GUIDED SEARCH FOR DEADLOCKS IN ACTOR-BASED MODELS

Steinar Hugi Sigurðarson

Master of Science

Software Engineering

June 2011

School of Computer Science

Reykjavík University

M.Sc. RESEARCH THESIS

ISSN 1670-8539

Guided Search for Deadlocks in Actor-Based Models

by

Steinar Hugi Sigurðarson

Research thesis submitted to the School of Computer Science
at Reykjavík University in partial fulfillment of
the requirements for the degree of
Master of Science in **Software Engineering**

June 2011

Research Thesis Committee:

Dr. Marjan Sirjani, Supervisor
Associate Professor, Reykjavik University, Iceland

Dr. Yngvi Björnsson, Co-Supervisor
Associate Professor, Reykjavik University, Iceland

Dr. Luca Aceto
Professor, Reykjavik University, Iceland

Prof.dr. Frank S. de Boer
Senior Researcher, Centrum Wiskunde & Informatica, Netherlands.
Professor, Leiden Institute of Advanced Computer Science, Netherlands.

Guided Search for Deadlocks in Actor-Based Models

Steinar Hugi Sigurðarson

June 2011

Abstract

The success of model checking is based on its ability to uncover errors in designs of software and protocols. Even a small reactive concurrent system can exhibit complex behavior. Such systems may have state-spaces larger than explicit state model checkers can verify. In practice, finding an error with a model checker is more useful than proving a property. Informed search algorithms use heuristic strategies with problem-specific knowledge to find solutions more efficiently than uninformed algorithms. Generally, such heuristics estimate the distance from a given state to a goal state. We present seven heuristics for guiding search algorithms through the state-space of actor-based models to a deadlock. Our methods can find a deadlock more efficiently than uninformed searches for some actor-based models. The A* search algorithm guarantees an optimal solution and returns the shortest counter-example. These methods are supported by a tool that performs directed model checking of the deadlock property. The objective is to detect difficult errors that might not be found by simulation or by conventional model checkers before reaching an upper bound or state-space explosion.

Keywords: Model Checking, Actor Model, Guided Search, Heuristic, Deadlock.

Stýrð leit að sjálfheldum í gerandabundnum hugbúnaðarlíkönum

Steinar Hugi Sigurðarson

Júní 2011

Útdráttur

Árangur líkanaprófana liggur í hæfni þeirra til að sýna fram á villur í hönnun hugbúnaðar og samskiptastaðla. Jafnvel lítil gagnverkandi kerfi geta haft flókna hegðun. Sökum þess að vinnsluminni tölvunnar er takmarkað geta slík kerfi haft stöðurými sem er stærra en það sem mögulegt er að prófa. Þess vegna er ekki hægt að sannreyna réttmæti kerfisins með þeim hætti. Líkanaprófun kemur að mestu gagni þegar hún sýnir fram á áður óþekkta villu. Stýrðar leitaraðferðir nota brjóstvitsaðferðir til þess að stýra leitinni á hagkvæmari hátt en óstýrðar aðferðir gera. Almennt veita brjóstvitsaðferðir upplýsingar um fjarlægðina frá ákveðinni stöðu í þá markstöðu sem er næst henni. Við kynnum sjö brjóstvitsaðferðir sem geta stýrt leit, í gegnum leitartré gerandabundinna hugbúnaðarlíkana, að stöðu þar sem kerfið er í sjálfheldu. Leitir sem eru stýrðar af þessum aðferðum finna sjálfheldur fyrir sum líkön á hagkvæmari hátt en óstýrðar leitir. Með A* leit tryggjum við að ef lausn finnst þá er engin styttri lausn til í leitartrénu. Aðferðirnar eru útfærðar í hugbúnaði sem framkvæmir líkanaprófun fyrir sjálfheldur í gerandabundnum líkönum. Markmiðið er að finna erfiðar villur sem koma sjaldan upp og myndu hugsanlega ekki finnast með hermingu eða hefðbundnum líkana-prófunum áður en leitin nær efri mörkum vinnsluminnis.

Lykilorð: Líkanaprófun, gerandabundið, stýrð leit, brjóstvitsaðferð, sjálfhelda.

Acknowledgements

I would like to thank my supervisors, Dr. Marjan Sirjani and Dr. Yngvi Björnsson, for guiding my way through the past year which has truly been remarkable. It has been a privilege working with such eminent and enthusiastic researchers. I greatly appreciate their patience and thank them for always being prepared to assist and share their knowledge and experience with me.

My dear friend and study-partner, Árni Hermann Reynisson, has been invaluable for the past five years at the Reykjavík University. I feel privileged for having been able to share thoughts and ideas with such talented individual. I thank him for always being interested in any problems which needed solving and his unconditional commitment to helping his friends. He is truly a wizard and shall be referred to by his well deserved nickname, Gandalf, from here on out. I thank all of my friends at the university who have been a great support throughout my studies, both directly and indirectly.

My sincere and greatest thanks go to my best friend and partner, María Sif Sigurðardóttir, for her unprecedented patience and support.

For proofreading this thesis, I thank Guðmundur Jósepsson and, my sister, Sólveig Hrönn Sigurðardóttir. Special thanks go to my dear niece, Ísabella Sól Gunnarsdóttir, for preventing a proof copy of this thesis from being used a score-board for the Eurovision Song Contest.

I thank the The Icelandic Research Fund for supporting this research.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

This thesis is a study on the use of formal methods for finding deadlocks in actor-based models. We have developed heuristics for guiding search algorithms more efficiently to deadlock states and present experimental results using pure heuristic and A* search. For the research, we extended Modere (Jaghoori, Movaghar, & Sirjani, 2006), a model checker for the actor-based language Rebeca (Sirjani, De Boer, & Movaghar, 2005), to execute our experiments. This extension of Modere is limited to the deadlock-freedom property.

Society is increasingly reliant on IT (Information Technology) systems in our daily lives, even though we are not always aware of it when we browse the Internet, check the weather forecast on our smart phones, make phone calls and when we watch television. All of these tasks rely both on embedded systems and remote servers, providing the user with the desired service. The requirement for correctness is increasing and, although not life-threatening in these cases, errors can be very annoying and expensive for the service provider or the client. For such systems, money is usually the main concern. When your new mobile phone does not behave as expected, reacting incorrectly upon your input, you would presumably return it. These errors are inconvenient for the customer and can, on a large scale, cause considerable financial damage for the manufacturer. History has a few large-scale examples. In 1994, a bug was discovered in Intel's P5 Pentium floating point unit, causing the processors to produce incorrect results for division in extremely rare cases (*FDIV Replacement Program: Statistical Analysis of Floating Point Flaw*, 2004). The bug left Intel with a 475 million US dollars charge after replacing faulty processors, and a severely damaged image.

Sometimes, safety is the main concern. Digital flight-control applications such as autopilot and fly-by-wire systems, control software for nuclear power plants and traffic

control systems are safety-critical and even a minor defect can be catastrophic. On June 4, 1996, the European Space Agency launched the first Ariane 5 rocket. Only 37 seconds into the flight it veered of its flight path and self-destructed, incurring a loss of over 370 million US dollars (Dowson, 1997). The cause was an unhandled exception during conversion of 64-bit floating point to 16-bit signed integer value when the size of the floating point value was greater than could be represented by the signed integer. Since then it has become one of the most famous software bugs in history. Although no humans were injured in this case, errors can be life threatening. Due to a race condition in software controlling the Therac–25 radio therapy machine between 1985 and 1987, three cancer patients died as a result of receiving approximately 100 times the intended dose of radiation and another three were seriously injured (Leveson, 1993).

Testing effort grows exponentially with complexity and even small reactive systems can exhibit very complex behavior. Concurrency and non-determinism, as commonly seen in reactive systems, are hard to cover with standard testing techniques. Formal methods provide more effective verification techniques for such systems. Their aim is to assert that the system offers the described behavior and is correct with respect to certain properties. Investigations have shown that such methods could have revealed the errors in the systems described, Pentium FPU, Ariane 5 and Therac–25, prior to deployment and thus eliminating their impact.

An investigation report by the FAA and NASA on the use of formal methods concludes (Rushby, 1995):

> Formal methods should be part of the education of every computer scientist and software engineer, just as the appropriate branch of applied mathematics is a necessary part of the education of all other engineers.

1.1 Model Checking

Model checking is a formal verification technique developed to automatically validate properties of either hardware or software systems by exploring every reachable state in the system. Difficult errors, affecting the system only under rare circumstances, often go unnoticed through simulation and traditional testing methods. A model checker will explore these states and thus might reveal the error before it causes damage. In fact, model checking has been successfully applied to several systems, exposing previously unknown flaws in some cases. For example, five previously unknown errors were

discovered in the controller of the Deep Space 1 spacecraft, one of which was a major design flaw (Havelund, Lowry, & Penix, 2001). Other similar examples exist.

However, this technique comes with its own set of problems. Even relatively simple reactive systems may have wildly large state-spaces, many orders of magnitude larger than the memory of conventional computers can store. This is referred to as the state-space explosion problem. Several effective techniques have been developed to combat this problem, while still models of real-world systems may be too large.

1.2 Contribution

This thesis is a research on model checking of actor-based systems using informed search algorithms. The search is guided by heuristics, aiming to find a deadlock state, if such a state exists in the model, before exhausting the computer's memory. Moreover, we wish to find the shortest, or a relatively short, counter-example reproducing the error found.

The objective of this study is the following:

1. Develop methods for finding deadlock errors in actor-based models more efficiently than with conventional search methods.
2. Develop methods for finding the shortest possible counter-example (an optimal solution), for such errors, more efficiently than with conventional search methods.

The contributions of the thesis are seven heuristic strategies for guiding search algorithms through the state-space of actor-based models towards deadlock states. Furthermore, we present Guided-Modere, a model checker for the Rebeca language (Sirjani et al., 2005). Guided-Modere is an extension of Modere (Jaghoori et al., 2006), the standard model checker for Rebeca, and is designed as a flexible research framework with interchangeable search algorithms and heuristics. The implementation includes two of the most commonly used informed search algorithms in the field of artificial intelligence and the contributed heuristics. Rebeca is an actor-based modeling language with a formal foundation (Sirjani, Movaghar, & Shali, 2004).

Rebeca is event-driven and is designed for the verification of distributed concurrent systems. A Rebeca model consists of a finite number of rebecs (actors) which communicate via asynchronous message passing. In general, such models are not meant to terminate and thus do not contain terminal states. A deadlock in such a system occurs when the complete system is in a terminal state while at least one component

is in a non-terminal state. Our heuristics provide information based on the message queues of the rebecs in the model with the intention of guiding the search towards a state in which none of the rebecs have an unprocessed message. Such as state will have no events driving the system and is thus a deadlock state.

We present the following heuristics:

1. **Queue Size.** Uses the combined number of messages of rebecs in the system.
2. **Current Queue.** Relies on the number of messages in the queue of the rebec executed in a given state.
3. **Empty Queue.** Counts the number of enabled rebecs, preferring states in which the fewest rebecs have a message to process.
4. **Reductive Queue.** This heuristic detects reduction or increase of total messages in the system and rewards states causing reduction.
5. **Reductive Queue with Memory.** Rewards states reducing the number of messages with additional discount for the rebecs responsible for causing the reductions.
6. **Queue Difference.** Prefers a state sending the fewest messages.
7. **Queue Difference with Memory.** Prefers states sending the fewest messages with additional discount for the rebecs responsible for causing reductions.

Experimental results show significant reduction of expansions performed before finding a deadlock state for some models and the guided searches often return shorter counter-examples than depth-first search.

1.3 Overview of the Thesis

The thesis is divided into chapters as follows. Chapter 2 introduces the basic concepts of actor-based models, model checking and state-space search, describing both uninformed and informed search methods. Chapter 3 describes Guided-Modere. The heuristic strategies are defined and discussed in detail, followed by an overview of the implementation. In Chapter 4 we present experimental results for the contributed heuristics. Three case studies with results for the key search algorithms are discussed in-depth. Chapter 5 briefly discusses the related work. Chapter 6 presents conclusions and perspectives followed by suggestions about future work. An overview of terminology is provided in Appendix A. Appendix B contains extended results with additional searches and models.

Chapter 2

Background

2.1 Models of systems

Models of systems describe the behavior of systems in an accurate and unambiguous way (Baier, Katoen, & Others, 2008). They are described using a modeling or a programming language. Finite concurrent models can be presented as transition systems with a finite set of *states* and *transitions*, commonly described as a *finite-state automaton* or a *directed graph*. In explicit-state model checking, a state is identified by the current values of variables, previously executed statements and other relevant information about the system. The transitions represent actions which transform the system from one state to another.

2.2 Model Checking

The process of verifying finite-state reactive systems with respect to a property is referred to as *model checking*. The technique was originally developed by Clarke and Emerson in 1981 (E. Clarke & Emerson, 1982) and Queille and Sifakis in 1982 (Queille & Sifakis, 1982). Specifications for the systems are expressed using temporal logic. Typically, the model's state-space is explored in a brute-force manner where each reachable state is considered. That way it can be proven whether a given model satisfies or violates a property with absolute certainty. Errors which remain undiscovered by simulation and testing could potentially be revealed by model checking (Baier et al., 2008; E. Clarke, 1997).

$$(((\mathrm{F}\ p) \wedge (\mathrm{G}\ q)) \rightarrow (p\ \mathrm{U}\ r)).$$

Figure 2.1: Example LTL formula. Informally, it says p being finally true and q globally true infers that p will remain true until r becomes true.

Typical properties checked with model checking are deadlock freedom, invariants and request-response. As with the models, properties are described in a precise and accurate manner. This is usually done using a property specification language such as *temporal logic*. Temporal logic is an extension of traditional propositional logic with added operators which refer to the system's behavior from one state to another. Two commonly used languages are *LTL* and *CTL* (Pnueli, 1977; Vardi & Wolper, 1984; E. Clarke & Emerson, 1982). They allow specifications of properties regarding various conditions such as functional correctness, safety, liveness, reachability and fairness properties (E. M. Clarke, Emerson, & Sistla, 1986). An example LTL property is shown in Figure 2.1.

A successful run of a traditional model checker on a model for a specific property can have three outcomes: the property is valid, invalid or the model checker has exhausted the computer's memory. If the property was validated the model checker was able to either explore the entire state-space or find a path such that it could be concluded that the model satisfies the given property. On the other hand, if the outcome was invalid, the model checker has reached a state in which the provided property is violated. Ideally, the model checker will return some form of a counter-example showing a set of reproducible actions which will result in the violation of the property. There can be various reasons for a property violation, such as inconsistency between the model and the system design, error in the model, property specification error or a design error. In the case when execution is terminated prematurely because of the memory being exhausted, no information about the validity of the model can be derived.

Fairness

Fairness is an important aspect of reactive systems. For a concurrent system with more than one process we may wish to rule out infinite sequences that are considered unrealistic. For instance, we can consider only execution sequences where each process is executed infinitely often. This is sometimes necessary to establish liveness properties. An example of an unfair path is a path where only one process is executed in a system

consisting of two or more processes which are enabled infinitely often. This type of fairness is also known as *process fairness* (E. M. Clarke et al., 1986).

A *fair path* is a path which fulfills fairness constraints. These are the three most commonly used types of fairness (Baier et al., 2008):

Unconditional fairness. Every process is allowed to execute infinitely often.

Strong Fairness. Every process that is enabled infinitely often is allowed to execute infinitely often.

Weak fairness. Every process that is continuously enabled is allowed to execute infinitely often.

We can speak of certain types of fairness with respect to specific parts of a system, such as entering a critical section.

Deadlocks

Many sequential programs have terminal states and are allowed or expected to terminate. For concurrent systems, this is typically not the case. Such systems are usually not expected to halt and terminal states represent a design error and are thus not desired. If the complete system is in a terminal state with at least one component in a non-terminal state a deadlock has occurred (Baier et al., 2008). A typical deadlock scenario is where every component is waiting for another and the system cannot progress. When two cars approach opposite sides of a single-lane bridge, one must pass first. If both are waiting for the other to pass they will wait forever, even though both wish to continue their travel. Thus, they are in a deadlock configuration.

The State-space explosion problem

Traditional model checkers consider every reachable state in the model to prove that it satisfies a particular property. A major disadvantage of model checking is how poorly it scales. The state-space for a model can grow exponentially. For a complete explicit-state model checking, every state has to be examined (Edelkamp, Schuppan, Bosnacki, Wijs, & Aljazzar, 2009). A model with a branching factor of 10 has 10^5 states for the first 5 levels. General state-of-the-art explicit-state model checkers are able to handle up to 10^9 states (Baier et al., 2008). One can, however, easily think of models exceeding this limit. If we consider the game of chess, for example, a typical position has about

30 legal moves. Due to the 50 moves rule [1] we know that the game will eventually end and thus the state-space is finite. Each move has about 10^3 possibilities. A typical game will last about 40 moves until the resignation of either player or a draw. Even if we assume that we can predict when a party will resign, which we cannot, the state-space has reached a size of 10^{120}. If a model checker, which can handle one state every microsecond, could be created it would still require 10^{90} years to play out all possibilities for the first move (Shannon, 1950).

For our game of chess we would probably quickly exhaust the computer's memory. This is referred to as the *state-space explosion* problem and poses a major challenge to state-space search algorithms and model checking. Even a relatively simple reactive system can have a wildly large state-space.

A number of techniques have been introduced to attack this problem:

Symbolic model checking is a method that represents the state-space *symbolically* rather than explicitly. It uses *Binary Decision Diagrams* to represent relations and formulas (Burch, Clarke, McMillan, Dill, & Hwang, 1986). This approach has been used to verify systems with up to 10^{476} states (Lind-Nielsen, 1999).

Partial order reduction is an approach which exploits the commutativity of concurrently executed transitions by analyzing the independence of actions in order to reduce the number of orderings that need to be analyzed for the verification of a specific property (Peled, 1994).

Model Abstraction is another approach which automatically constructs a reduced abstract model, sufficient for a specific property. This approach has reportedly been used to successfully model check a system of 10^{1030} states (E. M. Clarke, Grumberg, & Long, 1994).

Directed Model Checking is a bug-hunting technique where selection from the enumerated successor states is prioritized in order to find short counter-examples quickly. Model checking algorithms exploit the specification of properties to lead the search towards their falsification. This approach is driven by the success of directed state-space exploration in the field of artificial intelligence and is among the key technologies to overcome the state-space explosion problem in model checking (Edelkamp et al., 2009).

Quantitative Model Checking returns quantitative information on the model. Probabilistic model checking is an example of this approach, using approximations

[1] A player can claim a draw if no capture has been made and no pawn has been moved in the last fifty consecutive moves.

such as Monte Carlo to decide whether a property is satisfied within a certain probability (Grosu & Smolka, n.d.). Such a model checker for Rebeca is proposed in (Behjati, Sirjani, & Nili Ahmadabadi, 2010).

Each approach has its strength in particular applications. For the remainder of this thesis we will primarily discuss directed model checking.

2.3 The Actor Model

In the actor model *actors* are treated as the universal primitives of concurrency. Instead of threads, the actor model uses objects as units of distribution and concurrency which provides a simple and natural concurrency model. Actors are self-contained and communicate through asynchronous message passing. Actors can be created dynamically and the topology of the system changes dynamically (Hewitt, 1977; Agha, Mason, Smith, & Talcott, 1997; Sirjani et al., 2004).

Similar to pure object-oriented languages, everything is an actor in the actor model. In a response to a message an actor has received it can send a finite number of messages to actors it knows, create a finite number of new actors and designate the behavior for the next message it will process. More importantly, all of the actors in the system can perform these actions concurrently and no assumptions are made regarding the order in which they occur.

While the actor model is a primitive model of computation it can easily express a wide range of concurrent systems and provides a natural extension of functional programming and object abstraction.

2.4 The Rebeca Language

Rebeca is an event-driven modeling language, based on the actor model, with a formal foundation for modeling and verification of concurrent and distributed systems (Sirjani et al., 2004). It consists of reactive objects which communicate via asynchronous message passing. *Reactive classes* are templates defined by the programmers from which they can instantiate any number of reactive objects called *rebecs* (Sirjani et al., 2004). In an object-oriented environment, this would correspond to the relationship between classes and objects. A reactive object consists of the following components:

Message Queue. Each message sent to a rebec is pushed to the end of the queue. Although unbounded in the actor model, an upper limit is defined for model checking.

Known Rebecs. A list of other rebecs a reactive object knows and is able to send messages to.

State variables. Variables which each rebec can read or modify when processing a message.

Message servers. The behavior of rebecs is described by their message servers. A rebec accepts messages from any rebec in the system to all its servers. Every rebec must have an *initial* message server which corresponds to a constructor of an object. They accept parameters and support non-deterministic assignments. Reactive classes have a defined upper bound for the message queue size of modeling purposes.

A Rebeca model consists of a set of reactive classes, rebecs and their initial configuration. An example model, implementing Dijkstra's self-stabilizing token ring for three nodes (Dijkstra, 1974), is shown in Listing 2.1. Rebecs of the reactive class *Node* have message queues with an upper bound of 5 messages.

```
reactiveclass Node(5) {
   knownrebecs { Node child; }
   statevars {
      boolean isLeader;
      int value;
   }
   msgsrv initial(boolean lead) {
      value = ?(0, 1, 2, 3);
      isLeader = lead;
      child.update(value);
   }
   msgsrv update(int parentValue) {
      if(isLeader && value == parentValue) {
         value = ((value + 1) % 4);
         child.update(value);
      } else if(!isLeader && value != parentValue) {
         value = parentValue;
         child.update(value);
} } }
```

```
main {
   Node n0(n2):(true);
   Node n1(n0):(false);
   Node n2(n1):(false);
}
```

Listing 2.1: Example Rebeca model: Dijkstra's self-stabilizing token ring.

Rebeca is supported by front-end tools which translate Rebeca models into modeling languages of existing model checkers and a tool which performs direct model checking, described in the following section. Rebeca has been extended to support synchronous message passing to enable modeling of locally synchronous and globally asynchronous systems (Sirjani et al., 2005). This study will use the original version, without the extension.

Modere

Modere (Jaghoori et al., 2006) is an explicit-state model checker for the Rebeca language. It is inspired by SPIN (Holzmann, 1997), the model checker for Promela, and uses nested depth-first search (Holzmann, Peled, & Yannakakis, 1996) for automata-theoretic model checking. It stores local states of each rebec separately using a "never release" memory management strategy. It is highly optimized regarding memory and state-space exploration, implementing partial-order reduction to combat the state-space explosion problem. It supports verification of temporal logic properties written in either LTL, which are automatically translated to Büchi automata, or CTL. Modere consists of a Java compiler which generates intermediate C++ code. Compiling the C++ code produces an executable file performing the model checking of the system. If a property violation is found a counter-example is returned in XML format. Modere can be used either through a command-line interface or an Eclipse plug-in with GUI. Different aspects of Modere are discussed throughout the remainder of the thesis when compared with Guided-Modere.

2.5 State-Space Search

A state-space is a set of states. State-space search is the process of finding a state in a state-space satisfying a goal condition. The set of all states is not initially known, but

only a single initial state. Other states, the successors of known states, are generated as needed. Such problems may be formalized by defining its four components (Russell, Norvig, Canny, Malik, & Edwards, 1995):

Initial state is the start state. In Rebeca, this would be the state where all rebecs are ready to be initialized. From each state a *successor function* provides the reachable successor states.

Actions available in a particular state. From each state in a Rebeca model the set of actions consists of the currently enabled rebecs, i.e. the ones which have a message to be processed.

Goal test which determines if a particular state is a goal state. This can be a set of explicitly defined states or an abstract property which needs to be evaluated with respect to the current state in the search.

Path cost function which assigns a numerical value to each path. It should reflect the performance measure of the search.

Each node represents a single state. Search algorithms start from the tree's *root node*, representing the *initial state*, with an empty *fringe*. When the algorithm expands a state s, it removes s from the fringe and adds its successors.

If actions are reversible, the search may enter an infinite loop and the search tree becomes infinite. For some search trees, this problem will never come up while with others it is unavoidable. Thus, a solvable problem can become unsolvable. We may wish to prevent this by allowing the search to expand each state only once. By pruning repeated states, generating only the portion of the tree which spans the state-space *graph*, we can reduce the search tree to a finite number of nodes. The only solution is to keep the expanded states in memory. We can use a data structure, from here on referred to as the *closed list*, which stores all the nodes of the tree which have been expanded. We store the fringe, nodes we have not expanded yet, in the same fashion using a data structure referred to as the *open list*. Worst case time and space requirements are proportional to the size of the state-space, $O(b^d)$ where b is the branching factor and d is the height of the tree. However, these requirements may be much smaller (Russell et al., 1995). For the following sections, nodes in the search tree are illustrated as shown in Figure 2.2. Nodes on the fringe, or the open list, are identified by a regular solid border, the current node by a thicker border and closed nodes, that have already been expanded, by a dashed border. Unknown states, which are yet to be discovered, are faded gray.

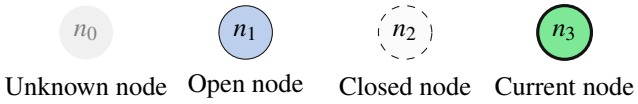


Figure 2.2: Notation for nodes used to demonstrate search algorithms.

2.6 Uninformed search

Uninformed search algorithms, also known as blind search algorithms, explore the search tree in a brute-force manner. They do not have additional information on the states beyond what is provided in the problem description. They only know how to generate successors and determine whether or not a particular state is a goal state (Russell et al., 1995).

Depth-First Search

Depth-first search is an uninformed search algorithm and is one of the most commonly used for model checking. It always expands the deepest node in the fringe, progressing deeper into the search tree until the nodes have no successor. Once such a node has been expanded it is removed from the fringe and the search retracts up to the previous level where the next sibling node is expanded. This is repeated until a goal state is reached or there are no states left on the fringe, in which case the property is satisfied. An example execution of depth-first search is illustrated in Figure 2.3.

Depth-first search is not complete as it may enter infinite loops and proceed down infinite paths. By using a closed list, infinite loops are eliminated and the search algorithm is complete for finite state-spaces.

Depth-first search is often implemented using a stack data structure for the fringe, obeying the LIFO (last-in, first-out) principle.

Nested depth-first search (NDFS) (Holzmann et al., 1996) is a variation of DFS with cycle detection and has been successfully implemented in several model checkers (Courcoubetis, Vardi, Wolper, & Yannakakis, 1992; Holzmann, 1997; Jaghoori et al., 2006). First, it executes an outer depth-first search until it reaches a state which violates the temporal property being verified. A second nested search, inner depth-first search, continues from that state searching for backwards edges to matching a state, creating a cycle. If the inner search does not find a satisfying backward edge, the outer search will continue from where it left off. Cycle detection is required for the verification of properties when the violating sequence must contain the violating state infinitely of-

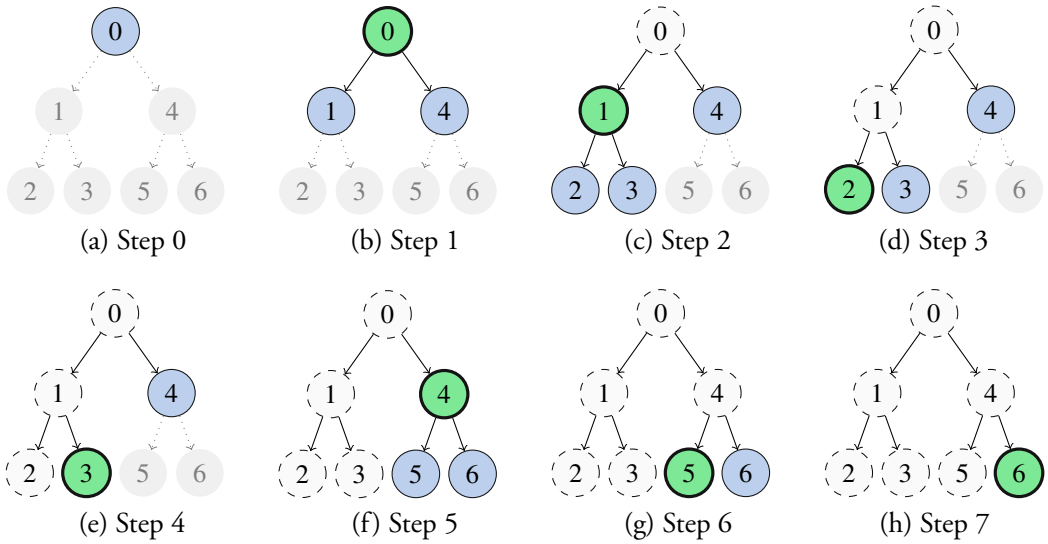


Figure 2.3: Depth-first search, example exploration. Order of selection is shown inside the nodes.

ten. In this thesis we focus on deadlocks only, which are reached through finite paths, and thus do not address nested search and cycle detection further.

Breadth-First Search

Breadth-first search is another uninformed search algorithm commonly used for model checking. Instead of going for the depth it expands all nodes on the fringe at a given depth before proceeding to nodes at the next level. It will not expand a state at depth $n + 1$ unless it has explored all the states at depth n.

It is usually implemented in the same manner as depth-first search except instead of using a stack for storing the fringe, new states are put in a FIFO (first-in, first-out) queue. All newly generated successor states are added to the end of the queue while the next state to expand is removed from the front.

Breadth-first search is complete for a finite branching factor. If the search tree contains a goal state, it will be found unless the search runs out of memory or reaches an upper bound.

One of the key benefits of using breadth-first search for model checking is that if a counter-example is found, it is guaranteed that no shorter counter-example exists and the solution is optimal. An example execution of the breadth-first search algorithm is illustrated in Figure 2.4.

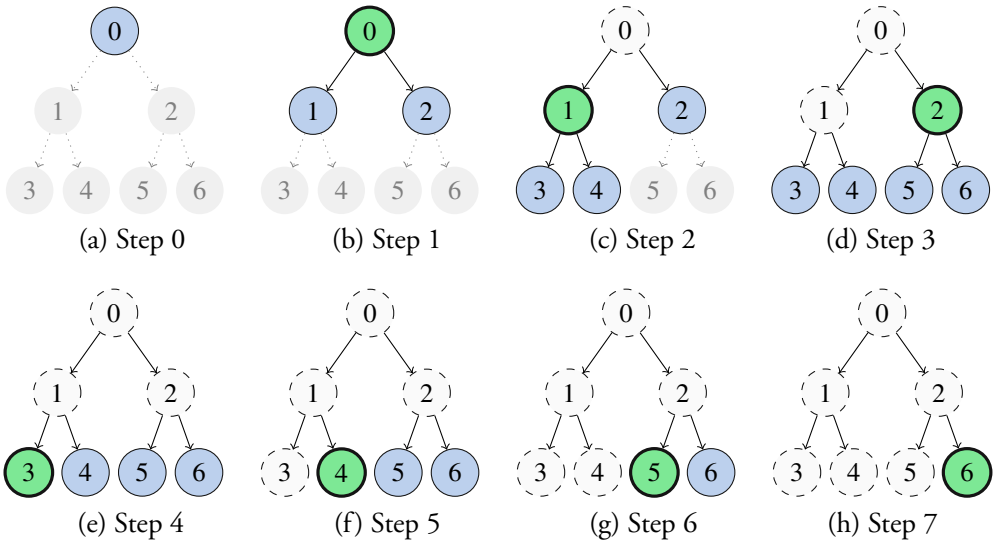


Figure 2.4: Breadth-first search, example exploration. Order of selection is shown inside the nodes.

Several variations of the algorithm have been implemented with model checking in mind, such as Parallel BFS (Barnat, Brim, & Chaloupka, 2003) and Distributed BFS (Barnat & Černá, 2006).

2.7 Informed Search

In practice, finding a bug with a model checker is more useful than proving a property (Yang & Dill, 1998). Informed search algorithms use heuristic strategies with problem-specific knowledge to find solutions more efficiently than uninformed algorithms. Generally, such heuristics estimate the distance from a given state to a goal state.

We consider two of the most commonly used informed search algorithms: pure heuristic search and A* search. Both belong to a family of *best-first search* algorithms. They use an *evaluation function*, denoted by $f(n)$, to prioritize the fringe, that is, to decide which node to expand next. Estimated cost of traversing from a node to a goal state is provided by a heuristic function, denoted by $h(n)$. The two algorithms have different evaluation functions, $f(n)$, and therefore different behavior, described in the following sections.

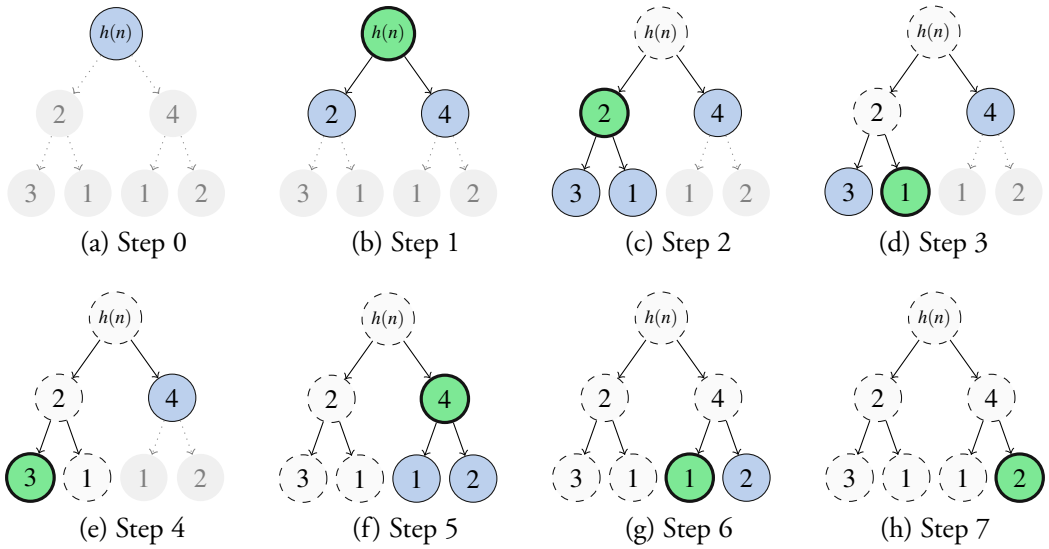
Pure Heuristic

Pure heuristic search, also known as *Greedy best-first* search, always expands the node that is the closest to the goal first, regardless of its distance from the initial state (Russell et al., 1995). The aim is to find a solution quickly even though it may not be an optimal one. The algorithm evaluates nodes using only the heuristic function. That is,

$$f(n) = h(n).$$

Pure heuristic search is similar to depth-first search in the sense that in practice it tends to follow the same path from the initial state to the goal but will retract as it hits a dead-end. Like depth-first search, it is not optimal, and the search may run into infinite loops and is thus not complete. However, by adding a closed list to the algorithm such loops are eliminated and completeness guaranteed for finite state-spaces.

Worst-case space and time complexity is $O(b^m)$ where b is the branching factor of the tree and m is the maximum depth of the search tree. In practice, the number of explored states may be much smaller with a good heuristic. Demonstration of pure heuristic search on an example tree is shown in Figure 2.5.



(a) Step 0 (b) Step 1 (c) Step 2 (d) Step 3

(e) Step 4 (f) Step 5 (g) Step 6 (h) Step 7

Figure 2.5: Pure heuristic search, example exploration. Heuristic values of states (the estimated distance to the nearest goal state) are shown inside the nodes.

A* Search

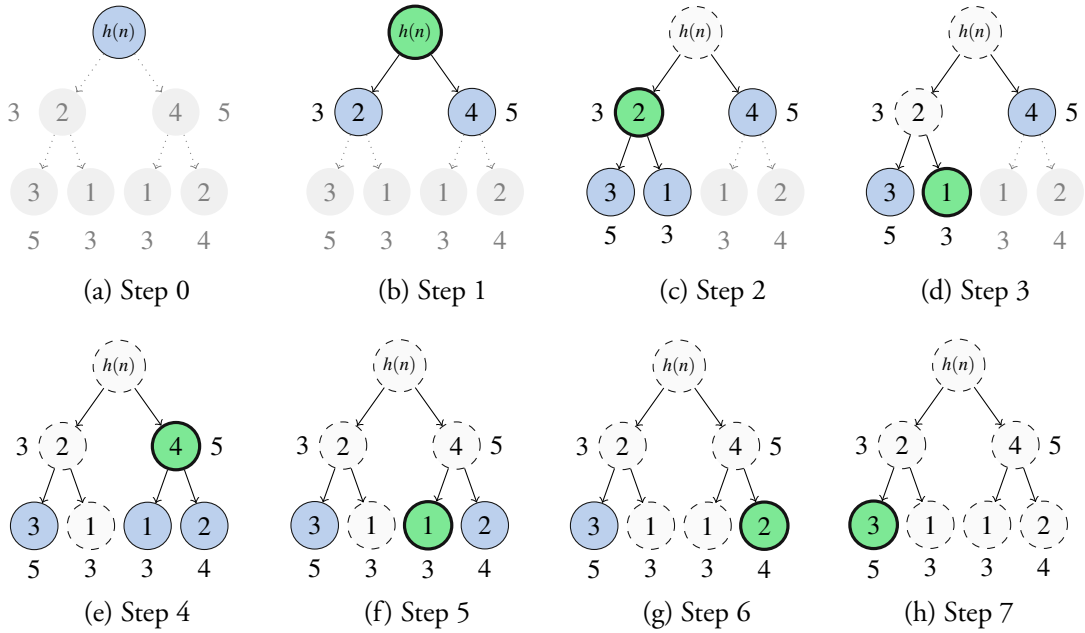
A* search ("A-star search") is the most popular best-first search algorithm. It combines the cost of reaching node n, the path cost function $g(n)$, with the estimated distance to the nearest goal state from n, the heuristic function $h(n)$ (Russell et al., 1995). That is, the evaluation function is the estimated length of a shortest path from the initial state to a goal state going through node n. Thus,

$$f(n) = g(n) + h(n).$$

A* is complete, that is, if a solution exists it will eventually be found, unless the algorithm runs out of memory and state-space explosion occurs. Furthermore, if the heuristic function is admissible, that is, it never overestimates the true distance to the goal, we are guaranteed to find the shortest solution first.

Weighted A* Search

Experience has shown that for some problems A* search spends a great amount of time exploring paths with insignificant difference in cost. The requirement of optimality causes the algorithm to spend time choosing between candidates with roughly the same cost. We may wish to relax the requirement of optimality in exchange for a quicker solution. We can do that by adding a weight factor to the two functions, the cost



(a) Step 0 (b) Step 1 (c) Step 2 (d) Step 3

(e) Step 4 (f) Step 5 (g) Step 6 (h) Step 7

Figure 2.6: A* search, example exploration. Heuristic values (the estimated distance to the nearest goal state) are shown inside the nodes and the evaluation function values (the total path cost) outside of them.

function $g(n)$ and heuristic function $h(n)$. The effect of $g(n)$ in the evaluation function $f(n)$ is to add a breadth-first component to the search. In fact, if $h(n)$ is removed the resulting evaluation function $f(n) = g(n)$ is that of breadth-first search. By removing the $g(n)$ component the search is a pure heuristic search (Pearl, 1984).

Weighted A* search has the evaluation function

$$f(n) = (1 - w)\, g(n) + w\, h(n),$$

where $w \in [0, 1]$ is the weight factor. By varying w the desired mixture can be achieved to match the reliability attributed to the heuristic function being used. Higher value of w will put more responsibility on the heuristic and reduce the requirement of an optimal solution and vice versa. The factors 0, $\frac{1}{2}$ and 1 would correspond to uniform-cost, A* search and pure heuristic search, respectively. In this study, the algorithm is used only to relax the requirement of optimality. Thus, our experiments will be confined to $w \in [\frac{1}{2}, 1]$.

Heuristic Functions

The efficiency of informed search is largely based on the quality of the heuristic function used. Traditionally, the returned evaluation of a heuristic function for a particular state is the estimated distance to the nearest goal state. This estimate is denoted by $h(n)$.

$$h(n) = \text{estimated distance to the nearest goal state.}$$

A heuristic strategy is *admissible* (or *optimistic*) if it never overestimates the distance from a given state to the nearest goal state. This is important if we want to guarantee that the first solution found is an optimal path, i.e. that no shorter solution exists. Model checkers using DFS search tend to return longer paths which makes it more difficult for the user to identify the error. In practice, a path relatively short, compared to an optimal path, will usually suffice. However, non-admissible heuristics can be useful and are widely used (Dr, 2009). *Consistent*, or *monotone*, heuristics are monotonically non-decreasing along the shortest path to a goal state. The estimated distance to a goal state will never exceed the estimated distance of a successor state with the added cost of traversing between them. Formally, heuristic function $h(n)$ is consistent if and only if for every node n and every successor q of n,

$$h(n) \leq c(n, q) + h(q),$$

where $c(n, q)$ is the cost of traversing from n to q. A consistent heuristic function is always admissible, but an admissible strategy can be inconsistent (Pearl, 1984).

When searching with inconsistent but admissible heuristics we must be careful when discarding previously expanded states if we wish to guarantee optimality. The node on the closed list may have a greater estimated total distance to a goal state than the new state, in which case we must re-open it. In theory, exponential increase in the number of expanded nodes may happen. However, this approach works well in practice (Edelkamp et al., 2009).

Although the accuracy of a heuristic is important, one must also consider how efficient it is computationally. After all, we could develop the perfect heuristic by performing a breadth-first search and return the measured distance to the nearest goal state. However, there is no gain in using such a heuristic. In general, we consider heuristic h_1 better than h_2 if its *effective branching factor* is less than that of h_2, it does not overestimate and the computation time is not too large (Russell et al., 1995).

An algorithm A_2^* is said to *largely dominate* A_1^* if every node expanded by A_2^* is also expanded by A_1^* except, perhaps, some nodes for which $h_1(n) = h_2(n) = C^* - g^*(n)$, where C^* is the cost of an optimal solution and $g^*(n)$ is cost of an optimal path from the initial state to node n. In other words, A* largely dominates its rivals if it performs fewer or an equal number of expansions except, possibly, for some paths where both heuristics return the cost of a shortest path (Pearl, 1984).

2.8 Directed Model Checking

Directed model checking is one of the key techniques developed to overcome the state-space explosion problem. Algorithms reorder the states to be expanded so that states more likely to violate the required property are expanded first. An evaluation function estimates the cost of the shortest possible path from a state to a violating state. Both of the informed search algorithms discussed have been used successfully for such model checkers (Edelkamp, Lafuente, & Leue, 2001; Groce & Visser, 2004; Hoffmann, Smaus, Rybalchenko, Kupferschmid, & Podelski, 2007; Edelkamp et al., 2009). They are discussed further under Related Work.

Chapter 3

Guided-Modere

Guided-Modere is an extension of Modere, the standard model checker for Rebeca, and was developed over the course of this research. The goal is to create a flexible research framework to experiment with different search algorithms and heuristics. Furthermore, we wish to study the efficiency of heuristic search for Rebeca models with respect to execution time, expanded nodes and length of counter-examples.

The implementation includes several variations of our heuristics, pure heuristic search algorithm, three A* search algorithms and three blind search algorithms. It is our platform for experimenting with, and measuring the performance of, different search algorithms and heuristics.

3.1 Heuristics

In this section, we describe the heuristic functions $h(n)$ used to inform search algorithms. Each strategy accepts a state as input and returns a non-negative integer which is the estimated distance to the nearest deadlock state.

A Rebeca model is expressed semantically as a labeled transition system $\mathcal{M} = \langle S, A, T, s_0 \rangle$ where S is the set of global states, A is the set of actions, $T \subseteq S \times A \times S$ is the set of transitions and s_0 is the set of initial states. A state in Rebeca is defined as the combination of the local states of all rebecs in the system, $s = \Pi_{j \in J} s_j$, where I is the set of rebecs and $s \in S$ is the global state. The local state s_j of a rebec r_j is identified by the values assigned to its local variables, v_j, and its message queue, m_j, including information about the sender, destination message server and parameters. We say that

rebec r_j is enabled when the number of messages in its queue, $|m_j|$, is greater than zero.

In general, a deadlock is a situation where two or more actions are waiting for each other finish. The actor model is event-driven and Rebeca only allows asynchronous message passing. Thus, the system cannot deadlock as long as there are events driving it. A rebec will never wait for a process to finish, gaining access to a mutual resource or a reply from another rebec. The system will run into a deadlock state if and only if there are no enabled rebecs, that is, $\sum_{j \in J} |m_j| = 0$, assuming the model has no terminal states. All of our heuristics exploit this fact and share the intention of driving the search towards states which are more likely to result in a deadlock.

We define and implement seven heuristics, presented in the following sections. All of the heuristics are admissible. In the literature, admissible heuristics are often assumed consistent, implying that consistency is desirable. Only two of the heuristics above are consistent: Queue Size and Empty Queue. Although generally considered worse for A* search, recent studies have shown that inconsistency can be beneficial. Inconsistent heuristics are able to escape regions of poor heuristic values before incurring significant cost (Zahavi, Felner, Schaeffer, & Sturtevant, 2007). Due to the relatively small values returned by our inconsistent heuristics, compared to the path cost, they are not expected to perform well with A* search as it will explore the state-space similar to breadth-first search. Additionally, due the reopening of states, A* searches using those heuristics may expand more nodes than breadth-first search. Thus, they are intended for pure heuristic search only.

The state of the system at node n in the search tree is referred to as n_{state} and the parent state of n_{state} as n_{parent}.

Queue Size

If we think of a reduction of a system where no new messages can be sent, only the messages rebecs have in their queues already will be processed. The number of actions required to drive a system to a deadlock state from a given state s is never less than the number of unprocessed messages in s so even with this reduction the system will not run into a deadlock state until they have all been processed. The *Queue Size* heuristic uses the total number of unprocessed messages in the system as the estimated distance to a goal state, preferring states with few messages to states with many. The Small

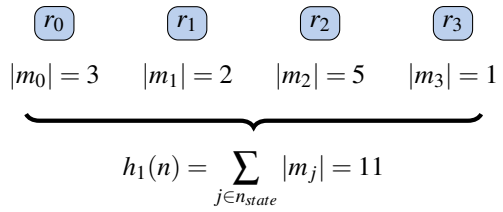


Figure 3.1: Queue Size heuristic: Example of the evaluation of a node.

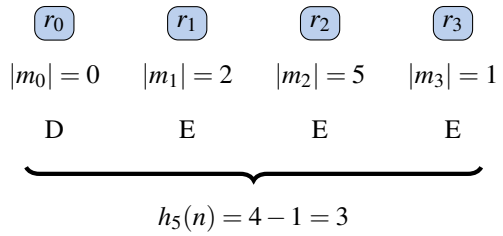


Figure 3.2: Empty Queue heuristic: Example of the evaluation of a node. Above the brace is a string representation of the state.

Queue heuristic is illustrated in Figure 3.1 and formally defined in Equation 3.1.

$$h(n) = \sum_{j \in n_{state}} |m_j|. \tag{3.1}$$

Between every node n and its successor node q exactly 1 message has been removed from the message queue of an enabled rebec. We know the cost of reaching q from n is $c(n,q) = g(q) - g(n) = 1$. Therefore, we can show that $h(n) \leq c(n,q) + h(q)$ holds for every n in the search tree. Since a goal state will have no messages, $h(n) = 0$ where n is a goal state. Given the above we know that $h(n)$ is *consistent* and, therefore, also *admissible* and will return an optimal solution when used with A* search.

This heuristic largely dominates all the others and will perform better with A* search.

Empty Queue

The *Empty Queue* heuristic counts the number of rebecs with no messages and assigns better heuristic values to states with higher values. This drives the search towards paths closer to a deadlock state and possibly with less branching factors, depending on the number of non-deterministic choices made by the enabled rebecs. This heuristic is identical to H_{ap} presented in (Edelkamp et al., 2001) for Promela models and the deadlock heuristic of PROVAT presented in (Lin, Chu, & Liu, 1988).

This strategy can be described as the Hamming distance between a state and a goal state for a property in which all rebecs are disabled. We can think of a state with three enabled rebecs as the string "*EEEDD*" where *E* means enabled and *D* disabled. Using the same notation we can easily define a deadlock state, "*DDDDD*" and see that the Hamming distance between the two strings, in this case 3, is equal to the value returned by the Empty Queue heuristic. This strategy is illustrated in Figure 3.2 and formally defined in Equation 3.2.

$$h_5 = |n_{state}| - \sum_{j \in n_{state}} \begin{matrix} 0 & if |m_j| > 0, \\ 1 & otherwise. \end{matrix} \tag{3.2}$$

Initial states will receive heuristic values equal to the number of rebecs in the system, since all of them are enabled. For every n and every successor q of n, we know n can have at most one more enabled rebec than q because of a transition. In other words, at most one more rebec has become disabled. Thus, this heuristic is consistent and will not require reopening of states when guiding an A* search.

Current Queue

The *branching factor* of a node n is affected by two attributes of the state:

1. The number of enabled rebecs in n_{state}.
2. The number of non-deterministic choices made in n_{state}.

We refer to the last executed rebec as the *current* rebec of a state. The *Current Queue* heuristic aims to "drain" rebecs by favoring nodes whose current rebec has the smallest message queue. The intention is to make the execution less fair, focusing on rebecs becoming disabled and thus potentially reducing the branching factor along the path. Potentially isolated parts of the system could become disabled, if the model has any. The behavior is illustrated in Figure 3.3 and formally defined in Equation 3.3.

$$h(n) = |m_{current}|. \tag{3.3}$$

In Equation 3.3, $m_{current}$ is the message queue of the most recently executed rebec in state n_{state}. A search path using this heuristic function should empty out rebecs with the smallest queues before proceeding to those with more messages which might reactivate the rebec again. Once a rebec has become disabled the node will receive a heuristic value equal to the second smallest queue (which has now become the smallest). Be-

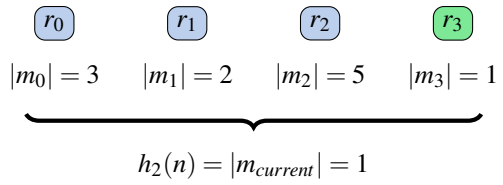


Figure 3.3: Current Queue heuristic: Example of the evaluation of a node. The rebec r_3 was executed last and is the current rebec.

tween such nodes the condition for consistent heuristics is broken. This heuristic never overestimates the distance to the nearest goal and is, therefore, admissible.

Reductive Queue

Essentially, message servers that do not send any messages can cause deadlocks in Rebeca models. The *Reductive Queue* heuristic utilizes this fact by comparing the total number of messages of a state and its parent. States are categorized as follows:

1. Number of messages has been reduced.
2. Number of messages has not changed.
3. Number of messages has increased.

States in each category receive the best, neutral and worst heuristic value, respectively. The initial states do not have a valid parent and are all assigned a value equal to the number of rebecs in the system.

$$
\begin{aligned}
p &= \sum_{k \in n_{parent}} |m_k| \\
q &= \sum_{j \in n_{state}} |m_j| \\
h(n) &= \begin{cases} \textit{best} & if\ q < p \\ \textit{neutral} & if\ q = p \\ \textit{worst} & otherwise \end{cases}
\end{aligned} \tag{3.4}
$$

The values chosen for *best*, *neutral* and *worst* in Equation 3.4 are 0, $\lfloor \frac{1}{2} \sum |m| \rfloor$ and $\sum |m|$, respectively. The heuristic is not consistent as the difference between resulting heuristic values of two consecutive states can be greater than the added cost (which is 1) if the two states are in distinct categories. Since the size of the message queue is the upper limit it will never return a value greater than the path cost to the nearest goal.

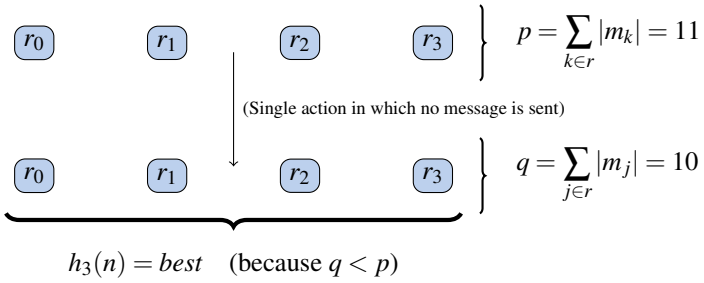


Figure 3.4: Reductive Queue heuristic: Example of the evaluation of a node.

Therefore, we know that the Reductive Queue heuristic is admissible. The heuristic is illustrated in Figure 3.4 and formally defined in Equation 3.4.

Reductive Queue with Memory

A deadlock error is often related to a single rebec in the system. In some cases, it will reduce the number of messages periodically, for example, in every other execution. The *Reductive Queue* heuristic strategy will only identify and favor a state immediately after performing such an execution. This variant of the heuristic keeps track of how often a rebec has reduced or increased the number of messages in the past. When no reduction or increase has taken place, the state will receive the *neutral* heuristic value, as before, but with a discount based on how likely executions of this rebec are to reduce the number of messages.

A state receiving maximum discount would get the heuristic value $\frac{1}{2}neutral$ which, in this case, is equal to *best*. However, that can only happen if a state reduces the number of messages on every execution and the discount would not apply. Should a rebec reduce number of messages on every other execution, the value of a non-reducing state would be $\frac{3}{4}neutral$. To maintain admissibility no penalty is given to rebecs which are likely to increase the size of the message queue. This heuristic is defined formally in Equation 3.5.

$$
\begin{aligned}
p &= \sum_{k \in n_{parent}} |m_k|, \\
q &= \sum_{j \in n_{state}} |m_j|, \\
h(n) &= \begin{cases} \textit{best} & if\ q < p, \\ \frac{1}{2}\textit{worst} + \frac{1}{2}(1 - \frac{reductions_r}{executions_r})\textit{worst} & if\ q = p, \\ \textit{neutral} & otherwise. \end{cases}
\end{aligned} \tag{3.5}
$$

The discount is meant to serve as a tie-breaker between two non-reducing and non-increasing states. For the same reason as *Reductive Queue*, this heuristic is not consistent.

Queue Difference

This heuristic strategy is identical to the *Reductive Queue* strategy except for the returned values. Instead of returning *best*, *neutral* and *worst* it relies on the number of messages created by the last action. This strategy will assign different values to states sending three or more messages, whereas the Reductive Queue heuristic will not. The heuristic is defined formally in Equation 3.6.

$$
h(n) = 1 + \sum_{j \in n_{states}} |m_j| - \sum_{k \in n_{parent}} |m_k|. \tag{3.6}
$$

This strategy will violate the requirement for consistency when $\sum_{j \in n_{state}} |m_j| > 1 + \sum_{k \in n_{successor}} |m_k|$. This will happen every time a message server sends more than one message. Thus, this heuristic is not consistent. As the path cost from a state to a goal state will never be less than the number of messages created, the strategy is admissible. The number of messages sent is usually far smaller than the total number of messages left to be processed. This causes the A* search to behave similar to breadth-first search and expand more nodes than better informed search algorithms. This behavior could be overcome by multiplying the heuristic value by a factor greater than 1, but doing so would sacrifice the admissibility. Thus, the heuristic strategy is not expected to return satisfying results with A* search but has good potentials with pure heuristic search and, perhaps, weighted A* search with a sufficiently small factor.

Initial states receive a value of 1, as if no difference took place. Should we count each initialization as a sent message, returning a heuristic value equal to the number of rebecs, chances are that only one rebec would ever be expanded at level 1 since the successor states would almost certainly have lower values.

Queue Difference with Memory

The *Queue Difference* strategy suffers from the same problem as *Reductive Queue* when it comes to states which have not affected the total number of messages. This heuristic is very similar to *Reductive Queue with Memory* except it keeps track of the difference in the number of messages instead of only incrementation and reduction. A rebec sending as many messages as it consumes will have a memory value of 0 while a rebec only consuming messages but sending none would have a memory value of 1. The heuristic is defined formally in Equation 3.7.

$$
\begin{aligned}
p &= \sum_{k \in n_{parent}} |m_k| \\
q &= \sum_{j \in n_{state}} |m_j| \\
h(n) &= \begin{cases} 1 + q - p - (1 - \frac{reductions_r}{executions_r}) & if\ q = p \\ 1 + q - p & otherwise \end{cases}
\end{aligned} \tag{3.7}
$$

When the difference in messages is 0, a state will receive a discount based on the memory value. The maximum discount has the same effect on the heuristic value as half a message. However, that can only happen when a rebec has a negative difference, i.e. -1, on every execution and, therefore, the discount would not apply. Thus, the discount range is equal to 0 to 0.5 messages.

As before, the discount is only used as a tie-breaker for equal states. The aim is to favor rebecs which have a history of decreasing the total number of messages when the model checker has two or more otherwise equally-valued states. To maintain admissibility, no penalty is given to rebecs having, on average a positive difference, and, of course, no discount either. For the same reasons as the Queue Difference heuristic, this one is not consistent.

As with the non-memory version, due to the extreme under-estimating of the heuristic, it is not expected to perform well with A* search but has good potential with pure heuristic search.

3.2 Combining Heuristics

Our heuristics provide information regarding different aspects of the message queue. In fact, we may wish to consider them as individual functions and combine them for improved results. We propose combining the two consistent heuristics with the inconsistent lower-value heuristics as follows:

1. **Queue Size and Empty Queue.** Both heuristics provide valuable estimates while their values can become significantly different for the same state. Balancing between those two could potentially be valuable for some models.
2. **Queue Size and Current Queue.** Combined, the two will favor states with a near-disabled rebec while reducing the total number of messages at the same time.
3. **Queue Size and Reductive Queue.** While the Queue Size strategy is a good estimate for the distance to the nearest goal state it will not distinguish between a path increasing the number of messages and one reducing them. By combining it with Reductive Queue we have added this information.
4. **Queue Size and Queue Difference.** Similarly, the Queue Difference will provide the Queue Size strategy with information on the number of sent messages.
5. **Empty Queue and Current Queue.** The Empty Queue heuristic aims to decrease the total number of enabled rebecs and the Current Queue focuses on how close the system is to having another disabled rebec. Thus, this pair could potentially provide more valuable information than either heuristic could individually.
6. **Empty Queue and Reductive Queue Size.** This combination will reward states with the fewest enabled rebecs and those reducing the total number of messages.
7. **Empty Queue and Queue Difference.** The Empty Queue lacks the detection of message server sending multiple messages. Queue Difference can provide that information.

The general formula for combined heuristics is their average with the possibility of altering the weight of either one. Thus,

$$h_{a+b}(n) = w\ h_a(n) + (1 - w)\ h_b(n),$$

where h_a and h_b are the heuristic functions to be combined and $w \in [0, 1]$ is the weight factor. The upper bound is the weighted average of the upper bounds of the heuristics. If both are admissible then the resulting heuristic is admissible as well and if both are consistent, the resulting heuristic is consistent as well.

In this study we only consider the combinations listed above with $w = \frac{1}{2}$. Exploring the potentials of other combinations and different factors between the heuristic is left to future research.

3.3 Inverted Heuristics for Guidance Towards Queue Overflow

As previously mentioned, the actors in the actor model have an unbounded queue. However, Modere has an upper bound defined by the model designer for modeling purposes (Jaghoori et al., 2006). Queue overflow occurs when a message is sent to a rebec with a full message queue. Here, we present a general approach to adapt our heuristics to guide the search towards such a state by inverting the heuristic with respect to its upper bound. Thus,

$$h_{inv}(n) = h_{max}(n) - h(n).$$

An inverted heuristic is denoted by [heuristic name]$_{inv}$.

By applying the *inverted heuristic*, the search will avoid deadlocks. Since we need only one of the rebecs to overflow, most of the heuristics are over-estimating. In fact, of the inverted heuristics only Current Queue$_{inv}$ is admissible. The other inversions use the messages queue of all of the rebecs and are thus evaluating states as if all of the queues need to overflow. Therefore, Queue Size$_{inv}$ is the more likely to provide good information than the others. Its definition is virtually identical to the channel overflow heuristic proposed in (Lin et al., 1988).

Since queue overflow is outside the scope of this study, the topic will not be discussed further. Experimental results for the inverted heuristics are listed in Appendix C.

3.4 Implementation

Standard Modere generates C++ code which implements Nested-DFS search (NDFS), with partial-order reduction, to traverse the state-space of the system. The program code is highly coupled to other parts of the code, which is presumably a result of optimization. Although performance is of great importance when it comes to model checkers, a researching developer will find it difficult to modify Modere to implement other algorithms for two reasons: Identifying and fully understanding the search related code is time-consuming and chances of introducing errors while modifying it are high.

In Guided-Modere, the entire search implementation was thus abstracted and separated from other parts of the code. Search algorithms implement the *template method pattern* (Johnson, Gamma, Helm, & Vlissides, 1995) which enables the developer to extend pre-existing search algorithms and re-implement only the parts that have different behaviors. Furthermore, the state-space search is encapsulated so that other parts of the model checker require no knowledge of the inner behavior of the search. Most of the commonly seen explicit state search algorithms share the same skeleton and, in many cases, parts of their behavior. The template method pattern is applicable for such algorithms. Additionally, the lists used by search algorithms for storing states are abstracted with the template method pattern for reusability and to simplify implementation of new search algorithms.

A search algorithm can optionally use a heuristic function. Heuristic functions in Guided-Modere implement the *strategy pattern*, also known as *policies*. The strategy pattern is applicable to classes which vary only in their behavior, sharing the same skeleton and interface. Essentially, all heuristic functions share the same interface, accepting a state as input and returning a single estimate in the form of a number. The search algorithm has no further interest in the heuristic and, therefore, its inner behavior can be encapsulated. Our heuristics must be interchangeable; any strategy should be usable with any search algorithm, regardless of whether that search algorithm will use it or not. In such cases, the strategy pattern is applicable.

Each search algorithm and each heuristic needs to be registered with the *search registry* and *heuristic registry*, respectively, in order to become available to the user. Both registries implement the *singleton pattern* and all registrations must take place before execution begins. In addition to the NDFS algorithm provided by standard Modere, we implemented and added four new state-space search algorithms to Guided-Modere using the aforementioned interface and three types of node lists.

The following search algorithms are supported:

- Modere's NDFS
- Breadth-first search, using FIFO node list.
- Depth-first search, using LIFO node list.
- Pure heuristic, using priority node list.
- A* search, using priority node list.
- Weighted A*, with run-time configurable weight factor.
- A* LIFO, with ties handled in a LIFO manner, instead of FIFO.

The informed search algorithms, pure heuristic and A* algorithms, can be used with all of the heuristics.

Chapter 4

Experimental Results

Guided-Modere is supported by two front-end command-line tools: a runner and a counter-example viewer. The runner handles the process of compiling a Rebeca model, performing the requested executions, collecting relevant data and generating a report. The second tool, counter-example viewer, conveniently displays the counter-examples for debugging and analysis.

We ran our experiments on various problems and protocols from the literature. Each of them was modeled in Rebeca and verified for deadlock freedom before adding errors occurring at different places in the state-spaces. Many commonly experienced modeling errors will be discovered at certain depth in the search tree, regardless of state variable configuration and order of events. Such errors did not yield interesting results besides showing the well-known difference between depth-first search and breadth-first search. More interesting in the context of this study are errors which occur only in rare situations, preferably in models with a relatively large state-space. Moreover, we are interested in difficult errors that might not be found by simulation or by conventional model checkers, implementing depth-first or breadth-first search, before reaching an upper bound or state-space explosion.

4.1 Setup

Experiments were executed on machines with Dual Core Intel(R) Xeon(TM) CPU 3.20GHz processors and 2GB RAM. The machines ran Rocks release 5.0 (V)/CentOS release 5 (Final) with Linux kernel 2.6.18–53.1.14.el5. Java code was built with Sun Java version 1.6.0–22 and C++ code using G++ (GCC) 4.1.2 20070626 without compiler optimization. Results concerning execution time are the average of either 10 or

20 consecutive executions, as noted. Modere's bound for maximum depth the default 10,000. Guided-Modere had no such limit implemented.

4.2 Self-stabilizing Token Ring

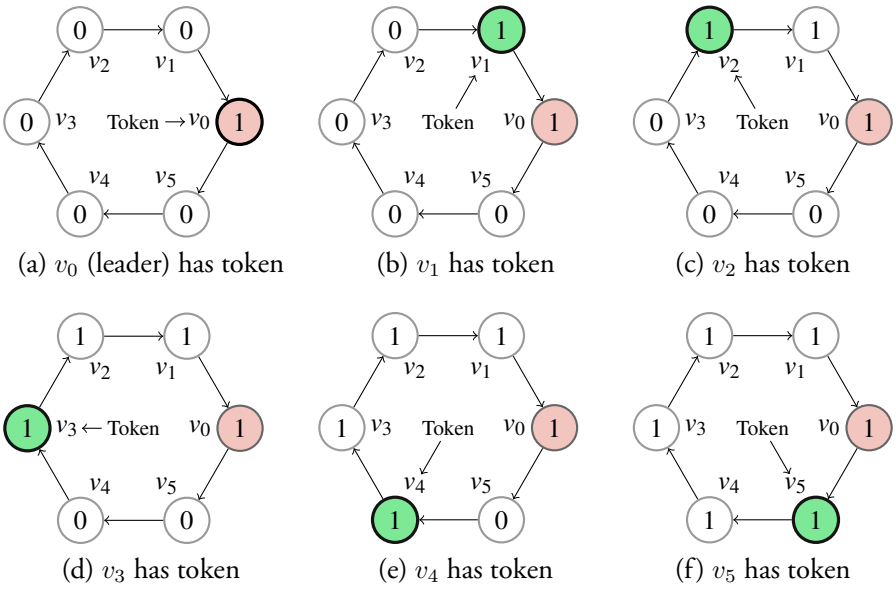
Edsger W. Dijkstra presented a self-stabilizing token ring using guarded commands in an article on self-stabilization (Dijkstra, 1974). Starting from an arbitrary initial state, the token ring will converge to a legitimate state in a finite number of steps, such that exactly one node holds the token. Each node v in the graph knows its clockwise, or left hand, neighbor as its parent p and node v_0 is the leader. The value of node v is denoted $S(v)$ and is eventually in $\{0, \ldots, n-1\}$ where n is the number of nodes in the graph. In a legitimate state, a single node in the token ring holds the token allowing access to a mutual resource. The token is then passed on from a parent node to its child. Pseudo-code for the algorithm run by each node is shown as Algorithm 1.

Algorithm 1 Edsger W. Dijkstra's token ring

```
1: if v = v_0 then
2:    if S(v) = S(p) then
3:       S(v) := S(v) + 1 mod n
4:    end if
5: else
6:    if S(v) ≠ S(p) then
7:       S(v) := S(p)
8:    end if
9: end if
```

The implementation of the algorithm in Rebeca has one reactive class, Node, and a rebec for each node in the graph. Instead of having the nodes knowing their parent we invert the relations such that each node knows its child. Every time a node changes its value it will send the new value to its child. In order to verify the model from every initial state we initialize the value with a non-deterministically selected value in the range $\{0, \ldots, n-1\}$. The model is shown in Listing D.4.

The token ring model with 6 nodes is verified deadlock-free with 7,180,795 states. An example execution showing a single round of a stabilized token ring is illustrated in Figure 4.1.



(a) v_0 (leader) has token (b) v_1 has token (c) v_2 has token

(d) v_3 has token (e) v_4 has token (f) v_5 has token

Figure 4.1: Token ring: Illustration of algorithm 1 with 6 nodes where v_0 is the leader. Value of each node, S(v), is shown inside it. Edges lead from child to parent.

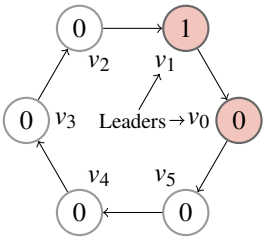


Figure 4.2: A deadlock configuration in a token ring with two adjacent leaders, each with a different value.

Token Ring with Two Leaders

The token ring requires a unique leader. Should there be either no leader or more than one leader, the system may deadlock. A token ring without a leader will converge to a state in which all of the nodes have the same value, $S(v_1) = \ldots = S(v_{n-1})$, in a finite number of steps. In such a configuration, no updates will take place and the system is in a deadlock state. If there is more than one leader in the token ring, each of them will eventually have a different value from the next. Once the new value of one leader reaches the next, the leader will refuse the value as it does not match its own. Such a deadlock configuration, for two adjacent leaders, is illustrated in Figure 4.2.

The objective of this model is to experiment with errors that only occur when the entire system is in one of the relatively few configurations. Adding another leader to the token

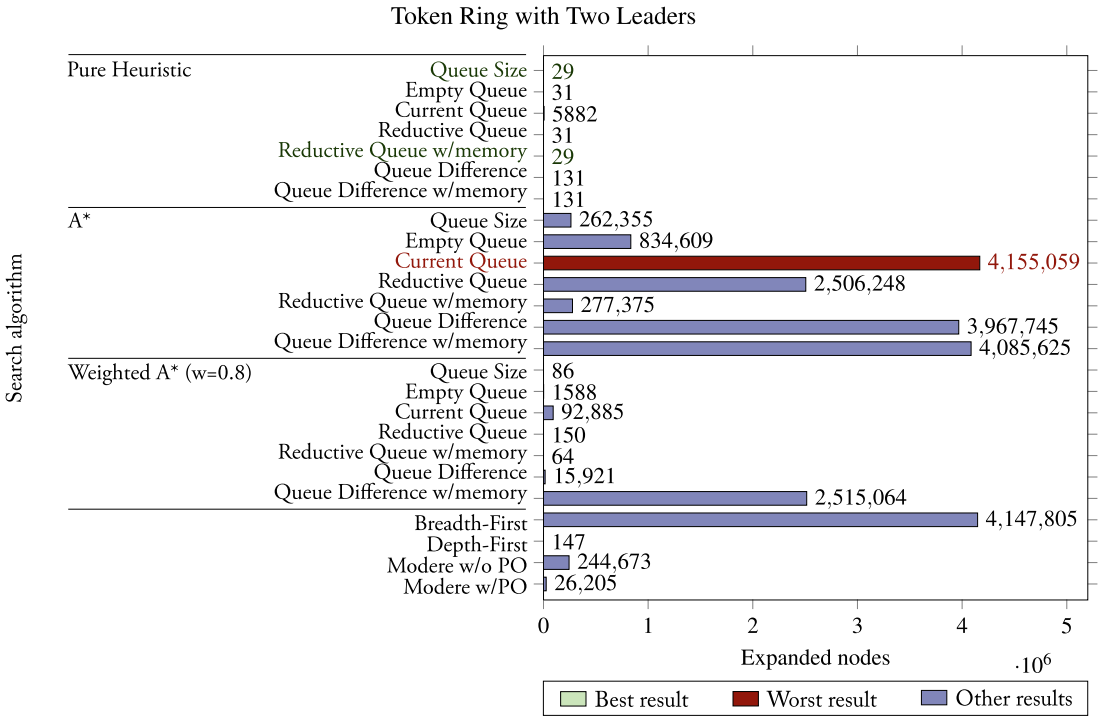


Figure 4.3: Results for a token ring with two leaders: Number of nodes expanded in the search tree before finding a deadlock.

ring produces this kind of error. With the added leader, adjacent to the previous one, the search tree contains 5,453,017 nodes with maximum depth of 56.

The number of nodes each search algorithm and heuristic expanded before finding a deadlock state is illustrated in Figure 4.3. The range is quite big, with breadth-first exploring 4,147,805 nodes (76% of the state-space) and pure heuristic search with Queue Size and Reductive Queue with Memory heuristics finding the deadlock after only 29 nodes (0.0005% of the state-space).

All of the pure heuristic searches, except when using Current Queue, found a deadlock after fewer than 150 expansions. The initial branching factor in this model is quite large and, therefore, it is important for a depth-seeking algorithm like pure heuristic search to make a good initial choice. In this case, the number of disabled rebecs is a good measurement since a stable system will have only one enabled rebec, making pure heuristic search with Empty Queue heuristic ideal. Since every message sent in the system is conditional, all of the heuristics depending on the total size of the message queue have a positive effect on the search. The Current Queue depends only on the queue of the rebec being executed, aiming to empty out the rebecs closest to becoming disabled and to make isolated parts of the system entirely disabled. Since each node

is either directly or indirectly connected to each other such isolation is impossible and the status of individual rebecs is not a good indicator in this model.

The branching factor for the initialization of each node is 6 and in each of the initializations the rebec will send exactly one message. Note that there is no requirement made that all rebecs are initialized before processing other messages of initialized rebecs.

Due to all of the rebecs sending the same number of messages, the evaluation function of A* search, $f(n) = g(n) + h(n)$, returns a worse value for a node reducing the number of messages at depth 3, than for node any non-reducing node at level 1. A non-reducing node at depth 2 will receive the same evaluation as a reducing node at level 3. Simply put, this creates an initial threshold for A*. Therefore, A* explores a great amount of nodes which have equal evaluation at low depth before proceeding deeper. After crossing this threshold the algorithm shows its strength over breadth-first search and finds a solution relatively quickly when used with the higher-value heuristics. As expected, A* search with the lower-value heuristics, Reductive Queue and Queue Difference performed similar to breadth-first search.

Weighted A* search with weight factor of 0.8 is sufficiently more willing to explore deeper to overcome this problem. However, by doing so the guaranteed optimality of A* is sacrificed (Pearl, 1984).

The difference between Modere and pure depth-first search is likely a result of different order of handling non-deterministic choices.

Most of the searches returned either an optimal or a near optimal counter-example, with a few exceptions. As expected, all A* searches returned optimal solutions. The length of each solution is shown in Figure 4.4. Depth-first and pure heuristic search returned longer paths than the other algorithms, while still shorter than the longest observed counter-example of 26 actions for this model. Modere returned an optimal solution.

Figure 4.5 illustrates the average execution times, of 20 consecutive executions, for all algorithms. They are closely related to the number of nodes expanded with the exception that uninformed searches are relatively faster than the heuristic searches. There is an insignificant difference between the pure heuristic searches, all of which returned a counter-example in less than 60 ms, with the exception of the Current Queue heuristic which required considerably more time as it expanded more nodes. A* search with the lower-value heuristics holds the records for longest executions, having

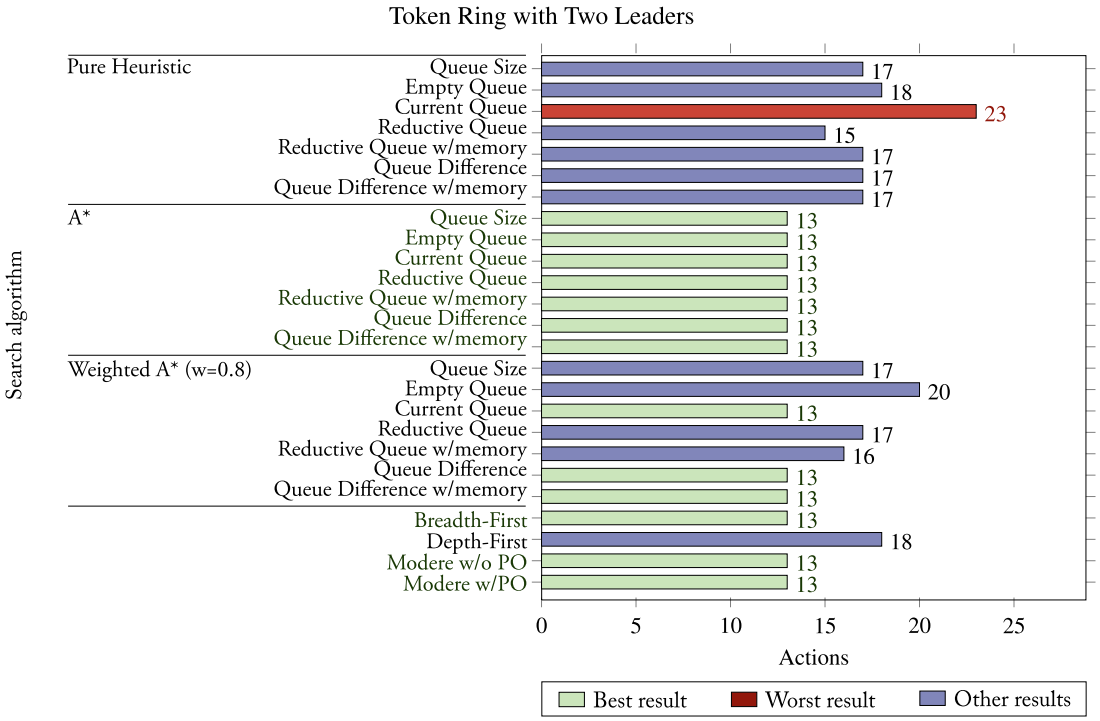


Figure 4.4: Results for a token ring with two leaders: Length of counter-examples returned, producing the deadlock error found.

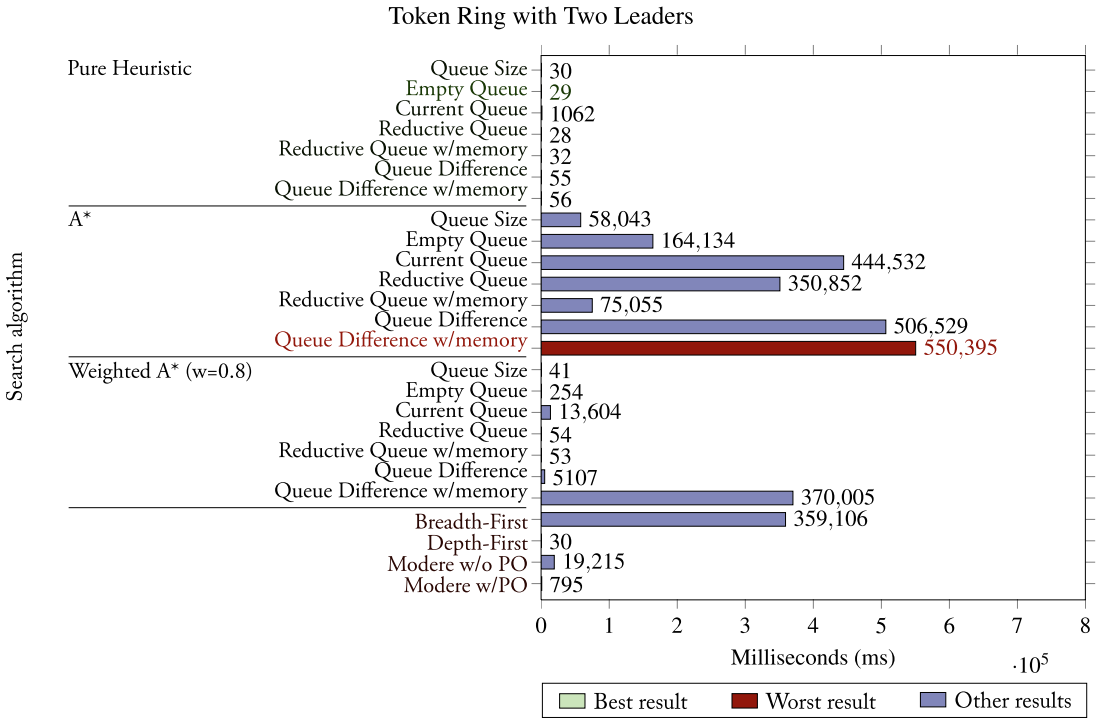


Figure 4.5: Results for a token ring with two leaders: Execution time.

explored nearly as many nodes as breadth-first search with the added overhead of the heuristics.

A* search with Empty Queue heuristic performed best of the searches guaranteeing optimal solution. This is due to the small overhead and the fact it expanded the fewest nodes of the optimal algorithms.

The best Guided-Modere search found a deadlock after expanding 29 nodes compared to 26,205 nodes for the best Modere search. Both found optimal solutions.

4.3 Dining Philosophers

The dining philosophers is another well-known problem in computer science, originally presented by Edsger W. Dijkstra as five computers competing for access to five shared drives. C. A. R. Hoare later presented the problem as five philosophers sharing a dinner around a circular table. The table has five plates, five forks and a large bowl of spaghetti in the middle (Hoare, 1978). As philosophers do, they are either thinking or eating. Each philosopher needs two forks to eat and always starts by picking up the fork on his left side and then the one on his right side. One can easily see that if all philosophers enter the table at the same time and pick up their left fork, they will all starve to death while waiting for the fork on their right side. The dining philosophers problem is illustrated in Figure 4.6

Various approaches have been proposed for solving the problem. Our model is based on the implementation used in (Jaghoori et al., 2006) and has one of the philosophers pick up the fork to his right first. This is sufficient to break the symmetry and no philosopher will starve. We have verified our model to be correct and deadlock free.

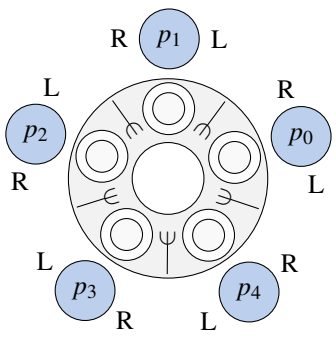


Figure 4.6: The Dining Philosophers problem.

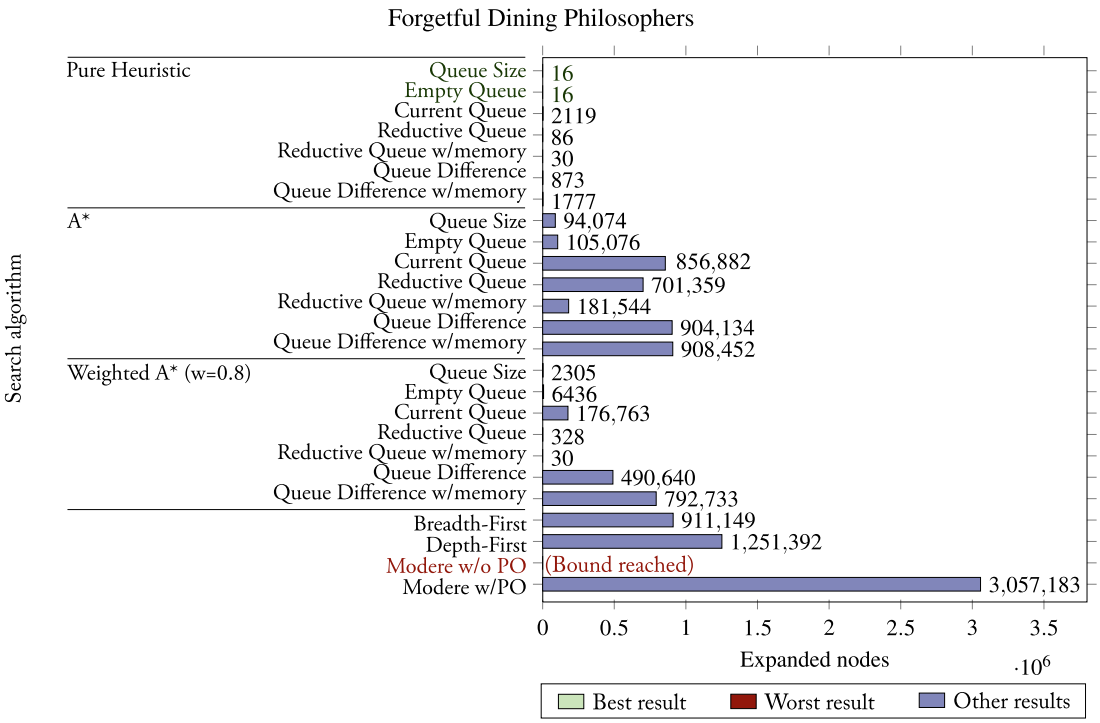


Figure 4.7: Forgetful Dining Philosophers: Number of nodes expanded in the search tree before finding a deadlock.

Forgetful Dining Philosophers

Now we may consider the possibility that one or more philosophers do not show up and what effect that may have on the system. The extended problem grants each philosopher the ability to remember the dinner and arrive as expected, to forget it temporarily and potentially remember it later or to forget the dinner completely and thus never arriving. Choice between the options is non-deterministic.

If a philosopher does not arrive the other two around him will benefit by gaining unconditional access to his forks. Should a philosopher arrive, but neither of those adjacent to him, he will be granted access to both of his forks each time requested and will never need to wait for longer than the time it may take each fork to respond. The model requires at least one philosopher to arrive, otherwise there will be no events driving the system and it deadlocks.

The heuristics will detect whether a philosopher has forgotten the dinner completely or not and evaluate the state accordingly. If a philosopher has forgotten it temporarily, the non-deterministic choice is made again in the same way as before. The heuristics manage to exploit this behavior and drive the search relatively quickly to a deadlock. The philosophers will be evaluated equally for the first three levels of the search tree,

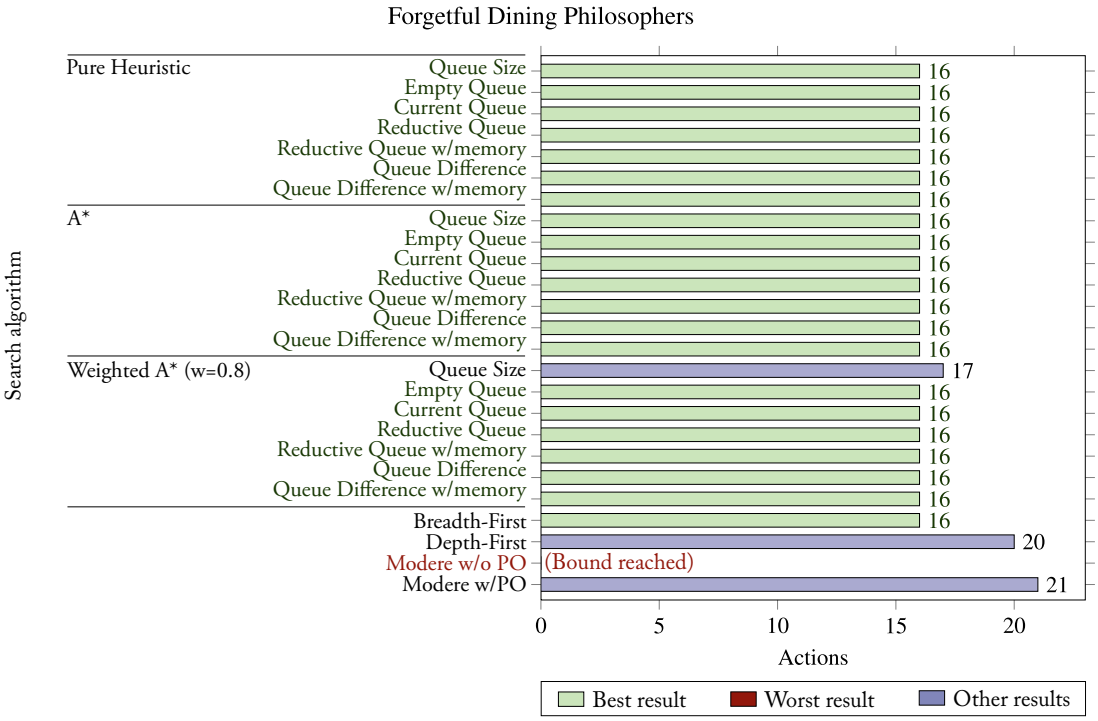


Figure 4.8: Forgetful Dining Philosophers: Length of counter-examples returned, producing the deadlock error found.

creating a threshold forcing A* search to expand a significant amount of similar nodes, which is clearly visible in Figure 4.7. Once the optimality requirement has been relaxed, namely in weighted A*, this threshold is proven quite useful as the cost of proceeding down a path with one or more non-forgetting philosophers is expensive. Due to bad initial choices the depth-first searches are expanding a large amount of nodes before eventually finding a deadlock state.

Weighted A* search with Reductive Queue and Reductive Queue with Memory heuristics performed well with 328 and 30 expanded nodes, respectively. This is lower than observed for this heuristic with similar models and thus luck may be involved. All of the pure heuristic searches expanded relatively few nodes with both of the Reductive Queue heuristics, Empty Queue and Queue Size under 100 nodes.

There is a significant difference between Modere, expanding over 3 million nodes, and the heuristic searches, all of which expanded fewer than 1 million nodes. The best heuristic search expanded 16 nodes, only 0.0005% of the 3,057,182 nodes that Modere, with partial order reduction, expanded. With Modere's default upper bound of 10,000 for the search depth, it did not return a result without the partial order reduction. Once the limit had been increased sufficiently, the deadlock was found after 3,323,899 node expansions.

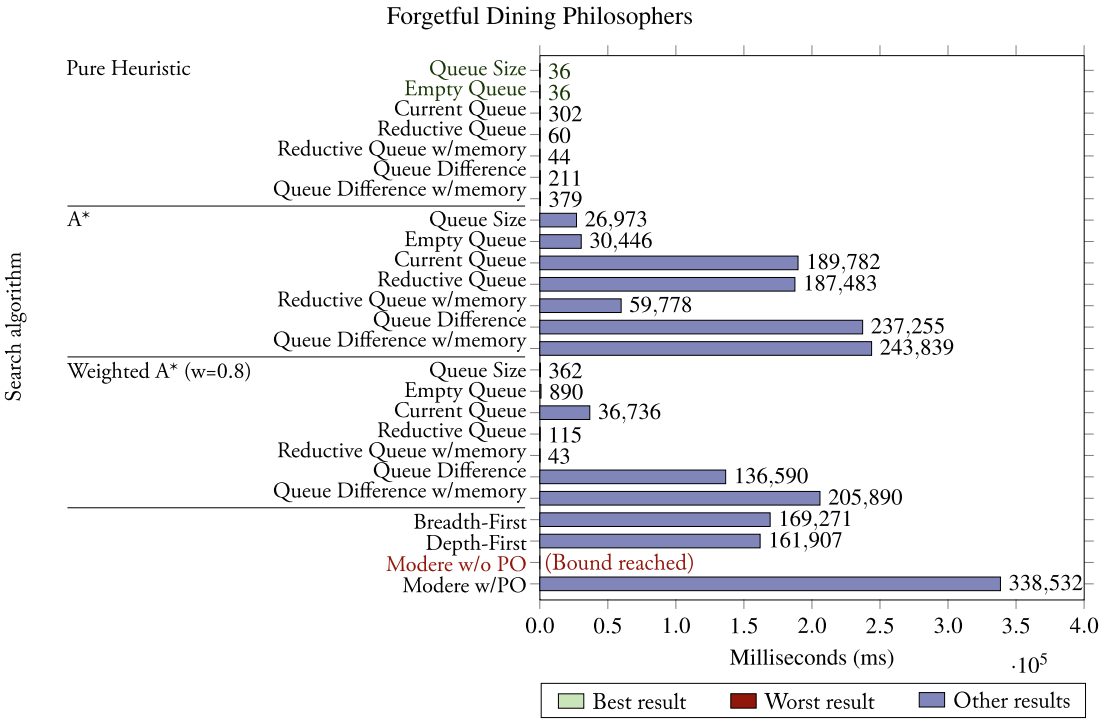


Figure 4.9: Forgetful Dining Philosophers: Execution time.

There is only a small difference in the length of counter-examples returned, listed in Figure 4.8. All informed searches found an optimal or a near optimal solution of 16 and 17 states. The depth-first searches returned a slightly longer solution of 20 states.

As shown in Figure 4.9, Guided-Modere found the deadlock after less time than standard Modere for all searches (average of 20 executions). The fastest guided search had a total execution time of only 36 ms, compared to 338,532 ms (5 min. and 38 sec.) for Modere. The fastest search with guaranteed optimality was A* with Queue Size heuristic, finding the deadlock in 26,973 ms (26 sec).

For this problem, Guided-Modere dominates Modere with shorter counter-examples, fewer expanded nodes and shorter execution times.

4.4 Needham-Schroeder Public-Key Protocol

The Needham-Schroeder Public-Key Protocol is a well-known communication protocol, proposed by Roger Needham and Michael Schroeder, providing mutual authentication of two clients communicating over an insecure network using a trusted server for key exchange (Needham & Schroeder, 1978).

We modeled the protocol using a multiplication of the value and the key as encryption, and division as decryption, requiring the public and private keys to be the same. Although not recommended for practical use, this was sufficient to verify the communication. Unfortunately, the initial protocol suffered from a man-in-the-middle vulnerability, identified by Gavin Lowe who proposed a fixed version (Lowe, 1995). However, we chose to implement the original protocol.

The protocol has seven steps which can be described as follows:

1. $A \rightarrow S$: A, B
2. $S \rightarrow A$: $\{K_b, B\}_{K_s^{-1}}$
3. $A \rightarrow B$: $\{N_a, A\}_{K_b}$
4. $B \rightarrow S$: B, A
5. $S \rightarrow B$: $\{K_a, A\}_{K_s^{-1}}$
6. $B \rightarrow A$: $\{N_a, N_b\}_{K_a}$
7. $A \rightarrow B$: $\{N_b\}_{K_b}$

Where A is the initiating client, or *initiator*, B responding client, or *responder*, and S the trusted key exchange server. K_a and K_b are the respective keys of clients A and B and K_s^{-1} is the signature of the key server S. N_a and N_b are the respective nonces generated by A and B. Nonce is a single-use generated key used to prevent replay attacks. We refer to a single execution of this protocol as a *conversation*.

Needham-Schroeder Protocol with Nonce error

This experiment focuses on errors triggered only when events occur in a particular order.

Clients generate *nonces* on two separate occasions. The first is before the initiating client sends its first message to the responding client and the second is before the responding client replies directly to the initiating client, sent in steps 3 and 6 respectively. By using a single state variable for storing nonces, regardless of where they were created, we have introduced an interesting deadlock error which is produced only when both clients initiate communication at the same time and the rebecs executed in a specific order. When both clients have an incorrect nonce for each other the system will deadlock.

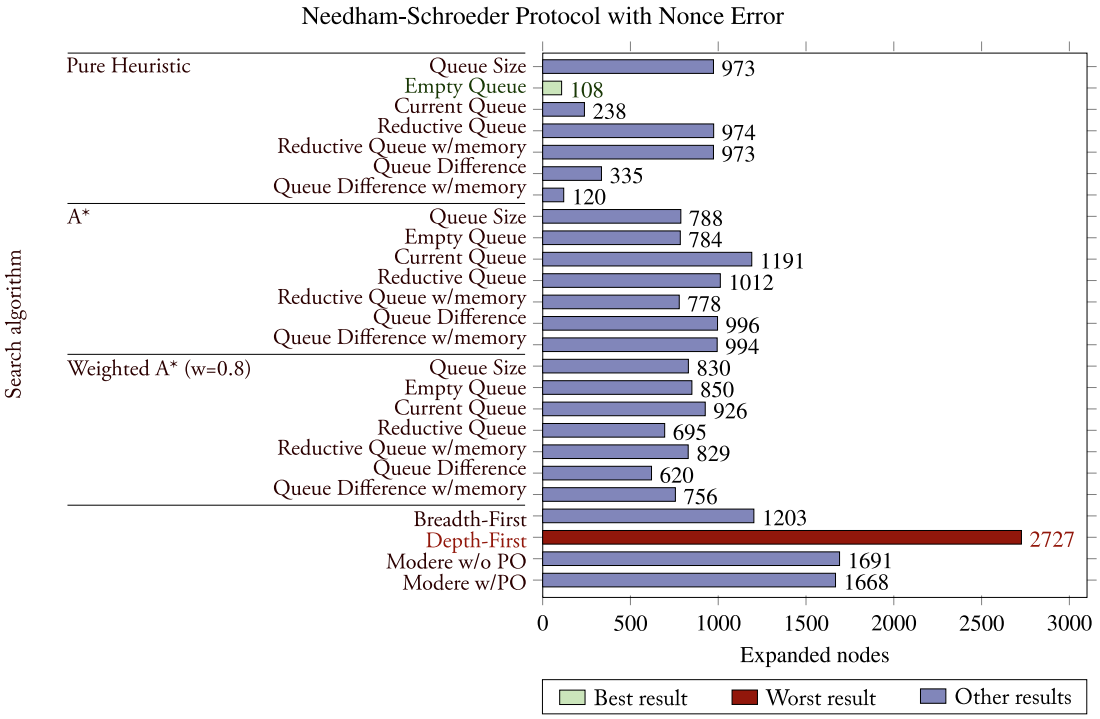


Figure 4.10: Needham-Schroeder public key protocol with nonce error: Number of nodes expanded in the search tree before finding a deadlock.

The initiator sends the responder its nonce in step 3 and receives it again in step 6. If the initiator has, in another conversation as the responder, overwritten the nonce in between, the verification will fail.

Another similar scenario is when the responder sends his nonce to the initiator in step 6 and receives it in step 7. If the reponder has executed step 3 as the initiator in another communication the verification in step 7 will fail. However, due to the use of ordered message queues (instead of unordered bags) this cannot occur when both communications are initialized simultaneously.

The state-space contains 5456 nodes with maximum depth of 912.

The number of nodes each search expanded before finding a deadlock state is illustrated in Figure 4.10. All the informed search methods required fewer nodes than the uninformed searches. Interestingly, breadth-first search expanded fewer nodes than depth-first search. The reason is the relatively small branching factor and a deadlock state is found at depth of 18 in the much deeper search tree.

Pure Heuristic search with the Empty Queue heuristic expanded the fewest nodes, finding a counter-example after only 108 nodes and is closely followed by Queue Difference with Memory which expanded 120 nodes. A plausible explanation for the

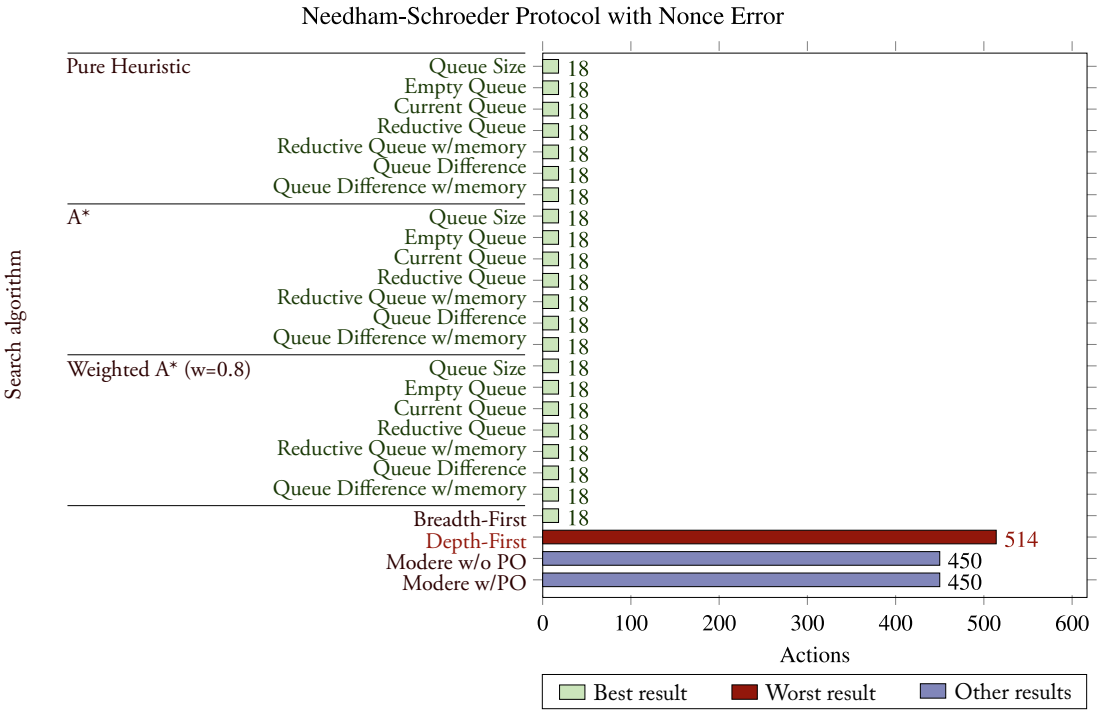


Figure 4.11: Needham-Schroeder public key protocol with nonce error: Length of counter-examples returned, producing the deadlock error found.

success of Queue Difference with Memory is its tendency to favor a single rebec over others, in this case, the server. The memory discount will provide the server with the benefit of the doubt as it is the only rebec causing negative difference in message numbers, before the two failed verifications which cause the deadlock. This will make the execution more fair, allowing both clients to replace their nonces in step 5, as responders, one after the other. Once one of the two conversations has failed, it will expand that branch and discover the second conversation failing as well. Exploration of the Empty Queue heuristic was quite similar, allowing the rebecs to go hand in hand until both reached deadlock in the same manner. Overall, pure heuristic search proved to be useful for this problem.

A* search expanded considerable more nodes compared to pure heuristic search, with a relatively similar number for most of the heuristics. A* search with the higher-value heuristics expanded much fewer nodes than breadth-first search, while the lower-value heuristic explored similar amount of nodes.

Counter-examples returned by the informed searches are significantly shorter than those from both depth-first searches, illustrated in Figure 4.11. All the informed searches return an optimal solution of 18 actions. Modere and pure depth-first search generated much longer counter-examples containing 450 and 514 actions, respec-

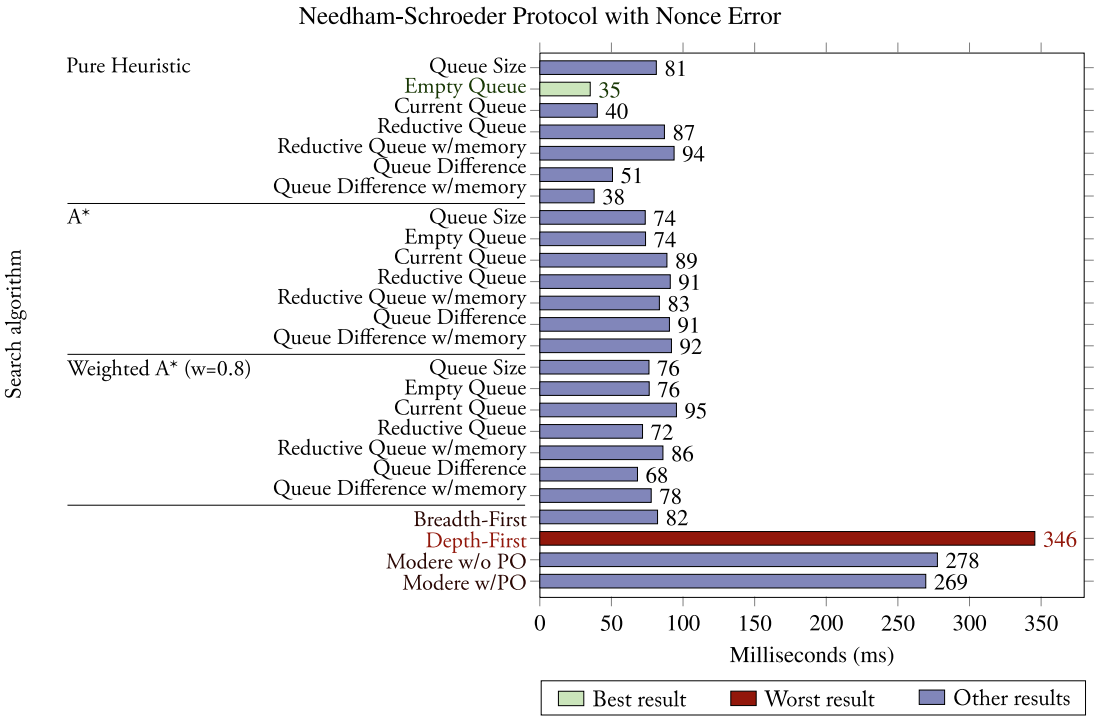


Figure 4.12: Needham-Schroeder public key protocol with nonce error: Execution time.

tively, 25.5 and 28.5 times the optimal solution. This is partly explained by the height of the search tree and the two concurrent conversations.

Figure 4.12 illustrates the execution time before a deadlock is found (average of 20 executions). The best pure heuristic searches were notably faster than the others, which was expected, knowing that they expanded the fewest nodes. Difference between execution times of the best A* and breadth-first search was 35%.

4.5 Combined Heuristics

Table 4.1 lists results for pairs of heuristics. For reference, results from Modere are appended to the table.

In comparison to the results of individual heuristics, the average of expanded nodes was significantly lower for the combination heuristics. The average was lower for all searches except for Needham-Schroeder model (deadlock version) with A* search algorithm. On the other hand, an individual heuristic had the fewest expanded nodes for 7 of the groups (model and search algorithm) compared to only 2 groups for the combined heuristics. Thus, the results indicate that the combination of heuristics will result

Search Algorithm	Heuristic A	Heuristic B	Token Ring w/2 leaders	Forgetful Philosophers	Needham Schroeder
Pure Heuristic	Empty Queue	Queue Size	52	16	108
	Empty Queue	Current Queue	241	33	409
	Empty Queue	Reductive Queue	30	24	974
	Empty Queue	Queue Difference	83	33	331
	Queue Size	Current Queue	33	33	87
	Queue Size	Reductive Queue	20	24	974
	Queue Size	Queue Difference	56	33	331
A*	Empty Queue	Queue Size	605,553	104,934	794
	Empty Queue	Current Queue	3,236,580	545,081	1174
	Empty Queue	Reductive Queue	2,149,159	553,891	1016
	Empty Queue	Queue Difference	3,053,137	542,811	998
	Queue Size	Current Queue	2,377,881	501,939	1013
	Queue Size	Reductive Queue	1,686,553	419,156	1016
	Queue Size	Queue Difference	2,381,821	613,367	995
A* w=0.8	Empty Queue	Queue Size	366	6436	850
	Empty Queue	Current Queue	13,489	72	789
	Empty Queue	Reductive Queue	218	155	695
	Empty Queue	Queue Difference	4434	150	624
	Queue Size	Current Queue	256	72	690
	Queue Size	Reductive Queue	152	155	695
	Queue Size	Queue Difference	91	150	624
Modere without PO			244,673	3,323,899	1691
Modere with PO			26,205	3,057,183	1668

Table 4.1: Expanded nodes before finding a deadlock using the average of two heuristics, compared to Modere.

in more stable heuristics and better average performance while individually they may perform better for specific models and errors. However, further experiments would be required to draw such a conclusion. Spikes in expansions with pure heuristic search as seen for individual heuristics were not observed for the combined heuristics. The Current Queue heuristic, which was not performing well on its own, proves valuable when paired with both the Queue Size and Empty Queue heuristics.

Further details on these results with length of counter-examples and execution times are provided in Appendix B.

4.6 When DFS is Faster

As mentioned earlier, DFS will perform similarly or better for errors which occur at a certain depth or a relatively small depth range in the search tree, regardless of the path chosen. In other cases, the heuristics may not detect error paths and, in the worst case, focus on error-free sections of the state-space and run out of memory before finding a violating state.

If optimal solutions are required, either breadth-first or A* search must be used. A* search with an admissible and consistent heuristic should expand fewer nodes than breadth-first search. Therefore, comparing the performance between the two is important, even though plain depth-first search may outperform the pure heuristic search.

This section is dedicated to the study of models where the depth-first search expands similar or fewer nodes than the heuristic searches. Only the number of expansions are illustrated while length of solutions and execution times are provided in Appendix B (Table B.4).

Token Ring

A token ring where the required relation between two of the nodes is broken is in fact no longer a ring. As a result, the token will eventually not be passed on, and the system deadlocks. Results for this model, with 5 nodes, are illustrated in Figure 4.13. The optimal path to a deadlock state consists of 11 actions and the maximum depth of the search tree is 96. The best heuristic search found a solution after expanding 14 states and the best blind search after only 15 states. For this kind of a problem, heuristic search is only beneficial if an optimal solution is required. The best A* search found an optimal solution after 15,595 node expansions while breadth-first search expanded 236,769 nodes.

Dining Philosophers

Standard Modere outperformed all other searches for the Dining Philosophers problem, when no measures had been taken to prevent deadlocking. The results are illustrated in Figure 4.14. Modere expanded only 31 nodes, executing the minimal set of actions required to produce the error. The best search of Guided-Modere expanded 224 nodes. Most of the searches returned optimal solutions, including those of Modere. Some of the A* searches outperform breadth-first search while others expanded more nodes, due to reopening of states. A* search with Queue Size heuristic and Reductive Queue with Memory heuristic return a guaranteed optimal path after 198,596 and 200,510 expansions, respectively. Compared to the 388,234 nodes breadth-first search expanded, that is a reasonably good performance.

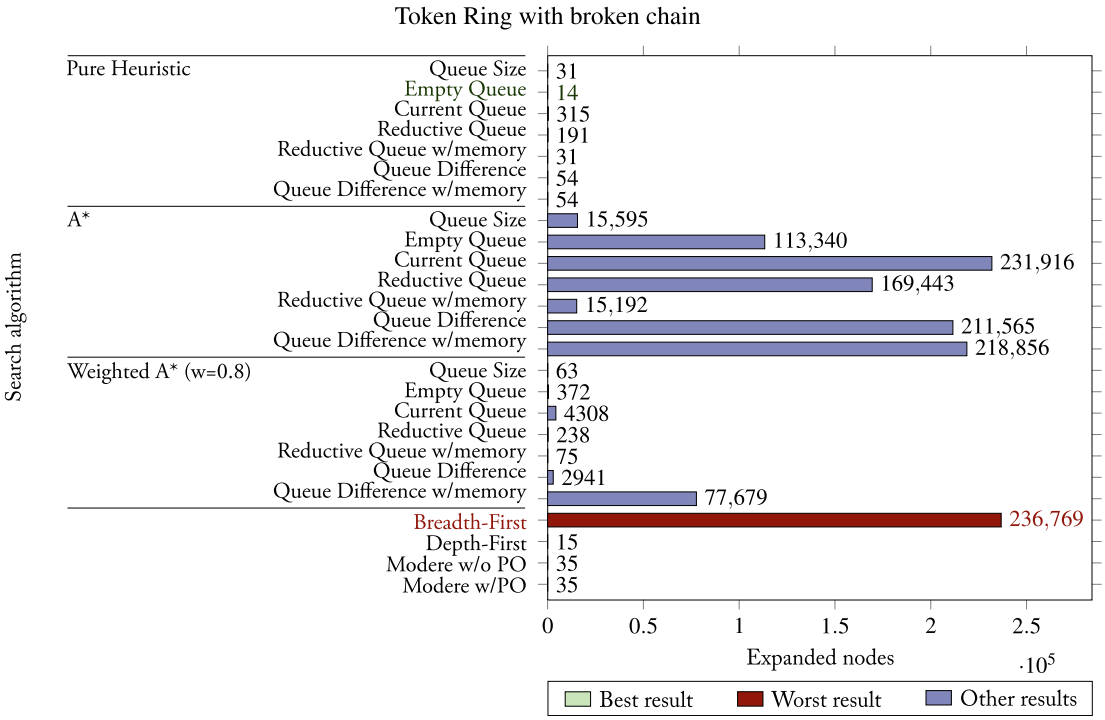


Figure 4.13: Token Ring with broken chain: Number of nodes expanded in the search tree before finding a deadlock.

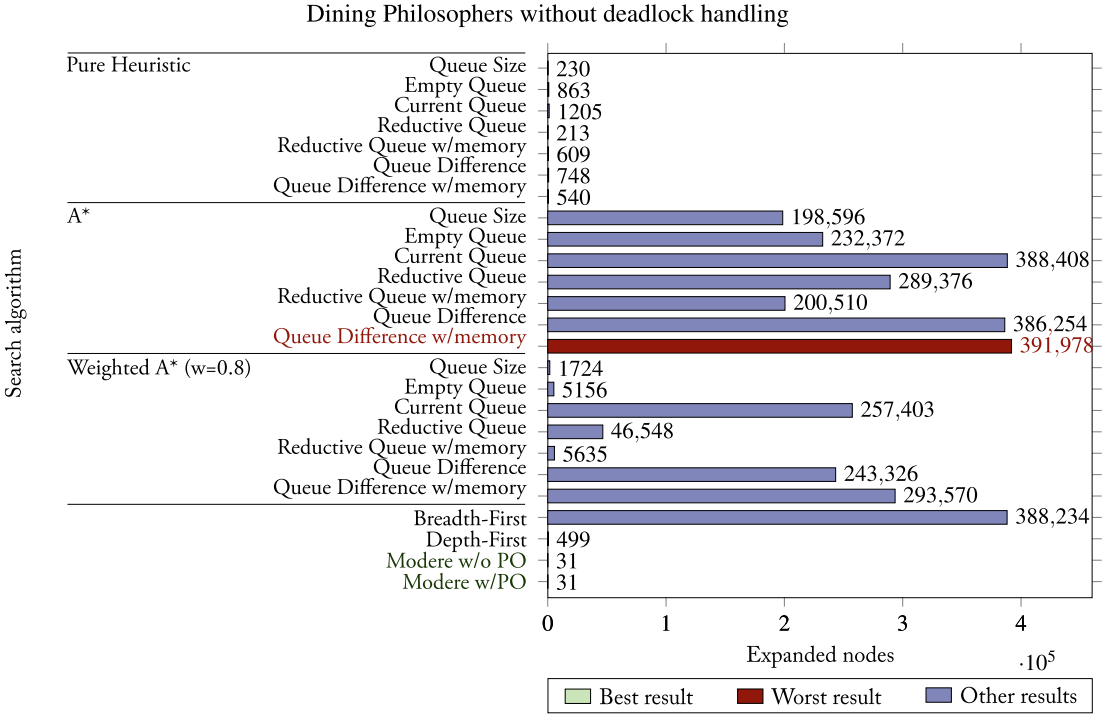


Figure 4.14: Dining Philosophers without deadlock prevention: Number of nodes expanded in the search tree before finding a deadlock.

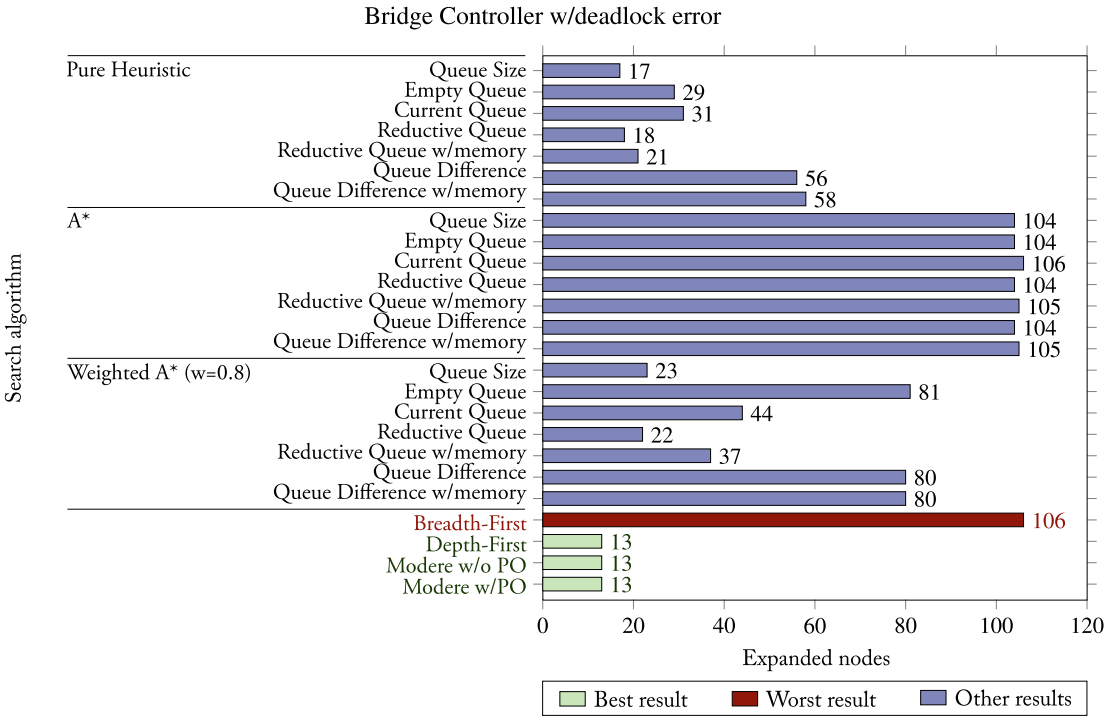


Figure 4.15: Bridge Controller with deadlock error: Number of nodes expanded in the search tree before finding a deadlock.

Bridge Controller

The Bridge Controller is a neat Rebeca model presented in (Sirjani et al., 2005). If we remove the send statement where a train announces it has left the bridge, the model will deadlock at depth 13 for all paths in the search tree. Thus, none of the heuristic search algorithms can outperform depth-first search, always finding the deadlock state in 13 expansions. The results for this model are listed in Figure 4.15. All searches return an optimal path since all deadlock paths are of the same length. Breadth-first search expanded all but one node in the 107 node search tree. Of the heuristic searches, pure heuristic search using the Queue Size heuristic performed the best, expanding 17 nodes before finding a deadlock state.

4.7 Performance

Modere is highly optimized with regard to CPU usage and memory consumption, for example, by using custom-written data structures. No such optimizations have been done in Guided-Modere as the emphasis has mainly been on building a flexible research framework. The code was written with readability and extendability in

mind rather than performance and it uses data structures from the standard library. Regardless, comparing the computational cost of each heuristic is necessary.

Due to the above-mentioned code optimizations in standard Modere we were unable to compile it successfully with compiler optimization. Regardless of the optimization level used, the resulting executable terminated with segmentation fault. Guided-Modere could, however, be compiled with full compiler optimization without issues. Experiments showed significant decrease in execution time as optimization levels were increased and showed execution times comparable with standard Modere expanding the same number of nodes. To avoid bias in the results, all experimental executions were compiled without optimization, unless otherwise noted.

Comparison tests were performed for executions that were guaranteed to expand the same exact number of nodes. This requirement was intended to prevent overhead caused by initialization and report generation from affecting the results. To accomplish this we model checked two models: Token ring with 5 nodes and dining philosophers with 4 philosophers, both of which were verified deadlock free. Algorithms included in the comparison are Modere without partial order reduction, depth-first search, breadth-first search and pure heuristic search with each of the heuristics. Modere with partial order reduction expanded fewer nodes and, due to re-opening of states, A* search with inconsistent heuristics expanded more nodes. Therefore, they were not included in the comparison.

The state-space of the token ring model, with 5 nodes, is 175,226 states and the dining philosopher model, with 4 philosophers, 46,010 states. Each pair of model and search algorithm was executed 10 times in a row, with and without compiler optimization. Comparison chart with average execution times of the token ring is shown in Figure 4.16 and dining philosophers problem in Figure 4.17. Comparison of average nodes expanded per second for the token ring in shown in Figure 4.18 and dining philosophers in Figure 4.19. Execution times discussed in this section are the average of 10 consecutive executions.

The rate of exploration, measured in nodes expanded per second, varies between one model to the other based on the models' complexity. Simple models will generally have higher rates than complex ones. The impact of compiler optimization is quite impressive, increasing the rate by a minimum of 27% (Token Ring, Queue Difference) and up to 59% (Dining Philosophers, Queue Difference with Memory).

As expected, Standard Modere has a higher rate of exploration than Guided-Modere when compiled with the same settings. Interestingly, the informed search algorithm

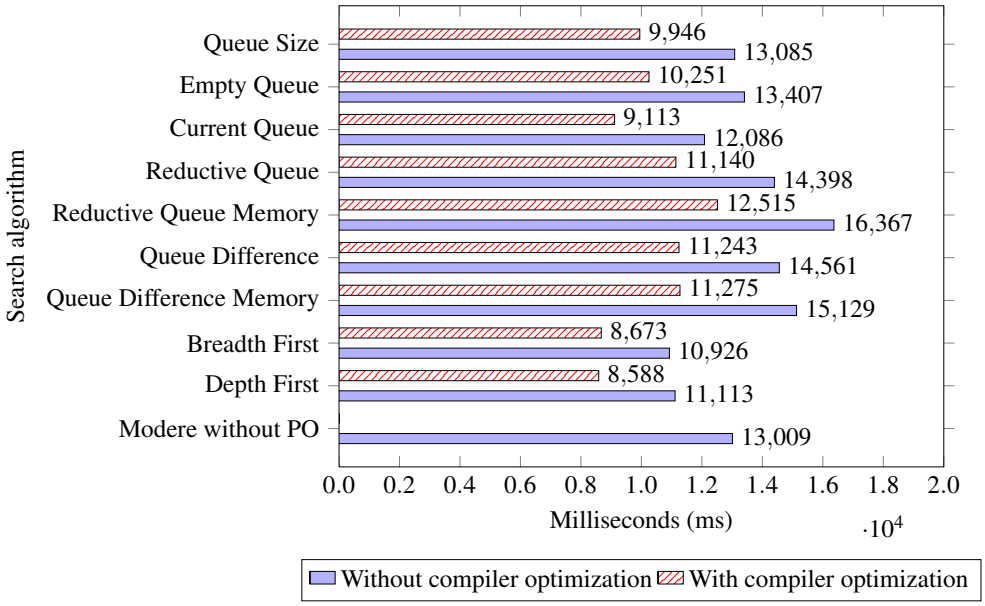


Figure 4.16: Token ring with 5 nodes: Execution time, with and without compiler optimization.

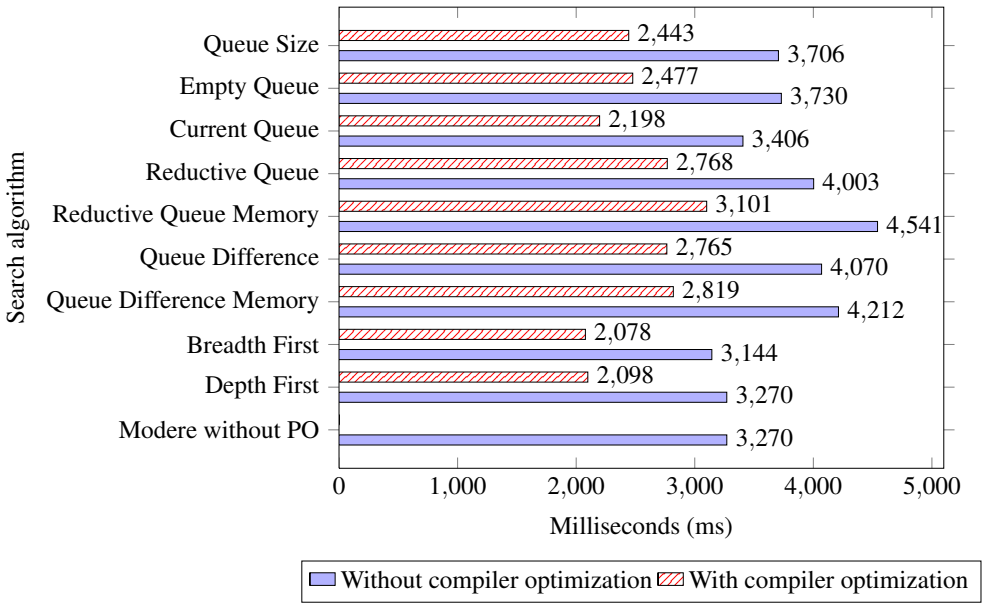


Figure 4.17: Dining Philosophers with 4 philosophers: Execution time, with and without compiler optimization.

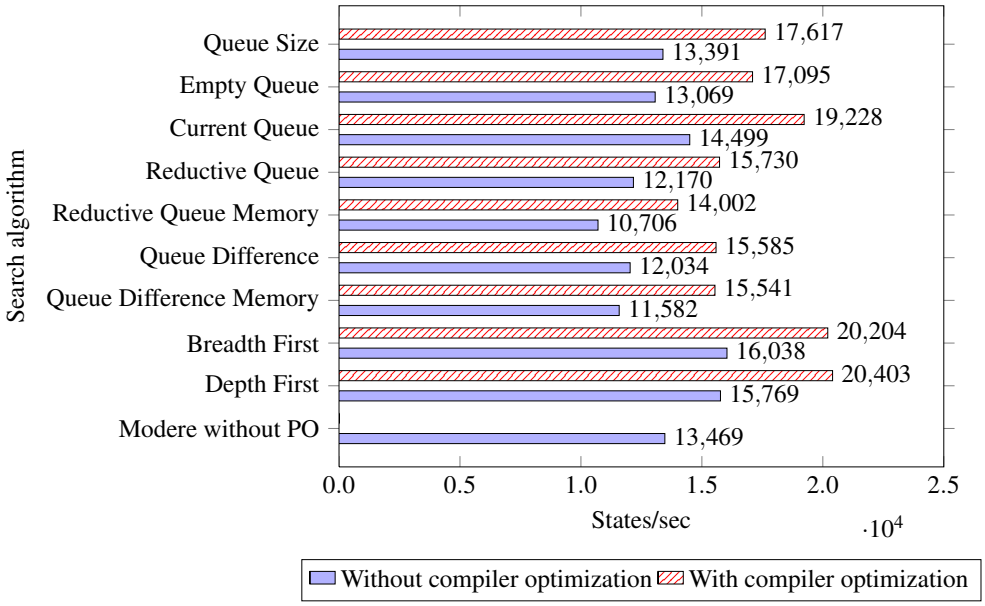


Figure 4.18: Token ring with 5 nodes: States expanded per second (states/sec), with and without compiler optimization.

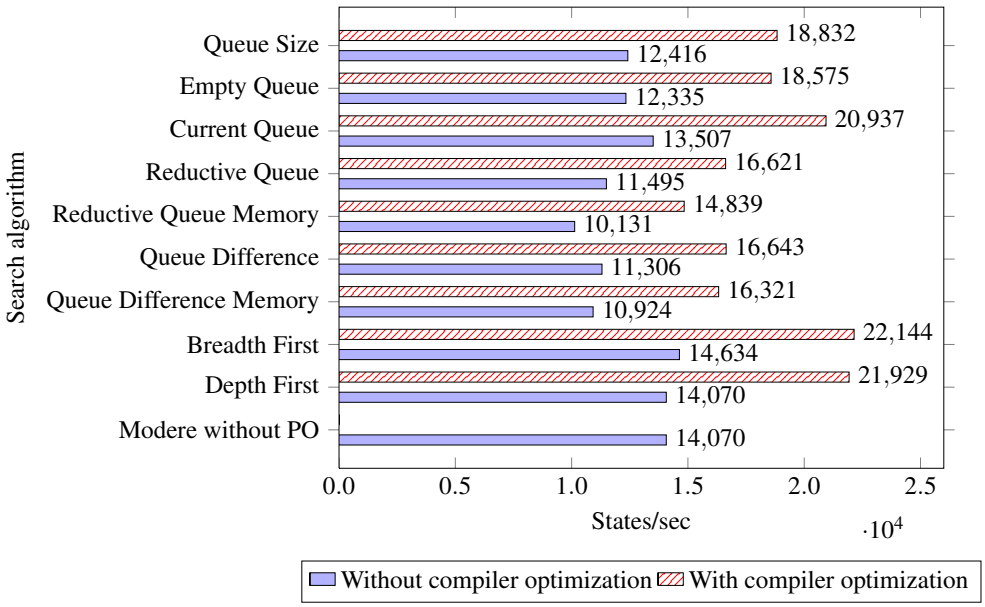


Figure 4.19: Dining Philosophers with 4 philosophers: States expanded per second (states/sec), with and without compiler optimization.

exceeds the blind search algorithms for some heuristics. This indicates that the additional overhead caused by those heuristics is small, although other factors may have an impact on this. Search algorithms in Guided-Modere use the following Standard Library and Standard Template Library data structures to achieve the required behavior:

- double ended queue for FIFO and LIFO node lists,
- priority queue for the node lists of Pure Heuristic and A* and,
- hash map, which all of the algorithms use for faster finding and removal of nodes.

The standard library priority queue has worst-case of $O(n)$ for adding a new node, $O(log(n))$ amortized. Thus, the rate of the informed search algorithms can vary between models depending on the order in which the nodes are expanded and their heuristic values. As the blind search algorithms depend only on the order in which the nodes are created this does not apply to them.

The heuristics can be categorized as follows: heuristics depending on the node's state, depending on the node's state and its parent, and heuristics depending on the parent and an external data structure for memory. Note that the information could be coded into the states; not doing so is mainly for practical purposes. The computational cost increases between each category. This is clearly visible in the comparison charts in Figures 4.18 and 4.19. Queue Size, Current Queue and Empty Queue depend only on the node itself and have the highest exploration rate. Reductive Queue and Queue Difference depend on the parent node as well, which requires additional computation and is visible in the reduced rate. In the third and computationally most expensive category are both memory algorithms which have a rate considerably lower than other heuristics.

Measurements indicated that the memory required for each node in Guided-Modere is on average 83 bytes greater than for standard Modere, the total size depending on the model being verified. The additional memory footprint is a result of nodes being stored as objects and the additional information required by the informed search algorithms. Without a doubt, this number can be decreased by refactoring and optimization. However, no such optimizations were attempted as they were outside the scope of the study.

Chapter 5

Related Work

Some form of directed model checking has been present from the beginning of model checking. Jan Hajek's *Approver*, first implemented in 1977, is assumed to be the first automated verification tool for communication protocols and used directed search for verification of safety properties (Edelkamp et al., 2009). In addition to classic communication protocols it was also capable of verifying other concurrent systems such as mutual exclusion algorithms. It used techniques dedicated to bug-finding instead of depth-first or breadth-first search, which have usually been applied to model checking.

In 1998 Yang and Dill published a paper on validation with guided search presenting a strategy called Target Enlargement which was very effective for some models (Yang & Dill, 1998). The idea was to make the target states bigger by computing their pre-image, the states that in one cycle can reach an error state, until the computer's memory limitation was reached. If the heuristics found problems, they usually did so in fewer states than breadth-first and depth-first search. They also show that their approach is more likely to find an error before state-space explosion occurs. In addition to Target Enlargement they propose a heuristic using Hamming distance as search metric. Their third technique, called Tracks, is similar to Target Enlargement except it uses approximate pre-images of error states based on subsets of the state variables, focusing the main state variables which control the behavior of the system. The fourth technique, Guideposts, relies on hints provided by the designer. The number of guideposts a path passes through is used as guidance for the search.

Edelkamp, Leue, and Lluch-Lafuente coined the term *directed model checking* in their paper on HSF-SPIN (Edelkamp et al., 2001). Based on SPIN and its Promela modeling language, HSF-SPIN used A*, best-first search (referred to as pure heuristic search

in this thesis) and an improved NFDS algorithm to establish safety and a large class of LTL-specified liveness properties. They present a formula-based heuristics for different classes of properties. One of them, called H_{ap}, is used for finding deadlocks and is virtually identical to our Empty Queue heuristic. Other heuristics aim at violation of liveness properties, safety properties and they allow designer-devised heuristics where the protocol designer can alter the definition of heuristics and explicitly define which states are dangerous. Without designer intervention, all reads, sends and conditions are considered dangerous. The paper includes test results for variety of protocols.

In (Hoffmann et al., 2007) predicate abstraction is used to generate heuristics for the verification of networks of extended timed automata in Uppaal. They build the entire abstract state-space before starting the search. During the search, states are mapped to their counterpart in the abstracted state-space and the error distance of the counterpart used as a heuristic estimate.

Groce and Visser present heuristic-guided model checking of Java programs in (Groce & Visser, 2004) using Java PathFinder. They present seven different heuristics, in short, based on the number executions of branches and byte-code instructions, branch coverage, number of blocked threads, amount of interleaving, thread preference and non-determinism avoidance (choose-free). Additionally, they allow the developer to define his own heuristic function or declare certain states “boring” or “interesting” by adding certain statements to model in question.

The PROVAT strategy presented in (Lin et al., 1988) has several heuristics, all of which depend only on the send and receive operations as in this study. They cover the selection of states from the open list, selection between actions available from a specific state and deciding whether to discard particular states. The heuristic proposed for finding deadlocks is somewhat similar to our Empty Queue heuristic. When selecting between transitions the deadlock heuristic always selects a transition performing a receive operation, choosing the one with the fewest messages among those. This corresponds to the behavior of our Current Queue heuristic, but in a different context. Their channel overflow strategy corresponds to our inverted version of Current Queue for guidance towards queue overflow, presented in Appendix C.

In 1996 Clarke and Wing proposed an external storage, distributed and directed on-the-fly model checker (Edelkamp & Jabbar, 1994).

Other studies have focused on attacking the state-space explosion problem for Rebeca models. Slicing-based reductions for Rebeca, with respect to a given property, are presented in (Sabouri & Sirjani, 2010). This approach can reduce the state-space for

some models and thus verify them before state-space explosion occurs. In (Behjati et al., 2010), a bounded rational verification approach is proposed using a Monte Carlo controlled reinforcement learning agent to model check large or infinite Rebeca models. The search is optimized for finding counter-examples and returns an approximate upon reaching the upper-bound for exploration.

Chapter 6

Conclusion and Future Work

In this thesis, we have presented seven heuristics which guide actor-based models towards deadlock states. Our experimental results indicate that they can significantly reduce the number of node expansions required before finding an error state. We have shown their ability to produce shorter counter-examples than the conventional depth-first search and in fewer node expansions than the optimal breadth-first search. For models with larger state-spaces than those considered in this study, blind searches could exhaust the computer's memory before reaching a goal, failing to find a deadlock error if one exists. The guided searches presented might reach that deadlock state before the state-space explosion occurs.

The requirement of optimality causes considerable overhead and pure heuristic search returned relatively short solutions for most of our experiments, expanding only a fraction of the states A* search explored. Weighted A* with $w = 0.8$ was a good balance between the two objectives, returning near-optimal counter-examples with fewer expansions than traditional A* search.

Two of the heuristics, Queue Size and Empty Queue, showed the best overall performance regarding node expansion and execution time. For A* search we recommend the Queue Size heuristic as it largely dominates the others and will thus perform better, as visible in the results. Since A* is an optimal search algorithm and all of the heuristics admissible, the counter-example returned is always optimal.

Further experiments can be done with Guided-Modere using the implemented heuristics. Other combinations of heuristics with different factors between them could potentially result in better performance. One could even create combinations targeting specific deadlock scenarios. Additionally, the inverted versions driving the search towards queue overflow require further research.

Heuristics based on information provided by the designer have been implemented in (Groce & Visser, 2004) and (Edelkamp et al., 2001), for example. Such a heuristic could be implemented for Guided-Modere. The model designer could tag message servers as either interesting or boring. Then the heuristic would guide the search such that states processing interesting messages are chosen over those processing a boring one. This strategy could be implemented by enabling either flags or annotations in the Rebeca code or by the identification of specific message servers at run-time.

Our research was limited to A* and pure heuristic search. IDA* search (Iterative-Deepening A*) (Korf, 1985) has been successfully applied to model checking (Edelkamp, Leue, & Lluch-Lafuente, 2004) and has been demonstrated to perform well in general with inconsistent heuristics (Zahavi et al., 2007). K-beam search and the non-pruning alternative k-best search (Felner, 2001) have been successfully applied to model checkers (Groce & Visser, 2004; Wijs & Lisser, 2007). MA* (Memory-bounded A*) and SMA* (Simplified-MA*) are designed to overcome the impractical memory requirements of A*. Simply put, once the search runs out of memory it expands the best node on the open list and prunes off the worst node (Russell, 1992).
All of these algorithms would make an interesting addition to Guided-Modere. The Weighted A* search could potentially be improved by dynamic weighting. Rather than keeping the weight constant throughout the search the heuristics could have the most weight initially, reducing it as the search gets deeper into search tree (Pearl, 1984). This approach provides an upper bound for the length of solution with respect to the optimal solution. Thus, the requirement for optimality could be relaxed up to a specific point, with regard to an optimal solution. In other words, the number of actions in the counter-example returned would never be more than, say, twice that of an optimal solution.

Modere implements partial-order reduction. Such a reduction would benefit the informed search algorithms and further reduce the required node expansions.

The study focuses only on the message queue and its relation to the deadlock property. The next big step would be developing property-based heuristics and performing heuristic search for violations of safety properties for actor-based models. Furthermore, applying the hybrid A*+Improved-Nested-DFS search algorithm presented in (Edelkamp et al., 2004) would enable the verification of liveness properties as well. The Target Enlargement technique proposed in (Yang & Dill, 1998) or predicate abstraction such as proposed in (Hoffmann et al., 2007) could result in reduced exploration. Both of these approaches could be of benefit in the search for violations of the deadlock property as well.

While outside of the scope of this study, the overhead of model checking with Modere could be reduced. In order to model check any model with either Modere or Guided-Modere the Rebeca code must be compiled to C++ code, which is then compiled to native code. Considerable portion of the native code, which is non-specific to the model in question, does not change from one model to the other and could be cached. Such optimization would reduce the time before the state-space search begins.

The methods presented in this thesis provide the ability to guarantee shortest counter-examples for deadlocks more efficiently than the conventional breadth-first search. Without the requirement of optimality they may find deadlock errors in models, where standard depth-first search would suffer state-space explosion, by exploring the state-space more efficiently.

Bibliography

Agha, G. a., Mason, I. a., Smith, S. F., & Talcott, C. L. (1997, January). A foundation for actor computation. *Journal of Functional Programming*, *7*(1), 1–72. Available from http://www.journals.cambridge.org/abstract_S095679689700261X

Baier, C., Katoen, J., & Others. (2008). *Principles of model checking*. The MIT Press. Available from http://mitpress.mit.edu/catalog/item/default.asp?ttype=2&tid=11481

Barnat, J., Brim, L., & Chaloupka, J. (2003). Parallel breadth-first search LTL model-checking. *18th IEEE International Conference on Automated Software Engineering, 2003. Proceedings.*, 106–115. Available from http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1240299

Barnat, J., & Černá, I. (2006). Distributed breadth-first search LTL model checking. *Formal Methods in System Design*, *29*(2), 117–134. Available from http://www.springerlink.com/index/94376Q360325P688.pdf

Behjati, R., Sirjani, M., & Nili Ahmadabadi, M. (2010). Bounded Rational Search for On-the-Fly Model Checking of LTL Properties. *Fundamentals of Software Engineering*, 292–307. Available from http://www.springerlink.com/index/Q22174422J82W260.pdf

Burch, J., Clarke, E., McMillan, K., Dill, D., & Hwang, L. (1986). Symbolic model checking: 10/sup 20/ states and beyond. *[1990] Proceedings. Fifth Annual IEEE Symposium on Logic in Computer Science*(4976), 428–439. Available from http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=113767

Clarke, E. (1997). Model checking. In *Foundations of software technology and theoretical computer science* (p. 54). Springer. Available from http://www.springerlink.com/index/xmq77a6tejwdc7nr.pdf

Clarke, E., & Emerson, E. (1982). Design and synthesis of synchronization skeletons using branching time temporal logic. *Logics of Programs*, 52–71. Available from http://www.springerlink.com/index/w1778u28166t2677.pdf

Clarke, E. M., Emerson, E. A., & Sistla, A. P. (1986, April). Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, *8*(2), 244–263. Available from http://portal.acm.org/citation.cfm?doid=5397.5399

Clarke, E. M., Grumberg, O., & Long, D. E. (1994, September). Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, *16*(5), 1512–1542. Available from http://portal.acm.org/citation.cfm?doid=186025.186051

Courcoubetis, C., Vardi, M., Wolper, P., & Yannakakis, M. (1992). Memory-efficient algorithms for the verification of temporal properties. *Formal methods in system design*, *1*(2), 275–288. Available from http://www.springerlink.com/index/p1706rk8030252r3.pdf

Dijkstra, E. (1974). Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, *17*(11), 643–644. Available from http://portal.acm.org/citation.cfm?id=361202

Dowson, M. (1997, March). The Ariane 5 software failure. *ACM SIGSOFT Software Engineering Notes*, *22*(2), 84.

Dr, K. (2009, February). Directed model checking with distance-preserving abstractions. *International Journal on Software Tools for Technology Transfer*, *1*(9), 10–37.

Edelkamp, S., & Jabbar, S. (1994). Large-Scale Directed Model Checking LTL.

Edelkamp, S., Lafuente, A., & Leue, S. (2001). Directed explicit model checking with HSF-SPIN. In *Proceedings of the 8th international spin workshop on model checking of software* (pp. 57–79). Springer-Verlag New York, Inc. Available from http://portal.acm.org/citation.cfm?id=380921.380930

Edelkamp, S., Leue, S., & Lluch-Lafuente, A. (2004). Directed explicit-state model checking in the validation of communication protocols. *International Journal on Software Tools for Technology Transfer (STTT)*, *5*(2), 247–267. Available from http://www.springerlink.com/index/k7kgd4thktmyf7k1.pdf

Edelkamp, S., Schuppan, V., Bosnacki, D., Wijs, A., & Aljazzar, H. (2009). Survey on Directed Model Checking. *Artificial Intelligence*, *5348*(April 2006).

FDIV Replacement Program: Statistical Analysis of Floating Point Flaw. (2004). Available from http://www.intel.com/support/processors/pentium/sb/cs-013005.htm

Felner, A. (2001). Improving search techniques and using them on different environments. *Science*(February). Available from http://www.ise.bgu.ac.il/faculty/felner/research/phd_thesis.ps

Groce, A., & Visser, W. (2004, October). Heuristics for model checking Java programs. *International Journal on Software Tools for Technology Transfer*,

6(4), 260–276. Available from http://www.springerlink.com/index/10.1007/s10009-003-0130-9

Grosu, R., & Smolka, S. (n.d.). Quantitative model checking. In *Proc. of isola* (Vol. 4, pp. 165–174). Citeseer. Available from http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.117.4967&rep=rep1&type=pdf

Havelund, K., Lowry, M., & Penix, J. (2001). Formal analysis of a spacecraft controller using SPIN. *IEEE Transactions on Software Engineering*, 749–765. Available from http://www.computer.org/portal/web/csdl/doi/10.1109/32.940728

Hewitt, C. (1977). Viewing control structures as patterns of passing messages. *Artificial Intelligence*, *8*(3), 323–364.

Hoare, C. a. R. (1978, August). Communicating sequential processes. *Communications of the ACM*, *21*(8), 666–677. Available from http://portal.acm.org/citation.cfm?doid=359576.359585

Hoffmann, J., Smaus, J., Rybalchenko, A., Kupferschmid, S., & Podelski, A. (2007). Using predicate abstraction to generate heuristic functions in Uppaal. *Model Checking and Artificial Intelligence*, 51–66. Available from http://www.springerlink.com/index/j3724375q6q3t804.pdf

Holzmann, G. (1997, May). The model checker SPIN. *IEEE Transactions on Software Engineering*, *23*(5), 279–295. Available from http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=588521

Holzmann, G., Peled, D., & Yannakakis, M. (1996). On nested depth first search. *American Mathematical Society*, *51*(12), 1336–1338. Available from http://books.google.com/books?hl=en&lr=&id=b2bvLuMILZkC&oi=fnd&pg=PA23&dq=On+nested+depth+first+search&ots=SsNvE2EM4n&sig=5gJnlT17zoCjJ2EPoXUaehnOui4

Jaghoori, M., Movaghar, A., & Sirjani, M. (2006). Modere: The model-checking engine of Rebeca. *Proceedings of the 2006*, 1810–1815. Available from http://portal.acm.org/citation.cfm?id=1141704

Johnson, R., Gamma, E., Helm, R., & Vlissides, J. (1995). Design Patterns: Elements of Reusable Object-Oriented Software. *Addison-Wesley*, *1*(February), 1–2. Available from http://www.best-seller-books.com/design-patterns-elements-of-reusable-object-oriented-software.pdf

Korf, R. (1985, September). Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, *27*(1), 97–109. Available from http://linkinghub.elsevier.com/retrieve/pii/0004370285900840

Leveson, N. (1993, July). An investigation of the Therac-25 accidents. *Computer*, *26*(7), 18–41.

Lin, F. J., Chu, P. M., & Liu, M. T. (1988). Protocol verification using reachability analysis: the state space explosion problem and relief strategies. *Proceedings of the ACM workshop on Frontiers in computer communications technology - SIGCOMM '87*, 126–135. Available from http://portal.acm.org/citation.cfm?doid=55482.55496

Lind-Nielsen, J. (1999). Stepwise CTL model checking of state/event systems. *Computer Aided Verification*, 316–327. Available from http://www.springerlink.com/index/0V4VLHJK5DHX9QFB.pdf

Lowe, G. (1995). An attack on the Needham-Schroeder public-key authentication protocol. *Information processing letters*, *56*(3), 131–133. Available from http://linkinghub.elsevier.com/retrieve/pii/0020019095001442

Needham, R. M., & Schroeder, M. D. (1978, December). Using encryption for authentication in large networks of computers. *Communications of the ACM*, *21*(12), 993–999. Available from http://portal.acm.org/citation.cfm?doid=359657.359659

Pearl, J. (1984). *Heuristics: intelligent search strategies for computer problem solving*. Addison-Wesley. Available from http://www.osti.gov/energycitations/product.biblio.jsp?osti_id=5127296

Peled, D. (1994). Combining partial order reductions with on-the-fly model-checking. In *Computer aided verification* (pp. 377–390). Springer. Available from http://www.springerlink.com/index/73348Q774295W8T2.pdf

Pnueli, A. (1977). The temporal logic of programs. *Foundations of Computer Science, Annual IEEE Symposium on*, *0*, 46–57.

Queille, J., & Sifakis, J. (1982). Specification and verification of concurrent systems in CESAR. In *International symposium on programming* (pp. 337–351). Springer. Available from http://www.springerlink.com/index/7X327643572334RW.pdf

Rushby, J. (1995). *Formal methods and their role in the certification of critical systems* (No. March). Citeseer. Available from http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.48.3999&rep=rep1&type=pdf

Russell, S. (1992). Efficient memory-bounded search methods. *Proceedings of the 10th European conference on Artificial intelligence*.

Russell, S., Norvig, P., Canny, J., Malik, J., & Edwards, D. (1995). *Artificial intelligence: a modern approach* (2nd editio ed., Vol. 74). Prentice hall Englewood Cliffs, NJ. Available from http://www.cis.uab.edu/courses/cs760/Spring-660-2007/7A-660-PLUS-SYLLABUS-DRAFT.pdf

Sabouri, H., & Sirjani, M. (2010, January). Slicing-based Reductions for Rebeca. *Electronic Notes in Theoretical Computer Science*, *260*, 209–224. Available from http://linkinghub.elsevier.com/retrieve/pii/S1571066109005210

Shannon, C. (1950). Programming a computer for playing chess. *Philosophical magazine*, *41*(7), 256–275. Available from http://www.iis.sinica.edu.tw/~{}tshsu/tcg2010/slides/slide6.pdf

Sirjani, M., De Boer, F., & Movaghar, A. (2005). Modular verification of a component-based actor language. *Journal of Universal Computer Science*, *11*(10), 1695–1717. Available from http://www.jucs.org/jucs_11_10/modular_verificatin_of_a/jucs_11_10_1695_1717_sirjani.pdf

Sirjani, M., Movaghar, A., & Shali, A. (2004). Modeling and verification of reactive systems using Rebeca. *Fundamenta*, *63*, 1–26. Available from http://portal.acm.org/citation.cfm?id=1227084

Vardi, M., & Wolper, P. (1984). Automata theoretic techniques for modal logics of programs. In *Proceedings of the sixteenth annual acm symposium on theory of computing* (Vol. 32, pp. 446–456). ACM. Available from http://portal.acm.org/citation.cfm?id=808711

Wijs, A., & Lisser, B. (2007). Distributed extended beam search for quantitative model checking. *Model Checking and Artificial Intelligence*, 166–184. Available from http://www.springerlink.com/index/9J01M10731380455.pdf

Yang, C. H., & Dill, D. L. (1998). Validation with guided search of the state space. *Proceedings of the 35th annual conference on Design automation conference - DAC '98*, 599–604. Available from http://portal.acm.org/citation.cfm?doid=277044.277201

Zahavi, U., Felner, A., Schaeffer, J., & Sturtevant, N. (2007). Inconsistent heuristics. In *Proceedings of the national conference on artificial intelligence* (Vol. 22, p. 1211). Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999. Available from http://www.aaai.org/Papers/AAAI/2007/AAAI07-192.pdf

Appendix A

Terminology Overview

Action. Actions transform the system from one state to another. A single action is represented by an edge between two states in a search tree. In Rebeca an action is the execution of one message server of a single rebec.

Admissible Heuristic. A heuristic is admissible if it never over-estimates the distance to the nearest goal.

Consistent heuristic. Heuristic which are monotonically non-decreasing along the shortest path to a goal state.

Counter-example. A set of actions, starting from the initial state, which produce a violation of a property or an error. The length of a counter-example is the number of actions it contains and is equal to the depth at which the node containing the violating state is in the search tree. A counter-example is also referred to as a *solution*.

Goal state. A state which satisfies the goal test of a search. In this study we refer to a goal state as either a node in the search tree or a state in a system where the system is in a deadlock configuration.

Heuristic. A strategy which estimates the distance from a particular node in the search tree to the nearest goal. The estimate of a heuristic is delivered via its heuristic function.

Largely dominate. A_2* largely dominates A_1* if always requires less or equal number of expansions before finding a goal state.

Model. A higher level specification of a system written in a modeling language.

Node. A node exists in a search tree and and represents exactly on state. Every node except for the root node has a parent and possibly children.

Optimal Solution. There exists no shorter solution than an optimal solution. In the context of this thesis, it represents the minimal set of actions which can produce the behavior in question.

Optimal Search. An optimal search is guaranteed to return an optimal solution, if a solution exists.

Search. We refer to a search as the execution of a search algorithm, with or without a heuristic, on a search tree.

Search Algorithm. An algorithm which searches a tree of nodes with or without information from a heuristic function.

Search tree. A collection of nodes forming a tree with a single root node.

State. A state is a snapshot of the system, identified by the values of variables, messages, previously executed statements and other relevant information.

System. The execution of a model or some other structure which has a certain behavior and is transformed from one state to another via actions.

Appendix B

Extended Results

B.1 Case studies

Results for the case studies discussed in Chapter 4 are presented on table format in Table B.1. Execution times are averages of 20 consecutive executions.

Task	Data	Token ring w/2 leaders	Philosophers, forgetful	Needham, deadlock
Modere with PO	Expanded states	26,205	3,057,183	1576
	Counter-example	13	21	450
	Execution time (ms)	795	338,532	269
Modere without PO	Expanded states	244,673	-	1691
	Counter-example	13	-	450
	Execution time (ms)	19,215	-	278
Depth First	Expanded states	147	1,251,392	2727
	Counter-example	18	20	514
	Execution time (ms)	30	161,907	346
Breadth First	Expanded states	4,147,805	911,149	1203
	Counter-example	13	16	18
	Execution time (ms)	359,106	169,271	82
Pure Heuristic / Queue Size	Expanded states	29	16	973
	Counter-example	17	16	18
	Execution time (ms)	30	36	81

Continued on next page

Continued from previous page

Pure Heuristic / Empty Queue	Expanded states	31	16	108
	Counter-example	18	16	18
	Execution time (ms)	29	36	35
Pure Heuristic / Current Queue	Expanded states	5882	2119	238
	Counter-example	23	16	18
	Execution time (ms)	1062	302	40
Pure Heuristic / Reductive Queue	Expanded states	31	86	974
	Counter-example	15	16	18
	Execution time (ms)	28	60	87
Pure Heuristic / Reductive Queue Memory	Expanded states	29	30	973
	Counter-example	17	16	18
	Execution time (ms)	32	44	94
Pure Heuristic / Queue Difference	Expanded states	131	873	335
	Counter-example	17	16	18
	Execution time (ms)	55	211	51
Pure Heuristic / Queue Difference Memory	Expanded states	131	1777	120
	Counter-example	17	16	18
	Execution time (ms)	56	379	38
A* / Queue Size	Expanded states	262,355	94,074	788
	Counter-example	13	16	18
	Execution time (ms)	58,043	26,973	74
A* / Empty Queue	Expanded states	834,609	105,076	784
	Counter-example	13	16	18
	Execution time (ms)	164,134	30,446	74
A* / Current Queue	Expanded states	4,155,059	856,882	1191
	Counter-example	13	16	18
	Execution time (ms)	444,532	189,782	89
A* / Reductive Queue	Expanded states	2,506,248	701,359	1012
	Counter-example	13	16	18
	Execution time (ms)	350,852	187,483	91
A* / Reductive Queue Memory	Expanded states	277,375	181,544	778
	Counter-example	13	16	18
	Execution time (ms)	75,055	59,778	83
A* / Queue Difference	Expanded states	3,967,745	904,134	996
	Counter-example	13	16	18
	Execution time (ms)	506,529	237,255	91

Continued on next page

Continued from previous page				
A* / Queue Difference Memory	Expanded states	4,085,625	908,452	994
	Counter-example	13	16	18
	Execution time (ms)	550,395	243,839	92
A* w=0.8 / Queue Size	Expanded states	86	2305	830
	Counter-example	17	17	18
	Execution time (ms)	41	362	76
A* w=0.8 / Empty Queue	Expanded states	1588	6436	850
	Counter-example	20	16	18
	Execution time (ms)	254	890	76
A* w=0.8 / Current Queue	Expanded states	92,885	176,763	926
	Counter-example	13	16	18
	Execution time (ms)	13,604	36,736	95
A* w=0.8 / Reductive Queue	Expanded states	150	328	695
	Counter-example	17	16	18
	Execution time (ms)	54	115	72
A* w=0.8 / Reductive Queue Memory	Expanded states	64	30	829
	Counter-example	16	16	18
	Execution time (ms)	53	43	86
A* w=0.8 / Queue Difference	Expanded states	15,921	490,640	620
	Counter-example	13	16	18
	Execution time (ms)	5107	136,590	68
A* w=0.8 / Queue Difference Memory	Expanded states	2,515,064	792,733	756
	Counter-example	13	16	18
	Execution time (ms)	370,005	205,890	78

Table B.1: Results of searches for models in case studies.

B.2 Dual Heuristics

Results for dual heuristic searches discussed in Chapter 4 are presented on table format with additional details in Table B.2. Execution times are averages of 20 consecutive executions.

Task	Data	Token ring w/2 leaders	Philosophers, forgetful	Needham, deadlock
Pure Heuristic / Empty Queue- Queue Size	Expanded states	52	16	108
	Counter-example	24	16	18
	Execution time (ms)	33	38	36
Pure Heuristic / Empty Queue- Current Queue	Expanded states	241	33	409
	Counter-example	18	16	18
	Execution time (ms)	63	39	51
Pure Heuristic / Empty Queue- Reductive Queue	Expanded states	30	24	974
	Counter-example	17	16	18
	Execution time (ms)	31	39	93
Pure Heuristic / Empty Queue- Queue Difference	Expanded states	83	33	331
	Counter-example	23	16	18
	Execution time (ms)	45	43	52
Pure Heuristic / Queue Size- Current Queue	Expanded states	33	33	87
	Counter-example	16	16	18
	Execution time (ms)	28	39	35
Pure Heuristic / Queue Size- Reductive Queue	Expanded states	20	24	974
	Counter-example	13	16	18
	Execution time (ms)	27	39	93
Pure Heuristic / Queue Size- Queue Difference	Expanded states	56	33	331
	Counter-example	18	16	18
	Execution time (ms)	39	39	51
A* / Empty Queue- Queue Size	Expanded states	605,553	104,934	794
	Counter-example	13	16	18
	Execution time (ms)	134,871	32,691	79
A* / Empty Queue- Current Queue	Expanded states	3,236,580	545,081	1174
	Counter-example	13	16	18
	Execution time (ms)	425,287	142,598	93
A* / Empty Queue- Reductive Queue	Expanded states	2,149,159	553,891	1016
	Counter-example	13	16	18
	Execution time (ms)	350,190	168,276	128
A* / Empty Queue- Queue Difference	Expanded states	3,053,137	542,811	998
	Counter-example	13	16	18
	Execution time (ms)	455,191	166,279	97

Continued on next page

Continued from previous page				
A* / Queue Size- Current Queue	Expanded states	2,377,881	501,939	1013
	Counter-example	13	16	18
	Execution time (ms)	318,612	130,784	87
A* / Queue Size- Reductive Queue	Expanded states	1,686,553	419,156	1016
	Counter-example	13	16	18
	Execution time (ms)	280,724	129,534	97
A* / Queue Size- Queue Difference	Expanded states	2,379,654	613,367	995
	Counter-example	13	16	18
	Execution time (ms)	372,277	181,395	96
A* w=0.8 / Empty Queue- Queue Size	Expanded states	366	6436	850
	Counter-example	20	16	18
	Execution time (ms)	91	975	82
A* w=0.8 / Empty Queue- Current Queue	Expanded states	13,489	72	789
	Counter-example	15	16	18
	Execution time (ms)	3075	49	75
A* w=0.8 / Empty Queue- Reductive Queue	Expanded states	218	155	695
	Counter-example	17	16	18
	Execution time (ms)	80	76	75
A* w=0.8 / Empty Queue- Queue Difference	Expanded states	4434	150	624
	Counter-example	15	16	18
	Execution time (ms)	1396	78	108
A* w=0.8 / Queue Size- Current Queue	Expanded states	256	72	690
	Counter-example	17	16	18
	Execution time (ms)	78	49	69
A* w=0.8 / Queue Size- Reductive Queue	Expanded states	152	155	695
	Counter-example	13	16	18
	Execution time (ms)	51	75	75
A* w=0.8 / Queue Size- Queue Difference	Expanded states	91	150	624
	Counter-example	13	16	18
	Execution time (ms)	54	77	72

Table B.2: Results of searches with dual heuristics for models in the case studies.

B.3 Additional searches

Table B.3 lists expanded states (above) and length of counter-examples (below) for the three case studies discussed in Chapter 4, for all of the deadlock seeking algorithms

implemented in Guided-Modere. Execution times are averages of 10 consecutive executions.

By default, ties in A* search are broken in FIFO manner (first-in, first-out). In table B.3 we refer to searches in which ties are broken in LIFO manner (last-in, first out), as *A* LIFO. *For these searches, A* behaves like DFS for equal states. The random heuristic is non-admissible and, thus, A* is not optimal with this heuristic.

Algorithm	Heuristic	Token ring w/2 leaders	Philosophers, forgetful	Needham, deadlock
Pure Heuristic / Random	Expanded states	12,589	2,447,970	704
	Counter-example	14	21	18
	Execution time (ms)	1471	405,343	63
A* / Random	Expanded states	317,758	404,948	612
	Counter-example	13	18	18
	Execution time (ms)	39,159	84,939	60
A* LIFO / Queue Size	Expanded states	263,752	89,833	783
	Counter-example	13	16	18
	Execution time (ms)	60,130	26,597	72
A* LIFO / Empty Queue	Expanded states	837,125	98,299	783
	Counter-example	13	16	18
	Execution time (ms)	169,728	29,483	72
A* LIFO / Current Queue Size	Expanded states	4,128,874	897,985	1196
	Counter-example	13	16	18
	Execution time (ms)	464,172	196,620	88
A* LIFO / Reductive Queue	Expanded states	2,539,630	707,399	1011
	Counter-example	13	16	18
	Execution time (ms)	365,172	194,751	90
A* LIFO / Reductive Queue Memory	Expanded states	262,633	168,231	783
	Counter-example	13	16	18
	Execution time (ms)	75,253	57,582	82
A* LIFO / Queue Difference	Expanded states	3,885,799	905,945	995
	Counter-example	13	16	18
	Execution time (ms)	513,644	246,303	89
A* LIFO / Queue Difference Memory	Expanded states	4,085,188	908,883	996
	Counter-example	13	16	18
	Execution time (ms)	561,434	252,652	91

Continued on next page

Continued from previous page

A* LIFO w=0.8 / Queue Size	Expanded states	31	106	824
	Counter-example	13	18	18
	Execution time (ms)	25	50	74
A* LIFO w=0.8 / Empty Queue	Expanded states	209	106	827
	Counter-example	14	18	18
	Execution time (ms)	75	50	74
A* LIFO w=0.8 / Current Queue Size	Expanded states	53,725	282,457	902
	Counter-example	13	16	18
	Execution time (ms)	10,612	59,191	73
A* LIFO w=0.8 / Reductive Queue	Expanded states	47	648	719
	Counter-example	17	17	18
	Execution time (ms)	38	178	72
A* LIFO w=0.8 / Reductive Queue Memory	Expanded states	55	35	828
	Counter-example	14	16	18
	Execution time (ms)	52	42	84
A* LIFO w=0.8 / Queue Difference	Expanded states	7971	476,309	609
	Counter-example	13	16	18
	Execution time (ms)	3468	125,092	65
A* LIFO w=0.8 / Queue Difference Memory	Expanded states	2,533,642	789,086	753
	Counter-example	13	16	18
	Execution time (ms)	378,597	202,297	74
A* w=0.9 / Queue Size	Expanded states	68	27	169
	Counter-example	16	16	18
	Execution time (ms)	49	39	38
A* w=0.75 / Queue Size	Expanded states	63	46	588
	Counter-example	13	17	18
	Execution time (ms)	39	40	60
A* w=0.7 / Queue Size	Expanded states	114	78	663
	Counter-example	14	17	18
	Execution time (ms)	68	48	64
A* w=0.6 / Queue Size	Expanded states	777	1483	682
	Counter-example	13	16	18
	Execution time (ms)	375	391	66
A* w=0.9 / Empty Queue	Expanded states	59	27	311
	Counter-example	18	16	18
	Execution time (ms)	31	37	45

Continued on next page

Continued from previous page				
A* w=0.75 / Empty Queue	Expanded states	1976	46	569
	Counter-example	15	17	18
	Execution time (ms)	442	39	57
A* w=0.7 / Empty Queue	Expanded states	3621	78	761
	Counter-example	13	17	18
	Execution time (ms)	1022	48	70
A* w=0.6 / Empty Queue	Expanded states	119,226	1223	749
	Counter-example	13	16	18
	Execution time (ms)	32,383	356	69
A* w=0.9 / Current Queue Size	Expanded states	144,890	442,278	353
	Counter-example	17	16	18
	Execution time (ms)	25,179	77,358	45
A* w=0.75 / Current Queue Size	Expanded states	427,593	204,032	890
	Counter-example	13	16	18
	Execution time (ms)	49,374	38,586	74
A* w=0.7 / Current Queue Size	Expanded states	1,328,083	238,702	765
	Counter-example	13	16	18
	Execution time (ms)	135,712	43,839	67
A* w=0.6 / Current Queue Size	Expanded states	4,159,812	201,919	870
	Counter-example	13	16	18
	Execution time (ms)	456,246	38,492	72
A* w=0.9 / Reductive Queue	Expanded states	110	282	293
	Counter-example	15	16	18
	Execution time (ms)	50	83	47
A* w=0.75 / Reductive Queue	Expanded states	1266	1770	813
	Counter-example	14	16	18
	Execution time (ms)	271	419	77
A* w=0.7 / Reductive Queue	Expanded states	86,260	43,710	590
	Counter-example	15	16	18
	Execution time (ms)	17,141	10,445	63
A* w=0.6 / Reductive Queue	Expanded states	1,395,616	255,467	797
	Counter-example	13	16	18
	Execution time (ms)	215,434	66,037	77
A* w=0.9 / Reductive Queue Memory	Expanded states	58	30	994
	Counter-example	16	16	18
	Execution time (ms)	54	43	91

Continued on next page

Continued from previous page				
A* w=0.75 / Reductive Queue Memory	Expanded states	73	30	817
	Counter-example	16	16	18
	Execution time (ms)	54	43	83
A* w=0.7 / Reductive Queue Memory	Expanded states	116	28	817
	Counter-example	14	16	18
	Execution time (ms)	62	41	83
A* w=0.6 / Reductive Queue Memory	Expanded states	36,950	3567	791
	Counter-example	13	16	18
	Execution time (ms)	12,812	1197	81
A* w=0.9 / Queue Difference	Expanded states	262,481	1,248,955	241
	Counter-example	18	16	18
	Execution time (ms)	66,313	286,281	44
A* w=0.75 / Queue Difference	Expanded states	831,652	670,971	755
	Counter-example	14	16	18
	Execution time (ms)	157,897	169,804	74
A* w=0.7 / Queue Difference	Expanded states	3,315,432	869,952	608
	Counter-example	13	16	18
	Execution time (ms)	443,438	213,873	65
A* w=0.6 / Queue Difference	Expanded states	3,300,044	771,059	767
	Counter-example	13	16	18
	Execution time (ms)	442,805	193,303	75
A* w=0.9 / Queue Difference Memory	Expanded states	276,903	281,272	777
	Counter-example	13	16	18
	Execution time (ms)	66,935	79,264	78
A* w=0.75 / Queue Difference Memory	Expanded states	3,245,379	928,105	759
	Counter-example	13	16	18
	Execution time (ms)	460,999	234,911	76
A* w=0.7 / Queue Difference Memory	Expanded states	3,289,511	893,905	754
	Counter-example	13	16	18
	Execution time (ms)	471,028	228,942	76
A* w=0.6 / Queue Difference Memory	Expanded states	3,889,076	909,852	993
	Counter-example	13	16	18
	Execution time (ms)	537,296	232,333	91

Table B.3: Results of additional searches for models in the case studies.

B.4 Errors where DFS is faster

Table B.4 lists detailed results for models where depth-first search succeed heuristic guided search with regard to expanded nodes and execution time. Execution times are averages of 10 consecutive executions.

Task	Data	Token ring, broken chain	Philosophers, no invert	Train, none can pass
Modere with PO	Expanded states	35	31	13
	Counter-example	15	31	13
	Execution time (ms)	42	66	44
Modere without PO	Expanded states	35	31	13
	Counter-example	15	31	13
	Execution time (ms)	43	66	48
Depth First	Expanded states	15	499	13
	Counter-example	15	499	13
	Execution time (ms)	22	350	25
Breadth First	Expanded states	236,769	388,234	106
	Counter-example	11	31	13
	Execution time (ms)	13,678	40,918	27
Pure Heuristic / Queue Size	Expanded states	31	230	17
	Counter-example	14	31	13
	Execution time (ms)	28	61	25
Pure Heuristic / Empty Queue	Expanded states	14	863	29
	Counter-example	11	76	13
	Execution time (ms)	21	121	26
Pure Heuristic / Current Queue Size	Expanded states	315	1205	31
	Counter-example	14	31	13
	Execution time (ms)	61	155	26
Pure Heuristic / Reductive Queue	Expanded states	191	213	18
	Counter-example	18	31	13
	Execution time (ms)	35	64	25
Pure Heuristic / Reductive Queue Memory	Expanded states	31	609	21
	Counter-example	14	76	13
	Execution time (ms)	27	118	25

Continued on next page

Continued from previous page				
Pure Heuristic / Queue Difference	Expanded states	54	748	56
	Counter-example	11	31	13
	Execution time (ms)	30	133	26
Pure Heuristic / Queue Difference Memory	Expanded states	54	540	58
	Counter-example	11	31	13
	Execution time (ms)	30	115	26
A* /Queue Size	Expanded states	15,595	198,596	104
	Counter-example	11	31	13
	Execution time (ms)	2461	27,574	27
A* /Empty Queue	Expanded states	113,340	232,372	104
	Counter-example	11	31	13
	Execution time (ms)	9768	32,039	27
A* /Current Queue Size	Expanded states	231,916	388,408	106
	Counter-example	11	31	13
	Execution time (ms)	16,664	49,808	27
A* /Reductive Queue	Expanded states	169,443	289,376	104
	Counter-example	11	31	13
	Execution time (ms)	14,901	43,710	27
A* /Reductive Queue Memory	Expanded states	15,192	200,510	105
	Counter-example	11	31	13
	Execution time (ms)	3035	34,091	27
A* /Queue Difference	Expanded states	211,565	270,378	104
	Counter-example	11	22	13
	Execution time (ms)	17,902	41,617	27
A* /Queue Difference Memory	Expanded states	218,856	0.0	105
	Counter-example	11	0.0	13
	Execution time (ms)	19,190	5	27
A* w=0.8 / Queue Size	Expanded states	63	1724	23
	Counter-example	14	31	13
	Execution time (ms)	29	303	25
A* w=0.8 / Empty Queue	Expanded states	372	5156	81
	Counter-example	11	31	13
	Execution time (ms)	47	579	27
A* w=0.8 / Current Queue Size	Expanded states	4308	257,403	44
	Counter-example	13	31	13
	Execution time (ms)	535	33,460	26

Continued on next page

Continued from previous page				
A* w=0.8 / Reductive Queue	Expanded states	238	46,548	22
	Counter-example	13	31	13
	Execution time (ms)	52	6971	25
A* w=0.8 / Reductive Queue Memory	Expanded states	75	5635	37
	Counter-example	11	31	13
	Execution time (ms)	42	838	26
A* w=0.8 / Queue Difference	Expanded states	2941	243,326	80
	Counter-example	11	31	13
	Execution time (ms)	720	37,598	27
A* w=0.8 / Queue Difference Memory	Expanded states	77,679	293,570	80
	Counter-example	11	31	13
	Execution time (ms)	8872	47,521	27

Table B.4: Results for models where DFS outperformed the heuristic searches.

Appendix C

Results for Queue Overflow Error

In this appendix we present experimental results for models with queue overflow. Guided searches are using inverted heuristics. Executions times are the average of 10 consecutive searches. An inverted heuristic is denoted by [heuristic name]$_{inv}$.

Experiments are executed on the following models:

1. **Producer Consumer** A send statement should be in an *else* block, but is executed unconditionally instead.
2. **Forgetful Philosophers** If a philosopher forgets temporarily, he will try to remember twice. Eventually the model may either deadlock or queue overflow, depending on which is reached first.
3. **Needham with dual retry.** If the verification of a nonce fails in step 7, the responder will attempt open a new conversion and ask the initiator to do so as well. Eventually the model may either deadlock or queue overflow, depending on which is reached first.
4. **Token Ring with overflow.** Token ring where a leader will update his child twice. Eventually the model will deadlock.

Results are listed in Table C.1. Execution times are averages of 10 consecutive executions.

Task	Data	Producer Consumer w/overflow	Dining Philosophers w/overflow	Needham-Schroeder w/overflow and deadlock	Token Ring w/overflow
Modere with PO	Expanded states	72	151,022	462	316
	Counter-example	72	38	117	9
	Execution time (ms)	50	8955	91	30
Modere without PO	Expanded states	72	323,947	471	1728
	Counter-example	72	37	117	9
	Execution time (ms)	50	29,401	91	95
Depth First	Expanded states	25	2,816,138	10,318	1325
	Counter-example	25	3628	2623	16
	Execution time (ms)	33	315,658	1680	72
Breadth First	Expanded states	1130	53,312	1203	6663
	Counter-example	21	7	18	6
	Execution time (ms)	62	14,406	82	857
Pure Heuristic / Queue Size$_{inv}$	Expanded states	49	68	13,653	762
	Counter-example	23	23	76	12
	Execution time (ms)	34	64	1161	70
Pure Heuristic / Empty Queue$_{inv}$	Expanded states	62	403	1977	21,594
	Counter-example	27	14	155	13
	Execution time (ms)	35	161	285	1740
Pure Heuristic / Current Queue Size$_{inv}$	Expanded states	258	42	2637	5425
	Counter-example	81	7	139	8
	Execution time (ms)	54	42	334	520
Pure Heuristic / Reductive Queue$_{inv}$	Expanded states	55	106	41,321	108
	Counter-example	23	17	80	11
	Execution time (ms)	34	78	3768	35
Pure Heuristic / Reductive Queue Memory$_{inv}$	Expanded states	45	109	328,101	555
	Counter-example	23	22	136	12
	Execution time (ms)	34	77	36,000	68

Continued on next page

Continued from previous page					
Pure Heuristic / Queue Difference$_{inv}$	Expanded states	83	396	23,637	2482
	Counter-example	23	11	85	17
	Execution time (ms)	35	158	2530	220
Pure Heuristic / Queue Difference Memory$_{inv}$	Expanded states	57	1601	7797	3982
	Counter-example	21	19	83	7
	Execution time (ms)	33	558	912	351
A* / Queue Size$_{inv}$	Expanded states	1087	4192	1506	4091
	Counter-example	21	7	18	6
	Execution time (ms)	70	1555	114	713
A* / Empty Queue$_{inv}$	Expanded states	1340	12,749	1506	14,522
	Counter-example	21	7	18	6
	Execution time (ms)	80	4676	115	1830
A* / Current Queue$_{inv}$	Expanded states	1386	1992	1281	6189
	Counter-example	21	7	18	6
	Execution time (ms)	81	740	94	961
A* / Reductive Queue$_{inv}$	Expanded states	1259	161,118	1363	1892
	Counter-example	21	7	18	6
	Execution time (ms)	81	62,566	111	429
A* / Reductive Queue Memory	Expanded states	1086	6939	1506	4066
	Counter-example	21	7	18	6
	Execution time (ms)	79	3193	134	866
A* / Queue Difference	Expanded states	1181	108,503	1380	3285
	Counter-example	21	7	18	6
	Execution time (ms)	76	42,357	111	648
A* / Queue Difference Memory	Expanded states	1179	132,562	1380	3285
	Counter-example	21	7	18	6
	Execution time (ms)	79	52,594	113	665
A* w=0.8 / Queue Size$_{inv}$	Expanded states	74	66	4721	454
	Counter-example	23	17	18	6
	Execution time (ms)	34	55	381	90
A* w=0.8 / Empty Queue$_{inv}$	Expanded states	216	6170	2978	38,459
	Counter-example	23	8	18	8
	Execution time (ms)	40	2190	247	3423

Continued on next page

Continued from previous page					
A* w=0.8 / Current Queue Size$_{inv}$	Expanded states	607	52	530	11,682
	Counter-example	21	7	18	7
	Execution time (ms)	54	47	58	1237
A* w=0.8 / Reductive Queue$_{inv}$	Expanded states	611	765	1209	2712
	Counter-example	23	25	18	7
	Execution time (ms)	57	303	107	371
A* w=0.8 / Reductive Queue Memory$_{inv}$	Expanded states	69	98	10,532	109
	Counter-example	23	23	18	6
	Execution time (ms)	35	81	1060	41
A* w=0.8 / Queue Difference$_{inv}$	Expanded states	746	8283	1283	2902
	Counter-example	21	7	18	6
	Execution time (ms)	60	3272	111	386
A* w=0.8 / Queue Difference Memory$_{inv}$	Expanded states	1159	359,963	2092	1938
	Counter-example	21	7	18	6
	Execution time (ms)	81	145,591	171	447
Pure Heuristic / Empty Queue- Queue Size$_{inv}$	Expanded states	51	92	733	233
	Counter-example	23	19	91	14
	Execution time (ms)	34	69	146	48
Pure Heuristic / Empty Queue- Current Queue Size$_{inv}$	Expanded states	75	36	2202	4370
	Counter-example	27	7	98	13
	Execution time (ms)	36	42	326	435
Pure Heuristic / Empty Queue- Reductive Queue$_{inv}$	Expanded states	50	301	1687	607
	Counter-example	23	25	101	7
	Execution time (ms)	34	151	268	100
Pure Heuristic / Empty Queue- Queue Difference$_{inv}$	Expanded states	54	1520	2273	2796
	Counter-example	23	17	83	14
	Execution time (ms)	34	612	362	303
Pure Heuristic / Queue Size- Current Queue Size$_{inv}$	Expanded states	61	43	1101	2070
	Counter-example	31	7	88	14
	Execution time (ms)	37	45	177	204
Pure Heuristic / Queue Size- Reductive Queue$_{inv}$	Expanded states	53	202	68,862	100
	Counter-example	23	22	124	12
	Execution time (ms)	34	114	7813	36

Continued on next page

Continued from previous page					
Pure Heuristic / Queue Size- Queue Difference$_{inv}$	Expanded states	51	54	49,967	198
	Counter-example	23	20	76	13
	Execution time (ms)	34	57	5107	47
A* / Empty Queue- Queue Size$_{inv}$	Expanded states	1319	9332	1508	9811
	Counter-example (actions)	21	7	18	6
	Execution time (ms)	84	3891	125	1524
A* / Empty Queue- Current Queue$_{inv}$	Expanded states	1421	4251	1329	10,892
	Counter-example (actions)	21	7	18	6
	Execution time (ms)	88	1763	106	1584
A* / Empty Queue- Reductive Queue$_{inv}$	Expanded states	1209	95,093	1363	6040
	Counter-example (actions)	21	7	18	6
	Execution time (ms)	84	41,943	120	1138
A* / Empty Queue- Queue Difference	Expanded states	1411	56,559	1380	10,363
	Counter-example (actions)	21	7	18	6
	Execution time (ms)	91	25,043	121	1781
A* / Queue Size- Current Queue$_{inv}$	Expanded states	1031	3579	1380	6836
	Counter-example (actions)	21	7	18	6
	Execution time (ms)	70	1450	110	1135
A* / Queue Size- Reductive Queue$_{inv}$	Expanded states	1361	105,782	1362	1895
	Counter-example (actions)	21	7	18	6
	Execution time (ms)	88	45,801	119	468
A* / Queue Size- Queue Difference	Expanded states	986	11,511	1382	3269
	Counter-example (actions)	21	7	18	6
	Execution time (ms)	71	5210	120	702
A* w=0.8 / Empty Queue- Queue Size$_{inv}$	Expanded states	183	392	3807	12,031
	Counter-example	23	18	18	12
	Execution time (ms)	39	189	347	1323
A* w=0.8 / Empty Queue- Current Queue Size$_{inv}$	Expanded states	524	61	1876	11,157
	Counter-example	21	7	18	7
	Execution time (ms)	53	54	157	1305
A* w=0.8 / Empty Queue- Reductive Queue$_{inv}$	Expanded states	164	2806	1990	4583
	Counter-example	21	13	18	7
	Execution time (ms)	38	1319	179	811

Continued on next page

Continued from previous page					
A* w=0.8 / Empty	Expanded states	696	6979	2093	10,453
Queue- Queue	Counter-example	21	8	18	6
Difference$_{inv}$	Execution time (ms)	63	3221	182	1633
A* w=0.8 / Queue	Expanded states	179	38	937	9220
Size- Current Queue	Counter-example	25	7	18	12
Size$_{inv}$	Execution time (ms)	39	43	90	1200
A* w=0.8 / Queue	Expanded states	102	177	1277	1140
Size- Reductive	Counter-example	23	21	18	6
Queue$_{inv}$	Execution time (ms)	36	113	122	216
A* w=0.8 / Queue	Expanded states	88	195	1294	601
Size- Queue	Counter-example	21	18	18	6
Difference$_{inv}$	Execution time (ms)	35	105	121	136

Table C.1: Results for queue overflow experiments with inverted heuristics.

Appendix D

Rebeca Models

This appendix contains the key models used in this study.

D.1 Dijkstra's Token Ring

```
/*
 * 2011 Reykjavik University
 * Author: Steinar Hugi Sigurdarson, steinar@steinar.is
 *
 * Djikstraās self-stabilizing token ring
 *
 * Source: Dijkstra EW. Self-stabilizing Systems in Spite of Distributed Control
 *       An On-Site Data Management System Application in. Communications of
 *       the ACM. 1974;17(11).
 *
 */

reactiveclass Node(6) {
   knownrebecs {
      Node child;
   }
   statevars {
      boolean isLeader;
      int value;
   }
   msgsrv initial(boolean lead) {
```

```
      value = ?(0, 1, 2, 3, 4); // Indexes of the nodes
      isLeader = lead;
      child.update(value);
   }
   msgsrv update(int parentValue) {
      if(isLeader && value == parentValue) {
         value = ((value + 1) % 5); // Last number should be the number of nodes
         child.update(value);
      }
      if(!isLeader && value != parentValue) {
         value = parentValue;
         child.update(value);
      }
   }
}

main {
   Node n0(n4):(true);
   Node n1(n0):(false);
   Node n2(n1):(false);
   Node n3(n2):(false);
   Node n4(n3):(false);
   Node n5(n4):(false);
}
```

Listing D.1: Dijkstra's self-stabilizing token ring with 6 nodes. (Result: satisfied)

```
/*
 * 2011 Reykjavik University
 * Author: Steinar Hugi Sigurdarson, steinar@steinar.is
 *
 * Djikstraãs self-stabilizing token ring with added deadlock
 * Deadlock error: Two leaders.
 *
 * Source: Dijkstra EW. Self-stabilizing Systems in Spite of Distributed Control
     An On-Site Data Management
 *       System Application in. Communications of the ACM. 1974;17(11).
 *
 */
```

```
reactiveclass Node(6) {
   knownrebecs {
      Node child;
   }
   statevars {
      boolean isLeader;
      int value;
   }
   msgsrv initial(boolean lead) {
      value = ?(0,1,2,3,4,5); // Indexes of the nodes
      isLeader = lead;
      child.update(value);
   }
   msgsrv update(int parentValue) {
      if(isLeader && value == parentValue) {
         value = ((value + 1) % 6); // Last number should be the number of nodes
         child.update(value);
      }
      if(!isLeader && value != parentValue) {
         value = parentValue;
         child.update(value);
      }
   }
}

main {
   Node n0(n5):(true);
   Node n1(n0):(true);
   Node n2(n1):(false);
   Node n3(n2):(false);
   Node n4(n3):(false);
   Node n5(n4):(false);
}
```

Listing D.2: Dijkstra's self-stabilizing token ring with 6 nodes and two leaders (Result: deadlock)

```
/*
 * 2011 Reykjavik University
 * Author: Steinar Hugi Sigurdarson, steinar@steinar.is
```

```
 *
 * Djikstraās self-stabilizing token ring
 *
 * Source: Dijkstra EW. Self-stabilizing Systems in Spite of Distributed Control
     An On-Site Data Management
 *       System Application in. Communications of the ACM. 1974;17(11).
 *
 */

reactiveclass Node(5) {
   knownrebecs {
      Node child;
   }
   statevars {
      boolean isLeader;
      int value;
   }
   msgsrv initial(boolean lead) {
      value = ?(0, 1, 2, 3, 4); // Indexes of the nodes
      isLeader = lead;
      child.update(value);
   }
   msgsrv update(int parentValue) {
      if(isLeader && value == parentValue) {
         value = ((value + 1) % 5); // Last number should be the number of nodes
         child.update(value);
      }
      if(!isLeader && value != parentValue) {
         value = parentValue;
         child.update(value);
      }
   }
}

main {
   Node n0(n4):(true);
   Node n1(n0):(false);
   Node n2(n1):(false);
   Node n3(n2):(false);
   Node n4(n3):(false);
```

```
}
```

Listing D.3: Dijkstra's self-stabilizing token ring with 5 nodes and leader updating its child twice. (Result: queue overflow)

```
/*
 * 2011 Reykjavik University
 * Author: Steinar Hugi Sigurdarson, steinar@steinar.is
 *
 * Djikstraâs self-stabilizing token ring with added deadlock
 * Deadlock: Ring is broken around node n3
 *
 * Source: Dijkstra EW. Self-stabilizing Systems in Spite of Distributed Control
    An On-Site Data Management
 *       System Application in. Communications of the ACM. 1974;17(11).
 *
 */

reactiveclass Node(6) {
   knownrebecs {
      Node child;
   }
   statevars {
      boolean isLeader;
      int value;
   }
   msgsrv initial(boolean lead) {
      value = ?(0, 1, 2, 3, 4); // Indexes of the nodes
      isLeader = lead;
      child.update(value);
   }
   msgsrv update(int parentValue) {
      if(isLeader && value == parentValue) {
         value = ((value + 1) % 5); // Last number should be the number of nodes
         child.update(value);
      }
      if(!(isLeader) && value != parentValue) {
         value = parentValue;
         child.update(value);
      }
```

```
    }
}

main {
    Node n0(n4):(true);
    Node n1(n0):(false);
    Node n2(n3):(false);
    Node n3(n2):(false);
    Node n4(n3):(false);
}
```

Listing D.4: Dijkstra's self-stabilizing token ring with 5 nodes and a broken chain. (Result: satisfied)

D.2 Dining Philosophers

```
/*
 * 2011 Reykjavik University
 * Author (of this version): Steinar Hugi Sigurdarson, steinar@steinar.is
 *
 * Dining Philosophers (with 4 philosophers)
 *
 * Source: C.a. Hoare, Communicating sequential processes, Communications of
 *   the ACM, vol. 21, 1978, pp. 666-677.
 *
 * Original problem due to E.W. Dijkstra.
 *
 * Original implementation: M. Jaghoori, A. Movaghar, and M. Sirjani, ãModere:
 *   The model-checking engine of Rebeca,ã Proceedings of the 2006, 2006,
 *   pp. 1810-1815.
 *
 */
reactiveclass Philosopher(7)
{
 knownrebecs {
  Fork leftFork;
  Fork rightFork;
 }
 statevars {
  boolean isEating;
  boolean hasLeftFork;
  boolean hasRightFork;
 }
 msgsrv initial() {
  hasLeftFork = false;
  hasRightFork = false;
  isEating = false;
  self.arrive();
 }

 msgsrv arrive() {
  leftFork.request();
 }
```

```
  msgsrv permit() {
    if (sender == leftFork) {
      if (!hasLeftFork) {
        hasLeftFork = true;
        rightFork.request();
      }
    } else {
      if (hasLeftFork && !(hasRightFork)) {
        hasRightFork = true;
        self.eat();
      }
    }
  }

  msgsrv eat() {
    isEating = true;
    self.leave();
  }

  msgsrv leave() {
    hasLeftFork = false;
    hasRightFork = false;
    isEating = false;
    rightFork.release();
    leftFork.release();
    self.arrive();
  }
}

reactiveclass Fork(7) {
  knownrebecs {
    Philosopher philLeft;
    Philosopher philRight;
  }
  statevars {
    boolean leftIsUsing;
    boolean rightIsUsing;
    boolean leftHasRequested;
    boolean rightHasRequested;
  }
```

```
msgsrv initial() {
  leftIsUsing = false;
  rightIsUsing = false;
  leftHasRequested = false;
  rightHasRequested = false;
}

msgsrv request() {
  if (sender == philLeft) {
    if (!leftHasRequested) {
      leftHasRequested = true;
      if (!rightIsUsing) {
        leftIsUsing = true;
        philLeft.permit();
      }
    }
  } else {
    if (!rightHasRequested) {
      rightHasRequested = true;
      if (!leftIsUsing) {
        rightIsUsing = true;
        philRight.permit();
      }
    }
  }
}
msgsrv release() {
  if (sender == philLeft && leftIsUsing){
    leftHasRequested = false;
    leftIsUsing = false;
    if (rightHasRequested) {
      rightIsUsing = true;
      philRight.permit();
    }
  }
  if (sender == philRight && rightIsUsing){
    rightIsUsing = false;
    rightHasRequested = false;
    if (leftHasRequested) {
      leftIsUsing = true;
```

```
      philLeft.permit();
    }
   }
  }
}

main {
  Philosopher phil0(fork0, fork3):(); // Inverted

  Philosopher phil1(fork0, fork1):();
  Philosopher phil2(fork1, fork2):();
  Philosopher phil3(fork2, fork3):();

  Fork fork0(phil0, phil1):();
  Fork fork1(phil1, phil2):();
  Fork fork2(phil2, phil3):();
  Fork fork3(phil3, phil0):();
}
```

Listing D.5: Dining Philosophers with 4 philosophers. (Result: satisfied)

```
/*
 * 2011 Reykjavik University
 * Author (of this version): Steinar Hugi Sigurdarson, steinar@steinar.is
 *
 * Dining Philosophers (with added ability to forget)
 * This version contains a deadlock error.
 *
 * Source: C.a. Hoare, Communicating sequential processes, Communications of
 *   the ACM, vol. 21, 1978, pp. 666-677.
 *
 * Original problem due to E.W. Dijkstra.
 *
 * Original implementation: M. Jaghoori, A. Movaghar, and M. Sirjani, ãModere:
 *   The model-checking engine of Rebeca,ã Proceedings of the 2006, 2006,
 *   pp. 1810-1815.
 *
 */
reactiveclass Philosopher(7)
{
```

```
knownrebecs {
  Fork leftFork;
  Fork rightFork;
}
statevars {
  boolean isEating;
  boolean hasLeftFork;
  boolean hasRightFork;
  int memory;
}
msgsrv initial(){
  hasLeftFork = false;
  hasRightFork = false;
  isEating = false;
  memory = ?(0,1,2);
  self.remember();
}
msgsrv remember() {
  if(memory == 0) {
    // Forgot it completely...
  } else if(memory == 1) {
    // Forgot it temporarily
    memory = ?(0,1,2);
    self.remember();
  } else if(memory == 2) {
    // Remembered
    self.arrive();
  }
}

msgsrv arrive() {
  leftFork.request();
}

msgsrv permit() {
  if (sender == leftFork) {
    if (!hasLeftFork) {
      hasLeftFork = true;
      rightFork.request();
    }
```

```
    } else {
      if (hasLeftFork && !(hasRightFork)) {
        hasRightFork = true;
        self.eat();
      }
    }
  }

  msgsrv eat() {
    isEating = true;
    self.leave();
  }

  msgsrv leave() {
    hasLeftFork = false;
    hasRightFork = false;
    isEating = false;
    rightFork.release();
    leftFork.release();
    self.arrive();
  }
}

reactiveclass Fork(7) {
  knownrebecs {
    Philosopher philLeft;
    Philosopher philRight;
  }
  statevars {
    boolean leftIsUsing;
    boolean rightIsUsing;
    boolean leftHasRequested;
    boolean rightHasRequested;
  }
  msgsrv initial() {
    leftIsUsing = false;
    rightIsUsing = false;
    leftHasRequested = false;
    rightHasRequested = false;
  }
```

```
  msgsrv request() {
    if (sender == philLeft) {
      if (!leftHasRequested) {
        leftHasRequested = true;
        if (!rightIsUsing) {
          leftIsUsing = true;
          philLeft.permit();
        }
      }
    } else {
      if (!rightHasRequested) {
        rightHasRequested = true;
        if (!leftIsUsing) {
          rightIsUsing = true;
          philRight.permit();
        }
      }
    }
  }
  msgsrv release() {
    if (sender == philLeft && leftIsUsing){
      leftHasRequested = false;
      leftIsUsing = false;
      if (rightHasRequested) {
        rightIsUsing = true;
        philRight.permit();
      }
    }
    if (sender == philRight && rightIsUsing){
      rightIsUsing = false;
      rightHasRequested = false;
      if (leftHasRequested) {
        leftIsUsing = true;
        philLeft.permit();
      }
    }
  }
}
```

```
main {
  Philosopher phil0(fork0, fork3):(); // Inverted

  Philosopher phil1(fork0, fork1):();
  Philosopher phil2(fork1, fork2):();
  Philosopher phil3(fork2, fork3):();
  Philosopher phil4(fork3, fork4):();

  Fork fork0(phil0, phil1):();
  Fork fork1(phil1, phil2):();
  Fork fork2(phil2, phil3):();
  Fork fork3(phil3, phil4):();
  Fork fork4(phil4, phil0):();
}
```

Listing D.6: Forgetful Dining Philosophers with 5 philosophers. (Result: deadlock)

```
/*
 * 2011 Reykjavik University
 * Author (of this version): Steinar Hugi Sigurdarson, steinar@steinar.is
 *
 * Dining Philosophers (with added ability to forget)
 * This version contains deadlock and queue overflow errors.
 *
 * Source: C.a. Hoare, Communicating sequential processes, Communications of
 *   the ACM, vol. 21, 1978, pp. 666-677.
 *
 * Original problem due to E.W. Dijkstra.
 *
 * Original implementation: M. Jaghoori, A. Movaghar, and M. Sirjani, ãModere:
 *   The model-checking engine of Rebeca,ã Proceedings of the 2006, 2006,
 *   pp. 1810-1815.
 *
 */
reactiveclass Philosopher(7)
{
  knownrebecs {
    Fork leftFork;
    Fork rightFork;
  }
```

```
statevars {
  boolean isEating;
  boolean hasLeftFork;
  boolean hasRightFork;
  int memory;
}
msgsrv initial(){
  hasLeftFork = false;
  hasRightFork = false;
  isEating = false;
  memory = ?(0,1,2);
  self.remember();
}
msgsrv remember() {
  if(memory == 0) {
    // Forgot it completely...
  } else if(memory == 1) {
    // Forgot it temporarily
    memory = ?(0,1,2);
    self.remember();
    // Queue overflow!
    self.remember();
  } else if(memory == 2) {
    // Remembered
    self.arrive();
  }
}

msgsrv arrive() {
  leftFork.request();
}

msgsrv permit() {
  if (sender == leftFork) {
    if (!hasLeftFork) {
      hasLeftFork = true;
      rightFork.request();
    }
  } else {
    if (hasLeftFork && !(hasRightFork)) {
```

```
      hasRightFork = true;
      self.eat();
     }
    }
  }

  msgsrv eat() {
    isEating = true;
    self.leave();
  }

  msgsrv leave() {
    hasLeftFork = false;
    hasRightFork = false;
    isEating = false;
    rightFork.release();
    leftFork.release();
    self.arrive();
  }
}

reactiveclass Fork(7) {
  knownrebecs {
    Philosopher philLeft;
    Philosopher philRight;
  }
  statevars {
    boolean leftIsUsing;
    boolean rightIsUsing;
    boolean leftHasRequested;
    boolean rightHasRequested;
  }
  msgsrv initial() {
    leftIsUsing = false;
    rightIsUsing = false;
    leftHasRequested = false;
    rightHasRequested = false;
  }

  msgsrv request() {
```

```
    if (sender == philLeft) {
      if (!leftHasRequested) {
        leftHasRequested = true;
        if (!rightIsUsing) {
          leftIsUsing = true;
          philLeft.permit();
        }
      }
    } else {
      if (!rightHasRequested) {
        rightHasRequested = true;
        if (!leftIsUsing) {
          rightIsUsing = true;
          philRight.permit();
        }
      }
    }
  }
  msgsrv release() {
    if (sender == philLeft && leftIsUsing){
      leftHasRequested = false;
      leftIsUsing = false;
      if (rightHasRequested) {
        rightIsUsing = true;
        philRight.permit();
      }
    }
    if (sender == philRight && rightIsUsing){
      rightIsUsing = false;
      rightHasRequested = false;
      if (leftHasRequested) {
        leftIsUsing = true;
        philLeft.permit();
      }
    }
  }
}

main {
  Philosopher phil0(fork0, fork3):(); // Inverted
```

```
 Philosopher phil1(fork0, fork1):();
 Philosopher phil2(fork1, fork2):();
 Philosopher phil3(fork2, fork3):();
 Philosopher phil4(fork3, fork4):();

 Fork fork0(phil0, phil1):();
 Fork fork1(phil1, phil2):();
 Fork fork2(phil2, phil3):();
 Fork fork3(phil3, phil4):();
 Fork fork4(phil4, phil0):();
}
```

Listing D.7: Forgetful Dining Philosophers with 5 philosophers and double remembering when temporarily forgetting. (Result: deadlock/queue overflow)

```
/*
 * 2011 Reykjavik University
 * Author (of this version): Steinar Hugi Sigurdarson, steinar@steinar.is
 *
 * Dining Philosophers (without any deadlock solution)
 *
 * Source: C.a. Hoare, Communicating sequential processes, Communications of
 *   the ACM, vol. 21, 1978, pp. 666-677.
 *
 * Original problem due to E.W. Dijkstra.
 *
 * Original implementation: M. Jaghoori, A. Movaghar, and M. Sirjani, ãModere:
 *   The model-checking engine of Rebeca,ã Proceedings of the 2006, 2006,
 *   pp. 1810-1815.
 *
 */
reactiveclass Philosopher(7)
{
 knownrebecs {
   Fork leftFork;
   Fork rightFork;
 }
 statevars {
   boolean isEating;
```

```
  boolean hasLeftFork;
  boolean hasRightFork;
}
msgsrv initial() {
  hasLeftFork = false;
  hasRightFork = false;
  isEating = false;
  self.arrive();
}

msgsrv arrive() {
  leftFork.request();
}

msgsrv permit() {
  if (sender == leftFork) {
    if (!hasLeftFork) {
      hasLeftFork = true;
      rightFork.request();
    }
  } else {
    if (hasLeftFork && !(hasRightFork)) {
      hasRightFork = true;
      self.eat();
    }
  }
}

msgsrv eat() {
  isEating = true;
  self.leave();
}

msgsrv leave() {
  hasLeftFork = false;
  hasRightFork = false;
  isEating = false;
  rightFork.release();
  leftFork.release();
  self.arrive();
```

```
  }
}

reactiveclass Fork(7) {
  knownrebecs {
    Philosopher philLeft;
    Philosopher philRight;
  }
  statevars {
    boolean leftIsUsing;
    boolean rightIsUsing;
    boolean leftHasRequested;
    boolean rightHasRequested;
  }
  msgsrv initial() {
    leftIsUsing = false;
    rightIsUsing = false;
    leftHasRequested = false;
    rightHasRequested = false;
  }

  msgsrv request() {
    if (sender == philLeft) {
      if (!leftHasRequested) {
        leftHasRequested = true;
        if (!rightIsUsing) {
          leftIsUsing = true;
          philLeft.permit();
        }
      }
    } else {
      if (!rightHasRequested) {
        rightHasRequested = true;
        if (!leftIsUsing) {
          rightIsUsing = true;
          philRight.permit();
        }
      }
    }
  }
```

```
  msgsrv release() {
    if (sender == philLeft && leftIsUsing){
      leftHasRequested = false;
      leftIsUsing = false;
      if (rightHasRequested) {
        rightIsUsing = true;
        philRight.permit();
      }
    }
    if (sender == philRight && rightIsUsing){
      rightIsUsing = false;
      rightHasRequested = false;
      if (leftHasRequested) {
        leftIsUsing = true;
        philLeft.permit();
      }
    }
  }
}

main {
  Philosopher phil0(fork4, fork0):(); // Injected error: These should be inverted

  Philosopher phil1(fork0, fork1):();
  Philosopher phil2(fork1, fork2):();
  Philosopher phil3(fork2, fork3):();
  Philosopher phil4(fork3, fork4):();

  Fork fork0(phil0, phil1):();
  Fork fork1(phil1, phil2):();
  Fork fork2(phil2, phil3):();
  Fork fork3(phil3, phil4):();
  Fork fork4(phil4, phil0):();
}
```

Listing D.8: Dining Philosophers with 5 philosophers and no deadlock prevention. (Result: deadlock)

D.3 Needham-Schroeder

```
/*
 * 2011 Reykjavik University
 * Author: Steinar Hugi Sigurdarson, steinar@steinar.is
 *
 * Rebeca implementation of the Needham-Schroeder Public Key Protocol
 * with two clients and one server.
 *
 * The encryption is value*key and decryption value/key. Due to this
 * simplification the private and public key must be the same.
 *
 *
 * Source
 *         Needham RM, Schroeder MD. Using encryption for authentication
 *         in large networks of computers. Communications of the ACM.
 *         1978;21(12):993-999.
 *
 */

reactiveclass Client(5) {
   knownrebecs {
      Server server;
      Client friend;
   }

   statevars {
      int id;
      int public;
      int private;
      int nonce;

      int idOther;
      int nonceOther;
      int pkOther;

      int pkServer;
   }
```

```
msgsrv initial(int _id, int _idOther, int _public, int _private, int
    _pkServer) {
   id = _id;
   public = _public;
   private = _private;
   nonce = 0;
   pkServer = _pkServer;

   idOther = _idOther;
   nonceOther = 0;
   pkOther = 0;

   self.open();
}

msgsrv open() {
   nonceOther = 0;
   pkOther = 0;

   // Reset nonce
   nonce = ?(1,2,3);
   nonce = nonce + id;


   // Go to step 1: A,B -> S
   if(idOther > 0) {
      server.step_1(id, idOther);
   }
}

// Step 1 is on server

// Come to Step 2: {pkb, B}_skas
msgsrv step_2(int _pkb_skas, int _id_skas)
{
   if(sender == server) { // Check sender
      if((_id_skas/pkServer) == idOther) { // Check decryption
         pkOther = _pkb_skas/pkServer;

         // Go to step 3: {Ia, A}_pkServer -> B
```

```
            friend.step_3(nonce*pkServer, id*pkServer);
         } // else: decryption error
      } // else: incorrect sender
   }

   msgsrv step_3(int _ia_pkServer, int _id_pkServer) {
      if(sender == friend) {
         nonceOther = _ia_pkServer/pkServer;
         idOther = (_id_pkServer/pkServer);

         // Go to step 4: B, A -> S
         server.step_4(id, idOther);
      } // else: incorrect sender
   }

   // Step 4 is on server

   // Come to Step 5: {pka, A}_skas
   msgsrv step_5(int _pka_skas, int _ida_skas) {
      if(sender == server) {
         // Go to step 6: {Ia, Ib}_pka -> A
         pkOther = _pka_skas/pkServer;
         idOther = _ida_skas/pkServer;

         // Reset nonce
         nonce = ?(1,2,3);
         nonce = nonce + id;

         friend.step_6(nonceOther*pkOther, nonce*pkOther);
      } // else: incorrect sender
   }

   msgsrv step_6(int _ia_pka, int _ib_pka) {
      if(sender == friend) {
         if((_ia_pka/private) == nonce) { // Check encryption
            nonceOther = (_ib_pka/private);
            // Go to step 7: {Ib}_pkb -> B
            friend.step_7(nonceOther*pkOther);
         } // else: decryption error
      } // else: incorrect sender
```

```
    }


    msgsrv step_7(int _ib_pkb) {
        if(sender == friend) {
            if((_ib_pkb)/private == nonce) { // Check encryption
                // Done
                self.open(); // Repeat
            } // else: decryption error
        } // else: incorrect sender
    }
}

reactiveclass Server(6) {
    knownrebecs {
        Client client_a;
        Client client_b;
    }

    statevars {
        int id;
        int idA;
        int idB;
        int public;
        int private;
        int pka;
        int pkb;
    }

    msgsrv initial(int _id, int _idA, int _idB, int _public, int _private, int
         _pka, int _pkb) {
        id = _id;
        public = _public;
        private = _private;

        pka = _pka;
        pkb = _pkb;

        idA = _idA;
        idB = _idB;
```

```
   }

   // Step 1 and step 4
   msgsrv step_1(int _idSender, int _id) {
      // Go to Step 2: {pkb, B}_skas
      if(sender == client_a) {
         client_a.step_2(pkb*private, _id*private);
      }
      else if(sender == client_b) {
         client_b.step_2(pka*private, _id*private);
      }
   }
   msgsrv step_4(int _idSender, int _idReceiver) {
      // Go to Step 5: {pka, A}_skas
      if(sender == client_a) {
         client_a.step_5(pkb*private, _idReceiver*private);
      }
      else if(sender == client_b) {
         client_b.step_5(pka*private, _idReceiver*private);
      }
   }
}

main {
   // NOTE: Due to the simplicity of the encryption/decryption pub must be equal
       to prv
   //                 ID,  ID_A, ID_B, pub,  prv,  pka,  pkb
   Server server (a,b) :(1, 2,   3,    10,   10,   20,   30);

   //                 ID,  ID_other, pub, prv, pkServer
   Client a   (server,b):(2, 3,      20,   20,  10);
   Client b   (server,a):(3, 0,      30,   30,  10);
}
```

Listing D.9: Needham-Schroeder Public Key Protocol. (Result: satisfied)

```
/*
 * 2011 Reykjavik University
 * Author: Steinar Hugi Sigurdarson, steinar@steinar.is
 *
```

```
* Rebeca implementation of the Needham-Schroeder Public Key Protocol
* with two clients and one server.
*
* This version contains a deadlock error.
*
* The encryption is value*key and decryption value/key. Due to this
* simplification the private and public key must be the same.
*
*
* Source
*         Needham RM, Schroeder MD. Using encryption for authentication
*         in large networks of computers. Communications of the ACM.
*         1978;21(12):993-999.
*
*/


reactiveclass Client(5) {
   knownrebecs {
      Server server;
      Client friend;
   }

   statevars {
      int id;
      int public;
      int private;
      int nonce;

      int idOther;
      int nonceOther;
      int pkOther;

      int pkServer;
   }

   msgsrv initial(int _id, int _idOther, int _public, int _private, int
       _pkServer) {
      id = _id;
      public = _public;
```

```
    private = _private;
    nonce = 0;
    pkServer = _pkServer;

    idOther = _idOther;
    nonceOther = 0;
    pkOther = 0;

    self.open();
}

msgsrv open() {
    nonceOther = 0;
    pkOther = 0;

    // Reset nonce
    nonce = ?(1,2,3);
    nonce = nonce + id;


    // Go to step 1: A,B -> S
    if(idOther > 0) {
        server.step_1(id, idOther);
    }
}

// Step 1 is on server

// Come to Step 2: {pkb, B}_skas
msgsrv step_2(int _pkb_skas, int _id_skas)
{
    if(sender == server) { // Check sender
        if((_id_skas/pkServer) == idOther) { // Check decryption
            pkOther = _pkb_skas/pkServer;

            // Go to step 3: {Ia, A}_pkServer -> B
            friend.step_3(nonce*pkServer, id*pkServer);
        } // else: decryption error
    } // else: incorrect sender
}
```

```
msgsrv step_3(int _ia_pkServer, int _id_pkServer) {
   if(sender == friend) {
      nonceOther = _ia_pkServer/pkServer;
      idOther = (_id_pkServer/pkServer);

      // Go to step 4: B, A -> S
      server.step_4(id, idOther);
   } // else: incorrect sender
}

// Step 4 is on server

// Come to Step 5: {pka, A}_skas
msgsrv step_5(int _pka_skas, int _ida_skas) {
   if(sender == server) {
      // Go to step 6: {Ia, Ib}_pka -> A
      pkOther = _pka_skas/pkServer;
      idOther = _ida_skas/pkServer;

      // Reset nonce
      nonce = ?(1,2,3);
      nonce = nonce + id;

      friend.step_6(nonceOther*pkOther, nonce*pkOther);
   } // else: incorrect sender
}

msgsrv step_6(int _ia_pka, int _ib_pka) {
   if(sender == friend) {
      if((_ia_pka/private) == nonce) { // Check encryption
         nonceOther = (_ib_pka/private);
         // Go to step 7: {Ib}_pkb -> B
         friend.step_7(nonceOther*pkOther);
      } // else: decryption error
   } // else: incorrect sender
}


msgsrv step_7(int _ib_pkb) {
```

```
      if(sender == friend) {
         // In the deadlock version situation check will fail since the
         if((_ib_pkb)/private == nonce) { // Check encryption
            // Done
            self.open(); // Repeat
         } // else: decryption error
      } // else: incorrect sender
   }
}

reactiveclass Server(6) {
   knownrebecs {
      Client client_a;
      Client client_b;
   }

   statevars {
      int id;
      int idA;
      int idB;
      int public;
      int private;
      int pka;
      int pkb;
   }

   msgsrv initial(int _id, int _idA, int _idB, int _public, int _private, int
       _pka, int _pkb) {
      id = _id;
      public = _public;
      private = _private;

      pka = _pka;
      pkb = _pkb;

      idA = _idA;
      idB = _idB;
   }

   // Step 1 and step 4
```

```
   msgsrv step_1(int _idSender, int _id) {
      // Go to Step 2: {pkb, B}_skas
      if(sender == client_a) {
         client_a.step_2(pkb*private, _id*private);
      }
      else if(sender == client_b) {
         client_b.step_2(pka*private, _id*private);
      }
   }
   msgsrv step_4(int _idSender, int _idReceiver) {
      // Go to Step 5: {pka, A}_skas
      if(sender == client_a) {
         client_a.step_5(pkb*private, _idReceiver*private);
      }
      else if(sender == client_b) {
         client_b.step_5(pka*private, _idReceiver*private);
      }
   }
}

main {
   // NOTE: Due to the simplicity of the encryption/decryption pub must be equal
       to prv
   //                 ID,  ID_A, ID_B, pub,  prv,  pka,  pkb
   Server server (a,b) :(1, 2,   3,    10,   10,   20,   30);

   //                 ID,  ID_other, pub, prv, pkServer
   Client a   (server,b):(2, 3,     20,   20,  10);
   // Since ID_other of both clients > 0 they will run at the same time. The
       implementation cannot handle that and will eventually deadlock.
   Client b   (server,a):(3, 2,     30,   30,  10);

}
```

Listing D.10: Needham-Schroeder Public Key Protocol with simultaneous conversations. (Result: deadlock)

```
/*
 * 2011 Reykjavik University
 * Author: Steinar Hugi Sigurdarson, steinar@steinar.is
```

```
*
* Rebeca implementation of the Needham-Schroeder Public Key Protocol
* with two clients and one server.
*
* This version contains both a queue-overflow error and deadlock error.
*
* The encryption is value*key and decryption value/key. Due to this
* simplification the private and public key must be the same.
*
*
* Source
*         Needham RM, Schroeder MD. Using encryption for authentication
*         in large networks of computers. Communications of the ACM.
*         1978;21(12):993-999.
*
*/

reactiveclass Client(5) {
   knownrebecs {
      Server server;
      Client friend;
   }

   statevars {
      int id;
      int public;
      int private;
      int nonce;

      int idOther;
      int nonceOther;
      int pkOther;

      int pkServer;
   }

   msgsrv initial(int _id, int _idOther, int _public, int _private, int
       _pkServer) {
      id = _id;
      public = _public;
```

```
    private = _private;
    nonce = 0;
    pkServer = _pkServer;

    idOther = _idOther;
    nonceOther = 0;
    pkOther = 0;

    self.open();
}

msgsrv open() {
    nonceOther = 0;
    pkOther = 0;

    // Reset nonce
    nonce = ?(1,2,3);
    nonce = nonce + id;


    // Go to step 1: A,B -> S
    if(idOther > 0) {
        server.step_1(id, idOther);
    }
}

// Step 1 is on server

// Come to Step 2: {pkb, B}_skas
msgsrv step_2(int _pkb_skas, int _id_skas)
{
    if(sender == server) { // Check sender
        if((_id_skas/pkServer) == idOther) { // Check decryption
            pkOther = _pkb_skas/pkServer;

            // Go to step 3: {Ia, A}_pkServer -> B
            friend.step_3(nonce*pkServer, id*pkServer);
        } // else: decryption error
    } // else: incorrect sender
}
```

```
msgsrv step_3(int _ia_pkServer, int _id_pkServer) {
   if(sender == friend) {
      nonceOther = _ia_pkServer/pkServer;
      idOther = (_id_pkServer/pkServer);

      // Go to step 4: B, A -> S
      server.step_4(id, idOther);
   } // else: incorrect sender
}

// Step 4 is on server

// Come to Step 5: {pka, A}_skas
msgsrv step_5(int _pka_skas, int _ida_skas) {
   if(sender == server) {
      // Go to step 6: {Ia, Ib}_pka -> A
      pkOther = _pka_skas/pkServer;
      idOther = _ida_skas/pkServer;

      // Reset nonce
      nonce = ?(1,2,3);
      nonce = nonce + id;

      friend.step_6(nonceOther*pkOther, nonce*pkOther);
   } // else: incorrect sender
}

msgsrv step_6(int _ia_pka, int _ib_pka) {
   if(sender == friend) {
      if((_ia_pka/private) == nonce) { // Check encryption
         nonceOther = (_ib_pka/private);
         // Go to step 7: {Ib}_pkb -> B
         friend.step_7(nonceOther*pkOther);
      } // else: decryption error
   } // else: incorrect sender
}


msgsrv step_7(int _ib_pkb) {
```

```
      if(sender == friend) {
         if((_ib_pkb)/private == nonce) { // Check encryption
            // Done
            self.open(); // Repeat
         } // else: decryption error
         else {
            // Added queue-overflow error
            friend.open();
            self.open();
         }
      } // else: incorrect sender
   }
}

reactiveclass Server(6) {
   knownrebecs {
      Client client_a;
      Client client_b;
   }

   statevars {
      int id;
      int idA;
      int idB;
      int public;
      int private;
      int pka;
      int pkb;
   }

   msgsrv initial(int _id, int _idA, int _idB, int _public, int _private, int
       _pka, int _pkb) {
      id = _id;
      public = _public;
      private = _private;

      pka = _pka;
      pkb = _pkb;

      idA = _idA;
```

```
      idB = _idB;
   }

   // Step 1 and step 4
   msgsrv step_1(int _idSender, int _id) {
      // Go to Step 2: {pkb, B}_skas
      if(sender == client_a) {
         client_a.step_2(pkb*private, _id*private);
      }
      else if(sender == client_b) {
         client_b.step_2(pka*private, _id*private);
      }
   }
   msgsrv step_4(int _idSender, int _idReceiver) {
      // Go to Step 5: {pka, A}_skas
      if(sender == client_a) {
         client_a.step_5(pkb*private, _idReceiver*private);
      }
      else if(sender == client_b) {
         client_b.step_5(pka*private, _idReceiver*private);
      }
   }
}

main {
   // NOTE: Due to the simplicity of the encryption/decryption pub must be equal
       to prv
   //                ID,  ID_A, ID_B, pub,  prv,  pka,  pkb
   Server server (a,b) :(1, 2,   3,    10,   10,   20,   30);

   //                ID,  ID_other, pub, prv, pkServer
   Client a   (server,b):(2, 3,     20,   20,  10);
   Client b   (server,a):(3, 2,     30,   30,  10);
}
```

Listing D.11: Needham-Schroeder Public Key Protocol with simultaneous conversations and dual retry. (Result: deadlock/queue overflow)

D.4 Bridge Controller

```
/**
* Source: Sirjani, M., De Boer, F., & Movaghar, A. (2005). Modular verification
    of a component-based actor language. Journal of Universal Computer Science,
    11(10), 1695-1717
* Modifications: Line 90: Commented out send statement, introducing a deadlock
    error.
*/
reactiveclass BridgeController(5)
{
      knownrebecs
      {
            Train t1;
            Train t2;
      }

      statevars
      {
            boolean isWaiting1;
            boolean isWaiting2;
            boolean signal1;
            boolean signal2;
      }

      msgsrv initial()
      {
            signal1 = false;  /* red */
            signal2 = false;  /* red */
            isWaiting1 = false;
            isWaiting2 = false;
      }

      msgsrv Arrive(){
            if (sender == t1){
                  if (signal2 == false){
                        signal1 = true; /* green */
                        t1.YouMayPass();
                  }
                  else{
```

```
                        isWaiting1 = true;
                  }
            }
            else{
                  if (signal1 == false){
                        signal2 = true; /* green */
                        t2.YouMayPass();
                  }
                  else{
                        isWaiting2 = true;
                  }
            }
      }

      msgsrv Leave(){
            if (sender == t1){
                  signal1 = false;  /* red */
                  if (isWaiting2){
                        signal2 = true;
                        t2.YouMayPass();
                        isWaiting2 = false;
                  }
            }
            else{
                  signal2 = false;  /* red */
                  if (isWaiting1){
                        signal1 = true;
                        t1.YouMayPass();
                        isWaiting1 = false;
                  }
            }
      }
}

reactiveclass Train(3)
{
      knownrebecs
      {
            BridgeController controller;
      }
```

```
	statevars
	{
		boolean onTheBridge;
	}

	msgsrv initial()
	{
		onTheBridge = false;
		self.Passed();
	}

	msgsrv YouMayPass(){
		onTheBridge = true;
		//self.Passed();
	}

	msgsrv Passed(){
		onTheBridge = false;
		controller.Leave();
		self.ReachBridge();
	}

	msgsrv ReachBridge(){
		controller.Arrive();
	}
}


main
{
	Train train1(theController):();
	Train train2(theController):();
	BridgeController theController(train1, train2):();
}
```

Listing D.12: Bridge Controller with deadlock error (Result: deadlock)

School of Computer Science
Reykjavík University
Menntavegi 1
101 Reykjavík, Iceland
Tel. +354 599 6200
Fax +354 599 6201
www.reykjavikuniversity.is
ISSN 1670-8539