it is possible to compare the behaviour of guided search in an explicit-state and symbolic setting. This is not as easy as it may at first seem however because the vagaries and unpredictability of BDD arithmetic make it hard to separate the symbolic from the underlying heuristic behaviour.*

In the next chapter, we unify the guided and unguided random-walk algorithms, both symbolic and explicit-state, into a single framework, and compare

their performance. We also study the PEP metric proposed in Chapter 5 in more detail, and propose a model classification based on PEP.

This page intentionally left blank.

Chapter 7

Unification and Model Classification

In the previous chapters, guided random-walk algorithms were implemented in symbolic and explicit-state contexts. The experimentation has showed that these algorithms are very good in terms of execution time and memory usage. In this chapter, we formalize those algorithms (symbolic and explicit-state) into a single framework and analyse the complexity. We also classify model-checking systems into categories based on their properties and suggest the best model-checking algorithm in each category. Moreover, the PEP presented in previous chapters has been revised to include a method of determining the factor. This chapter is based on the work [14, 12].

7.1 Introduction

In the previous chapters, we have extended the work of Qian and Nymeyer on symbolic guided model checking by adding the capability of random-walk based search. The algorithm is called symbolic guided/unguided random-

walk model checking. We also applied the random-walk mechanism into another explicit-state model checker SPIN by replacing its depth-first search algorithm by a guided/unguided random search. It is called GRANSPIN.

The contribution of this chapter is: (1) Integrate the symbolic and explicit-state guided random-walk algorithms into a single framework and analyse the complexity; (2) Carry out experiments that compare the explicit-state and symbolic algorithms, both guided and unguided, and using random walks; (3) Investigate a characterisation of models, called the *Process Error Participation* factor that predicts which model checker can be expected to give the best performance.

Our algorithmic contribution is described in Section 7.2, where we formalise the symbolic and explicit-state guided random-walk algorithms into one general algorithm, and analyse the complexity of all algorithms. The experimental work is described in Section 7.4, including the comparison of various random-walk based algorithms, both guided and unguided. Our summary is given in the last section.

7.2 The Algorithms

In Chapter 5, we developed a *symbolic* random-walk algorithm that is based on NUSMV and can be parameterised as abstraction-guided or unguided. The normal, unguided version of the algorithm was named SRMC, and the guided version SGRMC. Similarly, in Chapters 4 and 6, we developed an (unguided) *explicit-state* random-walk algorithm, named RANSPIN, and a guided explicit-state random-walk algorithm called GRANSPIN. The explicit-state algorithms are based on SPIN of course.

In this section we integrate all the above algorithms into a single algo-

rithm, or framework, called GRMC (which stands for Guided Random-walk Model Checking). We also analyse the complexity of the resulting algorithm. In Figure 7.1 we show GRMC. In this algorithm, if the variable *guided* is false,

---

GRMC algorithm

Input: ***transition system*** $M = (S, T, S_0, G)$,

heuFunc *heu*, ***boolean*** *guided*

Output: *goal path* | **not found**

1: **while** (! *terminationCond*)
2: $s = random(S_0)$
3: $visitedStates = \emptyset$
4: **while** (! $endCond(s)$)
5: $visitedStates = visitedStates \cup \{s\}$
6: **return** $getPath(visitedStates)$ **when** $(s \in G)$
7: $s = ranSucc(s, M, heu, guided)$
8: **return not found**

Figure 7.1: The Guided Random-walk Based Model-Checking Algorithm GRMC

the algorithm is a "pure" (i.e. unguided) random-walk algorithm. Otherwise the algorithm is guided. The algorithm randomly and repeatedly traverses the state space of the given transition system until *terminationCond* is satisfied. Each traversal is restricted in length by *endCond*. In each traversal, all visited states are recorded (line 5) in *visitedStates*. The algorithm will return the goal path if a goal is located (lines 6). Otherwise, the algorithm will return **not found** when the termination condition is satisfied (line 8). The

function *getPath* examines the times the states were visited and extracts the goal path. The function *random* chooses a state, uniformly and randomly, from a set of states (lines 2). The function *ranSucc* receives the current state s and randomly selects a successor state of s.

In the guided algorithm (variable *guided* is true), the function *ranSucc* generates all successor states first and assigns each successor state a heuristic cost (to reach the goal) by calling the input function *heu* of type ***heuFunc***. A successor state is then selected in a biased fashion. The smaller the heuristic cost of a state, the greater the probability that state will be selected. To achieve this, suppose that we have a set of states with heuristic costs $\{(s_0, h_0), \ldots, (s_n, h_n)\}$, and suppose that the maximum value of all the heuristic costs is $MAXHEU$. Each instance s_k is repeated $MAXHEU - h_k$ times, and a state is then selected randomly and uniformly. If the variable *guided* is false, the function *ranSucc* simply selects one state randomly and uniformly from the successor states of s.

Complexity: Let D be the average length of all random walks, and N the number of random walks before the goal is found or the termination condition is satisfied. The time complexity of GRMC is the product N and the time complexity for each random walk. The time complexity of a random walk is D times the time complexity of each step. Further assume that the average time of 1) generating the successor states at each step is η, 2) assigning heuristic values to successor states is γ, and 3) randomly selecting a successor state is μ. The guided algorithm costs $F_1 = F(N * D * (\eta + \gamma + \mu))$ and the unguided algorithm costs $F_2 = F(N * D * (\eta + \mu))$, whereas F is the cost function. In the worst case, when F_1 and F_2 are all $O(N * D)$, one could conclude that the guided algorithm and the unguided algorithm are equivalent in cost. However, in practice, $F_1 = F(N * D * (\eta + \gamma + \mu)) > F(N * D * (\eta + \mu)) = F_2$.

The space complexity of the algorithm is $O(D)$.

7.2.1 Symbolic representation

For comparison purposes, in this section we revisit part of our work from Chapter 5. Instead of using a heuristic function *heu*, SGRMC (or more precisely the symbolic function *ranSucc*) uses an abstract database *adb* to calculate the heuristic costs. The function *ranSucc* is presented in Figure 7.2. In this function, all successor states *ss* from the current state s are generated

---

function ranSucc(s, M, *adb*, *guided*)

10: $ss = getImage(s, M)$

11: **if** (*guided*)

12: ***costSet*** $p = \emptyset$

13: **for all** $b_i \in adb$

14: $ss'' = b_i \cap ss$

15: $p = p \ \cup \{(ss'', i)\}$ **unless** $(ss'' == \emptyset)$

16: $ss = guidedRandom(p, |adb|)$

17: $s = random(ss)$

18: **return** s

Figure 7.2: The symbolic random successor function in the SGRMC algorithm

firstly by calling the function *getImage* in line 10. the function *getImage* returns the successor states of s, namely $Img(s) = \{s' \mid (s, s') \in T\}$. Secondly, being symbolic, instead of assigning heuristic costs to individual successor states, we assign costs to sets of successor states that are equi-distant from

the goal state (according to the used abstraction). The abstract database *adb* contains this information. The (guided) algorithm divides the set of successor states *ss* into partitions [73]. As the set of states *ss* and elements b_i of *adb* are represented as BDDs, this partitioning is just the repeated application of the AND operator between *ss* and each b_i. Each partition is assigned a cost i (lines 13-15), where i is the abstract distance to the goal of (all) the states in that partition. All partitions and their corresponding costs are stored in the variable p of type ***costSet***. The ***costSet*** p is then sent to the biasing random selection function *guidedRandom* (line 16) to select a set of states, and the set of successor states is then restricted to that partition only. Finally a successor state is selected uniform randomly from this restricted set (line 17). The size of *adb* is the maximum heuristic cost $MAXHEU$.

Complexity: In this algorithm, the time to assign heuristic costs to the set of successor states is $F(\gamma' * |adb|) = O(|adb|)$, where γ' is the time required to extract each partition from *ss*. Because sets of states are represented as BDDs, the domain of the variables is $\{0, 1, \mathtt{x}\}$, where `x` can be either 0 or 1. (See Chapter 3 for more detail). The function *random* assigns a random value from $\{0, 1\}$ to `x`. The run-time of *random* needs to be considered when computing the complexity. The time complexity of the guided algorithm is hence $F_{1s} = F(N * D * (\eta + \gamma' * |adb| + \mu))$. The time complexity of the unguided algorithm is unchanged, $F_{2s} = F(N * D * (\eta + \mu))$.

7.2.2 Explicit-state representation

The guided random-walk algorithm was has been reported in Chapters 4 and 6. The algorithm was implemented by modifying SPIN and replacing its depth-first search algorithm by a guided random-walk algorithm. In this

section, we extract the function *ranSucc* from the algorithm and discuss its complexity.

The function *ranSucc* is shown in Figure 7.3. In this function, P is a set of processes and *pnc* a never-claim process that represents an LTL property. To move from the current state to a new state, the algorithm needs to make a move in some process p in P by executing one transition t of p (line 19 and 27), and thereafter, a move in the never-claim process (line 28). To reduce the run-time, *ranSucc* implements a so-called try-and-test strategy. In guided mode (line 12-18), all possible successor states from the current state are generated, and each corresponding transition is assigned a heuristic cost and stored in a variable *list* of type ***costSet***. A transition is then selected biased randomly from *list*, and tested whether it is valid. In unguided mode (lines 20-27), a process is selected randomly from P (line 21), and from that process, a transition is selected randomly (line 25). If this transition is not valid (non-executable), it is 'removed' and another transition is randomly selected. If eventually no transition is found to be executable, another process is randomly selected, and we try to find an executable transition in that process.

Complexity: Note that the function in line 13 selects the successor state of a transition without checking the guard condition of that transition. We assume that the time required to generate all 'potential' successor states (both valid and invalid) is η_1, and the try-and-test strategy at each step (over all processes in P) is η_2. It is easy to see that $(\eta_1 + \eta_2) \leq \eta$. If lists are implemented as arrays, the *random* functions will take constant time to select an element, hence the run-time can be ignored. The time complexity of the guided algorithm is therefore $F_{1e} = F(N * D * (\eta_1 + \eta_2 + \gamma))$, and of the unguided algorithm $F_{2e} = F(N * D * \eta_2)$.

function ranSucc(s, P, pnc, heu, $guided$)

10: ***costSet*** $list = \emptyset, s'' = s, plist = P$

11: **if** $(guided)$

12: **for all** $p \in plist,\ t \in allTrans(p, s)$

13: $s' = successor(t, s)$

14: $list = list \cup \{(t, heu(s'))\}$ **when** $(s' \neq s)$

15: **while** $((s'' == s) \wedge (list \neq \emptyset))$

16: $(t, h) = guidedRandom(list, MAXHEU)$

17: $list = list - \{(t, h)\}$

18: $s'' = execute(t, s)$

19: **else**

20: **while** $((s'' == s) \wedge (plist \neq \emptyset))$

21: $p = random(plist)$

22: $plist = plist - \{p\}$

23: $tlist = allTrans(p, s)$

24: **while** $((s'' == s) \wedge (tlist \neq \emptyset))$

25: $t = random(tlist)$

26: $tlist = tlist - \{t\}$

27: $s'' = execute(t, s)$

28: $s = move(s'', pnc)$

29: **return** s

Figure 7.3: The explicit-state random successor function in the GRANSPIN algorithm

7.2.3 Discussion of the complexity

As mentioned earlier, even though, in the worst case, F_1 and F_2 are both $O(N * D)$, in practice they will likely be different. The same comment can be made about F_{1s}, F_{2s}, F_{1e} and F_{2e}. Noting that $F_2 < F_1$ for the GRMC algorithm in Figure 7.1, $F_{2s} < F_{1s}$ for the SGRMC algorithm in Figure 7.2, and $F_{2e} < F_{1e}$ for the GRANSPIN algorithm in Figure 7.3, we conclude that the unguided random-walk algorithms, both symbolic and explicit-state, are faster than the guided random-walk algorithms. Because the guided algorithms use knowledge to bias the search towards the goal states, in practice we expect that the number of trials N (before reaching the goal path) will be smaller and the average length of the random walks D will be shorter than in the case of the unguided algorithms. We therefore expect the run-times of the guided algorithms to be smaller. An examination of earlier experimental results in [11, 9, 10] in fact confirms this.

A second observation is that the explicit-state unguided algorithm is generally faster than the symbolic unguided algorithm: $F_{2e} = F(N * D * \eta_2)$ is smaller than $F_{2s} = F(N * D * (\eta + \mu))$. This is trivial when $(\eta_1 + \eta_2) \leq \eta$. But it is more difficult to compare the performance of the explicit-state guided algorithm $F_{1e} = F(N * D * (\eta_1 + \eta_2 + \gamma))$ and the symbolic guided algorithm $F_{1s} = F(N * D * (\eta + \gamma' * |adb| + \mu))$. This will be the subject of our experiments in the next section[1].

[1]Note that if we further assume that the times to generate all successor states and assign heuristic costs in the two guided algorithms are the same, then $F_{1e} < F_{1s}$.

7.3 PEP determination

The abstraction used as a guide in the guided algorithms is based on a variable-dependency analysis. We gain more insight into the nature of the high- and low-PEP characterisation by examining the output of this analysis for the 2 versions of the Peterson protocol[2], shown in Figure 7.4. We see

Peterson	Peterson-Low
---- *tree layer 0*	---- *tree layer 0*
resource	resource
---- *tree layer 1*	---- *tree layer 1*
p1.state	_process_selector_
p1.k	p1.state
p2.state	p1.k
p2.k	p2.state
p3.state	p2.k
p3.k	---- *tree layer 2*
p4.state	p1.state2
p4.k	p1.j
p5.state	p2.state2
p5.k	p2.j
_process_selector_	

Figure 7.4: Partial results of the variable dependency analysis

in this figure the variable dependencies for original (high-PEP) system on the left, and the low-PEP system on the right, for a system comprising 5 processes. On the left we see that the state variables for all 5 processes are dependent on the variable *resource*. They are hence equally important, and as a result, it is difficult to define an effective (or *informed*) heuristic. On the right we see that the state variables of just 2 of the processes (*p1* and *p2*) are dependent on *resource*. Variables *p3*, *p4* and *p5* clearly are not involved (we refer to them as 'weak variables') and hence can be safely removed in an abstraction of the protocol.

[2]See section 7.4 for more detail of the protocol.

7.4 Experimentation

In this section, we carry out two series of experiments: 1) we compare the symbolic (un)guided random-walk algorithms with guided search SGMC and with NUSMV, and 2) we compare the explicit-state (un)guided random-walk algorithms with SPIN. We refer to the algorithms as follows: symbolic unguided random-walk - SRMC, symbolic guided random-walk - SGRMC, explicit-state unguided random-walk - RANSPIN, explicit-state guided random-walk - GRANSPIN, and rename AGMC in [73] as SGMC. We apply the algorithms SRMC, SGRMC, RANSPIN and GRANSPIN to two well-known problems: the leader-election protocol and the Peterson protocol (for N-processes sharing a critical section). Precisely the same heuristic mechanism (i.e. an abstract database) is used in both symbolic and explicit-state algorithms SGMC, SGRMC and GRANSPIN. As well, we apply the explicit-state algorithms to the General Inter-ORB Protocol (*GIOP*)[3]. *GIOP* is a real-world case-study. Note again, only default options are used for SPIN and NUSMV.

In Chapter 5, we defined the *Process Error Participation* (PEP) factor of a model. This characterises the number of concurrent processes in that model that are involved in an error state as a proportion of all processes. If few of the processes in a model are involved in causing an error, then the model has a low PEP factor. Conversely, if most or all of the processes in the model must 'work together' to cause the error, the PEP factor is high. A classic example of a high-PEP model is the dining philosopher's problem and deadlocking philosophers. Note the PEP factor is a function of the kind

[3]The specification of this protocol can be found at `http://tele.informatik.uni-freiburg.de/leue/sources/giop`. The most recent version of the protocol can be found at `http://www.inf.uni-konstanz.de/soft/research/projects/past_projects/fm_oo/fm_oo.shtml`.

of error, and you could say it measures the relative degree of concurrency of the error state.

7.4.1 Symbolic algorithms

Leader-election protocol

In Figure 7.5, we show the experimental results for all the symbolic random-

pr.	NuSMV			SGMC			SRMC			SGRMC		
	t	cx	m	t	cx	m	t	cx	m	t	cx	m
3	0.7	18	10.0	0.5	18	6.9	0.8	936.8	5.7	0.3	333.9	4.4
4	20.4	19	22.9	12.6	19	11.6	2.9	1566.7	9.5	2.3	1206.1	10.2
5	237.3	21	145.3	164.9	21	33.6	16.1	5262.1	11.3	10.1	3025.8	11.4
6	2108.6	23	645.1	755.2	23	121.4	89.3	16664.5	15.2	33.3	6126.3	13.3

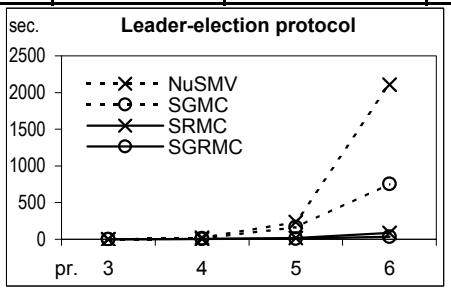


Figure 7.5: Performance of symbolic algorithms on the leader-election protocol

walk algorithms, and for SGMC and NUSMV, for the leader-election protocol. Some of this data has also been presented in Chapter 5 and is shown here for comparison purposes. In the table on the top of this figure, columns **pr.**, **t**, **cx** and **m** denote the number of concurrency processes in the system, the execution time in seconds, the length of the goal path, and the memory usage in Megabytes, respectively. The chart on the bottom plots just the execution times. Revealing is that neither the guided nor the unguided random-walk algorithms (SGMC and SGRMC) are able to get even close to the shortest goal

path that is generated by both NUSMV and SGMC, but in spite of this, both random-walk algorithms are an order of magnitude faster in execution time. As expected, the guided random-walk algorithm is faster than the unguided version, and returns a shorter goal path. The above behaviour has also been seen in other experiments reported in [11].

Peterson protocol

The results for the same algorithms for the Peterson protocol are shown in Figure 7.6. It again shows that both the random-walk algorithms are

pr.	NuSMV			SGMC			SRMC			SGRMC		
	t	cx	m	t	cx	m	t	cx	m	t	cx	m
3	0.1	16	1.7	0.1	16	1.8	0.0	45.1	1.3	0.0	32.9	1.4
4	25.0	40	12.1	167.8	40	26.8	1.3	698.4	8.4	0.7	386.3	8.0
5	1359.6	60	191.0	20577.2	60	587.1	13.6	1609.3	18.3	7.6	833.5	19.0

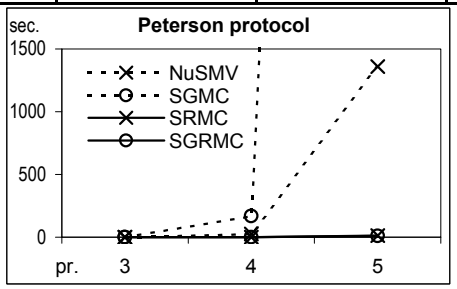


Figure 7.6: Performance of symbolic algorithms on the Peterson protocol

faster than NUSMV and SGMC, and that guided random search is the fastest. Noteworthy here though is the very poor execution time of guided search SGMC (and noting that this algorithm uses precisely the same abstraction heuristic as guided random search, which we see above is very fast). The Peterson protocol we study here is for a system in which processes share a single resource. The goal (error) state in this protocol occurs when more than one process gains access to this resource at the same time. All processes are

able to cause the protocol to move into an error state. With this high level of symmetry, we characterise the Peterson protocol as a high-PEP system.

We can modify the Peterson protocol, and make it into a low-PEP system, by stipulating that just (2) pre-determined processes can cause the protocol to move into an error state (all other processes are considered 'immune'). Intuitively, if the behaviour of only two processes can cause an error state, it is quite easy for the heuristic guide to bias the search (the heuristic needs to check the status of just these 2 processes). We expect then that the guided algorithm SGMC for this low-PEP system to be particularly effective (i.e. fast). Importantly, however, the PEP factor does not influence the performance of guided random-walk based algorithms in a significant way, for 2 reasons. Firstly, random-walk algorithms are still predominately random searches, so the guide does not play the same (important) role it plays in conventional A* search. Secondly, when you reduce the PEP factor, you also reduce the number of error states. This is not a problem for conventional search, but it is certainly a problem for random-walk based search as the greater the density of error states, the greater the chance of success when a search is random. We see these behaviours in the experimental results shown in Figure 7.7 for the low-PEP Peterson protocol.

7.4.2 Explicit-state algorithms

Leader-election protocol

In Figure 7.8, we show the experimental results for SPIN and for the random-walk based algorithms RANSPIN and GRANSPIN for the leader-election protocol.[4]

[4]We note here that Edelkamp has done experiments on guided search in SPIN using a tool he developed called HSF-SPIN [31]. However, Edelkamp's work focuses mainly on the number of states stored or expanded in the search, while our work focuses on the total

pr.	NuSMV			SGMC			SRMC			SGRMC		
	t	cx	m	t	cx	m	t	cx	m	t	cx	m
3	0.1	16	1.7	0.1	16	1.8	0.0	67.4	1.5	0.0	33.8	1.4
4	25.3	40	12.5	1.1	40	10.6	4.7	2127.7	10.5	1.9	743.9	9.0
5	1364.5	60	206.0	10.6	60	17.6	98.5	10877.5	31.1	24.9	2565.1	25.1

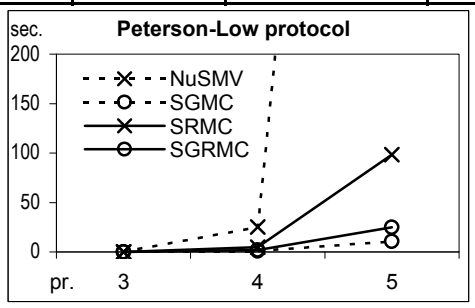


Figure 7.7: Performance of symbolic algorithms on the low-PEP Peterson protocol

pr.	SPIN			ranSPIN			granSPIN		
	t	cx	m	t	cx	m	t	cx	m
4	0.010	3865	2.7	0.000	718.6	2.4	0.000	78.3	2.4
8	0.020	9983	3.2	0.002	1374.9	2.5	0.000	131.5	2.4
12	0.020	9993	3.4	0.007	2382.7	2.6	0.000	164.9	2.4
16	0.030	9999	3.6	0.014	3434.4	2.8	0.001	281.0	2.5
20	0.040	9991	3.9	0.027	4930.4	3.1	0.002	445.2	2.5

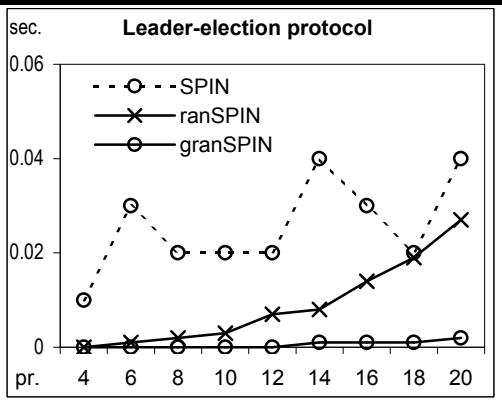


Figure 7.8: Performance of explicit-state algorithms on the leader-election protocol

execution time. Virtually no data that we could use was provided in [31].

As expected, both random-walk algorithms are faster and return shorter goal paths than SPIN. The performance of GRANSPIN is particularly impressive. We note that SPIN's underlying depth-first search of the state space is unstable, which means that depending on where the error state is possibly located, the search may be 'lucky' and find it quickly, or 'unlucky' and never find it (the available memory or time is exhausted). So, if we were to define another error state (in the right place), the results for SPIN may be very different. (We have done this as an exercise and confirmed the fact.) In contrast, random-walk based search algorithms are stable. In essence, the performance of these algorithms is independent of the location of the error state(s).

Peterson protocol

The experimental results of the explicit-state algorithms for the Peterson protocol are shown in Figure 7.9. SPIN performed poorly, showing its instability and its worst-case behaviour. The random-walk algorithms performed very well, with the guided algorithm GRANSPIN performing best of all. As in the case of our previous experiment with the symbolic algorithms, the PEP factor of the system makes little difference to the performance of the random-walk based algorithms[5].

General Inter-ORB Protocol (GIOP)

Kamel and Leue in [51] formalized and analysed the General Inter-ORB Protocol (GIOP). This protocol is a key component of the Common Object Request Broker Architecture (CORBA). The model used in that work violates the property such that states that multiple pending requests are not

[5]Note we do not have an implementation of SPIN that is based on guided heuristic search, which is where we would expect the PEP factor to play an important role.

pr.	SPIN			ranSPIN			granSPIN		
	t	cx	m	t	cx	m	t	cx	m
3	0.000	6519	2.7	0.005	2631.1	2.5	0.001	1083.4	2.4
4	0.410	9953	12.9	0.021	3654.9	2.6	0.007	2229.2	2.5
5	67.270	9993	839.2	0.036	3319.1	2.7	0.014	2936.8	2.5
6			*oom*	0.099	3051.4	3.2	0.022	3130.0	2.6
7				0.095	3258.8	3.4	0.024	3693.3	2.6
8				0.174	3752.5	4.2	0.032	4316.2	2.7
9				0.165	4698.8	4.4	0.042	4101.8	2.8
10				0.181	5611.7	4.6	0.056	5105.2	2.9
11				0.209	6997.5	5.3	0.049	4898.0	2.9

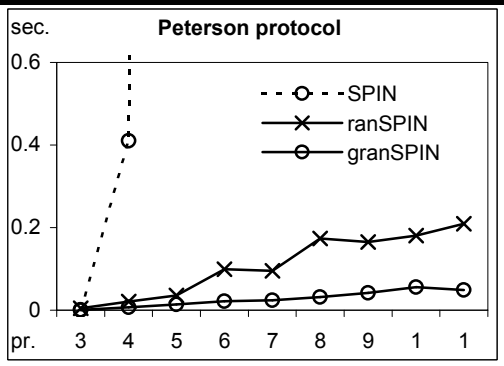


Figure 7.9: Performance of explicit-state algorithms on the Peterson protocol

allowed. The standard protocol as they called *GIOP3* contains 10 processes, including 2 *User* components. In this experiment, to study the performance of the random-walk algorithms in a real case-study, we repeat that validation and named it *GIOP*. We also followed the idea of making the larger system as they called *GIOP4* by increasing the size of the standard model *GIOP3*.

We analysed GIOP using SPIN (as they did) and using (un)guided RANSPIN. In Figure 7.10, we show the results. We see here that SPIN performs poorly (quickly running out of memory), RANSPIN performs very well, and our guided algorithm GRANSPIN has extraordinary performance. In the chart at the bottom of Figure 7.10, we highlight the performance of RANSPIN and GRANSPIN by reducing the y-axis scale to a maximum 0.02 second. In this chart, SPIN is not shown because its performance is off scale.

pr.	SPIN			ranSPIN			granSPIN		
	t	cx	m	t	cx	m	t	cx	m
10	1.450	260	65.1	0.003	293.7	2.6	0.001	181.9	2.5
11	12.010	343	501.5	0.004	373.4	2.7	0.000	183.7	2.5
12			*oom*	0.004	507.9	2.7	0.000	195.9	2.5
13				0.007	586.2	2.7	0.000	191.8	2.5
14				0.008	694.5	2.8	0.000	194.8	2.5
15				0.005	803.5	2.8	0.000	198.2	2.5
16				0.008	903.4	2.9	0.001	229.9	2.5
17				0.017	982.0	2.9	0.001	216.0	2.5
18				0.020	1073.8	2.9	0.005	257.1	2.6
19				0.008	227.9	2.9	0.002	206.7	2.5
20				0.009	279.4	2.9	0.001	188.5	2.5

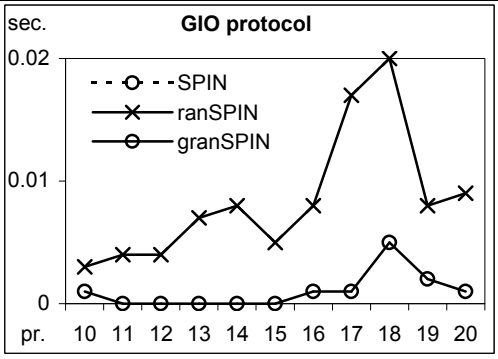


Figure 7.10: Performance of explicit-state algorithms on the General Inter-ORB protocol

7.4.3 Discussion - Model Classification

Comparing the execution time of explicit-state and symbolic random-walk algorithms in Figures 7.5, 7.6, 7.8 and 7.9, it is easy to see that the explicit-state algorithms are faster than the symbolic ones. One reason for this is the fact that the symbolic approach computes all successors of a given state, every time. This can be advantageous under certain conditions, but generally is not advantageous in a random walk, which only requires a single successor state during each traversal. In essence, the random walks are handicapped by the BDD baggage. The conclusion is that explicit-state models can most

profit from random-walk based search. An extra benefit of random-walk search is that it is insensitive to the location of errors (i.e. it is stable). In all our experiments, the guided random-walk algorithm was faster and produced shorter goal paths than the corresponding unguided version.

In the case of symbolic models, because of the presence of the guided model checker (SGMC), we need to consider the PEP factor in determining which search algorithm is likely to be best. Models that have a low-PEP factor can best be analysed by SGMC (because the heuristic can be well-informed), but models with a high-PEP factor should be analysed by SGRMC (because it is difficult to find a well-informed heuristic). The algorithm SGRMC is of course the symbolic equivalent of GRANSPIN.

In Figure 7.11, we depict graphically what we have said above. Given a

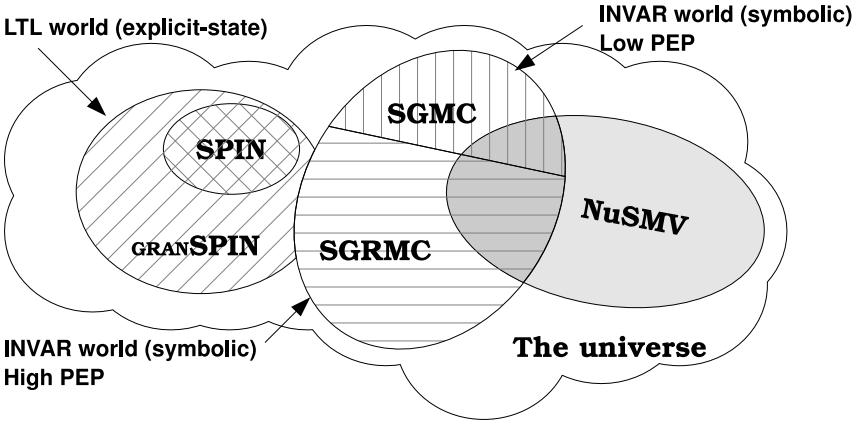


Figure 7.11: The classification of all checking models in term of performance

model, this depiction shows which algorithm you should use. We consider mainly the execution time as defining the performance in this figure, but usually the fastest algorithm also gives the shortest error path, and uses least memory. We should mention here that this depiction is incomplete in that we do not have a version of SPIN that is based on guided search. Like

SGMC, this tool is likely to benefit low-PEP models.

The SPIN tool uses LTL temporal logic of course, and the full power of this temporal logic is available to GRANSPIN as well. In this LTL world, GRANSPIN generally outperforms SPIN. This means that GRANSPIN can detect errors in more models than SPIN as well. In the symbolic world we are limited to invariant-property checking, unfortunately. Here we differentiate between guided conventional search and guided random-walk based search. This differentiation takes into account the PEP factor. Outside of the invariant world, NUSMV can of course handle general CTL properties.

7.5 Summary

In this work we have integrated and formalised the guided/unguided random-walk algorithms (partially) developed in earlier chapters. The unified explicit-state and symbolic framework is called GRMC. We have analysed the complexity of these approaches. This analysis suggests that the explicit-state random-walk search algorithms should generally be faster than the equivalent symbolic algorithms. Not surprisingly we have found that random-walk based algorithms have better performance when the state space is represented explicitly than symbolically. Computing the complexity of guided algorithms is of course very difficult. The reduction in the size of the state space that is searched when a well-informed heuristic (or abstraction) is used can be massive. If the heuristic is ill-informed, the guide may not help at all. For this reason a substantial amount of experimentation is required in this kind of research. Our experiments do strongly suggest that guidance is highly beneficial.

We also investigated more deeply the role of the PEP factor of models.

This very simple metric is surprisingly effective in predicting the performance improvement in conventional guided search. It is expected that a PEP analysis of a model could be used in practice to help derive a well-informed abstraction. Interestingly, the PEP factor does not appear to be important in guided random-walk based search. This is possibly because other aspects of the model need to be considered in a random-walk setting (the current definition is too simplistic). The static variable-dependency analysis that we use to generate a heuristic also warrants further investigation because there are other ways of producing abstract models that could result in even better heuristics.

Random-walk based search, whether guided or unguided, is not verification of course. At best, model simulation just increases the user's confidence that the system is correct. However, the experimental results show how well simulation scales, and how effective it is in locating bugs. Guided random walks in both settings are generally fast and frugal, and provide reasonable (but non-optimal) error paths. The ultimate aim of this research is to build a unified tool that allows the system engineer to (1) build explicit-state and symbolic-state specifications, (2) use an abstract model generator that will allow heuristics of known informedness to be generated, (3) debug using a random-walk based model simulator, and (4) verify using a conventional model checker.

We have presented a big-picture classification showing how the tools compare. The aim is to take a step in the direction of a unified tool that incorporates explicit-state and symbolic representations, an abstract model generator that will allow heuristics of known informedness to be generated, and interacting random-walk and conventional search.

In the next chapter, tools that implement our algorithms are described.

We present the architecture of the tools and user guides, and give examples how to use the tools.

Chapter 8

Tool Implementations

8.1 Overview

There are two tools:

GRANSPIN, which is an explicit-state model checker derived from SPIN [47], and

GRANGOLFER, which is a symbolic model checker based on GOLFER [73, 75, 76, 74], which in turn is based on NUSMV [18].

These tools implement the heuristic guided random-walk model-checking algorithms presented in Chapters 4-7. In particular, GRANSPIN implements all the algorithms presented in Chapters 4 and 6, and GRANGOLFER implements the algorithm called SGRMC in Chapter 5.

Command-line options allow users to choose between the new and the original algorithms. In GRANGOLFER, the random-walk algorithm has been implemented. This algorithm uses the same abstraction heuristic mechanism already in place in this tool (i.e. in GOLFER), making it straightforward for users to swap between guided conventional model checking and guided

random-walk based model checking. granSPIN required more work as there is no guidance mechanism in this tool, so both a heuristic generator and a random-walk based search needed to be built. The architecture and usage of granSPIN is presented in Section 8.2, and of granGOLFER in Section 8.3. A summary is given at the end.

8.2 granSPIN

8.2.1 The architecture of granSPIN

In Figure 8.1, we depict graphically the architecture of granSPIN. (See Figure 1 in [47] for the architecture of SPIN.) In the figure, modules/files that are new are shaded, and those that are original (i.e. belong to SPIN or to other programs such as `gcc`) are unshaded. The lightly shaded umbrella module named "Executable On-the-fly Verifier" is original although it does include newly developed modules. As in SPIN, there are 3 stages in checking a model:

1. generating the verifier (in the C programming language) from the input model
2. compiling the verifier (using a C compiler such as `gcc`)
3. running the executable verifier (program)

In generating the verifier, the input model (expressed in PROMELA), together with an LTL property, is sent to the "PROMELA Parser" module. The LTL property is translated into a PROMELA *never-claim* process by interacting with the "LTL Parser & Translator" module. The parsed model is then sent to the "Verifier Generator" module to generate the "Verifier files" (in the C language). Of course, the relevant algorithms in SPIN are embedded into

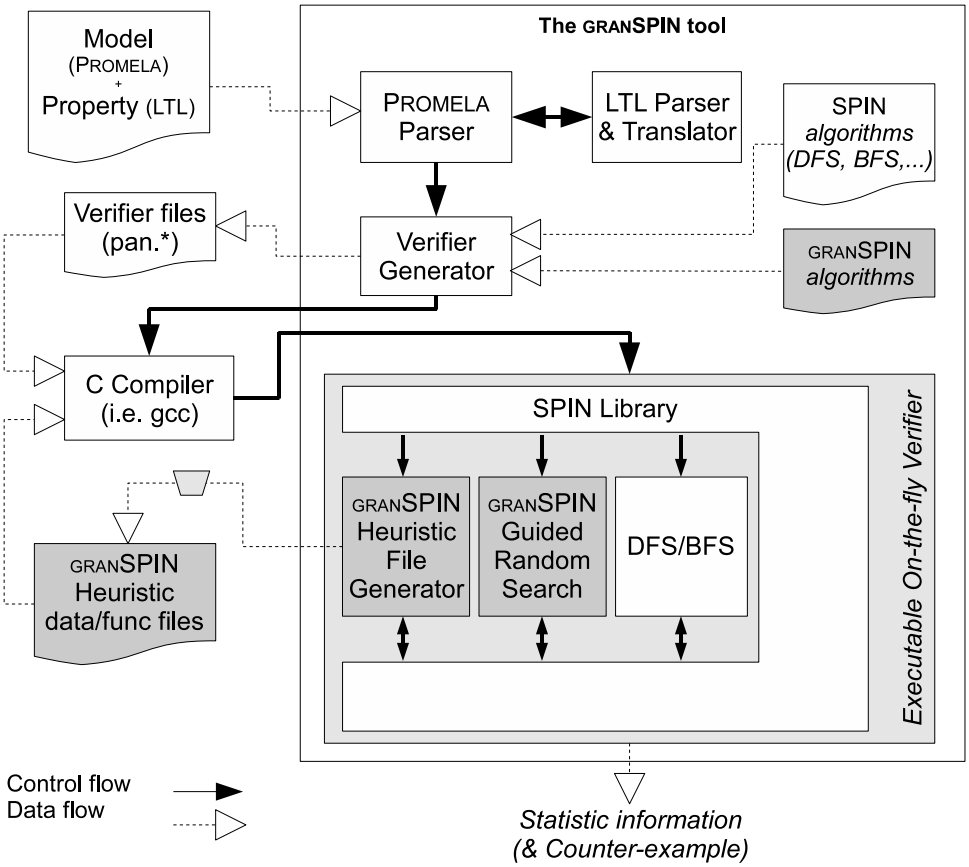


Figure 8.1: The architecture of GRANSPIN

the verifier files in this module. The new algorithms in GRANSPIN are also integrated into the verifier files in this step.

In the second stage, a C compiler such as `gcc` is invoked with the input "Verifier files" to generate an executable program (verifier). The heuristic in GRANSPIN is combined into this stage by incorporating additional files called "Heuristic data/func files". These files need to be generated manually by the user, and/or using the function called "Heuristic File Generator"

in GRANSPIN[1]. The model-checking options, such as choosing the algorithm (DFS, BFS or guided random-walk) in (SPIN and) GRANSPIN are given in this stage as compiler options. For example, the following command generates a verifier that uses the (unguided) random-walk algorithm to search for an error:

```
$gcc -o pan -DRW pan.c
```

After the executable verifier is generated, it is invoked to perform verification/checking. Depending on the compiler options selected by the user, a search algorithm is executed in this checking stage. The users may choose to use the original algorithm of SPIN such as (Nested) Depth-First Search (DFS) or Breadth-First Search (BFS), or the new algorithm in GRANSPIN such as Guided/Unguided Random-Walk Search. The outcome of the search may be a *counter-example*. Statistic information such as the total memory usage is generated as in SPIN. More information on random-walk search of GRANSPIN is also provided at this stage.

As mentioned in Section 8.1, SPIN cannot generate a heuristic. This is done in GRANSPIN in a function called "Heuristic File Generator". This generates "Heuristic data/func files" only, and stops immediately after generating the files. The heuristic files should be examined and possibly modified, and then sent to the compiling stage to generate a new guided random-walk model checking program (that include the heuristic).

[1]We have placed a small shaded trapezium in the data flow from the "Heuristic File Generator" module to indicate that the data flow may be manually modified by the users. In our current implementation, the "Heuristic File Generator" module is incomplete, and the user will need to modify the result before using it.

8.2.2 Command-Line Options

The 3 stages mentioned in the last section are the following:

Generating a Verifier

```
$spin -a <model file>
```

The result is a set of source files in the C programming language such as `pan.c` for the main program, `pan.h` for the (variable/data structure) declaration (header file), and `pan.t` for the transition declaration. For example, if the model of the dining-philosophers problem is given in the file `philo.pml`, then the command is:

```
$spin -a philo.pml
```

Compiling

```
$gcc -o pan <SPIN options> <granSPIN options> pan.c
```

For example, to use granSPIN's unguided random-walk algorithm (granSPIN's option `RW`) and set the SPIN's option `VECTORSZ` to a large number `1024`, we use:

```
$gcc -o pan -DVECTORSZ=1024 -DRW pan.c
```

For the complete options of SPIN, please refer to the SPIN manual. The granSPIN's options are given in the next section.

Running

```
$pan <pan options>
```

For example, to set the maximum search depth to `100000` steps in running the verifier, we use:

```
$pan -m100000
```

The complete options of `pan` are also given in the SPIN manual.

8.2.3 Compiling Options

In this section, we mention GRANSPIN's options only.

Required Options

RW Use GRANSPIN algorithm (required). The default algorithm is the unguided random-walk search. The try-a-child strategy is also the default in this option.

For example, to use the unguided random-walk algorithm and the try-a-child strategy, one could generate the verifier program by:

```
$gcc -o pan -DRW pan.c
```

Control Options

MAX_TRIALS=N *Termination Condition*: The maximum number of random walks. The default value is 1054.

RWNOLOOP *End-walk Condition*: Stop a walk immediately at the first loop (end-loop walk). The default is a walk ends at the maximum depth or at a leaf.

RWREFINE *Initialization Condition*: Start the next random walk from an arbitrary state of the previous walk. It starts a walk at an initial state by default.

RWSEED=seed Use the new `seed` for the random number generator.

For example, the following command will generate a model-checking program that uses the guided random-walk algorithm and performs a maximum 2000 random walks if it does not find an error. Each walk in the algorithm is initialised at a random state taken from the previous walk, except that the first walk is of course taken at an initial state. The seed for the random number generator used in the algorithm is 50.

```
$gcc -o pan -DRW -DMAX_TRIALS=2000 -DRWREFINE -DRWSEED=50 pan.c
```

Heuristic Generator Options

RWGRAPH Generate the transition graphs (control-flow) of all processes in the model in DOT format (See the *Graphviz* package at `http://www.graphviz.org`). The graph can be examined to modify the heuristic file `pan.z`, which is needed in compiling the guided random-walk search.

RWTRANS Generate the `pan.z` file. The file is used in compiling stage to generate the verifier program that uses the guided random-walk algorithms.

For example, the transition graph of the dining-philosophers problem in the file `philo.ps`, displayed in Figure 8.2, is generated by the following commands:[2]

[2]For the program `dot`, please refer to its manual at `http://www.graphviz.org`.

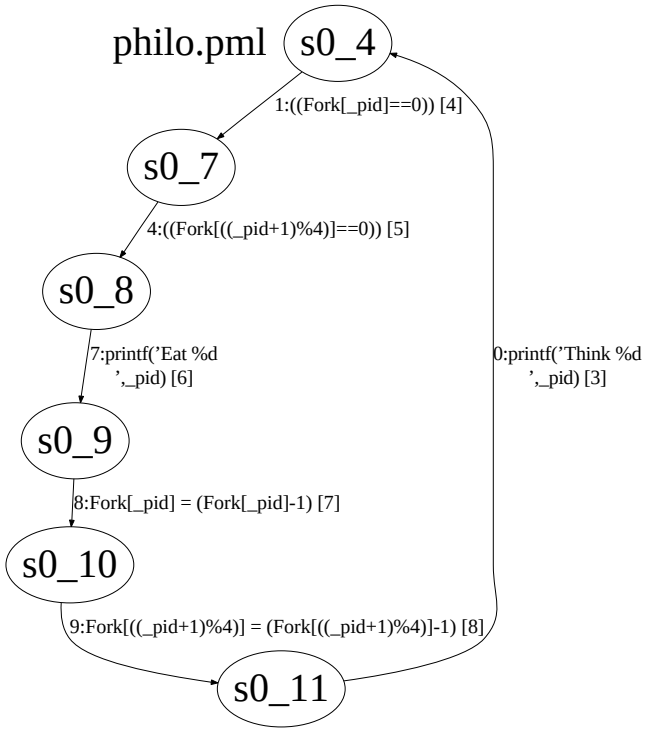


Figure 8.2: The transition graph of the dining-philosophers problem

```
$gcc -o pan -DRW -DRWGRAPH pan.c
$pan > philo.dot
$dot -Tps -o philo.ps philo.dot
```

Part of the heuristic file `pan.z` of the dining-philosophers problem, displayed in Figure 8.3, is generated by the following commands:

```
$gcc -o pan -DRW -DRWTRANS pan.c
$pan > pan.z
```

Note that the file `pan.z` has to be sent to the compiling stage to generate the executable model-checking program that uses the guided random-walk algorithms. As a result, the user needs to perform the compiling step twice:

```
#ifndef __PAN__Z__
#define __PAN__Z__

void set_heus(void) {

#if defined(GRWZ) || defined(GRW)

 state_heu = (int **) malloc(_NP_*sizeof(int*));

 state_heu[0] = (int *)malloc(15*sizeof(int));
 memset((void *)state_heu[0], 255, 15 * sizeof(int));

  state_heu[0][7] = 0;
  state_heu[0][8] = 5;
  state_heu[0][9] = 4;
  state_heu[0][10] = 3;
  state_heu[0][11] = 2;
  state_heu[0][4] = 1;

  state_heu_max = 5;

#else
```

Figure 8.3: Part of the heuristic file `pan.z` of the dining-philosophers problem

first to generate the `pan.z` file[3] and later for the guided random-walk based search program.

GRANSPIN's Algorithms Options

RWZ Use the unguided random-walk algorithm and the all-children strategy.

GRW Use the guided random-walk algorithm and the try-a-child strategy.

GRWZ Use the guided random-walk and the all-children strategy.

GRWHACK If this option is used together with `GRW` or `GRWZ` option, the heuristic named **h1** in Chapter 6 is used.

[3]Again, note that the user needs to modify the `pan.z` file before using it as this function is incomplete in this version.

GRWREV If this option is used together with guided modes (`GRW*` options), the reversed heuristic is used. See Chapter 6 for more details.

For example, the following command will generate the program that uses the guided random-walk algorithm granSPINt mentioned in Chapter 6[4]:

```
$gcc -o pan -DRW -DGRW pan.c
```

The following command will generate the program that uses unguided random-walk algorithm granSPINa:

```
$gcc -o pan -DRW -DRWZ pan.c
```

The following command will generate the program that uses guided random-walk algorithm granSPINa and uses the reversed heuristic:

```
$gcc -o pan -DRW -DGRWZ -DGRWREV pan.c
```

8.3 granGOLFER

8.3.1 The architecture of granGOLFER

The architecture of granGOLFER is described in Figure 8.4 (see Figure 8.1 in [71] for the architecture of GOLFER). The same notation as in Figure 8.1 is used in this figure. In this figure, the original modules/files that belong to GOLFER or NUSMV are unshaded, and those of granGOLFER are shaded[5]. The light shaded module "Directed Model Checking" is the GOLFER main

[4]Remember that the file `pan.z` must be provided beforehand.

[5]Note again that the small shaded trapezium indicates possible manual modification by the users.

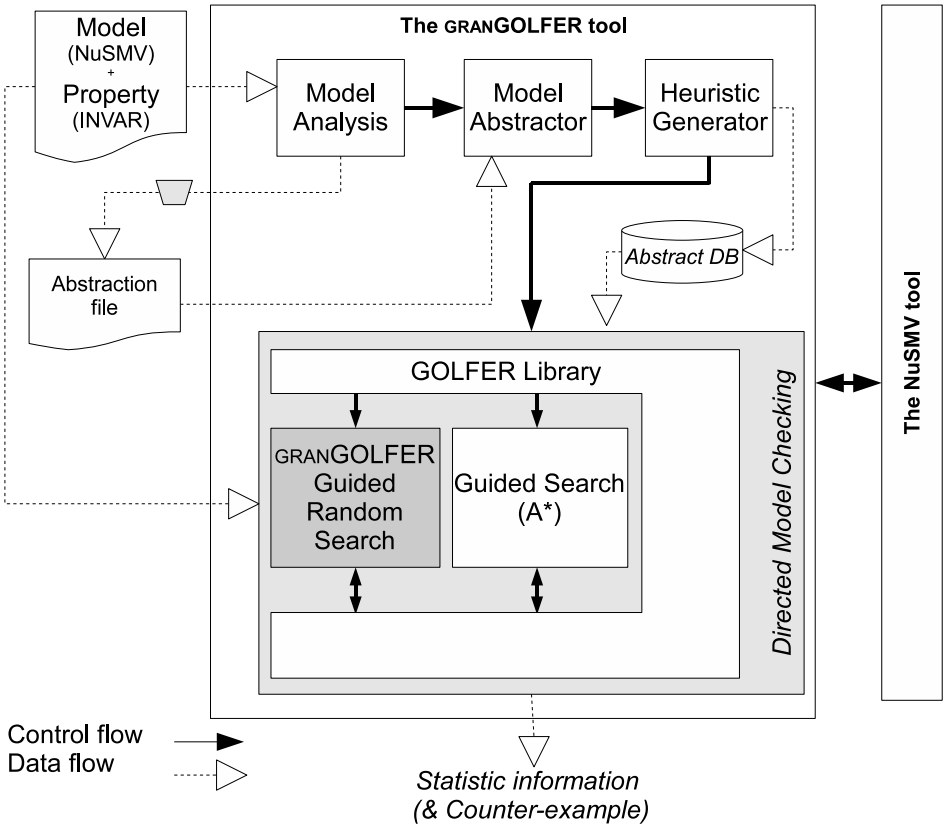


Figure 8.4: The architecture of GRANGOLFER

program with the new GRANGOLFER module. As in GOLFER, the tool interacts with NUSMV by calling its functions. This is indicated by the double arrow between the GRANGOLFER block and NUSMV block. As mentioned earlier in this chapter, GRANGOLFER is built on top of GOLFER by simply adding the new guided/unguided random-walk module. It is called "GRANGOLFER's Guided Random Search" in Figure 8.4. There are two stages in using GRANGOLFER: (1) generating the abstraction file, which contains a list of 'abstract away' variables, and (2) using the guided (random) search.

As in GOLFER, the abstraction file is generated by analysing the variable dependency of the input model ("Model Analysis" module). Actually,

the dependency graph of all variables in the model is built, and 'strong' variables are removed from the graph. The (names of) remaining 'weak' variables are output to the abstraction file. Following that, the "Model Abstractor" module receives the input model and the abstraction list and generates an abstract model. Upon receiving the abstract model from the "Model Abstractor" module, the "Heuristic Generator" module generates the abstract database "*Abstract DB*", and this is used in the heuristic (guided) search algorithm later.

In the second stage, the user needs to choose among granGOLFER's guided random search and GOLFER's guided search algorithms by proving options in the command line. A complete list of GOLFER's options can be found in [71]. The options for granGOLFER are described in the next section.

8.3.2 Command-line Options

Just like GOLFER, granGOLFER is invoked as an option in the NUSMV command line:

```
$NuSMV <NUSMV options> -golfer <GOLFER or granGOLFER options> <model>
```

The following options are for granGOLFER only.

Required Options

rmc Invoke the granGOLFER mode. The default algorithm in granGOLFER is the unguided random-walk algorithm. The default strategy used to select a successor state is all-children. This involves generating all 'real' successor states and then selecting one state randomly. The succes-

sor states are generated by performing a single operation on the BDD represented the current state and the BDD represented the transition relation of the model.

For example, the unguided random-walk search GRANGOLFER on the dining-philosophers problem (represented in the file `philo.smv`) can be invoked by the following command:

```
$NuSMV -golfer rmc philo.smv
```

Guided Random-Walk Search Options

rmc g [auto | manu 1 <abstract file>]

Invoke the GRANGOLFER in guided random search mode. In this mode, at each step in a walk, GRANGOLFER first generates the successor states from the current state, and then divides them into partitions, where each partition is assigned a heuristic cost taken from the abstract database *Abstract DB*. One partition is then selected in biased fashion so that the low-cost partitions are selected more often than the high-cost partitions. A successor state is then uniformly selected from the chosen partition.

If the option `auto` is given, the abstraction file is automatically generated as in GOLFER. Otherwise, the abstraction file is input by the option `manu 1 <abstract file>`.

For example, the guided random-walk search GRANGOLFER on the dining-philosophers problem can be invoked as follows. (Assume that the model is in the file `philo.smv` and the abstraction file is in the file `philo.abst`.)

```
$NuSMV -golfer rmc g manu 1 philo.abst philo.smv
```

Control Options

t Use the *random trail* option. In the random trail mode, no visited state is re-visited.

d <depth> *End-walk Condition*: Bound the length of a walk or trail to `depth`.

s <seed> The random generator is initialised by `seed`.

max <max_trials> *Termination Condition*: Bound the maximum of walks or trails to `max_trials`. The default value is 2020.

del <delta> eps <epsilon> *Termination Condition*: Bound the maximum number of walks or trails to $ln($`delta`$)/ln(1-$`epsilon`$))$.

rev Reverse the heuristic in guided random search mode.

For example, the guided random-trail search GRANGOLFER on the dining-philosophers problem, in which the maximum random trials is set to 1024 and the seed of the random generator is set to 50, can be invoked as follows:

```
$NuSMV -golfer rmc g t s 50 max 1024 manu 1 philo.abst philo.smv
```

The guided random-trail search GRANGOLFER on the dining-philosophers problem, in which the length of each trail is bounded to 100 steps, can be invoked as follows:

```
$NuSMV -golfer rmc g t d 100 manu 1 philo.abst philo.smv
```

8.4 Implementation Experiences

As mentioned earlier in this chapter, the tool GRANSPIN is based on SPIN. Modifying SPIN's source code to develop GRANSPIN was a challenging task. The first problem encountered was that the 'goto' statement in the C programming language was used to avoid using recursion to implement the (nested) depth-first search algorithm. While this probably results in a more efficient implementation, it also greatly reduces the readability of the source code. The second problem was simply to work out what the code does, without the assistance of any programming manuals. A third problem was that SPIN is a program that generates another program, which is a verifier. To modify SPIN's verifier, we have needed to understand how the verifier works, and then needed to modify SPIN's source code to generate the new verifier.

The development of GRANGOLFER at first appeared straightforward because NUSMV comes complete with programming manuals. However, these manuals do not provide sufficient detail, so it was still a difficult task to understand the source code.

The versions of SPIN and NUSMV that we modified are 4.2.7 and 2.3.1, respectively. Unfortunately these tools cannot be treated as "black-box": if the tools generated in this research are ever to be upgraded using new versions of SPIN or NUSMV, then these new versions would of course first need to be modified.

8.5 Distribution

Both GRANSPIN and GRANGOLFER can be obtained from our project website at `http://www.cse.unsw.edu.au/~anymeyer/projects`. In the website, the distributions together with the installation guides, and the user and program

manuals, are provided.

8.6 Summary

In this chapter, we have described the architectures of GRANSPIN and GRANGOLFER. The architectures are similar and consistent with the flow chart of the algorithm presented in Figure 1.1 in Chapter 1. The options and command-line usages of both tools are also given.

Chapter 9

Conclusion

Although model checking is used widely in the development of systems, its usefulness is limited by the size and complexity of the model. This research seeks to address this problem by providing the system developer with a formal tool that is able to hone in on temporally-defined errors on even very large systems, with minimal user interaction. There is no certainty of finding errors (the approach cannot verify a model), but experiments have consistently shown that it is fast and effective. In effect, sacrificing certainty for scalability has meant that the system developer can analyse far larger models than any conventional model checker can handle, and even handle models that are normally intractable. The developer can improve and fine-tune the performance by changing the heuristic (not studied in this work) and by changing the parameters of the random-walk strategy (restricting the length of the walk for example). It is the combination of guided and random-walk strategies that makes this research unique in the model-checking world. Unlike conventional model checking, in our approach the model of the system itself may need little or no compromise to find errors. The approach has been studied in both explicit-state and symbolic environments, and has resulted

in corresponding tools GRANSPIN and GRANGOLFER.

We considered symbolic models with only invariant properties in this work, and explicit-state models with properties expressed in LTL. If a model can be represented explicitly, our results show that the explicit-state guided random-walk algorithm is the fastest algorithm to check an LTL property.

9.1 Discussion

In this section we discuss the contributions that this thesis has made.

9.1.1 Random-Walk Search

The experiments have confirmed that the explicit-state random-walk algorithm is faster and consumes less memory than the depth-first search algorithm (used in SPIN). The difference is sometimes by many orders of magnitude. Importantly, our approach can still find errors for very large systems while conventional model checkers like SPIN cannot because of resource limitations. The random-walk algorithm has also been implemented symbolically. Experiments there also confirm that the new algorithm is faster and uses less memory than the breadth-first search used in NUSMV for example.

The disadvantage of the random-walk method is it may need to be terminated before it has found particular errors. In fact, a random-walk search is always terminated prematurely because it is always possible to carry out another random walk. So, even if the random-walk search has located errors, there may be more.

In this work we have proposed the use of random trails in which no state in a walk is re-visited. Trails can be used to derive a 'probabilistic' condition to terminate random walks. This makes it possible to conclude with a certain

probability whether further random walks are likely to discover an error, along the lines of work by Grosu and Smolka [42]. This avenue was not investigated further in this work however but it is a logical next step. Our focus has been the effectiveness of guided random-walk based research in finding errors under varying conditions. Our experiments suggest that it is not important what condition is used to terminate a random walk, and there is surprisingly little difference in effectiveness between walks which are bound and those which are un-bound.

9.1.2 Abstraction-Heuristic Guided Random-Walk Search

Abstraction-heuristic guided search analyses the given concrete model to build a variable dependency graph of the state space. This graph enables certain variables, which are deemed 'weak', to be identified, and these variables are abstracted away from the concrete model to build an abstract model. This abstract model functions as a heuristic in guided search. This mechanism is used in the 'conventional' guided model checker GOLFER, and also in the corresponding guided random-walk based model checker GRANGOLFER. This has enabled a direct comparison between the two methodologies to be made. Even though precisely the same heuristic mechanism is used in both methodologies, GRANGOLFER outperformed GOLFER in our experiments in terms of its ability to find errors and resource utilisation. This is somewhat counter-intuitive because the guided random-walk approach is still basically a randomised search in the state space.

9.1.3 Integration and Complexity

The symbolic and explicit-state algorithms have been integrated into a single framework called *Guided Random-walk Based Model Checking* (see Chap-

ter 7). The complexity analysis shows that the unguided random-walk search algorithms (both symbolic and explicit-state) are faster than the guided random-walk search algorithms. In practice, of course, because the guided random-walk search is 'smarter' than the unguided search, the run-time of the guided algorithm is expected to be smaller. The reason is that we can estimate the time and memory required to compute the heuristic in the guided approach, but we cannot estimate the saving that a heuristic makes during the search. A complexity analysis therefore cannot be used to measure the difference in performance or even understand what mechanisms are at work.

However, the complexity analysis does show that both the guided and unguided random-walk search algorithms are faster than a guided search algorithm. This is confirmed in our experiments.

A major advantage of using similar technologies to debug a model and to verify a model is that the transition in both the forward and backward directions between the corresponding two phases of a development can be done almost seamlessly. The same model, the same temporal property, and even the same heuristic can be used for both.

9.1.4 Model Characterisation and Classification

To achieve truly automated model checking, we need to understand better under what circumstances one tool is 'more suited' to analysing a model than another. We have found both explicit-state and symbolic models which can be analysed by one tool but are intractable by another. Why is this so? Just as in any science, the aim here should be to classify models. We have proposed here a software metric called the *Process Error Participation* (PEP), which measures the number of processes involved in producing a particular error. We have classified models into those that are low PEP and those that are high

PEP, and experimented in the effectiveness of heuristics on these models. It is found that the conventional guided search is fast in low PEP systems and relatively slow in high PEP systems, whereas in guided random-walk based search PEP appears to play no role. This matches our intuition that says that when there are few processes in a model that can cause the system to go into an error state, then for these models, a heuristic is most effective (in terms of run-time and length of goal path). Expressed another way, it is easier to devise a heuristic when processes exhibit non-uniform behaviour than when all processes behave similarly. In the case of high PEP symbolic models (when all processes behave similarly), the symbolic guided random-walk algorithm is the best in our experiments.

9.1.5 "Good" versus "Bad" Abstractions

One can say that our notion of goodness of abstraction in this work is the resulting speed-up of the search. *Can we predict when a significant speed-up will happen?* That is what the PEP factor is supposed to help us with. It does appear to be a good first step, and we have not been able to show it can be wrong, but much more experimentation, and theoretical work, needs to be carried out to verify its validity. In our experiments, we have novelly generated "bad" abstractions by reversing "good" abstractions, and the results do reveal the level of sensitivity of the performance to the quality of heuristic.

9.1.6 Summing Up

In this thesis we have shown that the heuristic guided random-walk algorithm can be effective in finding errors states in model checking. At the cost of verification, it can handle very large state spaces. It is also stable, finding

errors wherever we place them equally quickly in our experiments (there is no left-right bias as in depth-first search for example). Importantly, guided random-walk search is 'smart' as it uses domain knowledge to guide walks towards particular areas in the state space where errors are possibly located. We also proposed the use of random trails, in which no state is re-visited. We have shown that, using only maximal random trails, one can apply so-called Monte Carlo theory [42]. Finally, we have devised the PEP factor to classify models in terms of performance (of the algorithms).

9.2 Threats to Validity

We have carried out extensive experimentation in this research using the random-walk tools GRANSPIN and GRANGOLFER, and compared their performance with SPIN, NUSMV and GOLFER. We have also made other empirical comparisons between explicit-state and symbolic model checking. There can be many *threats* to the validity of experimental research. These threats need to be studied carefully to verify the soundness of the results.

9.2.1 Internal Validity

In our experiments, we have limited (i.e. bound) the maximum depth in searching for error states using the explicit-state (guided) random-walk algorithms (RANSPIN or GRANSPIN). Whenever a random walk reaches the bound, a new random walk is commenced. Changing this bound may affect the result. If the bound is too small, the error may never be found (as it may be out of reach). Moreover, if the bound is small, the search may carry out many random walks before reaching the goal, so there is a loss of efficiency. On the other hand, if the bound is too large, then the violation may be found

before reaching this bound, and in that case the bound will have no effect. Finding the right bound is hence an issue that can affect the internal validity of the results. This issue has been discussed in Section 4.3.2 in Chapter 4 and Section 6.3 in Chapter 6.

9.2.2 External Validity

The current version of SPIN and NUSMV we have used to develop our tools are 4.2.7 and 2.3.1, respectively. If our experiments were to be repeated using newer versions of these tools then the results may be different. Note however, this research deals only with heuristic guidance and random walks, and does not alter the algorithms in the kernel of SPIN and NUSMV. Qualitatively speaking, therefore, the comparisons we have drawn between convention model checking and random-walk based search are unlikely to be affected.

9.2.3 Construct Validity

The comparisons that we have focussed on in our experiments involve execution time, memory usage and the length of counter-examples. If the aim is to find error states in a very large state space, then execution time and memory usage will be important aspects. Maybe less important is the length of counter-examples, which are used by system engineers to understand why an error occurred and to 'repair' a specification.

There are of course a variety of other measures of performance such as the number of states that are visited and the number of counter-examples that are generated. The number of visited states may be important in a guided search, but it is generally not important in (guided) random-walk search because whenever new walks are undertaken, the visited states in

previous walks are deleted from memory. The number of visited states in any successful run is hence similar to the length of the counter-example. The number of counter-examples could be useful in comparing algorithms in certain application areas, and this could be future work.

9.3 Future Work

There are many directions in which this research could now proceed. The abstraction-heuristic method has been used but this is still largely a 'black-box'. Many ways of generating an abstract model are possible, but there is no research to indicate what features in an abstract model would provide the best heuristic in our guidance setting. As well, it is important that the abstract model be generated automatically, one would think, although it is very conceivable that user input into the abstraction process could be very useful. This is analogous to software where a debugger is 'directed' by the user to look in particular sections of code to find a bug.

Model classification also needs further work. The PEP factor is very course, and more sophisticated, finer-grained measures of concurrency and the error property are possible. Some attempt to automate and formalise the metric should be made.

While the tools GRANSPIN and GRANGOLFER are built on the well established tools SPIN and GOLFER, they are not designed to be user-friendly. Our tools are still very much experimental platforms, and not production tools. The algorithm that we have developed that integrates the explicit-state and symbolic guided random-walk based approaches also indicates that a single platform is possible. A 'seamless' tool environment that provides the user a variety of (guided) random-walk based debug tools and (guided) conventional

verification tools, in which models are expressed both symbolically or explicit state, is the ultimate goal of this work.

This page intentionally left blank.

Author's Publications

The following are the publications which have been produced by or in conjunction with the author during his PhD candidacy.

1. Thang H. Bui and Albert Nymeyer, "RANSPIN: A Random-Walk Based Model Checker", in *Proc. of Int. Workshop on Advanced Computing and Applications (ACOMP'08)*, (Ho Chi Minh City, Vietnam), pp. 38–48, Mar 2008.

2. Thang H. Bui and Albert Nymeyer, "The spin on guided random search in verification", in *IEEE Intl Conf. on Software Testing Verification and Validation Workshop (ICSTW'08)*, pp. 170–177, Mar 2008.

3. Thang H. Bui and Albert Nymeyer, "Formal Verification Based on Guided Random Walks", in *Proc. of the 7th Int. Conf. on integrated Formal Methods (iFM'09)*, (Düsseldorf, Germany), volume 5423 of LNCS, pp. 72–87, Springer-Verlag, Feb 2009.

4. Thang H. Bui and Albert Nymeyer, "Guided Model Checking using Random Walks (extended abstract)", in *Accepted for representation at 12th Int. Conf. on Computer Aided System Theory (EuroCAST'09)*, Feb 2009.

5. Thang H. Bui and Albert Nymeyer, "Formal model simulation: can it be guided?", in *Proc. of the 1st Int. Symp. on Search based Software Engineering (SSBSE'09)*, (Windsor, UK), pp. 93–96, IEEE Computer Society, May 2009.

6. Thang H. Bui and Albert Nymeyer, "Heuristic Sensitivity in Guided Random-Walk Based Model Checking", in *Proc. of the 7th IEEE Int. Conf. on Software Engineering and Formal Methods (SEFM'09)*, (Ha Noi, Vietnam), pp. 125–134, IEEE Computer Society, Nov 2009.

7. Thang H. Bui and Albert Nymeyer, "Integrated Guided Model Checking and Model Simulation", in *Submitted for publication*, 2009.

Bibliography

[1] E. Alba and F. Chicano. Searching for liveness property violations in concurrent systems with ACO. In *Proc. of the 10th Annual Conf. on Genetic and Evolutionary Computation (GECCO'08)*, pages 1727–1734, New York, NY, USA, 2008. ACM.

[2] E. Alba, F. Chicano, M. Ferreira, and J. Gomez-Pulido. Finding deadlocks in large concurrent java programs using genetic algorithms. In *Proc. of the 10th Annual Conf. on Genetic and Evolutionary Computation (GECCO'08)*, pages 1735–1742, New York, NY, USA, 2008. ACM.

[3] E. Alba and J. M. Troya. Genetic algorithms for protocol validation. In *Proc. of the 4th Int. Conf. on Parallel Problem Solving from Nature*, volume 1141 of *LNCS*, pages 870–879, London, UK, 1996. Springer-Verlag.

[4] G. Avrunin, J. C. Corbett, and M. B. Dwyer. Benchmarking finite-state verifiers. *Int. Journal on Software Tools for Technology Transfer*, 2(4):317–320, 2000.

[5] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *Proc. of the ACM SIGPLAN*

2001 Conf. on Programming Language Design and Implementation, pages 203–213. ACM Press, 2001.

[6] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Proc. of the 5th Int. Conf. on Tools and Algorithms for Construction and Analysis of Systems (TACAS'99)*, pages 193–207. Springer-Verlag, 1999.

[7] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transaction*, C-35(8):677–691, Aug 1986.

[8] T. H. Bui and A. Nymeyer. RANSPIN: A random-walk based model checker. In *Workshop on Advanced Computing and Applications (ACOMP'08)*, pages 38–48, Mar 2008. In print.

[9] T. H. Bui and A. Nymeyer. The spin on guided random search in verification. *IEEE Int. Conf. on Software Testing Verification and Validation Workshop (ICSTW'08)*, pages 170–177, 2008.

[10] T. H. Bui and A. Nymeyer. Formal model simulation: Can it be guided? In *Proc. of the 1st Int. Sym. on Search Based Software Engineering (SSBSE'09)*, pages 93–96. IEEE Computer Society, May 2009.

[11] T. H. Bui and A. Nymeyer. Formal verification based on guided random walks. In *Proc. of the 7th Int. Conf. on Integrated Formal Methods (iFM'09)*, volume 5423 of *LNCS*, pages 72–87. Springer-Verlag, Feb 2009.

[12] T. H. Bui and A. Nymeyer. Guided model checking using random walks (extended abstract). In *Accepted for representation at the 12th Int. Conf. on Computer Aided Systems Theory (EuroCAST'09)*, Feb 2009.

[13] T. H. Bui and A. Nymeyer. Heuristic sensitivity in guided random-walk based model checking. In *Proc. of the 7th IEEE Int. Conf. on Software Engineering and Formal Methods (SEFM'09)*, pages 125–134. IEEE Computer Society, Nov 2009.

[14] T. H. Bui and A. Nymeyer. Integrated guided model checking and model simulation. In *Submitted for publication*, 2009.

[15] J. R. Burch, E. M. Clarke, and D. E. Long. Representing circuits more efficiently in symbolic model checking. In *Proc. of the 28th Conf. on ACM/IEEE Design Automation (DAC'91)*, pages 403–407, New York, NY, USA, 1991. ACM.

[16] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proc. of the 5th Annual IEEE Symp. on Logic in Computer Science*, pages 1–33, Washington, D.C., 1990. IEEE Computer Society Press.

[17] F. Chicano and E. Alba. Ant colony optimization with partial order reduction for discovering safety property violations in concurrent models. *Information Processing Letters*, 106(6):221–231, 2008.

[18] A. Cimatti, E. M. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: A new symbolic model verifier. In *Procs. of the 11th Int. Conf. on Computer Aided Verification (CAV'99)*, volume 1633 of *LNCS*, pages 495–499. Springer, 1999.

[19] E. . Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, 1994.

[20] E. M. Clarke and E. A. Emerson. Synthesis of synchronization skeletons for branching time temporal logic. In *Workshop on Logic of Programs*, volume 131 of *LNCS*, pages 52–71, Yorktown Heights, 1981. Springer-Verlag.

[21] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Proc. the* 12^{th} *Int. Conf. on Computer Aided Verification (CAV'00)*, volume 1855 of *LNCS*, pages 154–169. Springer-Verlag, 2000.

[22] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.

[23] E. M. Clarke and J. M. Wing. Formal methods: state of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, 1996.

[24] J. M. Cobleigh, L. A. Clarke, and L. J. Osterweil. The right algorithm at the right time: Comparing data flow analysis algorithms for finite state verification. In *Proc. of the* 23^{rd} *Int. Conf. on Software Engineering*, pages 37–46. IEEE Computer Society, 2001.

[25] J. C. Culberson and J. Schaeffer. Searching with pattern databases. In *Proc. of the* 11^{th} *Biennial Conf. of the Canadian Society for Computational Studies of Intelligence*, volume 1081 of *LNCS*, pages 402–416. Springer-Verlag, 1996.

[26] E. W. Dijkstra. The humble programmer. *Communications of the ACM*, 15(10):859–866, 1972.

[27] K. Dräger, B. Finkbeiner, and A. Podelski. Directed model checking with distance-preserving abstractions. In *Proc. of the* 13^{th} *Int. SPIN*

Workshop on Model Checking of Software (SPIN'06), volume 3925 of *LNCS*, pages 19–34. Springer, 2006.

[28] M. B. Dwyer, S. Elbaum, S. Person, and R. Purandare. Parallel randomized state-space search. In *Proc. of the 29th Int. Conf. on Software Engineering (ICSE'07)*, pages 3–12, Washington, DC, USA, 2007. IEEE Computer Society.

[29] S. Edelkamp. Promela planning. In *Proc. of the 10th Int. SPIN Workshop on Model Checking of Software (SPIN'03)*, volume 2648 of *LNCS*, pages 197–212. Springer, 2003.

[30] S. Edelkamp and S. Jabbar. Large-scale directed model checking LTL. In *Proc. of the 9th Int. SPIN Workshop on Model Checking of Software (SPIN'06)*, volume 3925 of *LNCS*, pages 242–245. Springer, 2006.

[31] S. Edelkamp, S. Leue, and A. Lluch-Lafuente. Directed explicit-state model checking in the validation of communication protocols. *Int. Journal on Software Tools for Technology Transfer*, 5:247–267, 2004.

[32] S. Edelkamp and A. Lluch-Lafuente. Abstraction databases in theory and model checking practice. In *Proc. of Workshop on Connecting Planning Theory with Practice, Int. Conf. on Automated Planning and Scheduling (ICAPS)*, 2004.

[33] S. Edelkamp, A. Lluch-Lafuente, and S. Leue. Directed explicit model checking with HSF–SPIN. In *Proc. of the 8th Int. SPIN Workshop on Model Checking of Software (SPIN'01)*, volume 2057 of *LNCS*, pages 57–79. Springer, May 2001.

[34] Y. Eytani, K. Havelund, S. D. Stoller, and S. Ur. Towards a framework and a benchmark for testing tools for multi-threaded programs.

Concurrency and Computation: Practice & Experience, 19(3):267–279, 2007.

[35] M. M. Gallardo, J. Martinez, P. Merino, and E. Pimentel. αSPIN: A tool for abstract model checking. *Int. Journal on Software Tools for Technology Transfer*, 5:165–184, 2000.

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

[37] P. Godefroid and S. Khurshid. Exploring very large state spaces using genetic algorithms. *Int. Journal on Software Tools for Technlogy Transfer*, 6(2):117–127, 2004.

[38] S. Gradara, A. Santone, and M. Luisa Villani. Formal verification of concurrent systems via directed model checking. In *Electronic Notes in Theoretical Computer Science*, volume 185, pages 93–105. Elsevier, 2007.

[39] J. Ch. Grégoire. State space compression in SPIN with GETSs. In *Proc. of the 2nd Int. SPIN Workshop on Model Checking of Software (SPIN'96)*, pages 90–108. American Mathematical Society, 1996.

[40] A. Groce and W. Visser. Heuristics for model checking Java programs. *Int. Journal on Software Tools for Technology Transfer*, 6(4):260–276, 2004.

[41] R. Grosu, X. Huang, S. A. Smolka, W. Tan, and S. Tripakis. Deep random search for efficient model checking of timed automata. In F. Kordon and O. Sokolsky, editors, *Monterey Workshop*, volume 4888 of *LNCS*, pages 111–124. Springer, 2006.

[42] R. Grosu and S. A. Smolka. Monte Carlo model checking. In *Proc. of the 11th Int. Conf. on Tools and Algorithms for Construction and Analysis of Systems (TACAS'05)*, volume 3440 of *LNCS*, pages 271–286. Springer, 2005.

[43] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Trans. on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[44] P. Haslum. Model checking by random walk. In *Proc. of ECSEL Workshop*, 1999.

[45] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Proc. of the 29th SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 58–70. ACM, 2002.

[46] J. Hoffmann, J-G. Smaus, A. Rybalchenko, S. Kupferschmid, and A. Podelski. Using predicate abstraction to generate heuristic functions in UPPAAL. In S. Edelkamp and A. Lomuscio, editors, *Proc. of the 4th Workshop on Model Checking and Artificial Intelligence (MoChArt'06)*, volume 4428 of *LNCS*, pages 51–66. Springer, 2006.

[47] G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.

[48] G. J. Holzmann. The engineering of a model checker: The gnu i-protocol case study revisited. In D. Dams, R. Gerth, S. Leue, and M. Massink, editors, *Proc. of the 5th and 6th Int. SPIN Workshops on Theoretical and Practical Aspects of SPIN Model Checking*, volume 1680 of *LNCS*, pages 232–244, London, UK, 1999. Springer-Verlag.

[49] G. J. Holzmann. *The SPIN Model Checker*. Addison Wesley, 2004.

[50] M. Jones and E. Mercer. Explicit state model checking with Hopper. In *Proc. of the 11th Int. SPIN Workshop on Model Checking of Software (SPIN'04)*, volume 2989 of *LNCS*, pages 146–150. Springer, 2004.

[51] M. Kamel and S. Leue. Formalization and validation of the general inter-orb protocol (GIOP) using Promela and SPIN. In *Int. Journal on Software Tools for Technology Transfer*, pages 394–409, 2000.

[52] A. Kuehlmann, K. L. McMillan, and R. K. Brayton. Probabilistic state space search. In *Proc. of the 1999 IEEE/ACM Int. Conf. on Computer-Aided Design (ICCAD'99)*, pages 574–579, Piscataway, NJ, USA, 1999. IEEE Press.

[53] S. Kupferschmid, K. Dräger, J. Hoffmann, B. Finkbeiner, H. Dierks, A. Podelski, and G. Behrmann. Uppaal/DMC - abstraction-based heuristics for directed model checking. In O. Grumberg and M. Huth, editors, *Proc. of the 13th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'07)*, volume 4424 of *LNCS*, pages 679–682, Berlin Heidelberg, July 2007. Springer-Verlag.

[54] S. Kupferschmid, J. Hoffmann, H. Dierks, and G. Behrmann. Adapting an AI planning heuristic for directed model checking. In A. Valmari, editor, *Proc. of the 13th Int. SPIN Workshop on Model Checking of Software (SPIN'06)*, volume 3925 of *LNCS*, pages 35–52. Springer, 2006.

[55] S. Kupferschmid, J. Hoffmann, and K. G. Larsen. Fast directed model checking via russian doll abstraction. In C. R. Ramakrishnan and J. Rehof, editors, *Proc. of the 14th Int. Conf. on Tools and Algorithms for*

the Construction and Analysis of Systems (TACAS'08), volume 4963 of *LNCS*, pages 203–217. Springer, 2008.

[56] K. L. McMillan. *Symbolic model checking*. Kluwer Academic Publishers, Boston, MA, 1993.

[57] T. Menzies, D. Owen, and B. Cukic. Saturation effects in testing of formal models. *Proc. of the 13th Int. Sym. on Software Reliability Engineering (ISSRE'02)*, 00:15–26, 2002.

[58] T. Menzies, D. Owen, and J. Richardson. The strangest thing about software. *IEEE Computer*, 40(1):54–60, 2007.

[59] M. Mihail and C. H. Papadimitriou. On the random walk method for protocol testing. In D. L. Dill, editor, *Proc. of the 6th Int. Conf. on Computer Aided Verification (CAV'94)*, volume 818 of *LNCS*, pages 132–141, Standford, California, USA, 1994. Springer-Verlag.

[60] H. Mony, J. Baumgartner, V. Paruthi, R. Kanzelman, and A. Kuehlmann. Scalable automated verification via expert-system guided transformations. In *Proc. of Int. Conf. on Formal Methods in Computer-Aided Design (FMCAD'04)*, volume 3312 of *LNCS*, pages 159–173. Springer, 2004.

[61] N. J. Nilsson. *Principles of artificial intelligence*. Morgan Kaufmann Publishers Inc., 1980.

[62] A. Nymeyer and K. Qian. Heuristic search algorithm based on symbolic data structures. In *Proc. of the 16th Aust. Joint Conf. in Artificial Intelligence*, volume 2903 of *LNAI*, pages 966–979. Springer-Verlag, 2003.

[63] D. Owen, B. Cukic, and T. Menzies. An alternative to model checking: verification by random search of AND/OR graphs representing finite-state models. In *Proc. of the 7th IEEE Int. Sym. on High-Assurance Systems Engineering (HASE'02)*, pages 119–128. IEEE Computer Society, October 2002.

[64] D. Owen and T. Menzies. Lurch: a lightweight alternative to model checking. In *Proc. of the 15th Software Engineering & Knowledge Engineering (SEKE'03)*, pages 158–165, 2003.

[65] D. Owen, T. Menzies, M. Heimdahl, and J. Gao. On the advantages of approximate vs. complete verification: Bigger models, faster, less memory, usually accurate. In *Proc. of IEEE/NASA Software Engineering Workshop (SEW'03)*, pages 75–81, 2003.

[66] R. Pelánek. Typical structural properties of state spaces. In *Proc. of the 11th Int. SPIN Workshop on Model Checking of Software (SPIN'04)*, volume 2989 of *LNCS*, pages 5–22. Springer, 2004.

[67] R. Pelánek. Model classifications and automated verification. In S. Leue and P. Merino, editors, *Proc. of the 12th Int. Workshop on Formal Methods for Industrial Critical Systems (FMICS'07)*, volume 4916 of *LNCS*, pages 149–163. Springer, 2007.

[68] R. Pelánek, T. Hanžl, I. Černá, and L. Brim. Enhancing random walk state space exploration. In *Proc. of the 10th Int. Workshop on Formal Methods for Industrial Critical Systems (FMICS'05)*, pages 98–105. ACM Press, 2005.

[69] D. Peled. Combining partial order reductions with on-the-fly model-checking. In *Proc. of the 6th Int. Conf. on Computer Aided Verification*

(CAV'94), volume 818 of *LNCS*, pages 377–390, London, UK, 1994. Springer-Verlag.

[70] A. Pnueli. The temporal semantics of concurrent programs. *Theoretical Computer Science*, 13:45–60, 1981.

[71] K. Qian. *Formal Verification using heuristic search and abstraction techniques*. PhD in Computer Science, School of Computer Science and Engineering – The University of New South Wales, Kensington, NSW 2052, Australia, 2006.

[72] K. Qian and A. Nymeyer. Abstraction-based model checking using heuristical refinement. In *Proc. of the 2nd Int. Symp. on Automated Technology for Verification and Analysis (ATVA'04)*, volume 3299 of *LNCS*, pages 165–178. Springer-Verlag, 2004.

[73] K. Qian and A. Nymeyer. Guided invariant model checking based on abstraction and symbolic pattern databases. In *Proc. of the 10th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'04)*, volume 2988 of *LNCS*, pages 497–511. Springer-Verlag, 2004.

[74] K. Qian and A. Nymeyer. Language-emptiness checking of alternating tree automata using symbolic reachability analysis. In *Proc. of the 3rd Workshop on Model Checking and Artificial Intelligence (MoChArt'05)*, volume 149(2) of *Electronic Notes in Theoretical Computer Science*, pages 33–49. Elsevier, 2006.

[75] K. Qian, A. Nymeyer, and S. Susanto. Abstraction-guided model checking using symbolic IDA* and heuristic synthesis. In *Proc. of the 25th Int. Conf. on Formal Techniques for Networked and Distributed Systems*

(FORTE'05), volume 3731 of *LNCS*, pages 275–289. Springer-Verlag, October 2005.

[76] K. Qian, A. Nymeyer, and S. Susanto. Experiments in multiple abstraction heuristics in symbolic verification. In *Proc. of the 6th Sym. on Abstraction, Reformulation and Approximation (SARA'05)*, volume 3607 of *LNAI*, pages 290–304, July 2005.

[77] J. P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proc. of the 5th Colloquium on Int. Sym. on Programming*, volume 137 of *LNCS*, pages 337–351, London, UK, 1982. Springer-Verlag.

[78] N. Rungta and E. G. Mercer. Generating counter-examples through randomized guided search. In *Proc. of the 14th Int. SPIN Workshop on Model Checking of Software (SPIN'07)*, volume 4595 of *LNCS*, pages 39–57, Berlin, Germany, July 2007. Springer-Verlag.

[79] T. C. Ruys and E. Brinksma. Managing the verification trajectory. *Int. Journal on Software Tools for Technology Transfer*, 4(2):246–259, 2003.

[80] D. Sahoo, J. Jain, S. K. Iyer, D. L. Dill, and E. A. Emerson. Predictive reachability using a sample-based approach. In *Proc. of Correct Hardware Design and Verification Methods (CHARME'05)*, volume 3725 of *LNCS*, pages 388–392. Springer, 2005.

[81] K. Seppi, M. Jones, and P. Lamborn. Guided model checking with a bayesian meta-heuristic. In *Proc. of the 4th Int. Conf. on Application of Concurrency to System Design (ACSD'04)*, pages 217–226. IEEE, 2004.

[82] S. Shyam and V. Bertacco. Distance-guided hybrid verification with GUIDO. In *Proc. of Conf. on Design, Automation and Test in Europe*

(DATE'06), pages 1211–1216, 3001 Leuven, Belgium, Belgium, 2006. European Design and Automation Association.

[83] H. Sivaraj and G. Gopalakrishnan. Random walk based heuristic algorithms for distributed memory model checking. In *Proc. of the 2nd Int. Workshop on Parallel and Distributed Model Checking (PDMC'03)*, volume 89(1) of *Electronic Notes in Theoretical Computer Science*, pages 51–67. Elsevier, July 2003.

[84] U. Stern and D. L. Dill. Parallelizing the Murφ verifier. In *Proc. of the 9th Int. Conf. in Computer Aided Verification (CAV'97)*, pages 256–278, London, UK, 1997. Springer-Verlag.

[85] D. A. Stuart, M. Brockmeyer, A. K. Mok, and F. Jahanian. Simulation–verification: Biting at the state explosion problem. *IEEE Transactions on Software Engineering*, 27(7):599–617, 2001.

[86] E. Tronci, G. D. Penna, B. Intrigila, and M. Venturini. A probabilistic approach to automatic verification of concurrent systems. In *Proc. of the 8th Asia-Pacific on Software Engineering Conf. (APSEC'01)*, pages 317–324, Washington, DC, USA, 2001. IEEE Computer Society.

[87] M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification (preliminary report). In *Proc. of the Symp. on Logic in Computer Science*, pages 332–344. IEEE, 1986.

[88] C. West. Protocol validation in complex systems. *ACM SIGCOMM Computer Communication Review*, 19(4):303–312, 1986.

[89] P. Wolper and D. Leroy. Reliable hashing without collision detection. In *Proc. of the 5th Int. Conf. on Computer Aided Verification (CAV'93)*, volume 697 of *LNCS*, pages 59–70. Springer-Verlag, 1993.