# Leveraging distributed explicit-state model checking for practical verification of liveness in hardware protocols

by

Brad Bingham

M.Sc., The University of British Columbia, 2007

B.Sc., The University of Victoria, 2005

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

**DOCTOR OF PHILOSOPHY**

in

THE FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES

(Computer Science)

**THE UNIVERSITY OF BRITISH COLUMBIA**

**(Vancouver)**

April 2015

# Abstract

Protocol verification is a key component to hardware and software design. The proliferation of concurrency in modern designs stresses the need for accurate protocol models and scalable verification tools. Model checking is an approach for automatically verifying properties of designs, the main limitation of which is state-space explosion. As such, automatic verification of these designs can quickly exhaust the memory of a single computer.

This thesis presents PReach, a distributed explicit-state model checker, designed to robustly harness the aggregate computing power of large clusters. The initial version verified *safety* properties, which hold if no error states can be reached. PReach has been demonstrated to run on hundreds of machines and explore state space sizes up to 90 billion, the largest published to date.

Liveness is an important class of properties for hardware system correctness which, unlike safety, expresses more elaborate temporal reasoning. However, model checking of liveness is more computationally complex, and exacerbates scalability issues as compared with safety. The main thesis contribution is the extension of PReach to verify two key liveness-like properties of practical interest: *deadlock-freedom* and *response*. Our methods leverage the scalability and robustness of PReach and strike a balance between tractable verification for large models and catching liveness violations.

Deadlock-freedom holds if from all reachable system states, there exists a sequence of actions that will complete all pending transactions. We find that checking this property is only a small overhead as compared to safety checking. We also provide a technique for establishing that deadlock-freedom holds of a *parameterized system* — a system with a variable number of entities.

*Response* is a stronger property than deadlock-freedom and is the most common liveness property of interest. In practical cases, *fairness* must be imposed on system models when model checking response to exclude those execution traces deemed inconsistent with the expected underlying hardware. We implemented a novel twist on established model checking algorithms, to target response properties with action-based fairness. This implementation vastly out-performs competing tools.

This thesis shows that tractable verification of interesting liveness properties in large protocol models is possible.

# Preface

The work of this thesis was mostly conducted at the Integrated Systems Design Lab at the University of British Columbia, Point Grey campus. Some development of the PReach tool and experiments were done at Intel Corporation (Ronler Acres), in Hillsboro, Oregon. Chapters 3 – 6 are based on publications that have already appeared. Accordingly, some of the text and figures are based upon material written or drawn by my co-authors. These publications, along with my role in the research and writing are described in detail below.

1. Chapter 3 is based on: Brad Bingham, Jesse Bingham, Flavio M. de Paula, John Erickson, Gaurav Singh, and Mark Reitblatt. Industrial strength distributed explicit state model checking. In *Parallel and Distributed Model Checking (PDMC)*, pages 28–36, IEEE Computer Society, 2010.

2. Chapter 5 is based on: Brad Bingham, Mark Greenstreet, and Jesse Bingham. Parameterized verification of deadlock freedom in symmetric cache coherence protocols. In *Formal Methods in Computer-Aided Design (FMCAD)*, pages 186–195, FMCAD Inc., 2011.

3. Chapter 4 is based on: Brad Bingham, Jesse Bingham, John Erickson, and Mark Greenstreet. Distributed explicit state model checking of deadlock freedom. In *Computer Aided Verification (CAV)*, volume 8044 of *Lecture Notes in Computer Science*, pages 235–241, Springer-Verlag, 2013.

4. Chapter 6 is based on: Brad Bingham and Mark Greenstreet. Response property checking via distributed state space exploration. In *Formal Methods in Computer Aided Design (FMCAD)*, pages 15–22, FMCAD Inc., 2014.

In addition, Chapter 2 has small portions that originated from the above publications. My individual contributions to publication 1 include

- Created the initial prototype of the distributed model checking algorithm.
- Implemented and evaluated various approaches to load balancing to address both performance and thread crashes.
- Gathered statistics on Erlang communication throughput and latency relative to message size, motivating state-batching optimizations.

For the other publications, I performed the bulk of the research, with co-authors contributing with brain-storming discussions, manuscript writing and feedback. While they are not co-authors on publication 4, Jesse Bingham helped by running our tool on proprietary Intel models, and Jim Grundy provided some models he authored which we used as benchmarks.

# Table of Contents

# List of Tables

# List of Figures

# List of Algorithms

# List of Acronyms

| | |
|---|---|
| BDD | Binary Decision Diagram |
| CEGAR | Counter-Example Guided Abstraction Refinement |
| CMP | Chou-Mannava-Park |
| CTL | Computation Tree Logic |
| DEMC | Distributed Explicit-state Model Checking |
| DF | Deadlock-Freedom |
| DFS | Depth-First Search |
| EMC | Explicit-state Model Checking |
| FSCC | Fair Strongly Connected Component |
| HIR | Heuristic Inference Rule |
| LTL | Linear Time Logic |
| OA | Over-Approximate |
| OST | Outstanding-Search Table |
| OWCTY | One-Way-Catch-Them-Young |

| | |
|---|---|
| PTFA | Predecessor Trace Fair Actions |
| PtP | Pass-the-Path |
| SCC | Strongly Connected Components |
| **ST** | Search Table |
| UA | Under-Approximate |
| **WHT** | Witness Hash Table |

# Acknowledgments

Thanks to Mark Greenstreet for his supervision. His patience, advice and enthusiasm have been a constant inspiration. My supervisory committee of Alan Hu, John Harrison and Karthik Pattabiraman have all been helpful sources with regard to my studies. Thanks to Flavio M. de Paula, Jesse Bingham and John Erickson for all their hard work on the PReach project — without them this thesis would not exist. Flemming Andersen also provided invaluable support on this project, which included guidance during two internships with his group. I thank John Erickson and Jim Grundy for their mentorship during these work terms. I appreciate the kindness and support from the ISD lab students over the years, and my good friends at UBC and beyond. Finally, I thank my family members for their deep interest in my education, and for always lending a hand.

*For my parents, Rosemary and Paul, my role models in life.*

# Chapter 1

# Introduction

Concurrency pervades modern hardware and software. From instruction-level parallelism exploited in CPU execution pipelines, to multiple threads running on a multicore machine, to a fully distributed computation on a compute cluster, concurrency is critical to performance. Modern CPU designs are becoming increasingly parallel. Energy and temperature considerations with current technologies prohibit faster clock speeds [97]. Because single-core performance has reached its limit, multicore architectures are proliferating [3]. Such hardware designs require careful considerations for the protocols that support this concurrency through inter-core and inter-cache communication.

These hardware protocols are often devised and described at a high level by computer architects. The concurrent nature of these protocols make them particularly prone to subtle design bugs, as it is difficult for the human to consider all possible interleavings of concurrent events. Unchecked, any fundamental bugs or mistakes may not be exposed until the protocol has been implemented in a hardware description language, amid many other details, and simulations are run. But as simulation is almost necessarily incomplete with respect the the possible inputs, these bugs may escape to later stages of design and ultimately persist into fabricated chips and sold to customers. Errors that are discovered late in this design, implementation and production process can have extremely expensive consequences. Economically, we wish to catch these bugs as early in the design cycle as possible, for example, before a hardware description exists. This drives the effort known as *formal verification* — the attempt to prove formal properties of a computer chip design. Automated methods are easiest to use, but can require massive computational resources. To harness the required resources, we seek to scale our verification capabilities and make use of clusters of commodity machines.

Let's consider a simple example protocol that controls a traffic light. Suppose the light controls 4 directions: north, south, east, west, and has the standard green, yellow and red signals. A controller will dictate which signals are on when the traffic light is first powered on, and then how they change over time from there. The controller uses a timer to measure how long to stay in the current state before changing. It may also use additional information such as the time of day or sensory data that indicates the presence of a waiting vehicle. At the protocol level, we omit this timing information and consider what the controller allows as far as changes to the signals. For example, we wouldn't expect a good controller to allow a light to transition from green directly to red; we would expect a yellow light in between. For a simple controller, the *state* of the traffic light is precisely the signals that are turned on in the 4 directions.

How do we check if our controller is correct? One approach is to simply test it out and see if any accidents occur. Perhaps this is done in a controlled environment where no moving cars are involved. But how much of this testing is sufficient? Ideally we would like to exhaustively check *every possible configuration* and make sure all of them are okay. We need a formal notion of which states are "bad states"; such a description is called a safety property. A safety property of this system could be "if north is green then east is red", as we can all agree that violating this statement is indeed quite unsafe. But establishing safety isn't enough for a traffic light. Indeed, a light that has all 4 signals turned red forever is safe, but not very functional. This motivates the notion of liveness properties, that reason about events that will eventually occur. A liveness property of this system could be “north will eventually be green”, because otherwise the southbound cars could wait indefinitely.

Once we decide on safety and liveness properties, we can use verification tools to automatically verify them. To show safety, we need to exhaustively list all states that the system allows, and check that none of them meet the “bad” criteria. Liveness is more difficult, as this reasons about sequences of states that may be of unbounded length. The computational cost of both of these checks is mainly determined by the number of reachable states, that is states that could occur in the controller. Unfortunately, this number tends to increase rapidly as more detail is added to the system. In this example, determining the number of reachable states of this system would require a more formal description, but in practice it is often proportional to the number of *possible* states. Assuming that exactly one signal is always on in every direction, the number of possible states for this system (recalling there are 4 directions and 3 signals), is $3 \times 3 \times 3 \times 3 = 3^4 = 81$. How does this change if more detail is added to the system? Suppose that instead of 4 directions we have an addition direction to consider (say, from the north-west), and in addition there’s a left turn arrow, *i.e.* there’s now 5 directions and 4 signals. Now the number of possible states is $4^5 = 1024$, a huge increase. This is an example of *combinatorial explosion*.

In order to deal with state-explosion, many ideas have been proposed. One of our main approaches is to *distribute* the exhaustive check among many computers, to leverage their aggregate memory and computing power. Such techniques are incentivized by modern computers, as commodity multicore machines connected via commodity networks are sufficient to drastically increase our verification capabilities.

Much of this thesis focuses on the liveness problem. But with most verification research, it’s best to address safety first, as it is easier to deal with. Thus we present our tool called PReach, which lists all possible states of a system and checks them for safety. The motivation for this model checker was straightforward, as we had colleagues in industry with large model checking problems, and existing tools lacked the robustness and scalability they needed. PReach can handle systems with enormous numbers of states — on the order of billions — using a hundred machines or more. While the original PReach was very effective for verifying *safety* properties, it highlighted the need to check *liveness properties*, which are generally harder to deal with. Because it involves reasoning about sequences of states, model checking of liveness is susceptible to a much more severe form of the combinatorial explosion problem than safety. Furthermore, reasoning about sequences introduces mathematical technicalities that can make the properties to be checked incomprehensible to the architects. To address both the computational and the psychological complexity of liveness checking, we consider three special cases that both simplify the problem and address a large number of the liveness issues that arise in practice. In particular, we extend PReach to check for deadlock-freedom and response properties.

In the context of the traffic light example, a deadlock freedom property could be *"if new cars stop arriving at the intersection, all waiting cars will eventually face a green light".* This is important for hardware designs as it can demonstrate that the system will not get "stuck" in a state where pending operations cannot complete. An example response property is *"every car that arrives will eventually face a green light".* This is a stronger statement than the deadlock freedom property, as it says something about liveness of the intersection even when there's a steady stream of traffic. In practice, this is a more difficult property to specify and verify. Finally, we also give a method for proving *parameterized* deadlock freedom properties. This means that we could show that an intersection with *any* number of directions satisfies a deadlock freedom property. Such techniques are of particular importance when combinatorial explosion exhausts the capacity of our verification tools.

In summary, this dissertation attacks the problem of automatically proving liveness properties of real industrial hardware protocols. We proceed in the next section by formalizing the problem and our contributions using standard verification terminology.

## 1.1 Formalized Problem

Consider the problem of verifying correctness of a hardware design. Often, a verification engineer will formally express some key aspects of the design in a specification language, which implicitly describes a directed graph called a *transition system* (or simply a *system*). This is typically achieved by stating system variables, initial conditions on their values, and conditions for changes to their values. Such descriptions vary in detail, ranging from the protocol level, which models event-based nondeterministic behavior, to cycle-accurate models to bit-accurate models. The properties we would like to prove about the system are often categorized as *safety properties* and *liveness properties.* These are formally distinguished below, but a commonly used intuitive taxonomy quotes Lamport: “something will *not* happen” for the former and “something *must* happen” for the latter [80]. *Model checking* [44] refers to algorithms and tools for automatically verifying these properties. In this thesis, we focus on *explicit-state* model checking (EMC), which represents each *state* (a valuation of the system variables) as a distinct object in memory. The state-space explosion problem limits the effectiveness of model checking; as more detail is added to the model the number of possible states grows exponentially. Typically the number of reachable states grows in proportion to the number of possible states, which places increasing demands on computational resources.

Each vertex of the system corresponds to a state; the arcs of the system are called transitions. If $p$ is a predicate over the system variables, then the set of states with a variable valuation that satisfies $p$ are called $p$-states. The transition system includes a predicate for the initial states. State $s$ is called *reachable* if there is a path from an initial state to $s$. A *trace* is a walk of possibly infinite length that begins at an initial state. Except in highly degenerate transition systems, there are an uncountably infinite number of traces. We define a safety property as an assertion over all traces for which a counterexample trace has finite length. An example is “Always $p$”, which means that every reachable state is a $p$-state. Verifying a safety property using EMC amounts to enumerating the reachable states and checking something "local" to each state. A liveness property is an assertion over all traces for which a counterexample trace has infinite length. An example is "Eventually Always $p$", which means that every infinite trace has an infinite postfix of $p$-states. Liveness properties are more computationally intensive to verify because they require reasoning over traces rather than reachable states; liveness verification calls for algorithms that detect and analyze strongly connected components of the state transition graph [47, Section 22.5].

To cope with state-space explosion and computers with limited memory, judicious choices must be made when deciding which aspects of the design are included in the model. If a model checking procedure exhausts memory resources, then removing details from the model will usually reduce the size of the reachable state space. Suppose the model in question has is parameterized by an integer that monotonically increases the reachable state space size. Then, reducing the value of this parameter may render model checking tractable. For example, a cache coherence protocol description may be parameterized by the number of caches; clearly increasing this number leads to a greater number of variables and concurrent transitions, and a larger reachable state-space. Perhaps the design of interest has 32 caches, but we can only model check the protocol model with 4 caches. The expectation is that safety or liveness violations in the smaller model will have analogous violations in the real design, thus increasing our confidence when model checking succeeds. Of course, this is does not rule out the possibility that 32 cache design allows violations that are only present with more than 4 caches. Two different approaches for further improving confidence are: (1) using tools and techniques to increase our model checking capacity, and (2) using a parameterized verification technique, which proves something about the cache design for *any* parameter value, including 32. Both approaches are explored in this thesis.

Despite the complementary nature of safety and liveness, the majority of verification research and practice focuses on the former. This is due to safety being easier conceptually, theoretically and computationally. Our view is that liveness checking is a crucial component of the verification task. Not only is liveness usually a desirable property for systems to adhere to, but liveness checking can also catch modeling errors. Suppose the user intends to specify a system that will have reachable states $S$, but the user makes a mistake when writing the model and the resulting system only reaches a subset of the intended states $\hat{S} \subset S$. If all states of $\hat{S}$ satisfy safety but some state of $S \setminus \hat{S}$ does not, then the safety violation will be obscured by the user error. However, simple liveness checks can reveal errors that were hidden by a modelling mistake and give the user confidence that the corrected model faithfully captures the system as intended.

In this thesis we focus on two specific classes of properties of finite state systems. Because we emphasize industrial application, we choose properties that practicing designers will find easy to interpret. Likewise, our emphasis on large-scale problems leads us to choose properties that are both useful to the designer *and* computationally tractable for automatic verification.

The first liveness property that we consider is *deadlock-freedom* (DF). The basic idea is that given a system model, we identify some actions as injecting new transactions into the system (for example, requesting a read to or write from memory); other actions make progress on some pending transaction; and some actions complete one or more pending transactions. Let $q$ denote *quiesence*, a description of states in which there are no pending transactions.DF states that from any reachable state, the model can reach a $q$-state; in other words, all pending tasks can be completed. In CTL (computational tree logic — described in more detail in Section 2.1), we can express DF as $\mathsf{AG}\,\mathsf{EF}\,q$. $\mathsf{AG}$ means "along all paths (*i.e.* at all reachable states), and $\mathsf{EF}\,q$ means "there exists a path to a $q$ state". Thus, $\mathsf{AG}\,\mathsf{EF}\,q$ is CTL for "from every reachable state, there is a path to a state that satisfies q." We present efficient approaches for verifying DF, which can also be applied to the more general version $\mathsf{AG}\,(p \rightarrow \mathsf{EF}\,q)$ which means "from any reachable state that satisfies $p$, there exists a path to a state that satisfies $q$." Note that the existence of a path does not guarantee that the path is taken. For example, an unending sequence of new tasks that pre-empt some older task might continue to arrive; or the model may allow the system to make non-deterministic choices, only some of which lead to completing all pending tasks. To show that every task is eventually completed, we need a stronger property than DF.

Intuitively, a *response property* says that every request for some service is eventually granted. Let $p$-states be those in which the service has been requested, and $q$-states be those in which the service has been completed. A response property specifies that every trace that reaches a $p$-state will later reach a $q$-state. In CTL we write this as $\mathsf{AG}\,(p \rightarrow \mathsf{AF}\,q)$; this means that from all reachable $p$-states ($\mathsf{AG}\,p$), *all* paths eventually reach a $q$-state. In contrast, the $\mathsf{E}$ in $\mathsf{EF}\,q$ (as used to describe DF) says that there *exists* a path along which $q$ is eventually satisfied.

Adding a bit more notation, well note that there are two commonly used logics for reasoning about liveness: CTL and LTL (linear-time logic). The differences are described in Section 2.1 — neither is strictly more expressive than the other. In LTL, $\Box$ indicates "henceforth" (*i.e.* the property holds for all subsequent states) and $\Diamond$ denotes "eventually" (*i.e.* the property holds for some subsequent state). The CTL formula $\mathsf{AG}\,(p \rightarrow \mathsf{AF}\,q)$ is equivalent to the LTL formula $\Box(p \rightarrow \Diamond q)$. When describing response properties, the LTL notation is more commonly used in the research literature, and we will use the LTL notation here[1].

In practice, it is necessary to impose *fairness* assumptions on the system that formally express which traces are deemed to be realistically allowed by the expected underlying implementation. For example, an arbiter that always grants to one requester and not the other could be viewed as unrealistic and ruled out using fairness. More generally, most models consist of many concurrently operating subsystems. It is common to have a scenario with entities $a$, $b$, and $c$ where $a$ makes a request of $b$ while $c$ can continue indefinitely working on its own. Without fairness, we must consider traces where $a$ makes a request and all subsequent actions are those of $c$. For many models, it is unrealistic to consider scenarios where $c$ performs an arbitrary number of actions while $b$ does nothing. We need

---

[1] Some authors use $\mathsf{G}$ instead of $\Box$ and $\mathsf{F}$ instead of $\Diamond$ in LTL formulas. We use $\Box$ and $\Diamond$ notation as it will not be confused with the CTL notation that describes DF properties.

| Property Type | Increasing Capacity | Parameterized |
|---|---|---|
| Safety | PReach MC [22] | CMP Method [40] |
| Deadlock-Freedom | PReach-DF MC, CTL MC | [23] |
| Response Properties | LTL MC, PReach-Resp MC | [93, 18, 56] |

Table 1.1: Verification methods

a way to exclude these unrealistic, also referred to as *unfair*, traces. If *Fair* is a predicate over traces expressing these assumptions, then $Fair \rightarrow \Box(p \rightarrow \Diamond q)$ says that every fair trace that includes a $p$-state will include a $q$-state in the future, or equivalently, that any trace that visits a $p$-state and then cycles forever without visiting a $q$-state necessarily violates fairness.

We expect designers are more interested in response properties than DF because it guarantees something good will happen as opposed to it merely being possible. However, DF avoids the technical and conceptual difficulties associated with specifying fairness, is less computationally complex, and we believe many real protocols that violate response in fact also violate DF. Intuitively, DF can express that pending transactions in the system can complete if new transactions stop being injected. On the other hand, response properties can reason about a particular kind of transaction always eventually completing regardless of other activities.

The cross product of the property types $\{$*Safety, Deadlock-Freedom, Response Property*$\}$ and the goals of $\{$*increasing capacity, parameterized verification*$\}$ gives 6 pairs, as summarized in Table 1.1. We briefly cover this table and expand on each entry in Section 1.3.

The work of this thesis is formulated using the specification language of the Mur$\varphi$ model checker [52], a guarded command language [50] for finite state transition systems. However, we believe the methods are general to the languages used by other model checkers [65, 41, 5].

Model checking of safety properties has several well-understood enhancements that aim to increase capacity, including abstraction, symmetry reduction and partial order reduction. Our contribution in this area is the distributed explicit-state model checking (DEMC) tool called PReach, which utilizes the aggregate memory of hundreds of machines and is entirely compatible[2] with these other techniques. We have extended PReach to check both DF (PReach-DF) and response properties (PReach-Resp) in a distributed environment.

Parameterized verification is known to be an undecidable problem, so human guidance is usually a key ingredient. There are a number of previous works for both safety and liveness properties in this area. The Chou-Mannava-Park (CMP) method [40] for safety properties has been demonstrated as an effective approach when applied to Mur$\varphi$ models with a symmetric parameter. We built upon this method and developed a theory for proving parameterized DF properties, also using Mur$\varphi$ models with a symmetric parameter, that relies on PReach support for MC-DF checks [23]. See Chapter 2 for a survey of techniques that deal with proving that LTL formulas hold in parameterized systems.

To summarize, this thesis addresses the following problem.

> The ubiquity of highly concurrent hardware and software designs calls for verification tools that support both safety and liveness. Given the conceptual and computational complexity of specifying and verifying liveness, we need verification methods that support straightfoward liveness properties and that can be applied to industrial-scale problems.

## 1.2 The Explicit-State Approach

Before I discuss our contributions that utilize explicit-state model checking, it is important to justify the use of this old and simple method. Indeed, landmark contributions have

---

[2] Abstraction can be performed to generate the input model, and symmetry reduction is built in to the Mur$\varphi$ front-end. Partial order reduction, on the other hand, is not supported by PReach but it has been investigated in a parallel model checking setting [100].

been made with *symbolic* model checking techniques that represent a set of states (or approximation thereof) with a boolean formula over system variables, and the next-state function with boolean formula over primed and unprimed variables. The first widespread symbolic model checkers used *binary decision diagrams* (BDD) [35] and demonstrated that the state-space explosion can be curbed for many systems of practical importance [90]. Since then, other symbolic approaches have been employed, including those that utilize SAT queries and interpolants [95, 33]. These methods are necessary for what is referred to as *hardware model checking*, where a sequential circuit is modeled as an and-inverter graph with some modest number of input wires. Explicit-state model checking (EMC) is often infeasible for these kinds of models as the number of input combinations blows up exponentially. However, we note that while symbolic approaches are essential for certain problem domains, there is no "silver bullet" to dealing with state-space explosion. As noted by Hu [67, Section 1.2.3], there are $2^{2^n}$ boolean functions on $n$ variables, and any representation scheme that uses a polynomial number of bits $O(P(n))$ can represent at most $2^{O(P(n))}$ of these functions.

EMC tends to be appropriate when the reachable state space features sufficient non-uniformity. In such cases, symbolic expressions may degenerate to a large formula that resembles a disjunction of conjuncts, where each conjunct describes only a small number of states. Then, symbolic operations will in a sense require just as much work as explicit-state, but with a larger constant factor in terms of time and memory cost. The aforementioned work by Chou *et al.* [40] suggests that EMC is the right choice when checking cache coherence protocols. To paraphrase:

1. BDD performance is more sensitive to the data structures used to describe the protocol;
2. EMC can take advantage of symmetry reduction [70];
3. EMC is better suited for disk-based techniques;

4. SAT-based methods are not known to outperform BDD-based methods on cache coherence protocols.

In addition, EMC seems to be better suited for distributed implementations, at least compared with BDDs. Existing approaches involve maintaning the BDD on a single machine until it is about to exceed the memory capacity, and then carefully split the BDD in order to balance memory usage among two machines *and* minimize duplication [61, 60]. In contrast, distributing the reachable state space among machines is trivially achieved by using a uniform random hash mapping [104]. We note that Bradley's original IC3 model checking paper involves experiments that use multiple cores and machines, but appears to saturate in performance at 8 to 12 threads [33]. There are also promising IC3 methods for showing liveness [34], but to our knowledge the parallel performance has not been investigated.

## 1.3 Verification Methods of this Thesis

PReach (*P*arallel *REACH*ability) is a distributed explicit-state model checker based on Mur$\varphi$ [22]. It is designed to be scalable and harness the aggregate computing power of clusters of machines. PReach verifies Mur$\varphi$ models and borrows Mur$\varphi$'s C++ implementations of key model checking components, such as a hash table to store states and compilation of Mur$\varphi$ models to C++ code for the efficient computation of state successors. The main module that handles communication and coordination of threads is written in the functional language *Erlang* [2] and is roughly 1000 lines of code. This separation of the distributed algorithm from model checking details offers both efficiency and simplicity. Our experiments have shown that in addition to linear speedups, PReach can utilize hundreds of compute nodes to explore the state space of the largest Mur$\varphi$ models ever checked — on the order of 100 billion states. The clean code and extensibility of PReach has been leveraged in subsequent projects [23, 26, 27].

First, we implemented a sequential DF checking procedure as part of a method for parameterized verification of DF [23]. For a symmetric system $\mathcal{S}$ described in a guarded command language and parameterized by $n$, we seek to show that for any $n$, each reachable state of $\mathcal{S}(n)$ has a path to some $q$-state, i.e., $\mathcal{S}(n) \models \mathsf{AG}\,\mathsf{EF}\,q$, where $q$ is a predicate[3]. Our approach is inspired by, and builds upon the CMP method [40] which establishes properties of the form $\mathcal{S}(n) \models \mathsf{AG}\,p$ for predicate $p$, which may include universal quantifiers over the parameterized set. They use a counter example guided abstraction refinement (CEGAR) approach [45]. First, an abstraction $\mathcal{A}$ of $\mathcal{S}(n)$ for *every* $n > k$ is computed, which essentially keeps $k$ fixed parameter entities fully modeled, and overapproximates the behavior of all others. Next, a trace of $\mathcal{A}$ is found via model checker that reaches a $\neg p$-state. The user examines the trace and devises a *non-interference lemma* $\phi_1$ which is the abstraction of an assumed invariant of $\mathcal{S}(n)$. The intention is that if $\phi_1$ holds of $\mathcal{S}(n)$ for all $n > k$, then the spurious trace cannot occur. This is achieved by strengthening $\mathcal{A}$ with $\phi_1$ by restricting when actions of the abstracted parameter entities may occur, and model checking is used again to verify $\mathsf{AG}\,(p \wedge \phi_1)$. The user continues this process until the abstraction satisfies $\mathsf{AG}\,(p \wedge \phi_1 \wedge ... \wedge \phi_m)$, a counterexample is found, or the user is unable to find the next useful $\phi_i$. If $\mathsf{AG}\,(p \wedge \phi_1 \wedge ... \wedge \phi_m)$ is established, the assumed invariants along with $p$ are actual invariants of $\mathcal{S}(n)$ because $\mathcal{A}$ simulates $\mathcal{S}(n)$ for $n > k$.

Adapting this method to proving parameterized DF presents challenges. We need to show that every reachable state in $\mathcal{S}(n)$ has a "witness" — a path to some $q$-state. In order to find such paths, it's often necessary to include the actions of abstracted parameter entities. However, because the abstraction *overapproximates* these actions, it is unsound to assume that they may occur as there's no guarantee that corresponding actions exist in the concrete system.

Suppose we have used the CMP method to verify some safety properties of param-

---

[3] Typically, $q$ is universally quantified over the parameterized set, and is interpreted by to the set of *quiescent* states. In cache coherence protocols, this corresponds to states where there are no pending messages.

eterized system $\mathcal{S}(n)$ via abstract system $\mathcal{A}$. As the transitions of $\mathcal{A}$ are overapproximate, there could exist reachable states for which no concrete analogue is reachable in $\mathcal{S}(n)$ (for any $n$). In our method, we employ a *mixed abstraction* which augments $\mathcal{A}$ with *underapproximate* transitions $U$; denote the overapproximate transitions with $O$. Transitions of $U$ are guaranteed to have a concrete analogue, and thus soundly imply existential path properties in $\mathcal{S}(n)$. To check such properties, we use state space enumeration to explore states reached through $O$-transitions, and for each reachable state a path is found composed of $U$-transitions to a $q$-state. PReach implements this model checking procedure. If there exists a reachable state $\tilde{s}$ for which there is no $U$-path to a $q$-state, a counterexample trace is generated. Following an analogous CEGAR procedure to that of the CMP method, the user must either prove that $\tilde{s}$ is unreachable by strengthening $O$, or that such a path should exist from $\tilde{s}$ by weakening $U$. The latter is done by using inference rules that we refer to as "heuristics" [23]; see Chapter 5. The proof obligations of heuristics are discharged by model checking the mixed abstraction as described above. We applied this method to both the German and FLASH cache coherence protocols as case studies where DF for any number of caches was established.

Following up on this work we designed and implemented two *distributed* algorithms for verifying DF of systems. The user provides both the quiescent predicate $q$, along with a set of *helpful transitions* $\mathcal{H}$ that are expected to be a sufficient set to form a path from every reachable state to some $q$-state. Similar to the paths formed using $U$-transitions above, both algorithms perform a directed search using only $\mathcal{H}$-transitions. Identifying these transitions provides an expression of designer intent. This approach allowed us to discover a bug in the Peterson mutual exclusion model that had persisted in the Mur$\varphi$ distribution for about 20 years. We found that in most cases the average search path is short and PReach can perform this lightweight liveness check for a small additional overhead as compared to safety checking.

To attack the verification of a response properties, we implemented a variant of the *One-Way-Catch-Them-Young* (OWCTY) algorithm [73, 38] in PReach, capable of checking response properties on systems augmented with fairness on actions. Previous OWCTY algorithm descriptions were aimed at checking arbitrary LTL formulas, used data structures that respresented states sets with full state descriptors, and generally did not include fairness assumptions. In PReach, states are hashed to 40 bit descriptors, so that checking membership of a set is easy but member states cannot easily be reconstructed. Our approach uses forward reachability only, and augments the hash table entries with various bookkeeping bits to emulate membership in the various sets. We have found this implementation to be capable of checking response properties on fair systems with hundreds of millions of reachable states.

## 1.4 Thesis Statement

Given some hardware design model, the class of properties that can be described as DF or response are of practical interest. Capacity constraints limit the effectiveness of automatic, sequential model checking approaches, especially for protocol level descriptions. Many designs can be modeled as a system that is parameterized, for example by number of caches or addresses. Two approaches are to set the parameter as large as our model checking tools and hardware resources allow, or to employ a parameterized verification method. These approaches trade-off human effort with strength of the verification result with respect to the original design. Another trade-off is verifying response properties versus DF; the former is a stronger result, but the latter is easier both from a computational and human effort perspective. Regardless of the approach, there is particular importance to mitigate the burden on the user. By leveraging the stable, tested and large scale distribution of the state-space provided by the PReach model checker, we are able to achieve scalable methods for liveness verification.

This brings us to the thesis statement:

*This thesis develops and demonstrates tractable, practical and scalable distributed explicit-state model checking methods for establishing liveness properties of practical importance for large-scale models of hardware protocols.*

### 1.4.1 Contributions

1. PREACH: an industrial strength parallel, explicit-state model checker capable of checking the largest published Mur$\varphi$ models by distributing the computation across hundreds of machines. My contributions include:
    (a) Created the initial prototype of the distributed model checking algorithm.
    (b) Implemented and evaluated various approaches to load balancing to address both performance and thread crashes.
    (c) Gathered statistics on Erlang communication throughput and latency relative to message size, motivating state-batching, load balancing, and other optimizations.
2. PARAMETERIZED DF: A novel method for establishing DF properties in symmetric parameterized systems. This is the first work in parameterized DF, and complements the CMP method for parameterized safety properties.
3. PREACH-DF: Two approaches for DEMC of DF. Shows that given simple and easily available user input, can model check DF on large models for a small overhead compared with state-space enumeration.
4. PREACH-RESP: A novel algorithm implementation for distributed explicit-state model checking of response properties on fair systems. Demonstrated that in practice the OWCTY algorithm takes only a modest constant number of expansions per state — far less than the worst-case performance.

### 1.4.2 Roadmap

The rest of the thesis is organized as follows. Chapter 2 is related work, comparing and contrasting our contributions with literature in DEMC, parameterized verification, and LTL model checking. The Section 2.1 provides technical background and some definitions used throughout the thesis. Chapter 3 summarizes the PReach model checker. Chapter 5 explains our approach to parameterized DF proofs. Chapter 6 covers the implementation of the distributed OWCTY-like algorithm. Finally, Chapter 7 summarizes the thesis and points to areas of future work.

# Chapter 2

# Related Work

Here, we survey the related research from the literature and compare it with our work. First, Section 2.1 provides some definitions, notation for CTL and LTL temporal logics and background for the Mur$\varphi$ description language. These are used throughout the rest of this thesis and are relevant to related work. The next three sections divide the previous work into categories. Section 2.2 describes parallel and distributed explicit-state model checking with a focus on tools. Section 2.3 examines parameterized verification with an emphasis on techniques. Section 2.4 describes LTL model checking with a focus on algorithms. We conclude with a summary in Section 2.5 that puts the contributions of this thesis into context.

## 2.1 Preliminaries

A system $\mathcal{S}$ is a triple $(S, I, T)$ where $S$ is a set of states, $I \subseteq S$ are the initial states, and $T \subseteq S \times S$ is the transition relation. System $\mathcal{S}$ may be viewed as a digraph $G = (S, T)$. The reachable states of a system are the states $s$ for which a path exists in $G$ from an initial state to $s$. A trace is a walk of possibly infinite length in $G$ from an initial state. If $S' \subseteq S$, let $\langle S' \rangle$ be the subgraph of $G$ induced by the vertices of $S'$.

A system has a set of variables each with finite range that each state is a valuation of. If $p$ is a predicate on state variables, then we say that $s$ is a $p$-state if $p(s)$ is true. We sometimes overload this terminology; if $A$ is a set of states, then an $A$-state is a member of $A$. If $v$ is a variable of $\mathcal{S}$ and $s$ is a state, then $v(s)$ is the value of $v$ in $s$.

### 2.1.1 Temporal Logic

Here we give a brief explanation of CTL and LTL. For a formal description, see [68].

CTL is a logic over paths of $G$, and are meaningful in any state of $G$. Symbols $\mathsf{A}$ and $\mathsf{E}$ are branching quantifiers, where $\mathsf{A}$ means "for all paths from the current state" and $\mathsf{E}$ means "at least one path from the current state". Each branching quantifier is immediately followed by a path-specific quantifier:

- $\mathsf{X}\ \phi$ means $\phi$ holds in the next state on the path.
- $\mathsf{F}\ \phi$ means $\phi$ holds eventually on the path.
- $\mathsf{G}\ \phi$ means $\phi$ holds everywhere on the path.
- $[\phi\ \mathsf{U}\ \psi]$ means that $\phi$ holds at least until some state where $\psi$ holds, and that $\psi$ eventually holds.
- $[\phi\ \mathsf{W}\ \psi]$ means that $\phi$ holds at least until some state where $\psi$ holds. If $\phi$ holds everywhere on the path then $\phi\ \mathsf{U}\ \psi$ holds for any $\psi$.

Here, $\phi$ and $\psi$ are CTL formulas consisting of braching/path-specific quantfier pairs or state predicates. CTL formulas may be combined with the usual logic connectives: $\neg$, $\wedge$, $\vee$ and $\rightarrow$.

Unless otherwise stated, when we say that CTL formula $\psi$ holds in $\mathcal{S}$, it means that $\psi$ holds in every initial state. As examples,

- $\mathsf{AG}\, p$ means "always-globally $p$", or $p$ holds in all states along all paths;
- $\mathsf{AG}\,\mathsf{AF}\, p$ means "always-globally, always-eventually $p$", or in all states along all paths, $p$ holds eventually'

- $\mathsf{AG}\,\mathsf{EF}\,p$ means "always-globally, exists-eventually $p$", or in all states along all paths, there exists a path to a state where $p$ holds.

LTL is a logic over all infinite traces the system allows. As with CTL, the logical connectives have the usual meaning and $p$ is a state predicate. As LTL reasons over a set of traces, there is no notion of branching quantifiers, only path-specific quantifiers:

- $\Box\ \phi$ means $\phi$ holds everywhere along the trace.
- $\Diamond\ \phi$ means $\phi$ holds eventually along the trace.
- $\bigcirc\ \phi$ means $\phi$ holds in the next state on the trace.
- $\phi\ \mathcal{U}\ \psi$ means that $\phi$ holds at least until $\psi$ holds, and $\phi$ eventually holds along the trace.
- $\phi\ \mathcal{R}\ \psi$ means that $\phi$ holds at least until $\psi$ holds, but $\phi$ need not eventually hold along the trace.
- $p$ means that $p$ holds in the first state of the trace.

We use both CTL and LTL in this thesis. Neither is strictly more expressive than the other. For example, the CTL formula $\mathsf{AG}\,\mathsf{EF}\,p$ *cannot* be expressed in LTL, and the LTL formula $\Diamond\Box p$ *cannot* be expressed in CTL. Even though response properties are expressible in both CTL and LTL, we choose standard LTL notation as it is consistent with the literature on response properties.

### 2.1.2 The Mur$\varphi$ Description Language

Here we give an overview of the Mur$\varphi$ description language which is the input language of the Mur$\varphi$ model checker [52] and PReach. For the Mur$\varphi$ user manual and a tutorial, see [98, Murphi 3.1]. Examples of full Mur$\varphi$ models can be found in Figures 5.2 and 5.3, as well as Appendix A.

A Mur$\varphi$ system gives the variables that dictate the state space, a predicate for the initial states, and implicitly describe the transition graph through a sequence of guarded commands, or rules. It also contains a list of state invariants. When model checking commences, initial states are constructed and then their successors computed according to the list of rules. Each rule has a *guard*, which is a predicate on states. If the guard holds true in state $s$, the rule is said to *fire* and the command part of the rule specifies an atomic update to apply to $s$ to generate successor state $s'$. If all reachable states satisfy all invariants, model checking completes once all reachable states have been visited. If some reachable state does not satisfy an invariant, checking halts and a counterexample trace is printed.

We will construct a traffic light controller model as an example, to highlight the various syntactic elements.

The keyword `const` is used to specify constants.

```
const NUM_DIRECTIONS 4;
```

The keyword `type` is used for type definitions. Some built-in types include `boolean` and `num` (for integer values). In order to exploit symmetry reduction, a `scalarset` type must be used.

```
type DIR : scalarset(NUM_DIRECTIONS);
```

This indicates an index set of symmetric entities. The values of the indices cannot be compared with less-than or greater-than, only for equality. Enumerated, record and array types can be defined.

```
COLOR: enum green, yellow, red;
SIGNAL: record color : COLOR; waiting_car: boolean; end;
```

The system variables come next, using the keyword `var`. In our example, the only variable is an symmetric array of type `SIGNAL`:

```
var intersection: array[DIR] of SIGNAL;
```

The `startstate` keyword precedes a command for the initial states. These assignments are atomic, and take the form `variable := value;`. For loops may be used to range over index sets such as scalarsets. The initial assignment may be given a name — here we choose the `Init`.

```
startstate "Init"
   for d : DIR do
      intersection[d].color := red;
      intersection[d].waiting_car := false;
   end;
end;
```

This will initialize each direction's color to red and set the waiting car flag to false. Note that any uninitialized variables will be set to the special value `undefined`, regardless of type. It is a Mur$\varphi$ runtime error to read a variable with value `undefined` unless it is with the special function `isundefined`.

A `ruleset` is a group of rules generated from an index set. For example, we could use the following to allow `green` to `yellow` transitions for all directions.

```
ruleset d: DIR do
   rule "green_to_yellow"
      intersection[d].color = green
         ==>
      intersection[d].color := yellow;
   end;
end;
```

Mur$\varphi$ predicates (and guards) may include the symbols `&`, `|`, `!`, `->`, `=` and `!=` for logical and, or, not, implication, and comparing for equality and inequality, respectively. Some other rules we might want:

```
ruleset d: DIR do
   rule "yellow_to_red"
```

```
      intersection[d].color = yellow
         ==>
      intersection[d].color := red;
   end;
   rule "red_to_green"
      intersection[d].color = red
         ==>
      intersection[d].color := green;
   end;
   rule "car_arrives"
      !intersection[d].car_waiting
         ==>
      intersection[d].car_waiting := true;
   end;
   rule "car_proceeds"
      intersection[d].car_waiting & intersection[d].color = green
         ==>
      intersection[d].car_waiting := false;
   end;
```

Finally, we include invariants that all states must satisfy, otherwise Mur$\varphi$ will halt with a counterexample trace. This also introduced `forall` expressions that form a conjunct of predicates depending on values from an index set.

```
invariant "No_Collision"
   forall d1 : DIR do forall d2 : DIR do
      i != j ->
     ((intersection[d1].color = green & intersection[d1].car_waiting) ->
        !(intersection[d1].color = green & intersection[d1].car_waiting))
end end;
```

This invariant says that at most one direction will have a car waiting and a green signal at a time. But this invariant doesn't hold of our current system, as all 4 directions might be green with a car waiting. We could address this by only allowing a change from red to green when all lights are red, changing the guard of rule `red_to_green`.

```
rule "red_to_green"
   forall d1: DIR do intersection[d1].color = red end
      ==>
   ...
```

This new system satisfies the invariant, but probably isn't a good traffic light since at most one direction can be green at a time. In order to allow opposing directions to both be green at once would likely require us to rethink the types and symmetry. We won't go into this level of detail, as what we have presented is enough to follow the Mur$\varphi$ code of this thesis.

## 2.2 Parallel and Distributed Model Checking

The two most common methods of performing model checking are explicit-state space enumeration and the use of symbolic representations. For some industrial protocols, explicit-state model checking is considered to be the more effective verification technique [104], and hence is employed by numerous tools such as Mur$\varphi$ [52], SPIN [65], TLC [114] and Java PathFinder [63]. Most of these model checkers explore the state space in a sequential manner which can be a hindrance in terms of both memory usage and runtime when verifying systems with a large state space. This motivates the use of distributed explicit-state model checking (DEMC) tools. These tools may be categorized into those that handle safety only [104, 82, 96, 32] and those that handle some form of liveness [13]. DEMC for safety properties has received more attention. Generally, liveness checking is more complex both computationally and conceptually, yet it remains of industrial importance. To address this need, the main focus of this thesis is to design scalable solutions for interesting liveness properties of real models.

The PReach model checker [22] is an implementation of the Stern-Dill algorithm for DEMC [106]. It reuses back-end computation components of the Mur$\varphi$ model checker [52] and reads models written in the Mur$\varphi$ language, which has become the *de facto* industry standard for modelling hardware protocols. This algorithm statically partitions the state space among processing nodes according to a uniform hash function, where each node is said to *own* a set of states. When a state is expanded, the successors are sent to their respective owners where they are stored and subsequently queued for expansion, and duplicate states are ignored. Thus, state ownership specifies the compute node that is responsible for storing a given state. In PReach, a hash table implementation borrowed from Mur$\varphi$ [103] is used for each node to record the states it has encountered. See Figure 2.1 for a flow-chart overview of Stern-Dill DEMC.

The fact that state ownership is independent of which compute node performs the expansion leads to load balancing schemes on the states that are queued for expansion (the *work queue*). That is, as long as a given state is initially sent to its owner to check for hash table membership, *any* compute node may perform the state expansion step. The need for load balancing in DEMC has previously been identified by Behrmann in the context of timed automata model checking [19], and later by Kumar and Mercer [77]. We contrast the latter scheme with that of PReach in Section 3.3. Also, when the work queue becomes long, we may store most of its states on disk [105] to keep the majority of main memory dedicated to the hash table. Because the work queue is written and read sequentially with large blocks of states per disk access, using disk storage has minimal impact on performance compared with using memory only. States are batched into messages with typically 100 to 1000 states before sending between compute nodes. This batching method has been used by others in the context of DEMC [106, 96].

The closest tool to PReach is a model checking tool that has been extensively described in the literature called DiVinE [111, 12, 10, 4, 6, 7, 13]. The tool originated in Jiří Barnat's PhD work [15], and the project has been ongoing since. Initially the tool was

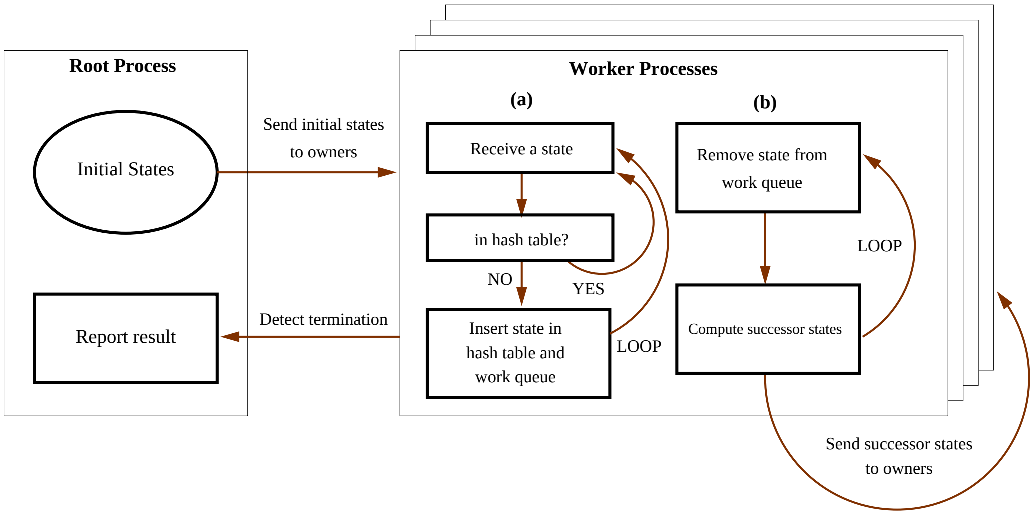


Figure 2.1: Stern-Dill DEMC flow-chart. Initial states are computed by a designated root process and then sent to their respective owners. These worker processes receive incoming states and check them for membership in a hash table. If the state is present, it is ignored. Otherwise, the state is inserted into the hash table and queued for expansion. Worker processes alternate between threads (a) and (b), where the former receives states and the later computes state successors and sends them off to owners. Termination is detected by the root process when all reachable states have been expanded or a state that violates safety is found.

intended to take advantage of distributed memory for model checking LTL formulas, and served as a platform for algorithm evaluation and comparison. This led to further research of algorithms and computing environments. Barant, Brim, and Ročkai [8] investigated the restricted problem of model checking *weak* LTL formulas using DiVinE. An LTL formula is weak if the product automaton of the formula and system that accepts counterexamples has the property that in a given SCC every state is either accepting or none of them are accepting. The same authors wrote an insightful paper on the challenge of applying partial order reduction in a parallel setting [10]. On the compute environment side, DiVinE was extended to run on CUDA enabled graphics cards [4]. They were able to utilize *two* graphics cards to speed up the model checking computation, the only work I am aware of to do so. This work was for LTL model checking using the MAP Algorithm (see Algorithm 2.3). The initial state space is enumerated on a single multicore machine, and then the reachable state graph is converted to a sparse matrix representation. This graph representation is partitioned into at most two pieces and copied to the memory of the CUDA devices where the MAP algorithm runs. DiVinE has also been used to evaluate the benefit of exploiting Flash memory for model checking algorithms that formerly utilized magnetic disk to store states [6]. Recently, DiVinE 3.0 was released with features aimed at software model checking [7]. Taking as input LLVM bytecode, it is capable of model checking C/C++ programs, along with other languages that compile to LLVM.

While the DiVinE tool has explored many aspects of parallel and distributed LTL model checking, its scalability and applicability to industrial problems is unconfirmed. One publication reports it handling state spaces consisting of as many as 419 million states [11]. However, another researcher has reported scalability issues when running DiVinE on 16 machines [54]. The developers of DiVinE state that it is likely to crash when the compute nodes are heterogenous, as it is targeted towards homogeneous clusters [13]. DiVinE has no explicit algorithms for dealing with strong fairness assumptions, and model checking with more than a handful of these assumptions renders the computation intractable (more details provided in Section 6.7). While DiVinE has optimizations for models with weak fairness assumptions [16], it is only possible to attach weak fairness to *all* rules, not a subset. Thus, DiVinE cannot catch the Mur$\varphi$ Peterson bug explained in Chapter 4. DiVinE is unable to verify our notion of deadlock-freedom, as it cannot be specified in LTL. As far as I am aware, the benchmarks used to measure performance of DiVinE are substantially smaller than the problems arising from real, industrial coherence and communication protocols considered in this dissertation.

LTSmin [32] is a model checking tool that handles a variety of model inputs and supports (sequential) explicit-state, symbolic BDD-based, and DEMC of safety. It is designed in a modular fashion where input models are translated into an internal representation that the model checking algorithms use, so it is straightforward to add new modelling languages. The Mur$\varphi$ language is not currently supported, nor is DEMC of liveness properties. The majority of work surrounding LTSmin appears to focus on multicore, shared memory model checking [85]. One paper that deals specifically with DEMC attacks the problem of model checking models with unbounded recursive data types [31]. In these models, states are not restricted to a fixed size, which leads to difficulties when hashing, distributing and comparing for equality. The largest model verified with LTSmin's DEMC approach is about 570 million states [32].

Some other examples of well-known explicit-state parallel model checkers are adaptations of Mur$\varphi$ and SPIN. "Parallel Mur$\varphi$" [104] is a parallel version of the Mur$\varphi$ model checker [52] based on parallel and distributed programming paradigms. Eddy Mur$\varphi$ [96] has improved speed over Parallel Mur$\varphi$ by the separation of concerns between next-state generation and communication during distributed model checking. It was intended to be a distributed model checker for Mur$\varphi$. Unlike previous work, Eddy Mur$\varphi$ was implemented to take advantage of emerging multicore CPUs by using two MPI threads per compute node (one for computation and one for communication). As of this date, there is no active maintainer of Eddy Mur$\varphi$. We have found that a recent version the software reports inconsistent numbers of reachable states on multiple runs for the same, simple models. Thus, we do not believe that Eddy Mur$\varphi$ is trustworthy in its current form. The 4500+ lines of C++ code that comprise Eddy make it more complicated than PReach. The simplicity and stability of PReach make it the obvious foundation for our investigation of DEMC for liveness properties.

PSpin has also been used for performing distributed model checking with the capability of handling up to 2.8 million states [82]. A distributed version of the Spin model checker [65] for checking safety only, PSpin departs somewhat from the Stern-Dill approach by allowing the hash function to be less uniform. Using a hash function that only depends on subset of the system variables can in some cases improve performance. In particular, this non-uniform hashing exploits the structure of models written in Spin's specification language, Promela. These models describe the system through processes with some independence between them. Therefore, if many transitions only change variables that the hash function does *not* depend on, successor states need not be communicated to another worker and remain locally owned. The PSpin work shows that with ratios between the worker with the least states and the worker with the most states of as low as 0.71, large performance gains are possible when compared with uniform hashing. DEMC has also been studied for models described by Petri nets [75, 20].

### 2.2.1 Model Checking Deadlock Freedom

Our approach for verifying the deadlock freedom property $\mathsf{AG}\,\mathsf{EF}\,q$ involves firing helpful transitions provided by the user. This essentially finds paths that follow pending system transactions (say, servicing a cache-miss) to completion, and $q$-states describe states with no pending transaction. This is reminiscent of techniques used in other verification contexts. *Pipeline flushing* used by Burch and Dill in their seminal paper [37] shows refinement of processor pipelines by completing in-flight instructions that are beyond a commit point as part of the refinement map. The idea of iteratively firing certain commands to complete in-flight transactions is also similar to the *completion functions* used by Park and Dill [101], though again their goal was to verify refinement. Neither DiVinE [12] nor Eddy [96] are capable of checking CTL properties such as $\mathsf{AG}\,\mathsf{EF}\,q$, and as mentioned above, neither has been applied to large scale problems as has been done with PReach [22]. Our approach differs from the classical CTL model checking algorithm [44], which performs a pre-image fix-point computation from $q$ to compute the set of states that satisfy $\mathsf{EF}q$, and then checks that the reachable states are contained in this set. With an explicit-state algorithm computing the pre-image fixpoint of $q$ can be expensive. Suppose a Mur$\varphi$ rule has an update of the form $var := expr$, where the rule guard and $expr$ is independent of $var$. Then, $var$ could have *any* type-consistent value in a previous state, many of which could be unreachable states in the system. By using a forward search, we ensure that the space and time complexity is proportional to that of doing (explicit) state-space exploration. Doing CTL model checking only using forward searches has been investigated for symbolic model checking, *e.g.* the work of Iwashita *et al.* [72]. They show that using forward search only offers performance improvements for many CTL properties.

## 2.3 Parameterized Verification

Our work in verifying parameterized deadlock-freedom builds upon the CMP method [40] for verifying parameterized safety properties. The CMP method has motivated other works, including its formalization [76], automation [28], and application to industrial protocols [107, 99].

There have been many previous efforts to extend compositional techniques to parameterized safety property verification [42, 92, 93, 40, 76, 86, 28, 83, 99]. Pioneering work by Clarke *et al.* [42] showed that a mutual exclusion scheme with $n$ nodes arranged in a ring that passed around a token was correct for all $n$. This was established by means of a bisimulation between a system with $k$ nodes and one with $k+1$ nodes for all $k > 1$. The logic to describe parameterized properties was *Indexed* CTL* (or ICTL*), which is CTL* without the next-time operator[1], and indexed formulas of the form $\bigwedge_i f(i)$ or $\bigvee_i f(i)$ where $i$ ranges over the nodes and $f$ is a predicate over variables local to that node. More recently, the work of Lv *et al.* [86] gives a technique for automatic strengthening of a CMP-like abstraction of symmetric systems. If the abstraction has $m$ concrete nodes, they instantiate a system with $m+1$ nodes and use invariants of this system as proposed variants of the abstraction.

As for liveness-like properties, there are several notable works. McMillan's work on using compositional methods for LTL liveness properties [91] was applied to parametric liveness verification of the FLASH coherence protocol [93]. Using SMV, the user can

---

[1] CTL* is a logic that is a superset of both CTL and LTL.

automatically generate a sound abstraction, and then use lightweight theorem-proving commands for concretizing additional caches or proposing user-devised noninterference lemmas, based on counter example analysis. Although [93] focuses on a proof of safety, the same framework was used to show that whenever the directory is in the pending state, it is eventually not pending [94]. McMillan's proof relies on a handful of lemmas and fairness assumptions, designed and proven within SMV.

Fang *et al.* proposed an interesting technique called *invisible ranking* [56] which attempts to automatically guess ranking functions to prove response properties. The associated proof obligations (from [88]) are decided using some small-model theorems (conditions under which it is sufficient to check some property on only a small instantiation of the parameterized system) and BDD based methods for proposing invariants for the parameterized system. The authors have previously use what they refer to as *counter abstraction* for parameterized liveness verification [102]. Here, liveness of parameterized mutual exclusion protocols is established by an abstraction that maps counter variables to have a range of $\{0, 1, \infty\}$.

Baukus *et al.* employ *WS1S* (Weak Second-order theory of 1 successor — a decidable second-order logic) to perform liveness verification of parameterized systems [17], and verify response properties for the German protocol as a case study [18]. Like our approach, human effort is required, to select both abstract predicates and ranking predicates needed to create an appropriate abstraction. The complexity of deciding WS1S is well-known to be super-exponential, hence scalability of this approach seems unlikely. It is unclear if WS1S is expressive enough to model array variables indexed by the parametric type and with values over the parametric type. Such variables appear in the Mur$\varphi$ description of the Flash protocol.

The earliest example we could find where both over-approximative and under-approximative abstractions of a transition system are employed for verification is the work of Larsen and Thomsen [81]. They distinguish between *necessary* and *admissible* transitions; for a process to refine another it must over-approximate the former and under-approximate the latter. Of course our interests are in *abstraction* rather than *refinement*, which are in a sense inverses of each other. Later the work of Dams *et al.* [48] and independently Cleaveland *et al.* [46] used *mixed transition systems* which are defined with *two* transition relations to formulate abstractions that preserve both universal and existential properties of the modal $\mu$-calculus. Our mixed-abstractions are similar, but these authors focus on the problem of curbing state-space explosion of a fixed system using an abstraction that preserves certain properties, and we utilize abstractions for parameterized verification.

## 2.4 Model Checking LTL Formulas

The standard approach to LTL model checking [110] involves constructing the *product automaton* that is the synchronous cross product of the Büchi automaton[2] that accepts the negation of the property in question, and the Büchi automaton for the system itself. If the language accepted by the product automaton is empty, then the LTL property holds; otherwise, a counterexample trace is found. Counterexamples correspond to reachable cycles of the product automaton that do not include an accepting state (where all system states are accepting). LTL model checking suffers from state-space explosion and this approach takes time and space $\mathcal{O}(n2^{|\phi|})$, where $n$ is the number of states in the original system, $\phi$ is the LTL formula and $|\cdot|$ is formula size, *i.e.*, the number operators and state-predicates. In this standard approach, including the algorithms we will present shortly (Algorithms 2.1, 2.2 and 2.3), fairness assumptions are handled implicitly, *i.e.*, by adding them as antecedents on the formula to check.

Explicit-state methods for LTL model checking must utilize some method of decomposing graph $(V, E)$ into its strongly connected components (SCCs). Using a sequential

---

[2] A Büchi automaton is similar to a nondeterministic finite automaton with accepting states $F$, but accepts traces of infinite length — those that visit an $F$-state infinitely often.

algorithm for accepting cycle detection such as Tarjan's [108], SCCs may be found in $\mathcal{O}(|V| + |E|)$ time. This, along with several other algorithms for the same problem rely on a depth-first search (DFS) to compute SCCs, however, DFS is known to be $\mathcal{P}$-complete and therefore is not easily parallelizable. However, such DFS-based algorithms are unsuited to parallelization unless[3] $\mathcal{P} = \mathcal{NC}$ [9]. In comparison, the time complexity of our approach using PREACH for proving response properties differs by a factor of $r/s$ where $s$ is the speedup of using multiple threads. Manna and Pnueli presented a sequential algorithm for model checking response properties of fair transitions systems [89], but this not easily parallelizable and thus scalability is limited. Recently, Holzmann implemented some interesting liveness checking algorithms in a multicore version of SPIN [66]; however this approach will only find counterexamples of bounded length. Other work related to ours includes that of the authors of the LTSmin model checking tool, most notably their algorithms for parallel SCC decomposition on multicore machines [79, 55].

BFS-based algorithms, on the other hand, are amendable to parallel implementations. We present below (Algorithm 2.1) a simple such algorithm, referred to as *Basic Distributed Cycle Detection*[4]; but first some definitions.

A Büchi automata $(S, I, T, F)$ is a digraph with vertices $S$ and arcs $T \subseteq S \times S$. Initial states $I$ are a subset of $S$, as are the accepting states $F$. In our presentation, we typically express a Büchi automata as a graph $G = (S, I, T)$ and associated accepting state predicate $F$, so that different notions of trace acceptance are easy to write. Given $X \subseteq S$, REACHABLE$(G, X)$ computes the set of states reachable in $G$ in zero or more transitions starting from states of $X$. REACHABLE$^+(G, X)$ computes the set of states reachable in $G$ *in one transition or more*, starting from states of $X$.

Each time around the loop, line 9 will remove from *Cur* all $F$-states that are un-

---

[3] $\mathcal{P}$ is the complexity class of problems that are decidable in polynomial time. $\mathcal{NC}$, known as "Nick's Class", is the set of problems decidable in polylogarithmic time on a parallel computer with a polynomial number of processors.

[4] While variants of this approach are common in the literature, we are not aware of this particular formulation having been previously published.

**Algorithm 2.1** Basic distributed cycle detection

```
procedure BASIC($G = (S, I, T), F$)
    ▷ find accepting states reachable from themselves
    $Cur$: set of states
    $Old$: set of states
    $Cur \leftarrow \text{REACHABLE}(G, I) \cap F$        ▷ All reachable $F$-states
    $Old \leftarrow \emptyset$
    while $Cur \neq Old$ do
        $Old \leftarrow Cur$
        $Cur \leftarrow Cur \cap \text{REACHABLE}^{+}(G, Cur) \cap F$
    end while
    if $Cur = \emptyset$ then
        property holds
    else
        $Cur$ contains a cycle with an accepting state
    end if
end procedure
```

reachable from another $F$-state in $S$. Since any $F$-state on a cycle is reachable from itself, it will never be removed from $S$.

The DiVinE model checker supports a number of SCC detection algorithms [14]. We summarize their two main algorithms: *One-Way-Catch-Them-Young* and *Maximum-Accepting-Predecessor*. OWCTY was first described in [57] and a parallel implementation was considered in [38]. It is similar to Algorithm 2.1, but tags states of $Cur$ with a counter of the number of predecessors each state currently has in $Cur$. The reachability computation determines the counters for each state reachable from the given set, and function ELIMINATENOPREDS utilizes these counters to removes any states from $Cur$ without a predecessor in $Cur$, until a fixpoint is reached. This approach can result in fewer loop iterations than in Algorithm 2.1, as it can pre-emptively remove states. For example, if $\langle Cur \rangle$ includes a long path of states that each have one incoming and one outgoing arc, except for the start of the path which has no incoming arcs, this entire path will be removed by the ELIMINATENOPREDS operation. The MAP algorithm tags states of $Cur$ with an $F$-state (or **null**), called its maximum accepting predecessor. The $F$-states are statically assigned an ordering $\prec$, where the **null** value is $\prec$ any $F$-state. Before entering the loop, *Cur* contains all states that can be reached from some (reachable) $F$-state, all tagged with **null**. The MAPs of are updated by finding, for each state of $s \in Cur$, the $\prec$-greatest state that has a path to $s$ within $\langle Cur \rangle$. The while loop checks if there exists a state of *Cur* that is its own MAP; if so, this $F$-state necessarily lies on a cycle. Otherwise, any state appearing as a MAP of some state of *Cur* does not lie on a cycle and is thus removed. Likewise, any state with a MAP of **null** can be removed as well, as such states cannot lie on an accepting cycle.

While the high-level pseudocode appearing in Algorithms 2.1, 2.2 and 2.3, were presented as sequential, they have direct parallel and distributed message passing implementations by using parallel methods for the breadth-first search [14].

**Algorithm 2.2** One-Way-Catch-Them-Young (OWCTY)

```
procedure OWCTY($G = (S, I, T), F$)
    $Cur$: set of (state, integer) pairs                    ▷ states tagged with ints
    $Old$: set of (state, integer) pairs
    $Cur \leftarrow (\textsc{Reachable}(G, I) \cap F) \times \{0\}$
    $Old \leftarrow \emptyset$
    while $Cur \neq Old$ do
        $Old \leftarrow Cur$
        $Cur \leftarrow \textsc{ReachableCountPreds}(G, Cur)$
        $Cur \leftarrow \textsc{EliminateNoPreds}(G, Cur)$
    end while
    if $Cur = \emptyset$ then
        property holds
    else
        Cur contains accepting states on cycle
    end if
end procedure
```

A model checking algorithm is considered to be "on-the-fly", if it may find a counterexample before constructing the entire graph. In [59], an on-the-fly algorithm for LTL model checking is given that attempts to use as little memory as possible by performing a DFS when checking the product automaton. This algorithm, utilized by the SPIN model

**Algorithm 2.3** Maximum Accepting Predecessor (MAP)

```
procedure MAP($G = (S, I, T), F$)
    $Accepting$: set of states
    $Cur$: set of (state, state or null) pairs
    $\prec$: ordering on $F$-states
    $Accepting \leftarrow \textsc{Reachable}(G, I) \cap F$
    $Cur \leftarrow \textsc{Reachable}(G, Accepting) \times \{\textbf{null}\}$
    $Cur \leftarrow \textsc{FindMAPs}(G, Cur)$                         ▷ compute MAP fixpoint
    while $Cur \neq \emptyset \wedge \forall(s, m) \in Cur.\ s \neq m$ do ▷ terminate if a state is its own MAP
        $Cur \leftarrow \textsc{EliminateMAPs}(Cur)$               ▷ remove all states that are MAPs
        $Cur \leftarrow \textsc{FindMAPs}(G, Cur)$                     ▷ compute MAP fixpoint
    end while
    if $Cur = \emptyset$ then
        property holds
    else
        $Cur$ contains accepting state on a cycle
    end if
end procedure
```

checker, generates a Büchi automaton for the negated property called the *tableau graph* by recursively breaking down the formula in question, creating nodes tagged with a formula holding presently, and with another formula that it's successors will satisfy. For example, if the formula $\mu\,\mathcal{U}\,\psi$ holds presently in a node, the key identity $\mu\,\mathcal{U}\,\psi \equiv \psi \vee (\mu \wedge \bigcirc\,(\mu\,\mathcal{U}\,\psi))$ is applied. The node is split into 2 nodes, one with present formula $\mu$ and successor formula $\mu\,\mathcal{U}\,\psi$ and another with present formula $\psi$ and an empty successor formula. Once all present formulas have been processed in node $u$, a successor node $v$ is created with a present formula set to the successor formula of $u$. The appeal of this approach is that when paired with an on-the-fly method of exploring the model, this tableau graph may also be explored on-the-fly with parts of it generated on-demand. This avoids generating the entire product automaton in cases where the model does not satisfy the property.

LTL model checking can also be performed using symbolic methods and BDDs [36]. The nuSMV model checker [41] handles LTL formulas by translating them to CTL formulas with fairness constraints [43]. Unlike our work, this model checker is symbolic and not implemented to run on a distributed cluster of machines.

Recent work includes that of Kesten *et al.* [74], where a symbolic LTL model checking algorithm is presented that explicitly integrates fairness in terms of compassion and justice [88, 89]. In general, fairness assumptions are properties of “interesting” execution traces, and traces that do not satisfy fairness are considered unrealistic and should be ignored as potential counterexamples. Compassion, also known as strong fairness, assumes that some event occurring infinitely often will be followed by another event occurring infinitely often. Justice, also known as weak fairness, assumes that some event occurs infinitely often. The key idea is to find the set of *feasible* states, that is the set of states that lie on some fair trace of the product automata. If this set is empty for a negated LTL formula, then the original LTL formula holds. Let the *predecessor closure* of a set of states $A$ be the reachable states via reversed transition arcs, starting from $A$. Let $\mathcal{J}$ be a set of predicates $J_i$, where a fair trace visits a $J_i$-state infinitely often for all $i$. Let $\mathcal{C}$ be a set of ordered pairs of predicates $(p_i, q_i)$, where a fair trace visits a $q_i$-state infinitely often if it visits a $p_i$-state infinitely often for all $i$.

The algorithm of Kesten *et al.* first symbolically computes the reachable states $S$ of the system. Then, the following three steps are iterated until a fixpoint is reached.

1. Remove from $S$ all states without a successor in $S$.
2. For each $J_i \in \mathcal{J}$, remove from $S$ all states that are not in the predecessor closure of the $J$-states.
3. For each $(p_i, q_i) \in \mathcal{C}$, remove from $S$ all $p$-states that are not in the predecessor closure of the $q$-states.

Finally, return the predecessor closure of $S$.

Previous symbolic algorithms dealt with compassion by adding the conditions as an antecedent to the LTL formula to check. Another approach is to transform a compassion constraint $(p, q)$ into a justice constraint, which involves adding an auxiliary boolean variable $b$ to the system. This variable is initialized to *false* and every transition may nondeterministically set it to *true*, from which point it remains *true* with each subsequent transition. Intuitively, this flag is "guessing" a point in the computation trace at which no further $p$-states will be visited. Thus, any $(b \wedge p)$-state is considered an error state where computation halts, and cannot be on any accepting fair trace. The additional justice constraint is $b \vee q$; either a $q$-state is visited infinitely often, or $b$ holds infinitely often (and persistently), indicating that $p$ cannot hold infinitely often.

The authors of [74] compare the performance of these 3 approaches for dealing with compassion (algorithmically, as an antecedent, transform to justice), and conclude that handling both compassion and justice algorithmically has the best performance. This paper also gives a clear and concise overview of the development of LTL tools and techniques. Like nuSMV, the approach here is symbolic and not distributed.

Černá and Pelánek made the important observation that the sequential symbolic algorithms of Kesten *et al.* are easily adapted to DEMC algorithms [38]. One such algorithm is Streett-Detect-Cycle (Algorithm 2.4). Input $G = (S, I, T)$ is a transition graph $\mathcal{C}$ is the set of compassion constraints as above. The algorithm returns *false* if the contains a reachable fair cycle, and *true* otherwise. Although this algorithm does not take a Büchi automata as input and does not technically solve the LTL model checking problem, it demonstrates how to check if *any* trace satisfies strong, state-based fairness.

For set $X \subseteq S$, Elimination$(G, A)$ is the set of states obtained by removing any state from $X$ without a predecessor in $\langle X \rangle$ until a fixpoint is reached. The algorithm is distributed by assigning each state an owner thread with a random uniform hash function. Each thread is responsible for the bookkeeping of all states it owns, including which reachable states are currently members of $S$. As with Algorithms 2.1, 2.2 and 2.3, all computations Reachability and Elimination can be implemented with BFS and message passing between threads, similar to the Stern-Dill DEMC algorithm used by PReach (see Algorithm 3.1). Our adaptation of Algorithm 2.4 that deals specifically with response properties is given in Chapter 6.

**Algorithm 2.4** A LTL DEMC Algorithm with Strong Fairness [38], slightly adapted

```
procedure STREETT-DETECT-CYCLE($G = (S, I, T), \mathcal{C}$)
    $Cur \leftarrow$ REACHABLE($G, I$)
    $Old \leftarrow \emptyset$
    while $Cur \neq Old$ do
        $Old \leftarrow Cur$
        for all $(p_i, q_i) \in \mathcal{C}$ do
            $Cur \leftarrow (Cur - p_i) \cup$ REACHABLE($G, Cur \cap q_i$)
        end for
        $Cur \leftarrow$ ELIMINATION($Cur$)
    end while
    $Cur \neq \emptyset$
end procedure
```

## 2.5 Summary

This survey of the literature puts our contributions into context. We build directly on Stern and Dill's parallel Mur$\varphi$ tool [52, 106], this making the PReach tool a peer to Eddy [96]. The CMP method for parameterized safety verification is the foundation of our contribution in proving parameterized DF. Our approach of distributed model checking of DF is a new model checking technique. The directed search and utilization of "helpful rules" is inspired by previous verification methods [37, 101] that involved flushing of transactions. The earlier work verified safety properties via refinement and was domain specific. Our technique provide a simple but useful and computationally tractable verification of DF that can be applied to a wide range of models. Finally, we leveraged previous model checking observations about handling fairness algorithmically [74] and the high level OWCTY algorithm for LTL model checking [57, 38] to devise a new algorithm for distributed model checking of response properties.

We bring these insights and ideas together in the PReach model checker. PReach is a robust, scalable, distributed explicit-state model checker that has been applied to real-world verification problems for hardware protocols. This tool was leveraged to prototype efficient DEMC of response properties with fairness, a problem of practical interest. Our parameterized verification work is novel in the space of parameterized liveness as it extends the successful CMP method to support deadlock freedom properties. In practice, the additional effort to show DF is small once CMP has been used to show safety. Together, our algorithms and their implementation in PReach demonstrate that DEMC is a practical way to automatically verify liveness properties that are of real-world significance and that have previously been beyond the capability of model checking tools.

# Chapter 3

# The PReach Model Checker

PReach is a distributed explicit-state model checker based on Mur$\varphi$. It was designed to be a reliable, easy to maintain, scalable model checker that was compatible with the Mur$\varphi$ specification language and suitable for industrial protocols. PReach uses legacy Mur$\varphi$ C++ code to handle the CPU and memory intensive aspects of the model checking computation. The communication and coordination for the parallel/distributed part is implemented in the concurrent functional language Erlang, chosen for its parallel programming elegance. This leverages the complementary strengths of C++ and Erlang and offers compatability with existing Mur$\varphi$ models.[1]

We used the original Mur$\varphi$ front-end to parse the model description, and borrow Mur$\varphi$'s back-end code for maintaining local hash tables and performing state expansions. The Erlang layer handles communication, load balancing, termination detection and other aspects of the parallel algorithm. This approach offers a clean and simple implementation, with the core parallel algorithms written in under 1000 lines of code. This chapter describes the high level algorithm, including the various features that are necessary for the large models we target. To date, the PReach model checker has verified an industrial cache coherence protocol with approximately 90 billion states. To our knowledge, this is larger

---

[1] This chapter is based on the first PReach model checker publication [22].

than any published model verified with an explicit-state model checker. PReach is in ongoing development has been released to the public under an open source BSD license [30].

## 3.1 Algorithm

PReach is based on the Stern and Dill distributed explicit-state model checking (DEMC) algorithm [104], a distributed breadth-first search[2] that partitions the space across the compute workers (a.k.a. *nodes*)[3]. A message passing environment is assumed, as well as a uniform random hash function that associates an *owner* node with each state. Pseudocode for Stern-Dill is given in Algorithm 3.1; details such as termination detection, error trace generation, as well as all PReach-specific features have been omitted for simplicity. We have also done some minor code reformatting compared to the original paper.

Each worker has two main data structures $T$ and $WQ$, which are a set of states and a list of states, respectively (declared on lines 1 and 2). We say a state has been *visited* if it has been received by its owner, and a state is *expanded* if its successors have been computed. Set $T$ is maintained to contain the states owned by the worker that have been visited, while $WQ$ are those states that have been visited but not yet expanded. The computation begins with each worker executing the main routine Search. After initializing $T$ and $WQ$ as empty and synchronizing with a barrier, Search sends the initial states to their respective owners in lines 8 to 12. Next each worker enters a while loop (line 13) that iterates until termination has been detected. Termination detection checks that each node has an empty work queue and no messages are in flight by comparing that total number of messages sent to the total number received. We use the same termination detection algorithm as in [106]; details are presented in that paper.

---

[2] Strictly speaking, the search order is not breadth-first because the timing of communication actions results in various visitation interleavings. However, the algorithm is "breadth-first" in that each worker operates on a FIFO of states.

[3] The workers can run on distinct hosts or some can seamlessly sit on the same host, *e.g.* on a multicore machine.

In the body of the while loop, procedure GETSTATES is invoked on line 25, which iteratively pulls incoming states from the *mailbox* (or *message queue*), that is, the runtime queue of messages sent to this node but not yet received by the thread. Each state $s$ in GETSTATES that is not in $T$ is inserted into $T$ and appended to the tail of $WQ$. Once there are no more states to receive from the mailbox, control returns to line 15 where $WQ$ is checked for emptiness. If it is nonempty, we pop a state $s$ from $WQ$ and compute its successors. Each successor is first canonicalized for symmetry reduction [70] on line 18, and then the resulting state $s'_c$ is sent to its owner.

We now discuss PREACH's implementation. To harness fast and reliable code, PREACH uses existing Mur$\varphi$ code for many key functions involved in explicit-state model checking, such as:

- front end parsing of Mur$\varphi$ models,
- state expansion and initial state generation,
- symmetry reduction,
- state hash table $T$ look-ups and insertions,
- invariant and assertion violation detection,
- pretty printing of states (for counter-example traces).

To facilitate Erlang calls to Mur$\varphi$ functions, we wrote a light-weight wrapper on top of existing Mur$\varphi$ C code, which we call the *Mur$\varphi$ Engine*. We also employ the Mur$\varphi$ front-end that compiles the Mur$\varphi$ model into a C++ library.

Aside from space and time efficiency during model checking, using the Mur$\varphi$ Engine has another clear benefit: the Erlang code need only handle management of the distributed aspects of the algorithm, while the Mur$\varphi$ Engine handles key computations that pre-existing Mur$\varphi$ code can do quickly and correctly. The Erlang code, at a high level, resembles Algorithm 3.1 with key functions such as *Successors*(), *Canonicalize*(), and *Insert*() calling into the Mur$\varphi$ Engine. One notable exception to this paradigm is the work queue $WQ$; this is handled in Erlang code and uses an optimized disk-file to allow for efficient processing of very long queues.

**Algorithm 3.1** Stern-Dill DEMC

```
$T$ : set of states
$WQ$ : list of states

procedure SEARCH( )
    $T \leftarrow \emptyset$
    $WQ \leftarrow []$
    $barrier()$
    if I am the root then
        for each start state $s$ do
            Send $s$ to $Owner(s)$
        end for
    end if
    while $\neg Terminated()$ do
        GETSTATES()
        if $WQ \neq []$ then
            $s \leftarrow Pop(WQ)$
            for all $s' \in Successors(s)$ do
                $s'_c \leftarrow Canonicalize(s')$
                Send $s'_c$ to $Owner(s'_c)$
            end for
        end if
    end while
end procedure

procedure GETSTATES( )
    while there is an incoming state $s$ do
        $Receive(s)$
        if $s \notin T$ then
            $Insert(s, T)$
            $Append(s, WQ)$
        end if
    end while
end procedure
```

Stern and Dill's distributed model checking algorithm [106] partitions the state space among processes with a uniform random hash function. Processes are said to *own* states that hash to their process IDs. Once a state has been visited, its owner process is responsible for storing it locally. In PReach this is done with the Mur$\varphi$ model checker's hash table [103] which uses a predetermined number of bits[4] to represent each state. The use of hash compaction and bloom filters in explicit-state model checking is a thoroughly studied area [113, 53] and lends itself to practical approaches. Hash table compression admits a small probability that some state will erroneously be viewed as visited when it actually hasn't been. In our experience this probability is tiny; for example, a very large model checking experiment with about 100 billion states had only a 0.03% chance of a missed state [22]. The experiments in this chapter admit a much smaller probability than this; the `German6` model with over 316 million states had a probability of a missed state of less than $7.36 \times 10^{-5}$. If this probability were of practical concern, the user could simply re-run the tool using a different seed for randomization and reduce the probability of a missed state in *both* runs to less than $(7.36 \times 10^{-5})^2 < 5.42 \times 10^{-9}$.

## 3.2 Crediting

An important problem encountered when running large models with early versions of PReach was that some workers would slow in state expansion rate or completely crash. In a naive implementation of DEMC, each node sends all the successors of a state to the owners of the successors immediately upon expanding a state. After analyzing Erlang mailbox[5] size statistics, we found that the problematic nodes were accumulating a disproportionate number of messages in the mailbox. Simple tests of producer-consumer code written in Erlang have revealed that the time for an Erlang process to receive a single message in-

---

[4] This number is a configuration parameter. The results in this chapter use the default value of 40 bits.

[5] The *mailbox* stores the incoming messages in the Erlang runtime that have yet to be received by the Erlang program. We also write *message queue* when referring to *mailbox*.

creases in the number of mailbox messages. Once a worker falls behind the others for any reason, (such as one worker receiving many states at once or OS scheduling) it tends to permanently fall behind the others. As these mailbox messages piled up, the node would eventually allocate more memory and this would result in slower processing of messages. This effect cascaded until the node started paging and was unable to do anything at a reasonable speed.

Our remedy was to introduce a crediting mechanism. Each node is initially granted a number of credits $C$ per peer node; each credit allows one unacknowledged message from the peer. When a message containing states is received by another node, an acknowledgment is returned. This provides a hard bound on the total number of messages a node may have in its message queue at any one time, *i.e.*, $nC$ where $n$ is the number of nodes. We note that no performance penalty was observed due the additional acknowledgment messages. When no credits are available to send states to a particular node $p$, the states are locally queued in an *outbox*. A round-robin scheduler ensures that periodically, the number of credits for $p$ is checked and states waiting in the outbox are sent. In cases when the outbox for $p$ grows large, this queue of messages is temporarily written to disk to avoid running out of memory.

## 3.3 Light Weight Load Balancing

When a distributed program runs in a heterogeneous computing environment, the total runtime is determined by the runtime of the slowest node. If one node takes much longer than the others to finish its work, it does not matter how fast the others were. This can be caused by some nodes being intrinsically slower than others, or dynamically assigned more work than others. In the context of DEMC, the notion of "work" is typically measured as the number of states awaiting expansion, which is the length of $WQ$. Load balancing therefore plays an important role in efficiency, and as explained below, can prevent extreme slowdowns and crashes. Previous work by Behrmann [19] and Kumar and Mercer [77]

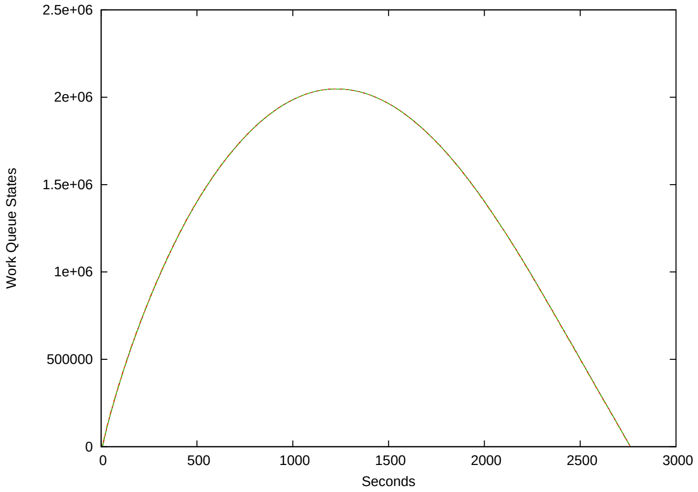


Figure 3.1: Load balancing with Kumar and Mercer's algorithm

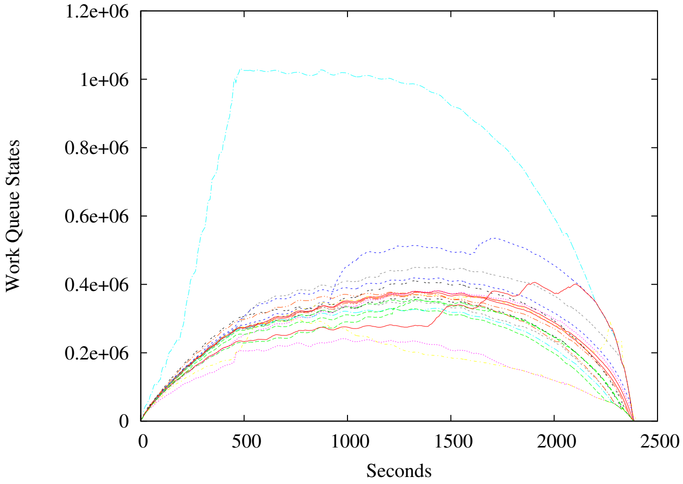


Figure 3.2: Light weight load balancing

observed the problem of an unbalanced work load in the context of DEMC. Surprisingly, although at the end of the computation all nodes have expanded roughly the same number of states (due to uniform hashing) [104], the dynamic work queue sizes during model checking can vary wildly between workers.

To address this problem, Kumar and Mercer proposed a rebalancing scheme that attempts to keep the work queues of all nodes similar in length throughout the computation [77]. This is achieved by comparing work queue sizes between hypercube adjacent nodes and passing states to neighboring nodes with smaller work queues. This is done in an aggressive manner, with the goal of keeping all work queues roughly the same lengths. In PReach, work queues are kept on disk and we are not concerned about minimizing the maximum work queue size (as Kumar and Mercer achieve). Rather, we seek to avoid nodes sitting idle with an empty work queue. Our scheme, which we call *light weight* load balancing, reduces the overhead of work queue balancing. Each node tracks the sizes of all other nodes' work queues by sending work queue size information along with state messages. When one node notices that a peer node has a work queue that is *LBFactor* times smaller than its own (for a fixed parameter *LBFactor*), it sends some of its states to the peer. Empirically, we found $LBFactor = 5$ to be a good choice; it allows enough load balancing to occur so that nodes complete around the same time without doing too much unnecessary rebalancing. If disk usage were an issue, one could use a smaller factor to keep the maximum work queues smaller at the expense of some extra load balancing.

To show the difference between the two schemes, we ran a simple $10^7$ state counter model using PReach with both schemes. In Figure 3.1 we see strict balancing that keeps the work queue sizes of all nodes identical. In Figure 3.2 we see light weight load balancing, which allows the work queues to vary somewhat. In both cases, however, all nodes complete around the same time. The benefit of the light weight scheme is that it is able to process states faster because it is doing less load balancing and it completes sooner, at 2404 seconds as opposed to the stricter scheme which finishes in 2768 seconds. Also notice that the longest work queue in Figure 3.2 is about half the length of all work queues in Figure 3.1. This is due to more work piling up as more computational resources are used for load balancing states, as opposed to the light weight scheme.

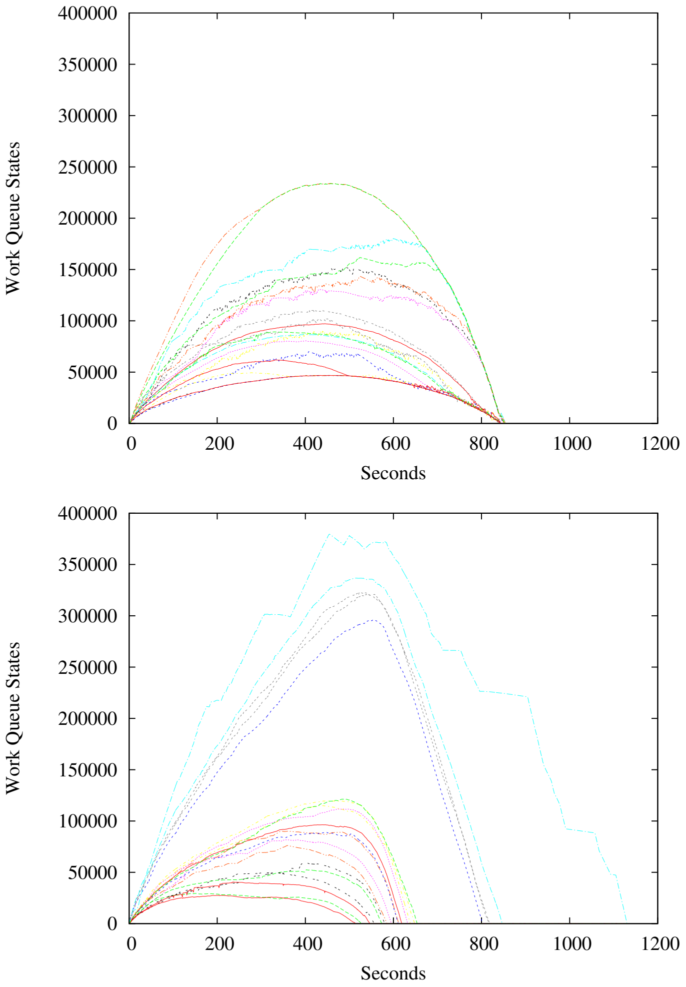


Figure 3.3: Load balancing effect on work queues. German model with 9 caches using 20 workers – load balancing enabled (top) and disabled (bottom).

To show the improvement between a load balanced run and a non-load balanced run, we ran several well known protocols: SCI, LDASH and German. These protocols are included in the Mur$\varphi$ distribution. We ran them on 60, 40, and 20 nodes, respectively. The results for the German protocol are in Figure 3.3, which clearly demonstrate the benefit of light weight load balancing. The comparisons for the LDASH and SCI protocols exhibit similar speedup and graph characteristics; see Figures 3.4 and 3.5.

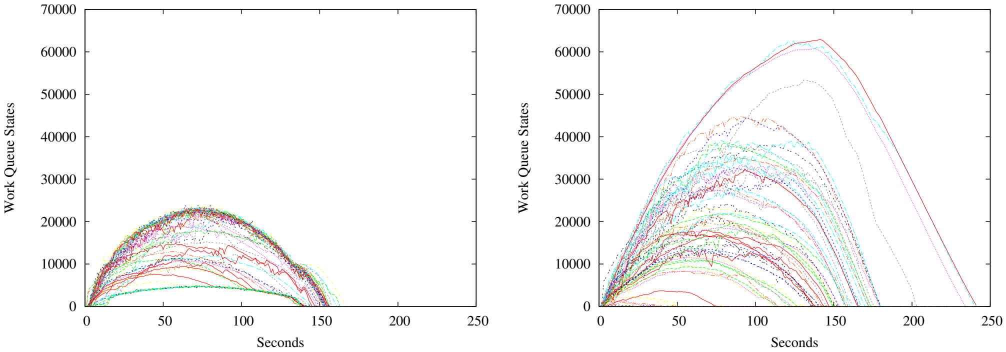


Figure 3.4: LDash model load balancing (60 workers); load balancing on (left) and off (right)

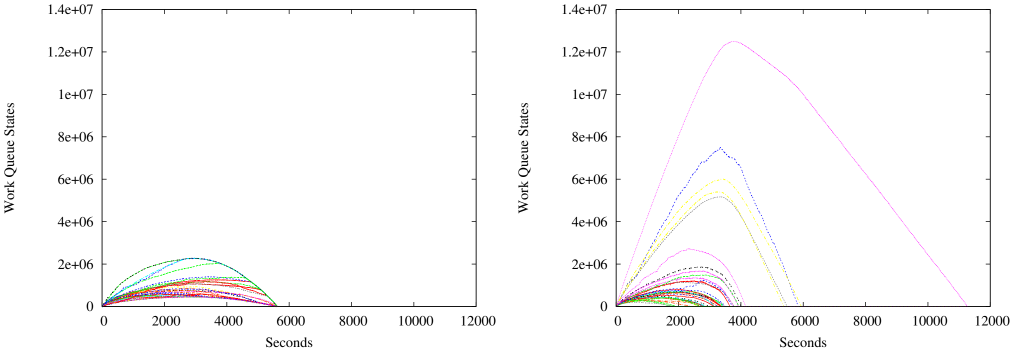


Figure 3.5: SCI model load balancing (40 workers); load balancing on (left) and off (right)

## 3.4 Batching of States

Stern and Dill's algorithm, and consequently PReach, involves communicating successor states to their respective owners (see line 19 in Algorithm 3.1). The number of reachable transitions is typically at least a factor of 4 larger than the number of reachable states; thus, we could easily end up communicating hundreds of billions of states in the large models we target. This motivates us to look at reducing the number of messages sent, which reduces the number of calls to Erlang's *Send* and *Receive* communication primitives, along with utilizing the network bandwidth more efficiently. Indeed, in simple producer-consumer Erlang benchmarking tests we have observed a factor of 10 to 20 speedup by sending states in batched lists of length 100 to 1000 as opposed to sending the states individually.

A simple batching variant of the Stern-Dill algorithm, which was implemented in an early PReach prototype is as follows. Each worker batches generated states in separate out queue for each peer node before sending. This mechanism is controlled by two parameters: *MaxBatch* and *FlushInterval*. A batch of states for a peer node is sent when *MaxBatch* states have accumulated for the peer, or of *FlushInterval* seconds have passed since the last batch to the peer, whichever happens earlier. Furthermore, the out queues will neglect waiting for a full batch of states before sending whenever the work queue is short. This typically occurs only near the start and near the end of the entire computation. We found that setting $MaxBatch = 1000$ and $FlushInterval = 1$ second achieved good performance. However, for very large models it is possible that different values would be needed, or an adaptive scheme where their values vary between nodes and over time. This was not explored in depth because while batching helps reduce runtime, some models suffered serious performance issues where a small number of nodes would become overwhelmed with states; this problem was described in Sect. 3.2. Fortunately, batching of states is compatible with both methods that alleviate this problem: load balancing and crediting. When load balancing is activated, it is known *a priori* how many states will be sent from one node's work queue to the other node's work queue, so the states can be trivially batched into one message. With crediting, we must choose a value for *MaxBatch*, but the timeout *FlushInterval* is unnecessary because *MaxBatch* states are sent to node $p$ whenever a credit is available for $p$ and the round-robin scheduler selects $p$. Thus, our batching approach involves only the parameter *MaxBatch*, which bounds the maximum number of states aggregated into one message.

Figure 3.6 shows the benefit of varying the *MaxBatch* parameter in conjunction with crediting.

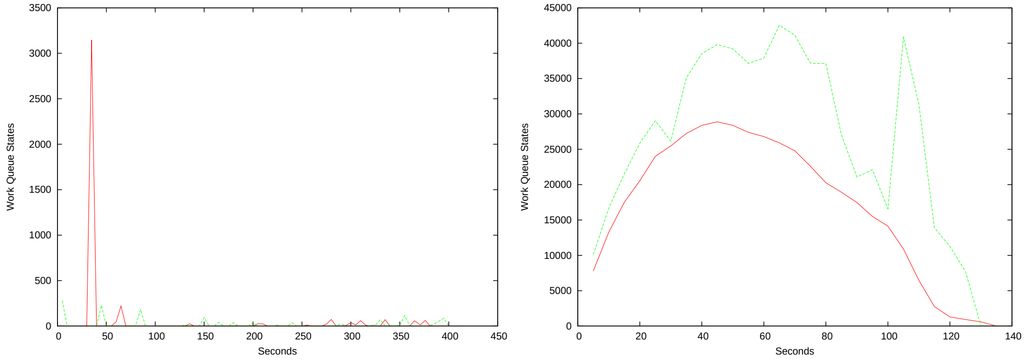


Figure 3.6: State batching effect on the counter model with $10^5$ states using 2 workers. Message queue lengths with $\mathit{MaxBatch} = 1$ (left) and $\mathit{MaxBatch} = 100$ (right).

## 3.5 PReach Pseudocode

In Algorithm 3.2 we show pseudocode for the PREACH algorithm. The basic outline of the code follows closely the original Stern-Dill algorithm. However, the key features of batching, crediting, and load balancing all require changes. First, in order to implement batching, the outgoing states are placed in the queue *OutQ* (line 22) instead of being sent directly to their destination. These states are actually sent out later, in line 25, for the current destination. We visit the outgoing queues in a round robin manner and attempt to send to one node after each state is expanded. Next, for crediting, we return acknowledgments (line 33) from states sent to us in GETSTATES. Each node keeps track of the credits it has for sending to every other node with *Cred*. (Note: not shown is the increment and decrement of $\mathit{Cred}[p]$ when sending states to or receiving states from $p$, respectively). The *SendQ* function contains the rest of the changes. It implements the logic that decides when we should send to a particular peer, using its estimate *WQEst* of how large the other node's *WQ* is. It checks to see that credits are available in *Cred*, and ensures that we do not send states to a node that is currently sending us load balancing states (line 43). Next, we check to see if the node is eligible to receive load balancing states (line 44; note that load balancing is only enabled if the worker's work queue length is greater than some

**Algorithm 3.2** PReach DEMC

```
 1: $T$ : set of states
 2: $WQ$ : list of states
 3: $OutQ[n]$ : array of list of states                    ▷ $n$ is number of nodes
 4: $Cred[n]$ : array of int                               ▷ $C$ is number of credits
 5: $WQEst[n]$ : array of int
 6:
 7: procedure SEARCH
 8:     $T \leftarrow \emptyset$; $WQ \leftarrow []$; $Cred \leftarrow [C, ..., C]$
 9:     $OutQ \leftarrow [[], ..., []]$; $WQEst[n] \leftarrow [0, ..., 0]$
10:     $CurDest \leftarrow 0$
11:     if I am the root then
12:         for each start state $s$ do
13:             Send $s$ to $Owner(s)$
14:         end for
15:     end if
16:     while $\neg Terminated()$ do
17:         GETSTATES()
18:         if $WQ \neq []$ then
19:             $s \leftarrow Pop(WQ)$
20:             for all $s' \in Successors(s)$ do
21:                 $s'_c \leftarrow Canonicalize(s')$
22:                 $Enqueue(s'_c, OutQ[Owner(s'_c)])$
23:             end for
24:         end if
25:         SENDQ($CurDest$)
26:         $CurDest \leftarrow (CurDest + 1) \pmod{n}$
27:     end while
28: end procedure
```

threshold *WQminLB*). Following any load balanced messages we check to see if we should send states owned by the current peer from the outbox. We try to wait until there are at least *MaxBatch* messages in a batch (typically *MaxBatch* = 100), but if the destination's work queue is low or if we don't have any work ourselves, then we will send smaller batches (line 51).

```
function GetStates()
    while there is an incoming state-message $M$ do
        ▷ $S$ list of states, $p$ sender ID, $L$ sender $WQ$ length
        $\{S, p, L\} \leftarrow Receive(M)$
        $SendAck(p)$
        $WQEst[p] \leftarrow L$
        for all $s \in S$ such that $s \notin T$ do
            $Insert(s, T)$
            $Enqueue(s, WQ)$
        end for
    end while
end function

function SendQ($dest$)
    if $Cred[dest] > 0 \;\wedge\; WQEst[dest] < LBFactor * length(WQ)$ then
        if $length(WQ) \geq WQminLB \;\wedge\; LBFactor * WQEst[dest] < length(WQ)$ then
            $NumStates \leftarrow min\{length(WQ), MaxBatch\}$
            Dequeue $NumStates$ states from $WQ$ and send to $dest$
        end if
        if $OutQ[dest] \geq MaxBatch \vee$
        $(OutQ[dest] > 0 \wedge (length(WQ) = 0 \;\vee\; WQEst[dest] < MaxBatch))$ then
            $NumStates \leftarrow min\{length(OutQ[dest]), MaxBatch\}$
            Dequeue $NumStates$ states from $OutQ[dest]$ and send to $dest$
        end if
    end if
end function
```

## 3.6 Results

Here we present a number of experiments detailing the performance of PReach. Most of these experiments were run on machines from a heterogeneous computing pool consisting primarily of 2.5-3.5GHz Core 2 and Nehalem class Intel machines. Typically 4-8GB of memory was available per worker, although most experiments did not use all of this. Table 3.1 presents a few of the larger models we have verified with PReach.

Though previous implementations of the Stern-Dill DEMC algorithm report linear speed-up [96, 104], it is important to show that such speed up is also achieved in our imple-

| Model | States ($\times 10^9$) | Nodes | Time (hours) | States per Sec per Node |
|---|---:|---:|---:|---:|
| Pet8 | 15.3 | 100 | 29.6 | 1493 |
| Intel3_5 | 10.1 | 61 | 24.7 | 1860 |
| Intel3_7 | 28.2 | 92 | 90.2 | 945 |

Table 3.1: Large model experiments. Here Pet8 is Peterson's mutual exclusion algorithm over 8 clients (with no symmetry reduction), and Intel3 is an Intel proprietary cache protocol (with symmetry reduction enabled). The last two rows are for Intel3 with respectively 5 and 7 transaction types enabled.

mentation, especially given that a high level language (Erlang) is handling communication details. Figure 3.7 shows the speed-up for $n$ nodes against Mur$\varphi$ for a few public domain models. In all cases except german7 (German's protocol [58] over 7 caches) we see near linear speed-up. The diminishing returns of german7 is expected for smaller models; the number of reachable states (after symmetry reduction) is less than $2 \times 10^6$, and Mur$\varphi$ completes checking it in only 315 seconds. In contrast, Mur$\varphi$ took an hour or more for the three other protocols (german9: 6242 seconds, mcslock2: 8386 seconds, ldash: 3583 seconds). The slope of the speedups of Figure 3.7 are all less than 0.3, where the slope of the mcslock2 speedup is especially underwhelming. This slope is determined by the ratio of runtime spent in the Erlang code versus the Mur$\varphi$ engine. For these models, the Mur$\varphi$ engine time (for state expansions, checking invariants, etc.) is relatively small. However, for the industrial Intel models in Table 3.1 the Mur$\varphi$ Engine takes longer for these operations, and we estimate the slope of the speedup for these models to be about 0.5.

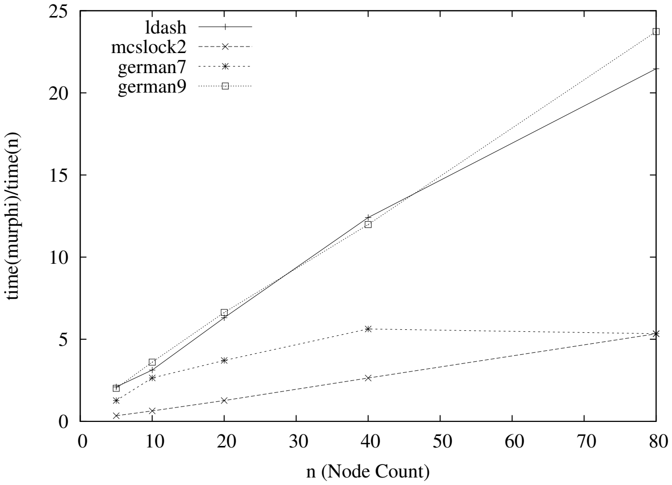


Figure 3.7: Speed up experimental results

# Chapter 4

# Model Checking of Deadlock Freedom

This chapter presents a practical method for verifying Deadlock Freedom (DF) properties in guarded command systems. Such properties are expressed in CTL as $\mathsf{AG}\,\mathsf{EF}\,q$ where $q$ is a set of *quiescent* states. We require the user to provide transitions of the system that are "helpful" in reaching quiescent states. The distributed search constructs a path consisting of helpful transitions from each reachable state $s$ to some state that is either quiescent or is known to have a path to a quiescent state, thereby demonstrating that such a path for $s$ exists. We extended the PReach model-checker with these algorithms. Performance measurements on both academic and industrial large-scale models shows that the overhead of checking deadlock-freedom compared with state-space enumeration alone is small.[1]

DF properties are a weak form of liveness. A system could satisfy DF and yet contain cycles of non-$q$-states, and thus some executions may visit an infinite sequence of non-$q$-states. However, DF properties provide a necessary but not sufficient condition for appropriate liveness properties to hold. We show that DF is straightforward to formalize, in particular it avoids the specification of fairness conditions. It is also less computationally

---

[1] This chapter is based on a published tool paper [26].

intensive to model check when compared with even simple liveness properties. We explore one such liveness property called *response* in Chapter 6.

## 4.1 Overview

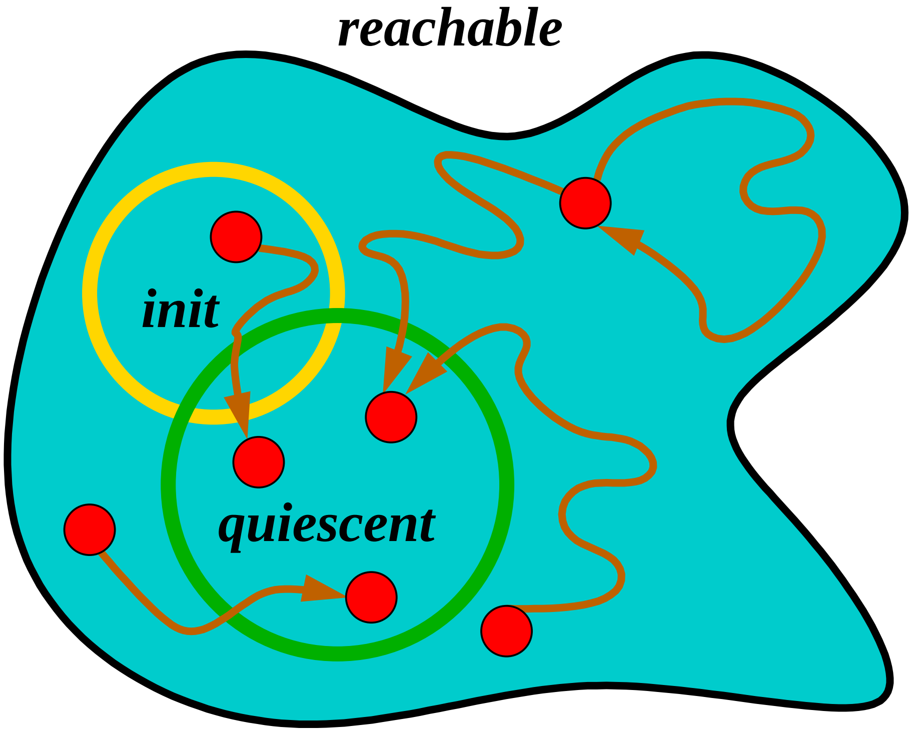


Figure 4.1: Illustration of deadlock freedom

Automatic checking of liveness properties is a challenging task. Approaches to address this generally require the user to carefully specify system *fairness* assumptions that are necessary for liveness to hold. Furthermore, checking liveness is computationally expensive, being more sensitive to the state-space explosion problem than simple state-space enumeration. A broad class of liveness failures of practical importance is *deadlock*, wherein one or more transactions are blocked, for example, due to a cyclic resource dependency [64]. In such a state, there exists no path to a state where all transactions have completed. This is our motivation for characterizing deadlock-freedom by a property $\mathsf{AG}\,\mathsf{EF}\,q$.[2] This property is illustrated in Figure 4.1, where some reachable states are highlighted and shown to have paths to a quiescent state. Notice that a system may have a cycle of non-quiescent states and still satisfy deadlock freedom. There only need be a path from all states, including those in such cycles, to some quiescent state.

A feature added to the PReach tool of the previous chapter and the focus of this chapter is an explicit state model checking technique to verify the CTL [44] property $\mathsf{AG}\,\mathsf{EF}\,q$. This property (of recent interest [62]) says "*for all reachable states, there exists a path to some q-state*". In our approach to verifying $\mathsf{AG}\,\mathsf{EF}\,q$, the system is modeled using guarded commands, and the user identifies a subset $\mathcal{H}$ of these commands as *helpful*. These commands are the ones that the user expects will cause the system to make progress towards $q$. If $s$ is a state and $s' \neq s$ can be reached from $s$ by performing a helpful command, then we say that $s'$ is a *helpful successor* of $s$. Thus, from any reachable state $s_1$,[3] we seek a *witness* – a path to some $q$ state. If $s_1$ is a $q$-state, then the path is trivial; otherwise, PReach computes $s_2$, a helpful successor of $s_1$. If $s_2$ is a $q$-state then we have found a witness path for $s_1$; otherwise, $s_3$, a helpful successor of $s_2$ is computed. This process iterates, building a witness $\rho = s_1, s_2, \ldots$ until a state $s_i$ is found where either

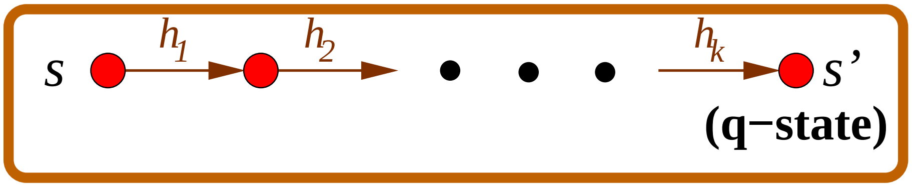
- $s_i$ is a $q$-state (Figure 4.2a), or
- $s_i$ has no helpful successor (referred to as *$\mathcal{H}$-stuck*; Figure 4.2b), or
- $s_i$ already appears in $\rho$ (Figure 4.2c).

In the first case, $\rho$ is a witness for $s_1$ and PReach continues with its standard state-space exploration algorithm. If a witness is found for every reachable state then we say that $\mathsf{AG}\,\mathsf{EF}_{\mathcal{H}}\,q$ holds, since the "$\mathsf{EF}$" paths consist entirely of transitions from $\mathcal{H}$. Clearly

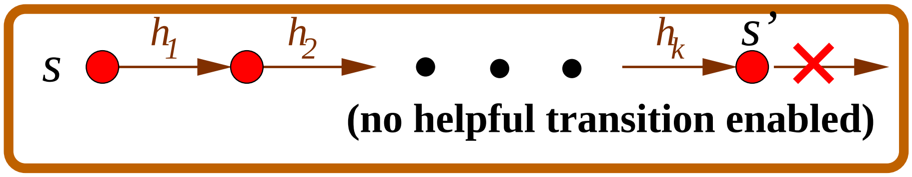
---

[2] Some literature and tools identify deadlock with the much weaker property that all reachable states have at least one (possibly unique) successor. We use the stronger form, $\mathsf{AG}\,\mathsf{EF}\,q$ throughout this chapter.

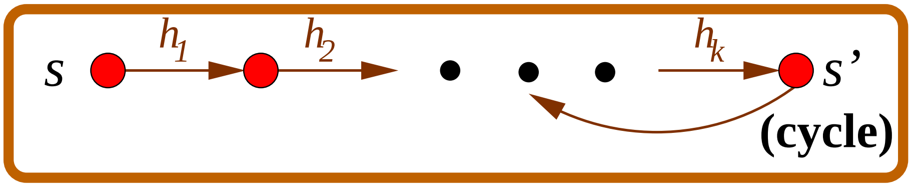
[3] PReach can also verify the more general property $\mathsf{AG}\,(p \rightarrow \mathsf{EF}\,q)$, but for this paper we assume for brevity that $p$ is *true*.



(a) A path is found to some $q$-state $s'$



(b) A path is found to some state $s'$ with no helpful transition enabled



(c) A path is found that cycles

Figure 4.2: Outcomes of a witness search

$\mathsf{AG}\,\mathsf{EF}_{\mathcal{H}}\,q$ is sufficient for $\mathsf{AG}\,\mathsf{EF}\,q$. In the other two cases, PReach halts and reports the path $\rho$ to the user. These cases do not imply $\neg\mathsf{AG}\,\mathsf{EF}\,q$. For example, if the helpful rule list is empty and there exists a reachable state that is not a $q$-state, then nontrivial witnesses will never be found. Likewise, PReach may choose a sequence of transitions that leads to a cycle even though a path to a $q$-state exists. While either error could be a false negative, as we report in Section 4.4, in practice such failures can show that behaviors of the model are not those intended by the designer and thereby reveal real errors.

In our experience, many guarded command models have a clear partition between commands that inject a new request and those that service existing ones. Therefore, it is not difficult to identify a suitable set of rules $\mathcal{H}$ and a quiescent state predicate $q$. We show through experiments that using only helpful commands to form paths is sufficient in practice to verify $\mathsf{AG}\,\mathsf{EF}\,q$. Assuming that EMC has already enumerated the reachable states as part of safety property verification, the additional time to verify deadlock freedom is usually a small fraction of the time as compared with state-space enumeration.

It is critical for performance to leverage the known witnesses during subsequent searches. Suppose a witness path $\rho_1$ has been found for state $s$, and $s$ is encountered on path $\rho_2$ while searching for witness path for $s'$. Clearly, the concatenation of paths $\rho_1$ and $\rho_2$ is a witness path for $s'$. PReach uses a dedicated state hash table for this purpose, called **WHT** (witness hash table). Each time a witness path is found for some state, it is added to the **WHT**; when we check if some state $s_i$ is a $q$-state, we also check for membership of **WHT**. Henceforth, we use $q \vee \mathbf{WHT}$ to denote states that are either $q$-states or members of **WHT**.

### 4.1.1 Specification Syntax

The Mur$\varphi$ syntax is extended to specify these deadlock freedom (DF) properties. The basic structure is

```
liveness "<Property Name>"
<P-predicate>
CANGETTO
<Q-predicate>
```

where `liveness` and `CANGETTO` are keywords, `<Property Name>` is a string that names the property, and `P-predicate` and `Q-predicate` are predicates describing $p$ and $q$, written in the same way as guard predicates are written in Mur$\varphi$ programs. For example, the German cache coherence protocol Mur$\varphi$ code contains a flag `ExGntd` that indicates if exclusive access has been granted to the (single) cache line. Suppose we wanted to check that if exclusive access is granted, there exists a sequence of actions that removes this access. In

Mur$\varphi$ syntax, this is written as

```
liveness "ExSurrendered"
ExGntd = true
CANGETTO
ExGntd = false
```

The set of helpful rules $\mathcal{H}$ are specified in terms of Mur$\varphi$ rules. Each rule, which defines a set of transitions, is expressed in Mur$\varphi$ as

```
rule "<Rule Name>"
<Guard>
==>
<Command>
```

The helpful rules are determined by passing *nonhelpful rules* flag `-nhr R1 ... Rk` to PReach[4]. Any rule with a name that contains any of the strings `R1 ... Rk` will be excluded from $\mathcal{H}$, and all other rules are included.

## 4.2 Algorithms

This section describes our algorithms that extend the general Stern-Dill paradigm of statically partitioning the state-space and assigning states to owner workers, as covered in Chapter 3. The data structure **WHT** is assumed to only store states that are owned by a given worker. In a sequential implementation, states could be inserted into **WHT** when the search for a witness path begins, since **WHT** is not consulted by subsequent searches until a witness path is found. The prefix of the search path must be maintained in order

---

[4] For our benchmarks, there are more helpful rules than nonhelpful ones, so it is easier to provide the latter.

to detect a cycle, but can be discarded once a witness path is confirmed. In a parallel implementation we assume state ownership is distributed across workers, *i.e.*, a worker will only insert owned states into **WHT**.

Under this paradigm, it is unsound to insert states into **WHT** while it may be consulted by other workers until it is *confirmed* that a witness path exists. This is due to searching for witnesses for states that query one another. Consider Figure 4.3, where worker 1 owns state $a$ and worker 2 owns state $b$, and transitions $(a, b)$ and $(b, a)$ are helpful. Assuming that neither $a$ or $b$ are quiescent, then DF does not hold since neither state has a witness. To find a witness path for $a$, worker 1 might query worker 2 to check if $b$ is a member of $q \vee$ **WHT**, and vice-versa. In a distributed setting, these two queries might occur at the same time. If (say) worker 1 reports a witness exists for $a$ (rather than a search is pending), then worker 2 may erroneously conclude that a witness exists for $b$. Therefore, our algorithms must be wait for a confirmation of a witness path before inserting states into **WHT**.

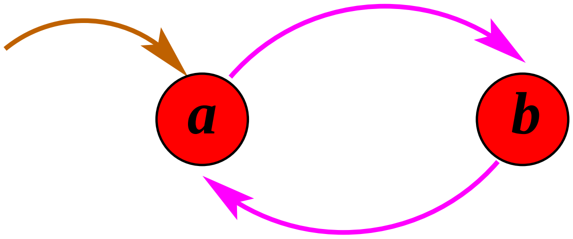


Figure 4.3: A subgraph that does not satisfy DF

### 4.2.1 Local Search

This search involves no communication to other workers during a witness search. When a worker in the distributed reachability algorithm encounters a new state $s$, the worker computes a path $\rho$ as described in Section 4.1. That is, starting with $[s]$, the path $\rho$ is constructed by iteratively computing a helpful successor of the current head of $\rho$. Once some state $s' \in q \vee$ **WHT** is encountered, all states of $\rho$ that are owned by this worker will be added to **WHT**, which necessarily includes $s$. This path computation is *not* distributed across workers, and thus, redundant path computations occur. While this approach scales poorly when the reachability analysis is run on a large number of machines, it provides a baseline that is free from communication overhead.

### 4.2.2 Pass-the-Path

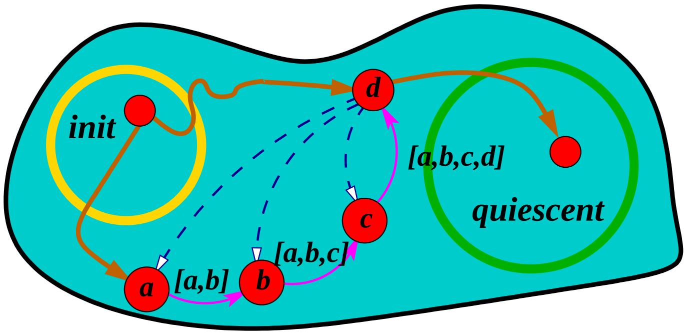
*Pass-the-path* (PtP) distributes the witness path searches by forwarding the current path prefix to the owner of the next state. When a state $s$ is found in the reachability analysis, if it is already in $q \vee \mathbf{WHT}$ then a witness path is known to exist and no further work is needed. Otherwise, an enabled helpful rule is chosen; the successor state, $s'$ is computed, and the search message $([s], s')$ is sent to the owner of $s'$. Here, $[s]$ is a list of states representing the current prefix path. This process continues constructing a prefix path $\rho$, communicating search messages of the form $(\rho, s_{cur})$ to the owner of $s_{cur}$, until a member of $q \vee \mathbf{WHT}$ is reached, a cycle is encountered, or no helpful commands are enabled. In the first case, the owners of all states along the path are notified and they update their **WHT**s, and otherwise a failure is reported. See Figure 4.4a for an example.

Notice that PtP allows redundant searches and acknowledgments to occur because workers keep no record of which states have pending searches for witness paths.

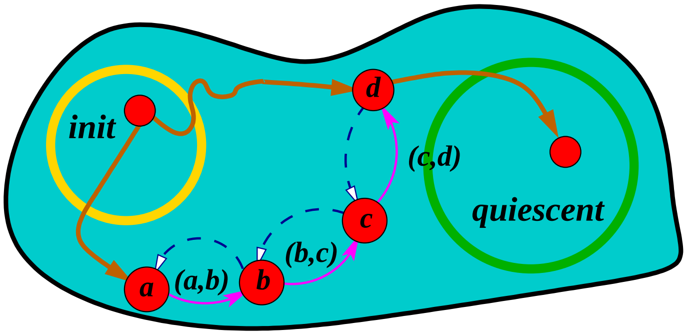
### 4.2.3 Outstanding Search Table

These redundant searches can be avoided if workers keep track of which states have outstanding witness searches. We have implemented such an approach where each worker maintains the pending searches in local table of states pairs **ST** (search table). This approach called OST has the benefit of search messages containing only a pair of states $(s, s')$. If such a message arrives and there is a pending search for $s'$ in the table, then $s$ is added to a list of states that must be acknowledged as having a witness once $s'$ is acknowledged.

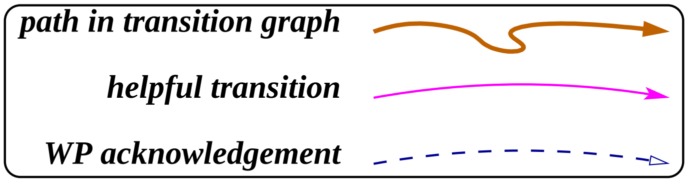
When starting a witness search for owned state $s$, if there is no element $(s, \cdot)$ in **ST**,



(a) PtP example. When state $d$ is encountered by the witness search, the owner of $d$ sends a message confirming the witness to for all states in the path.



(b) OST example. When state $d$ is encountered by the witness search, states in the path are confirmed to have a witness in reverse order, chaining through entries in the OST tables.



(c) Legend

Figure 4.4: PtP/OST example, assuming that $d$ is in **WHT**, but $a$, $b$ and $c$ are not.

then store $(s, \mathsf{null})$ in **ST**; find $s'$, a helpful successor of $s$; and send $(s, s')$ to the owner of $s'$. Otherwise, if $(s, \cdot)$ exists in **ST**, it signals that a witness search is already pending for $s$. When the owner of $s'$ receives $(s, s')$ and $s'$ is not $\mathcal{H}$-stuck, then

- If $s'$ is a member of $q \vee \mathbf{WHT}$, send an $(\mathit{WAck}, s)$ message to the owner of $s'$.

- Else, if **ST** contains an entry $(s', \cdot)$, then insert $(s', s)$ into **ST**. There is a pending witness search for $s'$ and the witness search for $s$ is blocked from propagating.
- Otherwise, insert $(s', s)$ into **ST**; find $s''$, a helpful successor of $s'$; and send the message $(s', s'')$ to the $owner(s'')$.

In the case that $s'$ is $\mathcal{H}$-stuck, a counterexample trace may be generated by sending message $(cex, [s, s'])$ to $owner(s)$ and chaining these messages backwards through the path until the state that started the search is found. When the owner of $s$ receives message $(WAck, s)$, insert $s$ into **WHT** and send message $(WAck, s_i)$ to the owner of non-null $s_i$ for every entry $(s, s_i)$ in **ST**, and delete these entries.

Notice that witness searches that cycle are not immediately detected, but will result in **ST** entries that are never removed. When state-space exploration completes and all sent messages have been received, the termination detection of PReach will observe that each node has an empty work queue and no messages are in-flight, but there are non-empty **ST**s. At this point a counterexample cycle can be generated similar to the $\mathcal{H}$-stuck case above, starting with some **ST** entry $(s_i, s_{i-1})$. Thus OST has the disadvantage of not detecting these cycles until all states have been visited, although heuristics could be designed that try to discover cycles earlier. Counterexample trace generation in this mode is not yet implemented.

### 4.2.4 Implementation Notes

For ease of implementation and for repeatable experiments, we restrict the helpful successor of a given state to be fixed but arbitrary. Thus, there is exactly one helpful successor of each state provided that at least one exists.

The Mur$\varphi$-based hash table described in Chapter 3 is once again utilized to store explored states. However, we "tag" each hash table entry with two bits – one to indicate if the state has been visited in the state enumeration and one to indicate if a witness has been confirmed. These bits are a small memory overhead as the hash table uses a 40 bit value to store each state.

The deadlock freedom algorithms can take advantage of load balancing similar to the standard PReach algorithm. Recall that PReach sends a large batch of states owned by a worker with a long **WQ** to a worker with a shorter one for expansion. In PtP and OST mode, there are two additional kinds of messages, witness search and witness acknowledge, are sent in large numbers between workers. These message types each have a designated outbox, and messages are batched in the same way as Basic PReach (Chapter 3). Thus, during a typical round of sending, a worker will send 3 messages to every other worker, one containing states for expansion, one containing witness searches, and one containing witness acknowledgments. When a worker receives a non-owned state $s$ via load balancing, a witness search will be initiated for $s$ regardless of $s$ existing in the **WHT** of its owner. The possibility of redundant witness searches could be avoided by adding extra information about witnesses for load balanced states, but this was not implemented.

We have experimented with a heuristic to avoid sending search messages to workers that are running slowly. The main idea is to leverage PReach's flow control mechanism to detect when a worker is slow to acknowledge received messages. Then, a worker may avoid sending $(\rho, s)$ to $owner(s)$ for $s \notin q$ by locally computing helpful successor $s'$ of $s$, and sending $([s] \circ \rho, s')$ to $owner(s')$. The trade-off here is if $s \in \mathbf{WHT}$ at $owner(s)$, then this local computation is redundant as is the eventual witness acknowledge message sent to $owner(s)$.

## 4.3 Performance

We ran PReach-DF on a variety of combinations of Mur$\varphi$ models and hardware configurations, summarized in Table 4.1. For each, we measured the performance of regular state-space enumeration (no EF), local search mode (Section 4.2.1), Pass-the-Path mode (Section 4.2.2) and Outstanding-Search-Table mode (Section 4.2.3). Local search mode takes an impractical amount of time for a few of these so we omit it from the results. The Mur$\varphi$ models used are the German and Flash cache coherence protocols, the Peterson mutual exclusion algorithm, the MCS lock mutual exclusion algorithm and an Intel proprietary cache coherence protocol. We use `GermanX` and `flashX` to denote these models configured with `X` caches and two data values; `peterson12` is Peterson's algorithm with 12 workers and `mcslock6` is the MCS Lock algorithm with 6 workers. All benchmarks and the PReach-DF source code is available online[25]. The compute server farms are as follows:

- multicore: 8 PReach workers on a 2 socket server machine, each processor is a Intel® Xeon® E5520 at 2.26 GHz with 4 cores.
- small cluster: 80 PReach workers on a homogenous cluster of 20 Intel Core i7-2600K at 3.40 GHz with 4 cores.
- large cluster: 100 PReach workers on a heterogenous network of contemporary Intel® Xeon® machines.

The column "runtime" is the mean runtime of three trials, with the exception of the large cluster runs which are based on one trial (marked with †). The "overhead" columns are the additional runtime relative to that of the "no EF" mode run of the same model run on the same hardware. In PtP mode, column "avg. path" is the number of helpful transitions needed to reach a state that is a member of $q \vee \mathbf{WHT}$, averaged over the searches launched for each non-$q$-state. This number is *also* the average number of times each non-$q$-state is acknowledged for insertion to **WHT**.

## 4.4 Summary

We have shown an efficient distributed algorithm and implementation for checking deadlock freedom properties. The simpler approach of PtP does some redundant work, but performs well on models of cache coherence where paths to $q$-states tend to be relatively short (see

Figures 4.5 and 4.6). The approach is intolerably slow for the Peterson mutual exclusion algorithm where these paths are much longer. In this case, our more involved OST algorithm has a favorable runtime and manageable memory overhead (as shown in Figure 4.7). In fact, our experiments reveal that the memory usage of OST is typically small relative to the amount dedicated to the **WHT**, and has a small performance gain over PtP for the benchmarks where it does not timeout. We find that using OST to check deadlock freedom is inexpensive – at most a 109% runtime penalty but typically much smaller. While we believe OST is generally a better choice, a closer comparison with PtP is a topic of future work. We hypothesize that the performance gap between them is a function of the average path length of a witness search; this could be verified by constructing a toy model where this average can be tuned.

The utility of our approach to model checking deadlock freedom was underscored when a counterexample trace was generated on the `peterson12` benchmark. After carefully checking our definitions of $q$ and $\mathcal{H}$, we found a critical typo in the Mur$\varphi$ model, which despite being an example model in the popular Mur$\varphi$ distribution, has persisted for nearly 20 years. Interestingly, the bug in question was not revealed by checking the safety property mutual exclusion, or by "Mur$\varphi$ deadlock" (states with no successors) or *even* by checking $\mathsf{AG}\,\mathsf{EF}\,q$. With the bug, there exists a state where a worker attempting to enter the critical section may not do so until another worker makes an attempt. Thus, the model does not satisfy $\mathsf{AG}\,\mathsf{EF}_{\mathcal{H}}\,q$.

Checking $\mathsf{AG}\,\mathsf{EF}_{\mathcal{H}}\,q$ has a "buy-one, get-one free" appeal. Once model checking has been done for safety properties, a small amount of human effort is needed to identify helpful rules and write a quiescence predicate, $q$. While this approach cannot check for subtle liveness errors, especially ones that rely on fairness constraints, $\mathsf{AG}\,\mathsf{EF}_{\mathcal{H}}\,q$ can be very effective at finding deadlocks and violations of designer intent as illustrated by the Peterson example.

| model | hardware | no EF | local | PtP | | OST |
|---|---|---|---|---|---|---|
| | | runtime | overhead | overhead | avg. path | overhead |
| `German9` | multicore | 1365.12 | 0.76 | 0.16 | 1.005 | 0.21 |
| `flash5` | multicore | 752.75 | 1.07 | 0.30 | 1.005 | 0.39 |
| `peterson12` | multicore | 4018.34 | 1.11 | timeout | - | 0.38 |
| `mcslock6` | multicore | 230.09 | 1.12 | 0.43 | 1.093 | 0.67 |
| `German9` | small cluster | 85.91 | 3.43 | 0.24 | 1.642 | 0.35 |
| `flash5` | small cluster | 57.46 | 1.52 | 0.32 | 1.307 | 0.52 |
| `flash6` | small cluster | 1854.42 | 1.56 | 0.23 | 1.021 | 0.29 |
| `peterson12` | small cluster | 254.27 | 9.50 | timeout | - | 0.50 |
| `mcslock6` | small cluster | 22.04 | 1.45 | 2.73 | 4.763 | 1.09 |
| `intel_small`† | large cluster | 1025.20 | 7.10 | 0.84 | 2.242 | 0.55 |
| `intel_large`† | large cluster | 49041.70 | timeout | 0.35 | 1.309 | 0.44 |

Table 4.1: Performance of DF checking algorithms. Model sizes in states: `German9` – 19844513, `flash5` – 24063542, `flash6` – 609827554, `peterson12` – 116039964, `mcslock6` – 12838266, `intel_` – 22738573, `intel_large` – 906695343.

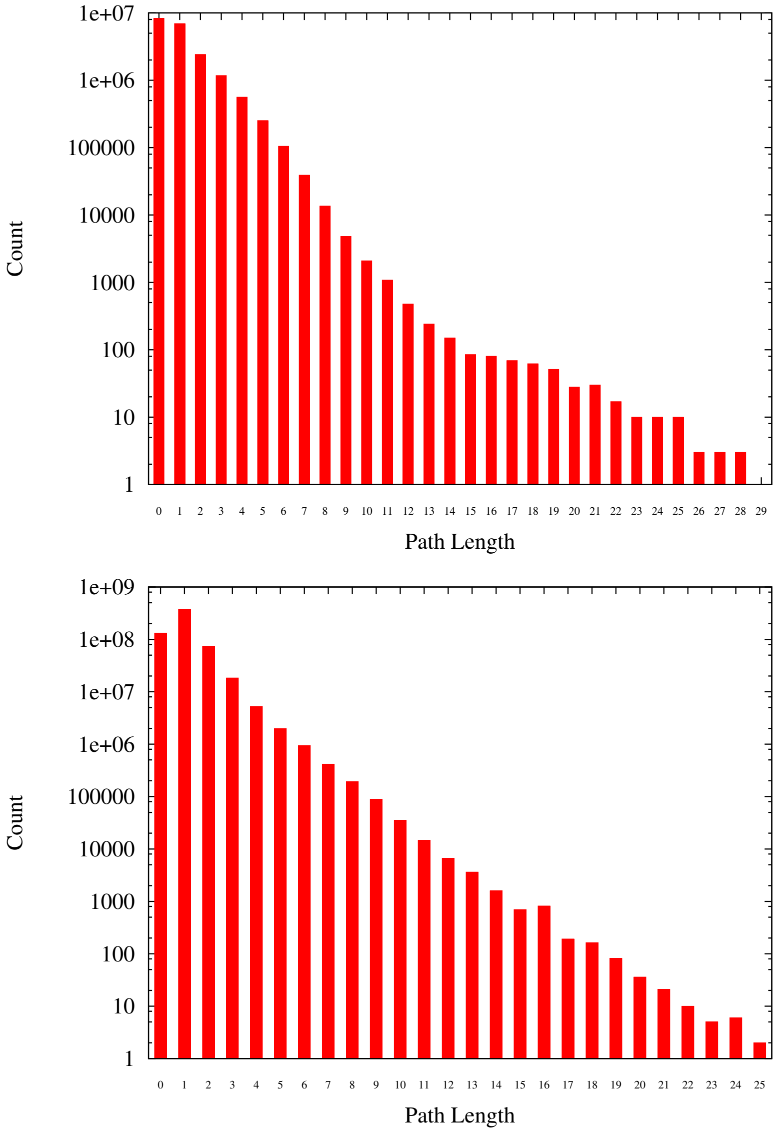


Figure 4.5: Semi-log histograms of path lengths in PtP mode, as defined in Table 4.1. The top is from `German9`/multicore, the bottom is from `flash6`/small cluster.

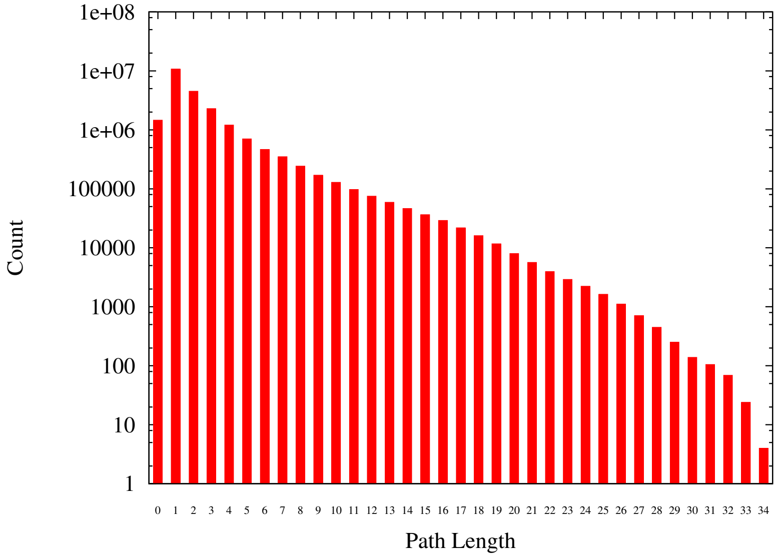


Figure 4.6: Semi-log path length histogram for `intel_`/large cluster in PtP mode

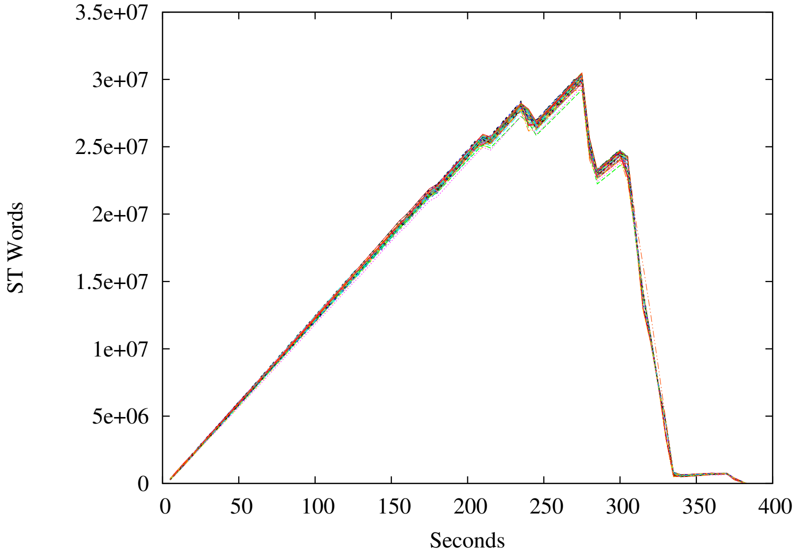


Figure 4.7: Memory usage of **ST** for `peterson12`/small cluster in OST mode. Here, "words" are 8 bytes; the maximum memory usage reached over all workers is about 233 MB.

# Chapter 5

# Parameterized Deadlock Freedom

Classical model checking assumes that the system under consideration is finite-state. Many researchers have explored techniques to generalize model checking to verify various classes of *parameterized* systems. A parameterized system $\mathcal{P}$ is a function that yields a finite-state system $\mathcal{P}(n)$ for all naturals $n \geq 1$. Here $n$ indicates the number of values involved in some type $\mathsf{P}$ (called the *parametric type*) used by the system, for example client IDs or memory addresses. A parameterized model checking problem asks if $\mathcal{P}(n)$ satisfies some given specification for all[1] $n$. Unless one puts severe restrictions on the class of systems, parameterized model checking is undecidable [1]. The systems we consider are *symmetric*, a restriction on those for the undecidable proof. However, in Section 5.2.4 we provide a proof that model checking symmetric paramaterized systems is also undecidable.[2]

A promising approach to parameterized model checking is based on abstraction and compositional reasoning [92, 93, 40, 76, 86, 28, 83, 99] and is typically used to verify universally-quantified (over $\mathsf{P}$) state assertions roughly as follows. An initial abstraction $\mathcal{A}_0$ is created from the syntax of $\mathcal{P}$. By construction, the transitions of $\mathcal{A}_0$ over-approximate those of $\mathcal{P}(n)$ for arbitrary $n > k$ (typically for some small $k$). If the safety property $\varphi$

---

[1] Many approaches, including ours, only verify $\mathcal{P}(n)$ for all $n \geq n_0$, where $n_0$ is a small constant. This is not a shortcoming since the $\mathcal{P}(n)$ where $n < n_0$ are either "uninteresting" or can be dispatched by finite-state model checking.

[2] This chapter is based on [23].

holds of $\mathcal{A}_0$, then we can soundly conclude it holds for $\mathcal{P}(n)$. However, often this is not the case and we must *strengthen* the transitions of $\mathcal{A}_0$ using a conjectured state invariant $\varphi_1$, yielding a tighter abstraction $\mathcal{A}_1$. This process iterates until we obtain a $\mathcal{A}_j$ wherein $\varphi$ along with all of $\varphi_1, \ldots, \varphi_j$ hold. At this point our parametric verification goal has been achieved. If $\varphi$ is *not* invariant of $\mathcal{P}(n)$, then eventually the user will either give up or notice a real counter-example. This approach is sound, in that if $\varphi$ does not hold of $\mathcal{P}(n)$ then $\varphi$ will not hold of $\mathcal{A}_0$, regardless of the strengthening applied.

Our contribution is an extension of the approach described above to handle Deadlock Freedom (DF) properties of the form of the CTL formula $\mathsf{AG}\,\mathsf{EF}\,q$ for a state predicate $q$ (more generally, we also consider $\mathsf{AG}\,(p \rightarrow \mathsf{EF}\,q)$). The key idea is to construct abstract transition relations that not only over-approximate those of $\mathcal{P}(n)$, but also transition relations that *under-approximate* $\mathcal{P}(n)$. The resulting verification framework is formalized in term of *mixed abstractions* [48], which are systems having two transition relations $O$ and $U$, which are respectively over-approximative (OA) and under-approximative (UA). As in the traditional approaches, $O$ is used to explore the reachable abstract states, which represent an over-approximation of the reachable states of $\mathcal{P}(n)$. During this exploration, paths of $U$ are explored to check that the existential CTL formula $\mathsf{EF}q$ holds for each reachable abstract state. If so, it is safe to conclude DF of $\mathcal{P}(n)$; the existence of a path in $U$ implies that of a corresponding path in $\mathcal{P}(n)$. Like $O$, it is straightforward to algorithmically construct an initial $U$; however, analogous to how $O$ might be too *weak*, we might find that $U$ is too *strong*.

One of our key contributions is a set of heuristic methods that allow the user to soundly weaken $U$. This is achieved by exploiting syntactic comparisons between the guarded-command transitions (*i.e.* Mur$\varphi$ rules) of the parameterized system and those of the OA transitions. If rule's guard is *not* weakened by the overapproximation, then we conclude the rule is UA. In cases when the guard has been weakened, we provide heuristic inference rules (HIRs) that allow us to conclude the rule is UA if some DF property holds of the mixed abstraction. These properties can be decided by DF model checking. Determining which HIR to apply for a particular rule is straightforward, and generating the relevant DF property for a heuristic is automatable. When a DF property for the mixed abstraction fails to hold, the required human insight is similar to that the method of Chou *et al.* [40], which is used to prove parameterized saftey properties. Another contribution is a theorem that supports DF verification when $q$ involves universal quantification over the parametric type.

Note that this is an *incomplete* method for solving the problem of parameterized DF for symmetric systems. The presented HIRs provide tools for the user to weaken the underapproximation and, in our experience with case studies, are sufficient to establish a proof. They are heuristic in that they may or may not be successful in proving deadlock freedom, but they are sound in that model checking certain side conditions is justification to add transitions to the underapproximation.

This chapter is organized as follows. We begin with a simple example of showing DF in a mutual exclusion scheme. Next, mixed abstractions are defined and some key lemmas are stated. In Section 5.3, we describe how mixed abstractions are modeled and computed in Mur$\varphi$. Section 5.4 discusses our approach for weakening $U$ by employing HIRs. Section 5.5 outlines the DF verification of the German and FLASH cache coherence protocols, and Section 5.6 concludes the chapter. For convenience, a list of chapter symbols is provided in Table 5.3. Appendix Sections C.1 and C.2 give specific examples of applying two kinds of HIRs.

## 5.1 A Simple Example

To keep this example simple, we have glossed over some technical details. These are addressed in footnotes that are collected at the end of this section on page 80. Sections 5.2–5.4 give a more detailed an formal description of our approach.

Consider the parameterized program of Figure 5.1 that describes a mutual exclusion

```
in $n$ : integer where $n > 1$
local $t$: integer where $t = 0$

$\overset{n}{\underset{i=1}{\|}} P[i]$ ::
    $\ell_0$ : loop forever do
        $\ell_1$ : noncritical
        $\ell_2$ : while $t \neq i$ do
            $\ell_3$ : $\langle$if $t = 0$ then $t := i$ end if$\rangle$
            $\ell_4$ : if $t = i$ then
                $\ell_5$ : critical
            end if
        end while
        $\ell_6$ : $t := 0$
    end loop
```

Figure 5.1: Program TURN (mutual exclusion by turn setting) [89, Fig. 2.8]

algorithm. This program can be described with the Mur$\varphi$ system of Figure 5.2. For brevity, the lines $\ell_2$ and $\ell_3$ are combined (modeled as `L3`, waiting to enter the critical section) as are the lines $\ell_4$ and $\ell_5$ (modeled as `L5`, the critical section). Line $\ell_0$ is not modeled; the system begins with all threads at `L1`. Instead of the value 0 for $t$, we use the built-in Mur$\varphi$ value `undefined`. In order to be a valid Mur$\varphi$ system, we must provide a specific parameter value for $n$. Any value greater than 1 will suffice; for this example, we arbitrarily choose 5. The Mur$\varphi$ system of Figure 5.3 is the corresponding abstract system with 1 thread explicitly modeled. The rest are subsumed into a new value called `Other`. This abstraction overapproximates the behavior of the concrete system any $n > 2$ threads, essentially by assuming that these "other" threads $2, ..., n$ could be at any line.

Some of these rules have identical syntax and names as their concrete counterparts. Since nothing about the state is assumed or overapproximated in the guards of such rules, the following rules are in fact *underapproximate*: `L1_to_L3`, `L3_to_L5`, `L5_to_L6`, `L6_to_L1`, and `Stutter`. However, the rules with `ABS` prefixes in the name do not meet this criteria. For example, compare the guard of the concrete rule `L5_to_L6` with the abstract rule

```
-- a mutual exclusion protocol
const NUM_THREADS : 5; -- assume 5 threads
type THREAD : scalarset(THREAD_NUM); -- a symmetric set
     LINE_NUM : enum L1, L3, L5, L6; -- enumerated type representing the line number
var  Line : array [THREAD] of LINE_NUM;
     t : THREAD;

startstate "Init" -- initial state
  for i : THREAD do Line[i] := L1; end;
end;

ruleset i : THREAD do

   rule "L1_to_L3" Line[i] = L1
      ==> Line[i] := L3; end;

   rule "L3_to_L5" Line[i] = L3 & isundefined(t) -- & is logical AND
      ==> Line[i] := L5; t := i; end;

   rule "L5_to_L6" Line[i] = L5 & t = i
      ==> Line[i] := L6; end;

   rule "L6_to_L1" Line[i] = L6
      ==> Line[i] := L1; undefine t; end;

end; -- ruleset

rule "Stutter" true
   ==>
end;
```

Figure 5.2: The Mur$\varphi$ system for program TURN

`ABS_L5_to_L6`. The guard of the latter rule assumes that there exists some $i > 1$ such that `Line[i] = L5`. Every rule of the abstraction contributes to the set of overapproximate transitions $O$, and the list of UA rules contribute to the set of UA rules $U$. The above abstraction, augmented with the set of UA rules $U$, is in fact a mixed-abstraction $\mathcal{A}$.

We seek to show that for any $n$, every reachable state $s$ of the concrete system $\mathcal{P}(n)$ has a path to some state $s'$ where `t` is `undefined`. These are states where no thread is in the critical section. We use model checking of the mixed abstraction to check $\mathcal{A} \models \mathsf{AG}\,\mathsf{EF}\,q$,

```
-- abstract system
const  THREAD_NUM : 1;
type  THREAD : scalarset(THREAD_NUM);
      ABS_THREAD : union {THREAD, enum{Other}}; -- new value "Other" abstracts threads > 1
      LINE_NUM : enum {L1, L3, L5, L6};
var Line : array [THREAD] of LINE_NUM;
    t : ABS_THREAD;

startstate "Init"
  for i : THREAD do Line[i] := L1; end;
end;

ruleset i : THREAD do
-- underapproximate rules
   rule "L1_to_L3" Line[i] = L1
      ==> Line[i] := L3; end;

   rule "L3_to_L5" Line[i] = L3 & isundefined(t)
      ==> Line[i] := L5; t := i; end;

   rule "L5_to_L6" Line[i] = L5  & t = i
      ==> Line[i] := L6; end;

   rule "L6_to_L1" Line[i] = L6
      ==> Line[i] := L1; undefine t; end;

end; -- ruleset

rule "Stutter" true
   ==>
end;

-- overapproximate rules
rule "ABS_L3_to_L5" isundefined(t)
   ==> t := Other; end;

rule "ABS_L5_to_L6" t = Other
   ==> undefine t; end;
```

Figure 5.3: Mur$\varphi$ system for the mixed abstraction of TURN

where $q$ is `isundefined(t)`. This DF property is expressed in Mur$\varphi$ syntax using new keywords `liveness` and `CANGETTO` as understood by PREACH. Here, the property is:

```
liveness "NoCrit"
```

```
   true CANGETTO isundefined(t)
```

This DF property holds if for every state reachable by $O$ transitions, there exists a path of $U$ transitions to a state where `t` is `undefined`.

When executed, a counterexample reveals that the underapproximate transition relation $U$ is too strong. The counterexample path begins with `ABS_L3_to_L5` firing from the initial state, reaching a state where `Line[1] = L1` and `t = Other`. Then, `L1_to_L3` fires, updating `Line[1]` with the value `L3`. At this state none of the UA rules are enabled, but non-UA rule `ABS_L5_to_L6` is enabled. We do not consider this rule UA because it assumes that whatever node `t` refers to in some concretization, call it `j > 1`, has `Line[j] = L5`. Of course, it is easy to see in this example that in any concrete system, `t = j` if and only if `Line[j] = L5`. However, for illustration we show how this can be proven using permutation. For any state where `t = Other`, consider swapping the "actual" value held by `t` with thread 1. This permutation exploits the symmetry of the system to force the thread value of `t` to be visible in the mixed abstraction; see Figure 5.4. The resulting states are described by the predicate `isdefined(t) & t != Other`[3]. We show that all reachable states satisfying *this* predicate have a path to some state that satisfies the guard of `L3_to_L5`, written as `exists i : THREAD do t = i & Line[i] = L5 end`[4]. Thus, we check the following property[5]:

```
liveness "H1"
   isdefined(t) & t != Other
   CANGETTO
   exists i : THREAD do t = i & Line[i] = L5 end
```

This DF property holds, which proves `ABS_L3_to_L5` is UA. Now, with this rule added to $U$, the property `NoDeadlock` is checked again and now holds. This completes the proof parameterized DF, *i.e.*, that for any $n > 1$, every reachable state of $\mathcal{P}(n)$ has a path to some state where no thread is in the critical section.

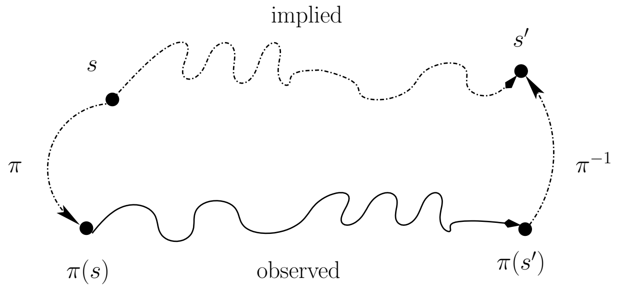


Figure 5.4: Finding paths through permutation. The path from $s$ to $s'$ is implied by finding a path from their permuted counterparts, $\pi(s)$ and $\pi(s')$.

## Notes

[3] The function `isdefined` does not exist in Mur$\varphi$, but we use it as shorthand for `!isundefined`, where `!` is logical NOT. The operator `!=` means "not equal to", *i.e.*, `t != Other` is the same as `!(t = Other)`.

[4] We would prefer to write something like `t = 1` instead of `isdefined(t) & t != Other`, and `t = 1 & Line[1] = L5` instead of `exists i : THREAD do t = i & Line[i] = L5 end`. However, Mur$\varphi$ will complain because the set of nodes are symmetric, and we therefore cannot refer to a particular value directly.

[5] For this check, instead of using all UA rules to find the requisite path, only rules that are "local" to `t` are used. These are rules that only change the $i^{th}$ index of array variables when `t = i`. This is to ensure that the changes of this path remain hidden when the inverse permutation is taken. In this example, rules `L1_to_L3` and `L6_to_L1` when `i = 1`, as well as `Stutter` are local to `t` when `t = i`.

## 5.2 Formal Framework

This section presents the formal framework that we use to verify quiescence properties of parameterized systems. Section 5.2.1 introduces *mixed abstractions* that have two transition relations: one that under-approximates the behaviors of the concrete systems and another that provides an over-approximation. Section 5.2.2 presents the idea of *insufficiency* – a mixed abstraction may have an under-approximation that is too strong to verify the desired quiescence property or an over-approximation that is too weak. Section 5.2.3 describes the parameterized systems that we consider. These are less general than those of [1]. Section 5.2.4 shows that safety property checking (and hence DF checking) remains undecidable even with our restrictions.

### 5.2.1 Mixed Abstractions

A *system* $\mathcal{S}$ is a tuple $(S, I, T)$ where $S$ is a set of *states*, $I \subseteq S$ is the set of the initial states, and $T \subseteq S \times S$ is the *transition relation.* We write $s_1 \rightsquigarrow_T s_2$ to denote that $(s_1, s_2) \in T^*$. A state $s$ is said to be $\mathcal{S}$*-reachable* (or simply *reachable* if $\mathcal{S}$ is understood) if $s_0 \rightsquigarrow_T s$ for some $s_0 \in I$. A *state predicate* $p$ is simply a subset of $S$; if $s \in p$ we call $s$ a $p$*-state.* Following standard CTL syntax, for state predicates $p$ and $q$, we write: $\mathcal{S} \models \mathsf{AG}\, p$ if all reachable states are $p$-states; $\mathsf{AG}\,(p \rightarrow \mathsf{EF}\, q)$, if for all reachable $p$-states $s$ there exists a $q$-state $s'$ such that $s \rightsquigarrow_T s'$; and $\mathsf{AG}\,\mathsf{EF}\, q$ to mean $\mathsf{AG}\,(\mathit{true} \rightarrow \mathsf{EF}\, q)$, *i.e.*, from any reachable state there is a path to some $q$-state.

To show that $\mathsf{AG}\,(p \rightarrow \mathsf{EF}\, q)$ can be inferred for a concrete system $\mathcal{S}$ by establishing properties of an abstraction $\mathcal{A}$ we employ Lynch and Vaandrager's notion of *forward simulation* [87]. Let $\mathcal{S}_1 = (S_1, I_1, T_1)$ and $\mathcal{S}_2 = (S_2, I_2, T_2)$ be two systems and $\theta \in S_1 \times S_2$ be an abstraction relation. With respect to $\theta$, we say that $T_2$ forward simulates (or "simulates" for short) $T_1$ if for every $(s_1, s_1') \in T_1$ and for all $s_2$ such that $(s_1, s_2) \in \theta$, there is some $s_2' \in S_2$ such that $s_2 \rightsquigarrow_{T_2} s_2'$ and $(s_1', s_2') \in \theta$. This allows system $\mathcal{S}_2$ to take multiple steps that may be invisible in $\mathcal{S}_1$ including possibly steps that have no "explanation" in $\mathcal{S}_1$. This general sense of simulation is motivated by our goal of showing the existence of trajectories. See Figure 5.5. The domain of $\theta$ may be a subset of $S_1$, in particular, restricted to the reachable subset of $S_1$. We say $T_2$ *simulates* $T_1$ when $\theta$ is clear from context. Now let $s_1, \ldots, s_\ell$ be a $T_1$-path, and suppose $T_2$ simulates $T_1$ with respect to

$\theta$. By induction on $\ell$, there exists an $T_2$-path $s'_1, \dots, s'_k$ and a non-decreasing surjection $f : \{1, \dots, \ell\} \to \{1, \dots, k\}$ such that $(s_i, s'_{f(i)}) \in \theta$ for all $1 \le i \le \ell$.

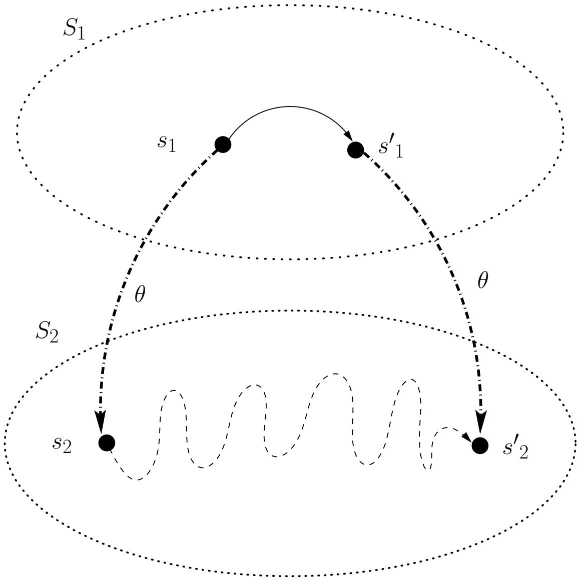


Figure 5.5: Simulation of $\mathcal{S}_1$ by $\mathcal{S}_2$

To show $\mathsf{AG}\,(p \to \mathsf{EF}\,q)$ using abstraction, the abstract system must, for soundness, over-approximate the set of reachable $p$-states, and under-approximate the set of paths from $p$-states to $q$-states. Thus, we introduce a *mixed abstraction* as defined below. Note that unlike Lynch and Vaandrager, we require $\theta$ to be a function.

**Definition 1.** *Let $\mathcal{S} = (S, I, T)$ be a system and let Reach be the $\mathcal{S}$-reachable states. A* mixed abstraction *of $\mathcal{S}$ (relative to $\theta : S \to S_{\mathcal{A}}$) is a quadruple $\mathcal{A} = (S_{\mathcal{A}}, I_{\mathcal{A}}, U, O)$ such that*

- *$S_{\mathcal{A}}$ is a set of abstract states,*
- *$I_{\mathcal{A}} \subseteq S_{\mathcal{A}}$ are the initial abstract states and satisfy $\theta(I) \subseteq I_{\mathcal{A}}$,*

- *$O \subseteq S_{\mathcal{A}} \times S_{\mathcal{A}}$ simulates $T$ with respect to $\theta \cap (\mathit{Reach} \times S_{\mathcal{A}})$, and*
- *$T$ simulates $U \subseteq S_{\mathcal{A}} \times S_{\mathcal{A}}$ with respect to $(\theta \cap (\mathit{Reach} \times S_{\mathcal{A}}))^{-1}$.*

Here $U$ and $O$ are respectively called the *under-approximate* (UA) and *over-approximate* (OA) transition relations of the mixed abstraction. When discussing mixed abstractions, we will often refer to $\mathcal{S} = (S, I, T)$ as the *concrete system*, to $S$ as the *concrete states*, etc.

The following serves as the basis for our approach to proving deadlock freedom for a class of parameterized systems. Let $p$ and $q$ be state predicates over $S_{\mathcal{A}}$, and suppose for all $(S_{\mathcal{A}}, I_{\mathcal{A}}, O)$-reachable $p$-states $s$ there exists a $q$-state $r$ such that $s \rightsquigarrow_U r$. We assert this by writing

$$\mathcal{A} \vDash \mathsf{AG}\,(p \to \mathsf{EF}\,q) \tag{5.1}$$

Hence, (5.1) holds of a mixed abstraction if for all $p$-states reachable using the over-approximative transition relation, there exists a path through the under-approximative transition relation to a $q$ state. See Figure 5.6 for a schematic of this relationship between the mixed abstraction and a concrete system.

**Lemma 1.** *Let $\mathcal{A}$ be a mixed abstraction of $\mathcal{S}$ relative to $\theta$ and let $p$ and $q$ be state predicates on $S_{\mathcal{A}}$. If $\mathcal{A} \vDash \mathsf{AG}\,(p \to \mathsf{EF}\,q)$ then $\mathcal{S} \vDash \mathsf{AG}\,(\theta^{-1}(p) \to \mathsf{EF}\,\theta^{-1}(q))$.*

*Proof.* Let *Reach* be the set of $\mathcal{S}$-reachable states. Let $w$ be any $\theta^{-1}(p)$-state in *Reach*. Because $O$ simulates $T$ with respect to $\theta \cap (\mathit{Reach} \times S_{\mathcal{A}})$, $\theta(w)$ is $(S_{\mathcal{A}}, I_{\mathcal{A}}, O)$-reachable, and furthermore $\theta(w)$ is clearly a $p$-state. Let $a_0, \ldots, a_m$ be a $U$-path from $\theta(w) = a_0$ to a $q$-state $a_m$. Because $T$ simulates $U$ with respect to $(\theta \cap (\mathit{Reach} \times S_{\mathcal{A}}))^{-1}$, for all $0 \leq i < m$ and all $w_i \in \theta^{-1}(a_i)$ there exists $w_{i+1} \in \theta^{-1}(a_{i+1})$ such that $w_i \rightsquigarrow_T w_{i+1}$. Therefore, taking $w_0 = w$, there is a path $w \rightsquigarrow_T w_m$ where $w_m \in \theta^{-1}(q)$. $\square$

Note that the definition of mixed abstraction explicitly mentions the reachable states of $\mathcal{S}$ (in the involved simulation relations), this is just our means of formalizing the minimal requirements a mixed abstraction must satisfy in order to prove Lemma 1. In other words,

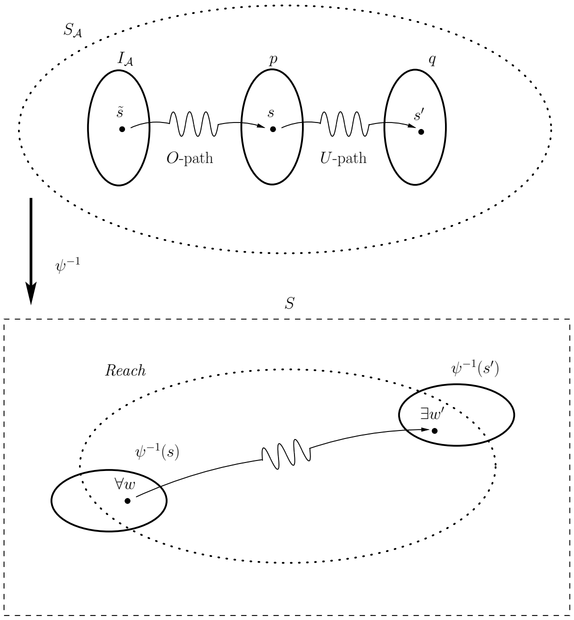


Figure 5.6: Illustration of Lemma 1. Suppose that for each $s \in p$ that is reachable from an initial state through $O$ transitions, there exists $s' \in q$ reachable from $s$ through $U$ transitions. Then every reachable $w$ of $\mathcal{S}$ that is a concretization of $s$ has a path to some state in the concretization of $s'$.

any methodology that aims to construct mixed abstractions must guarantee *at least* these simulations. We emphasize that this is different than asking the user of such a methodology to precisely characterize *Reach* (indeed our methodology does not make such a demand).

We conclude this section by stating a connection between $\mathcal{S}$ and the transitions of $U$ and $O$.

**Lemma 2.** *Suppose $\mathcal{S} = (S, I, T)$, $S_{\mathcal{A}}$, $I_{\mathcal{A}}$, and $\theta : S \to S_{\mathcal{A}}$ are as in Def. 1. If $U, O \subseteq S_{\mathcal{A}} \times S_{\mathcal{A}}$ satisfy*

1. *for all $(w, w') \in T$ such that $w$ is $\mathcal{S}$-reachable we have $(\theta(w), \theta(w')) \in O$, and*
2. *$(s, s') \in U$ implies for all $\mathcal{S}$-reachable $w \in \theta^{-1}(s)$ there exists $w' \in \theta^{-1}(s')$ such that $w \rightsquigarrow_T w'$,*

*then $(S_{\mathcal{A}}, I_{\mathcal{A}}, O, U)$ is a mixed abstraction for $\mathcal{S}$.*

*Proof:* Follows directly from Def. 1.

When performing reasoning that allows us to add (or remove) transitions from $U$ and $O$ in a mixed abstraction, we employ sufficient conditions of Lemma 2. This essentially restricts the definition of mixed abstractions to those where each transition of $T$ has a corresponding transition in $O$, and each transition of $U$ has a corresponding path in $T$.

### 5.2.2 Insufficiency

Lemma 1 allows us to infer $\mathcal{S} \models \mathsf{AG}\,(\theta^{-1}(p) \to \mathsf{EF}\,\theta^{-1}(q))$ if model checking (or any other means) verifies $\mathcal{A} \models \mathsf{AG}\,(p \to \mathsf{EF}\,q)$. However, the converse of the lemma does not hold in general (or even in common cases). Let us call the mixed abstraction $\mathcal{A}$ *insufficient* if $\mathcal{S} \models \mathsf{AG}\,(\theta^{-1}(p) \to \mathsf{EF}\,\theta^{-1}(q))$ holds but $\mathcal{A} \not\models \mathsf{AG}\,(p \to \mathsf{EF}\,q)$. If $\mathcal{A}$ is insufficient, it follows that there exists a $p$-state $a_p$ such that $a_0 \rightsquigarrow_O a_p$ for some $a_0 \in I_{\mathcal{A}}$ but there is no $q$-state $a_q$ such that $a_p \rightsquigarrow_U a_q$. There are two common causes for insufficiency:

- **OA Insufficiency.** There is no $(S, I, T)$-reachable $s_p \in S$ such that $\theta(s_p) = a_p$. Hence $a_p$ does not abstract any reachable state of $\mathcal{S}$. This is often caused by $O$ being too *weak*, *i.e.* there exists a proper subset $O' \subset O$ such that $(S_{\mathcal{A}}, I_{\mathcal{A}}, O', U)$ is a mixed abstraction of $\mathcal{S}$ wherein $a_p$ becomes unreachable. Thus the CMP method should be applied to strengthen the overapproximation.

- **UA Insufficiency.** There *is* $(S, I, T)$-reachable $s \in S$ such that $\theta(s) = a$, however none of the $T$-paths $s = s_0, \ldots, s_\ell$ where $s_\ell \in \theta^{-1}(q)$ (at least one such $T$-path must exist), are simulations of any $U$-paths. In practice, this is often caused by $U$ being too *strong*. In this case, there may exist a proper superset $U' \supset U$ such that $(S_{\mathcal{A}}, I_{\mathcal{A}}, O, U')$ is a mixed abstraction of $\mathcal{S}$.

For the mixed abstractions we use to verify our parameterized systems, we will observe that OA insufficiency is addressed in previous literature [40], however UA insufficiency is not. Our basic approach is to identify UA insufficiency from a counter-example trace. In practice, the transitions from $O$ that are needed in $U$ are often apparent from this counter-example. The basic idea is to show that for each such transition $(s, s')$ of $O$, there is a corresponding set of *paths* in the concrete system, *i.e.*, each state of $\theta^{-1}(s)$ has a path to some state of $\theta^{-1}(s')$. Sections 5.3 and 5.4 present how this can be done by syntactic pattern matching and model checking of the *abstract* system for properties with the form shown by Formula (5.1).

There is also a third flavor of insufficiency,

- **Abstract Quiescence Insufficiency.** Note that in Lemma 1, the quiescent predicate actually verified is of the form $\theta^{-1}(q)$, where $q$ is a predicate on the abstract states. Suppose, however that there does not exist a $q$ such that $\theta^{-1}(q)$ characterizes the desired set of quiescent concrete states. We experience this for our case studies: the desired quiescence predicate involves a universal quantification over the parametric type that the underlying simulation relation cannot precisely characterize. That is, if the concrete quiescent states are characterized by a predicate of the form $\forall i.\ \phi(i)$, then there is no abstract predicate $q$ such that $\theta^{-1}(q) = \forall i.\ \phi(i)$. We deal with this form of insufficiency via Theorem 1 in Section 5.4.1.

### 5.2.3 Parameterized Systems

For the approach presented in this chapter, a parameterized system $\mathcal{P}$ is a function mapping natural numbers to systems. We write $\mathcal{P}(n) = (S(n), I(n), T(n))$ to denote the components of $\mathcal{P}(n)$ for an arbitrary $n$. The states $S(n)$ are the type-consistent assignments to a set of state variables. For a state $w$ of a parameterized system and a state variable $v$, we write $v(w)$ to denote the value $w$ assigns to $v$. We allow four types of variables:

- finite types that are independent of $n$, such as booleans and enumerations; for simplicity, we denote all such types as $\mathsf{B}$
- a type which has cardinality $n$, denoted $\mathsf{P}^n$

- arrays indexed by $\mathsf{P}^n$ with elements in $\mathsf{B}$, denoted $\mathsf{array}\,[\mathsf{P}^n]$ of $\mathsf{B}$
- arrays indexed by $\mathsf{P}^n$ with elements in $\mathsf{P}^n$, denoted $\mathsf{array}\,[\mathsf{P}^n]$ of $\mathsf{P}^n$

We identify the set $\mathsf{P}^n$ with the numbers $\{1, \ldots, n\}$ called *nodes*; the only operations supported on nodes are equality comparison, assignment, and nondeterministic choice.[6] This ensures that for all $n$, $\mathcal{P}(n)$ is fully symmetric [69] in $\mathsf{P}^n$; here we give a brief review of this notion. Let us write $\lambda i.e$ to denote the array $a$ indexed by $\mathsf{P}^n$ where $a[i] = e$ and $e$ is an expression of the appropriate type. Let $\pi$ be a permutation on $\mathsf{P}^n$. We overload $\pi$ to act on $w \in S(n)$ by defining $\pi(w) \in S(n)$ to be the state such that for each state variable $v$, $v(\pi(w))$ is equal to:

- $v(w)$, if $v$ has type $\mathsf{B}$
- $\pi(v(w))$, if $v$ has type $\mathsf{P}^n$
- $\lambda i.(v(w)[\pi^{-1}(i)]$, if $v$ has type $\mathsf{array}\,[\mathsf{P}^n]$ of $\mathsf{B}$
- $\lambda i.\pi(v(w))[\pi^{-1}(i)])$, if $v$ has type $\mathsf{array}\,[\mathsf{P}^n]$ of $\mathsf{P}^n$

Then $(S(n), I(n), T(n))$ is called *fully symmetric* if for all $w, w' \in S(n)$ and all permutations $\pi$ on $\mathsf{P}^n$ we have both that $w \in I(n)$ iff $\pi(w) \in I(n)$, and $(w, w') \in T(n)$ iff $(\pi(w), \pi(w')) \in T(n)$. The following lemma has a simple inductive proof using the latter.

**Lemma 3** (Path Symmetry)**.** *For $w, w' \in S(n)$ we have $w \rightsquigarrow_{T(n)} w'$ if and only if $\pi(w) \rightsquigarrow_{T(n)} \pi(w')$.*

The method presented in this chapter is applicable to fully symmetric models. Section 5.3.1 gives the restrictions on parameterized systems that ensure our approach can be applied. We refer to such system as *admissible*.

---

[6] If $i$ and $j$ are nodes, a parameterized system is not allowed to perform a comparison such as $i < j$ or increment a variable $i := i + 1$.

### 5.2.4 Undecidability

Here we sketch a proof that model checking symmetric parameterized systems is undecidable. Given a Turing machine $M$ with no input, we simulate $M$ using at most $i$ tape cells with $\mathcal{S}(i)$. Each node of $\mathcal{S}(i)$ represents a tape cell. The description of $M$ can be encoded using boolean variables in $\mathcal{S}(i)$. Two phases of the parameterized system occur. First, nodes are chosen to represent tape cells in some order. Initially, all cells are *unused*. Each step nondeterministically chooses some unused cell and marks it as used. The first node selected holds the symbol for the leftmost tape cell. The second node selected holds the symbol for the second tape cell to the left and so on. Each node has two variables of type node that act as pointers to the left and right, so the ordering of cells is maintained through a doubly linked-list. Note that the tape cells are represented by nodes, and the state of the Turing machine is stored in global (*i.e.* non-parameterized) variables. We include an additional boolean variable **halts** which is initialized to *false*. Second, $M$ is simulated using the tape as represented by the doubly linked-list of nodes. If $M$ halts using at most $i$ cells, then our simulation will eventually reach a halting state and **halts** will be set *true*. Otherwise, simulation of $M$ on a tape with $i$ cells will eventually run off the right end of the tape, at which point the simulation enters a *TapeOut* state and remains there forever. Or, $M$ runs forever without using more than $i$ cells. In either case, **halts** remains *false*. If $M$ halts when run on empty input, then it uses some bounded number of cells before halting. If our simulation runs with at least that many nodes, then our simulation will reach a state where **halts** is set, so $\mathsf{AG}\,\neg\mathbf{halts}$ is not satisfied. Therefore, $\forall i.\ \mathcal{S}(i) \models \mathsf{AG}\,\neg\mathbf{halts}$. is satisfied iff $M$ does not halt when run on empty input. Note that deadlock-freedom is a more general property than safety, as $\mathsf{AG}\,(\neg p \rightarrow \mathsf{EF}\,\mathit{false}) \Rightarrow \mathsf{AG}\,p$. Since model checking deadlock-freedom of a symmetric parameterized system solves the co-halting problem, it is undecidable.

## 5.3 Syntactical Abstraction

We assume that the parameterized system is modeled by Mur$\varphi$ [51] or a similar guarded-command notation. Given a program $P$ to describe the parameterized system, we use well-established techniques [92, 93, 40, 76, 86, 28, 83, 99] to obtain an abstraction of $P$. Our formulation is inspired by the Krstic's "syntactic" approach [76]; Section 5.3.1 states restrictions that we assume on the form of $P$, and Section 5.3.2 summarizes the abstraction technique. In Section 5.4, we show how the abstraction can be generalized to produce an under-approximate transition relation, $U$, and how $U$ can be soundly weakened to prove quiescence properties.

### 5.3.1 Syntax and Restrictions

We assume that the guarded-command program that models the parameterized system satisfies certain syntactic restrictions described in this section. These restrictions ease the syntactical abstraction process and simplify reasoning about the program because many useful properties are guaranteed by construction. We say that such a program is *admissible*. From the case studies reported in Section 5.5, we've found that these restrictions are not problematic in practice.

An admissible program has a set of variables of the types indicated in Table 5.1. A state of the admissible program is a type-consistent assignment of values to these variables. If $e$ is a term, we write $e(s)$ to denote the value of $e$ in state $s$. In Mur$\varphi$, a guarded command is called a *rule* and has the form: *guard* $\rightarrowtriangle$ *action*, where the *guard* is a boolean-valued expression, and the *action* is a sequence of one or more assignments. We write $r : \rho \rightarrowtriangle \boldsymbol{a}$ to denote rule $r$ with guard $\rho$ and action $\boldsymbol{a}$.

The denotation $[\![r]\!]$ of $r$ is the set of tuples $(s, s') \in S \times S$ such that $\rho(s)$, and $s'$ is the state reached by performing action $\boldsymbol{a}$ from state $s$. Mur$\varphi$ has *rulesets* of the form:

```
ruleset i :  P^n do r(i) end;
```

where $r(i)$ is a rule (or a ruleset, as they may be nested). Here, $i \in \mathsf{P}^n$ is called the *ruleset* *parameter.* If $rs$ is the ruleset indicated above, then

$$[\![rs]\!] \quad = \quad \{(s, s') \mid \exists i \in \mathsf{P}^n.\ (s, s') \in [\![r(i)]\!]\} \tag{5.2}$$

A *local boolean predicate* $L$ is a propositional formula over the variables of type $\mathsf{array}\,[\mathsf{P}^n]\ \mathsf{of}\ \mathsf{B}$. For node $i$, we say $L[i]$ holds of a state if $L$ evaluates to true when its variables are assigned according to the $i^{th}$ array entries of the state. An admissible program must satisfy the following syntactic restrictions. Rulesets have guards that are a conjunct of:

- Boolean terms, composed of variables of type $\mathsf{B}$ or $\mathsf{array}\,[\mathsf{P}^n]\ \mathsf{of}\ \mathsf{B}$ indexed by a ruleset parameter, and the logical connectives `AND`, `OR` and `NOT`.
- At most one *forall condition*, appearing positively, of the form $\forall i \in \mathsf{P}^n.\ C[i]$ where $C$ is a local boolean predicate.
- Any number of $\mathsf{P}$-comparisons, of the form $v_1 = v_2$ or $v_1 \neq v_2$, where $v_1$ and $v_2$ are each either a ruleset parameter, a variable of type $\mathsf{P}^n$, or a variable of type $\mathsf{array}\,[\mathsf{P}^n]\ \mathsf{of}\ \mathsf{P}^n$ indexed by a ruleset parameter. Without loss of generality, we restrict each ruleset parameter to appear in at most one $\mathsf{P}$-comparison of equality.

The initial states and ruleset commands given by a sequence assignments of the following forms:

- Assignments of the form $b_1 := b_2$, $a^1_{\mathsf{B}}[i] := b_2$, $a^1_{\mathsf{B}}[i] := a^2_{\mathsf{B}}[i]$, where $b_1$ and $b_2$ are variables of type $\mathsf{B}$, $a^1_{\mathsf{B}}$ and $a^2_{\mathsf{B}}$ are variables of type $\mathsf{array}\,[\mathsf{P}^n]\ \mathsf{of}\ \mathsf{B}$, and $i$ is a ruleset parameter. RHS values may also be the constants *true* and *false*.
- Assignments of the form $p_1 := p_2$, $a^1_{\mathsf{P}}[i] := p_2$, $a^1_{\mathsf{P}}[i] := a^2_{\mathsf{P}}[i]$, where $p_1$ and $p_2$ are variables of type $\mathsf{P}^n$, $a^1_{\mathsf{P}}$ and $a^2_{\mathsf{P}}$ are variables of type $\mathsf{array}\,[\mathsf{P}^n]\ \mathsf{of}\ \mathsf{P}^n$, and $i$ is a ruleset parameter.

- *Forall updates* of the form $\forall i \in \mathsf{P}^n.\ a_{\mathsf{B}}[i] := \ell(i)$, where $\ell$ is a boolean function depending on variables of type $\mathsf{B}$, $\mathsf{P}^n$ and on the $i^{th}$ index of array variables.

The general form of the rulesets are as follows.

```
ruleset i1 : Type1 do
   ruleset i2 : Type2 do
      ...
         ruleset iN : TypeN do
             rule "Rule1" guard1 ==> command1;
             ...
             rule "RuleK" guardK ==> commandK;
         end;
      ...
   end;
end;
```

These restrictions ensure that guards in admissible programs do not contain disjunctions of comparisons between variables of type $\mathsf{P}$ and have no existentially quantified terms; updates in admissible programs do not contain if-then-else clauses. These constructs can be handled by a straightforward splitting into multiple rulesets. The Mur$\varphi$ systems for German and FLASH are admissible, and from this experience, we believe that the systems for many other symmetric protocols will be admissible or easily modified to produce an admissible equivalent.

### 5.3.2 Abstraction

Let $\mathcal{P}(n) = (S(n), I(n), T(n))$ be the denotation of an admissible program $P$. We want to construct a mixed abstraction, $\mathcal{A} = (S_{\mathcal{A}}, I_{\mathcal{A}}, O, U)$. In this section, we show how $S_{\mathcal{A}}$, $I_{\mathcal{A}}$, and $O$ can be readily found by syntactic transformations of the source-code of $P$. Section 5.4 extends this approach to the construction of $U$. To create these abstractions, we introduce a new type to represent type $\mathsf{P}^n$ from the concrete system; this type requires

| concrete type | abstract type | abstraction $\psi$ |
|---|---|---|
| $\mathsf{B}$ | $\mathsf{B}$ | $v(\psi(s)) = v(s)$ |
| $\mathsf{P}^n$ | $\mathsf{P}^k \cup \{\mathit{Other}\}$ | $v(\psi(s)) = \alpha_k(v(s))$ |
| array $[\mathsf{P}^n]$ of $\mathsf{B}$ | array $[\mathsf{P}^k]$ of $\mathsf{B}$ | $\forall i \in \mathsf{P}^k \ : \ v(\psi(s))[i] = v(s)[i]$ |
| array $[\mathsf{P}^n]$ of $\mathsf{P}^n$ | array $[\mathsf{P}^k]$ of $\mathsf{P}^k \cup \{\mathit{Other}\}$ | $\forall i \in \mathsf{P}^k \ : \ v(\psi(s))[i] = \alpha_k(v(s)[i])$ |

Table 5.1: Mapping of types from concrete system to mixed abstraction. The abstract state space is $S_{\mathcal{A}}$ and the abstraction function is $\psi : S(n) \rightarrow S_{\mathcal{A}}$. For a system variable $v$ and $s \in S(n)$, the leftmost column gives the type of $v$ in the concrete domain, the second column gives the type of $v$ in $S_{\mathcal{A}}$, and the third column specifies the value $v$ is assigned by $\psi(s)$ in terms of $v(s)$. Function $\alpha_k : \mathsf{P}^n \rightarrow \mathsf{P}^k \cup \{\mathit{Other}\}$ is defined as $\alpha_k(v) = v$ for $v \leq k$; $\alpha_k(v) = \mathit{Other}$ otherwise.

the user to choose a constant $k$. It is assumed throughout that $k$ is *at least* the greatest number of ruleset parameters for any ruleset in $P$ (typically, $k \leq 3$). Table 5.1 specifies how each variable of $\mathcal{P}$ is typed in $\mathcal{A}$ and how the abstraction function $\psi$ acts on $v$. Intuitively, $\psi(s)$ preserves $\mathsf{B}$ variable values, replaces values of type $\mathsf{P}^n$ greater than $k$ with *Other*, and restricts arrays to the indices $\mathsf{P}^k$ (hence all array entries $v[i]$ for $i > k$ are deleted by $\psi$). Although $\psi$ is a function, we will treat it as a relation, $\psi \subseteq S(n) \times S_{\mathcal{A}}$, and freely employ its inverse $\psi^{-1} \subseteq S_{\mathcal{A}} \times S(n)$. We call elements of $\mathsf{P}^k$ *non-abstracted* and elements of $\mathsf{P}^n \setminus \mathsf{P}^k$ *abstracted*. For every ruleset parameter $i$ interpreted as abstracted, all updates with $a_{\mathsf{B}}[i]$ or $a_{\mathsf{P}}[i]$ appearing on the LHS are deleted. All instances $a_{\mathsf{B}}[i]$ or comparisons depending on $a_{\mathsf{P}}[i]$ appearing positively in the guard are replaced with *true*; those appearing negatively are replaced with *false*. Instances of $i$ appearing on the RHS of assignments are replaced with *Other*. Finally, equality comparisons with $i$ appearing positively in the guard are replaced with *true*. The state variables of $\mathcal{A}$ have the same names as those of $\mathcal{P}$, with the types changed as shown in Table 5.1.

We now overload $\psi$ to map rules of $\mathcal{P}$ system to rules that generate the state transitions of $O$. Rules of $\mathcal{P}$ that are not in rulesets are copied without change of syntax (therefore, with the implied change of types), to $O$. If ruleset $r : \rho \twoheadrightarrow \boldsymbol{a}$ depends on $m$ ruleset parameters, consider the set of *rule instantiations*, obtained from assigning each ruleset parameter a value in $\mathsf{P}^n$. This set is partitioned as $r_0, ..., r_{2^m-1}$, where all rule instantiations of $R_j$ have the same partitioning of ruleset parameters into $\mathsf{P}^k$ and $\mathsf{P}^n \setminus \mathsf{P}^k$ (since there are $m$ ruleset parameters, there are $2^m$ possible partitions). Each set $r_j$ abstracts to an abstract ruleset $\widehat{r}_j$ according to the described syntactic transformation. We denote the set of corresponding abstract rulesets to concrete ruleset $r$ by $\psi(r) = \{\widehat{r}_0, ..., \widehat{r}_{2^m-1}\}$. Let $\widehat{r}_0 : \widehat{\rho}_0 \twoheadrightarrow \widehat{\boldsymbol{a}}_0$ denote the unique element of $\psi(r)$ such that all ruleset parameters are non-abstracted. Note that although the set of rule instantiations differ depending on the value of $n$, the set $\psi(r)$ does not, for any $n > k$, hence we can fix $n = k + 1$ to perform this abstraction.

In this chapter I will use the German cache coherence protocol [58] as an example; a Mur$\varphi$ model for the protocol appears in Appendix A. The `SendGntE` ruleset sends a message to inform a cache/node that it has been granted exclusive access to a cache line.

**Example:** The concrete rule from German `SendGntE` is

```
ruleset i : NODE do rule "SendGntE"
  CurCmd = ReqE ∧ CurPtr = i ∧ Chan2[i].Cmd = Empty
    ∧ ¬ExGntd ∧ forall j : NODE do ¬ShrSet[j] end ==>
  Chan2[i].Cmd := GntE; ShrSet[i] := true;
  ExGntd := true; CurCmd := Empty;
```

The abstraction contains two corresponding rulesets, one where $i$ is non-abstracted and one where $i$ is abstracted, with the former corresponding to $\widehat{r}_0$:

```
ruleset i : NODE_A do rule "ABS_SendGntE1"
CurCmd = ReqE ∧ CurPtr = i ∧ Chan2[i].Cmd = Empty
  ∧ ¬ExGntd ∧ forall j : NODE_A do ¬ShrSet[j] end ==>
Chan2[i].Cmd := GntE; ShrSet[i] := true;
ExGntd := true; CurCmd := Empty;

rule "ABS_SendGntE2"
CurCmd = ReqE ∧ CurPtr = Other ∧ ¬ExGntd
  ∧ forall j : NODE_A do ¬ShrSet[j] end ==>
ExGntd := true; CurCmd := Empty;
```

Notice the difference between the forall condition of `SendGntE` and the weaker one of `ABS_SendGntE1`, despite the identical syntax. The type `NODE` in the concrete system ranges over values from 1 to $n$, where as `NODE_A` in the abstract system ranges from 1 to $k < n$.

## 5.4 Verifying Universal Quiescence

We want to verify properties of the form

$$\mathcal{P}(n) \models \mathsf{AG}\,\mathsf{EF}\,Q_n \;, \tag{5.3}$$

where $\mathcal{P}(n)$ is a parameterized system and

$$Q_n \;=\; G \wedge \textstyle\bigwedge_{i \in \mathsf{P}^n} L[i] \tag{5.4}$$

is the quiescence property to be verified. Here, $G$ is a *boolean predicate*, meaning $G$ only depends on variables of type $\mathsf{B}$, while $L$ is a local boolean predicate (defined in Sect. 5.3.1). To verify (5.3), we construct a mixed abstraction, $\mathcal{A} = (S_{\mathcal{A}}, I_{\mathcal{A}}, O, U)$, and show that for all $O$-reachable states, there exists a $U$-path to a state that satisfies $Q_n$. To do so, we must address two key issues. First, $Q_n$ cannot be established directly from $\mathcal{A}$, as $Q_n$ refers to variables of the concrete system that do not appear in the abstraction. This is the "abstract quiescence insufficiency" defined in Section 5.2.2; Section 5.4.1 shows how it can be addressed. Second, $U$ may omit transitions that are required to reach states that satisfy $Q_n$. This is the UA insufficiency from Section 5.2.2, and we address it in Sections 5.4.2 and 5.4.3.

### 5.4.1 Universally Quantified Quiescence

To show (5.3), we need to show that $L[i]$ holds for all $i$, not just non-abstracted $i$. Intuitively, we show that $\mathcal{A}$ can reach a state where $L[i]$ holds for all non-abstracted $i$, then use Lemma 3 (Path Symmetry) to exchange any abstracted $j$ for which $L[j]$ might not hold with a non-abstracted $i$ and find a path that establishes $L[i]$. We then establish (5.3) by induction. For this approach to be valid, we must show that if $G$ holds, then for each non-abstracted node $i$, there is a $U$-path to a state that satisfies $L[i]$ whose concretization in $\mathcal{P}(n)$ does not falsify $L[j]$ for any abstracted node $j$. To do this, we introduce the notion of $L$-preserving transitions. For the remainder of this chapter, let *Reach* denote the reachable states of $\mathcal{S}(n)$.

**Definition 2.** *For local boolean predicate $L$, abstract transition $(s, s')$ is* $L$-preserving *if*

$$\begin{aligned} & \forall w \in \psi^{-1}(s) \cap \mathit{Reach}.\ \exists w' \in \psi^{-1}(s').\ w \rightsquigarrow_T w' \\ \wedge \quad & \forall i \in \mathsf{P}^n \setminus \mathsf{P}^k.\ w \in L[i] \Rightarrow w' \in L[i]\ . \end{aligned}$$

Abstract ruleset $\widehat{r}$ is $L$-preserving if all transitions in $[\![\widehat{r}]\!]$ are $L$-preserving. A mixed abstraction is called $L$-preserving if its UA transitions contain only $L$-preserving rules.

Let us fix a mixed abstraction $\mathcal{A} = (S_\mathcal{A}, I_\mathcal{A}, U, O_\mathcal{A})$ for $\mathcal{P}(n)$, where $n > k$. We also use $L$ to denote a local boolean predicate, and $\mathcal{B} = (S_\mathcal{A}, I_\mathcal{A}, U_\mathcal{B}, O_\mathcal{A})$ to denote a mixed abstraction with only $L$-preserving transitions for $U_\mathcal{B}$. Note that the same set $O_\mathcal{A}$ is used for overapproximate transitions of both $\mathcal{A}$ and $\mathcal{B}$. For $i, j \in \mathsf{P}^n$, define $\mathsf{P}_i^j \subseteq \mathsf{P}^n$ as $\{h \in \mathsf{P}^n : i \leq h \leq j\}$.

Let permutation $\pi_{j\leftrightarrow h}$ map elements of $\mathsf{P}^n$ according to

$$\pi_{j\leftrightarrow h}(i) = \begin{cases} j & \text{for } i = h, \\ h & \text{for } i = j, \\ i & \text{otherwise.} \end{cases} \tag{5.5}$$

Let $T$ be shorthand for $T(n)$. We can now state our main theorem for showing universally quantified quiescence.

**Theorem 1** (Universally Quantified Quiescence)**.** *Let $G$ denote a boolean predicate, $L$ a local boolean predicate, and let $\mathcal{A}$ and $\mathcal{B}$ be mixed abstractions of $\mathcal{P}(n)$, and assume that $\mathcal{B}$ is $L$-preserving. If*

1. $\mathcal{A} \models \mathsf{AG}\,\mathsf{EF}\,(G)$, *and*

2. $\mathcal{B} \models \mathsf{AG}\,(G \rightarrow \mathsf{EF}\,(G \wedge \bigwedge_{i \in \mathsf{P}^k} L[i]))$

*then* $\mathcal{P}(n) \models \mathsf{AG}\,\mathsf{EF}\,(G \wedge \bigwedge_{i \in \mathsf{P}^n} L[i])$.

*Proof.* For $1 \leq h \leq n$, let $J_h$ denote the property $\forall w \in G \wedge \mathit{Reach}.\ \exists w' \in (G \wedge \bigwedge_{i \in \mathsf{P}^h} L[i])$ where $w \rightsquigarrow_T w'$, and $\forall i \in \mathsf{P}^n_{k+1}.\ w \in L[i] \rightarrow w' \in L[i]$. Antecedent 2 implies $J_k$. Assume $J_h$ holds for $k \leq h < n$. Applying permutation $\pi_{1 \leftrightarrow h+1}$ to $J_k$ gives $\forall w \in G \wedge \mathit{Reach}.\ \exists w' \in (G \wedge \bigwedge_{i \in \mathsf{P}_2^{h+1}} L[i])$ where $w \rightsquigarrow_T w'$, and $\forall i \in \mathsf{P}^n_{k+1}.\ w \in L[i] \rightarrow w' \in L[i]$. This property with Antecedent 2 implies $J_{h+1}$ by transitivity. Thus, property $J_n$ follows by induction. The paths implied by Antecedent 1 composed with those of $J_n$ complete the proof by transitivity. $\square$

### 5.4.2 Abstract Rule Tags

Given a program $P$, we use the syntactical abstraction technique to produce an abstract program $\widehat{P}$. In the remainder of this paper, we use the term "ruleset" to refer to Mur$\varphi$-style rulesets with any degree of ruleset nesting including no such quantification – *e.g.* a "ruleset" could be a simple rule. We want to identify which rulesets of $\widehat{P}$ have denotations that are UA, and which are $L$-preserving, for a given local boolean predicate $L$. We use $r : \rho \twoheadrightarrow \boldsymbol{a}$ and $\widehat{r}_0$ as defined in Section 5.3.2, and use $\widehat{r}_j$ to denote an arbitrary element of $\psi(r)$.

We tag abstract rulesets with tags from the set $\{\mathsf{AUG}, \mathsf{AEG}, \mathsf{AUC}, \mathsf{AEC}\}$; the first two elements are called *guard tags*, and the last two are called *command tags*. These indicate reasons (in the guard and command, respectively) why the abstract ruleset is not trivially UA or $L$-preserving. An abstract ruleset can be tagged with *any* of the 16 subsets of these tags. AUG and AUC indicate that a universal quantifier has been abstracted; similarly AEG and AEC indicate that existential information has been abstracted.[7]

---

[7] Note that AEG and AEC are not indication of explicit existential quantifiers in the concrete

We call $\rho$ and $\widehat{\rho}_j$ *syntactically equivalent* if they are expressed with identical syntax, and $\rho$ contains no forall conditions. In this case, we attach no guard tags to $\widehat{r}_j$. Likewise, if $\boldsymbol{a}$ and $\widehat{\boldsymbol{a}}_j$ have identical syntax and contain no forall updates, then we attach no command tags. If a ruleset $r$ has no guard or command tags, then it is simple to show that $r$ is both UA and $L$-preserving for any local predicate $L$. Typically, the set of all such rulesets is insufficient to establish the desired quiescence property.

The elements of $\psi(r) \setminus \widehat{r}_0$ may not have syntatically equivalent guards because some ruleset parameter $i$ is abstracted, so the abstraction will syntactically change the guard (except for degenerate cases). These rulesets have guards that optimistically abstract away references to abstracted $i$; this is safe when constructing the OA but not the UA. Such rulesets are tagged with AEG (abstract existential in guard)[7]. For example, rule `ABS_SendGntE2` in Section 5.3.2 is tagged with AEG due to the `CurPtr = Other` conjunct. When $\rho$ contains a forall condition, it is necessarily weakened in every rule of $\psi(r)$. In this case, every ruleset of $\psi(r)$ including $\widehat{r}_0$ is tagged with AUG (abstract universal in guard).

Similarly, existential or universal updates may be missing from the command of an abstract ruleset, relative to the concrete version. If local update $a_b[i] := e_b$ appears in $\boldsymbol{a}$ for ruleset parameter $i$, then any ruleset of $\psi(r)$ where $i$ is abstracted (that is, where the update $a_b[i] := e_b$ vanishes), is tagged with AEC (abstract existential command).[7] If $\boldsymbol{a}$ contains a forall update, then every ruleset of $\psi(r)$ is tagged with AUC (abstract universal command).

**Examples:** Referring to the example in Section 5.3.2, abstract ruleset `ABS_SendGntE1` is tagged with AUG because the `forall` condition only ranges over concrete nodes. Abstract rule `ABS_SendGntE2` is tagged with AUG for the same reason, AEG because the guard conjunct `Chan2[i].Cmd = Empty` is removed, and AEC because the updates `Chan2[i].Cmd := GntE` and `ShrSet[i] := true` are abstracted away.

The concrete rule `RecvReqS` from the German protocol (Appendix A) is

---

system syntax; *existential* refers to the quantifier in the ruleset denotation, which ranges over the ruleset parameter.

```
  ruleset i : NODE do rule "SendGntE"
    CurCmd = ReqE ∧ CurPtr = i ∧ Chan2[i].Cmd = Empty
      ∧ ¬ExGntd ∧ forall j : NODE do ¬ShrSet[j] end ==>
    Chan2[i].Cmd := GntE; ShrSet[i] := true;
ExGntd := true; CurCmd := Empty;

  ruleset i : NODE do rule "RecvReqS"
    CurCmd = Empty & Chan1[i].Cmd = ReqS ==>
    CurCmd := ReqS; CurPtr := i; Chan1[i].Cmd := Empty;
    for j : NODE do InvSet[j] := ShrSet[j] end;
  end end;
```

The abstraction contains two corresponding rulesets, one where $i$ is non-abstracted and one where $i$ is abstracted.

```
  ruleset i : NODE_A do rule "ABS_RecvReqS1"
    CurCmd = Empty & Chan1[i].Cmd = ReqS ==>
    CurCmd := ReqS; CurPtr := i; Chan1[i].Cmd := Empty;
    for j : NODE_A do InvSet[j] := ShrSet[j] end;
  end end;

  rule "ABS_RecvReqS2"
    CurCmd = Empty ==>
    CurCmd := ReqS; CurPtr := Other;
    for j : NODE_A do InvSet[j] := ShrSet[j] end;
  end;
```

Both rules `ABS_RecvReqS1` and `ABS_RecvReqs2` are tagged with AUC, as the `for` update only ranges over concrete nodes.

### 5.4.3 Heuristics

Each tag assigned to a ruleset corresponds to a set of proof obligations for showing it is UA or $L$-preserving (for some local boolean predicate $L$). Either of these properties can

|  | Property |  |
|---|---|---|
| Tag | UA | $L$-preserving |
| AEG (Abstracted Existential Guard) | Heuristic 1 | Heuristic 2 |
| AUG (Abstracted Universal Guard) | Heuristic 3 | Heuristic 4 |
| AEC (Abstracted Existential Command) | None | Heuristic 2 |
| AUC (Abstracted Universal Command) | None | Heuristic 4 |

Table 5.2: Heuristics for ruleset tag/property pairs and associated obligations. "None" means there is no obligation to show. A ruleset with no *tags* is $L$-preserving, while one with no *guard tags* is UA.

be established by *discharging* through the corresponding heuristic according to Table 5.2. Once a tag is discharged we may safely ignore it as a potential reason why the desired property does not hold. Each of the heuristics involves model checking a mixed abstraction. In this section, the various heuristics are stated.

An abstract ruleset is called *local to $i$* (as a special case of having no tags) when the guard only depends on variables of type $\mathsf{B}$, $\mathsf{P}$, and the $i^{th}$ index array variables $a_{\mathsf{B}}$, and the command only updates the local state of non-abstracted $i$. Here, given an abstract or concrete state, the *local state* of $i$ is simply the values of all array variables at index $i$. The transitions that compose such rules are called *local transitions*. A mixed abstraction with UA set $U$ composed only of rulesets local to $i$ is denoted $\mathcal{A}_{\ell(i)}$. Assuming ruleset $\widehat{r}$ is UA, we write $\mathcal{A}_{\ell(i)} \models \mathsf{AG}\,(A \rightarrow \mathsf{EF}_{\widehat{r}}\,B)$ when every $O$-reachable $A$-state has a path to some $B$-state consisting of transitions of rules local to $i$ and *necessarily a single transition* of ruleset $\widehat{r}$. See Figure 5.7.

When showing rulesets are UA (Heuristics 1 and 3), note that the tags AEG or AUG indicate guards that are OA because they have abstracted away information about abstracted nodes. Our heuristics compute $O$-reachable states and exploit Lemma 3 (Path Symmetry) to find the possible local state of abstracted nodes under some boolean predicate. Then, if the local state of node $i$ does not have a required property, we find "hidden paths" composed entirely of rulesets local to $i$ that reach a state that does have the property. This assures that although some states in the concretization of abstract guard $\widehat{\rho}_j$

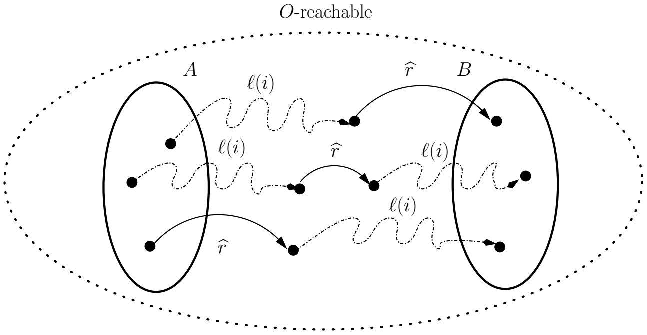


Figure 5.7: Illustration of $\mathcal{A}_{\ell(i)} \models \mathsf{AG}\,(A \rightarrow \mathsf{EF}_{\widehat{r}}\, B)$. Paths labeled $\ell(i)$ are paths composed of transitions local to $i$, and transitions labeled $\widehat{r}$ are transitions for that rule.

*do not* satisfy the corresponding concrete guard $\rho$, there is a guaranteed path that is not observable in the abstract system from every $\psi^{-1}(\widehat{\rho}_j)$ to a $\rho$-state. For simplicity, we present our heuristics for rulesets with at most one abstracted ruleset parameter; however, generalizing these heuristics to rulesets with more parameters is straightforward.

When showing rulesets are $L$-preserving, it must be checked that aspects of the guard and update that have been abstracted away do not affect $L$-preservation in the abstracted nodes; Heuristics 2 and 4 pertain to this check. The obligations for these heuristics require that a certain transition must fire on each path that justifies the deadlock freedom property. Intuitively, when the heuristic obligation holds, the concrete paths that justify the tagged ruleset in question $\widehat{r}$ to be UA must have a certain form. For abstracted node $i$, each path is

- a (possibly empty) path composed of transitions of rules local to $i$, followed by
- a transition of concrete rule $r$ (possibly changing non-local variables), followed by
- a (possibly empty) path composed of transitions of rules local to $i$.

Furthermore, we only seek a path when the starting state is an $L[i]$-state, and the final state must also be a $L[i]$-state. The $i$ for which this is shown depends on the heuristic.

Heuristic 2 reasons about those abstracted nodes that are abstracted ruleset parameters in $\widehat{r}$. Heuristic 4 reasons about those abstracted nodes that are *not* abstracted ruleset parameters in $\widehat{r}$. Note that we assume a ruleset has been proven UA before it is proven $L$-preserving.

A few definitions are needed for the heuristic statements. If $\widehat{r}$ is an abstract ruleset with ruleset parameter $i$, let $\widehat{r}|_{i=1} : \widehat{\rho}|_{i=1} \rightarrowtriangle \widehat{\boldsymbol{a}}|_{i=1}$ be the ruleset where all instances of $i$ are replaced with the constant value 1. Also, let $\mathit{relax}(\widehat{\rho}, i)$ be the guard $\widehat{\rho}$ but with the values of variables $a_{\mathsf{B}}[i]$ and $a_{\mathsf{P}}[i]$ unconstrained. If $A \subseteq S_{\mathcal{A}}$, let $\Gamma(A)$ denote the strongest boolean predicate implied by $A$.

We include proofs for all heuristics in Appendix B, and sketch the proof of Heuristic 1 here. are similar, and are omitted. Throughout these proofs, we fix a mixed abstraction $\mathcal{A} = (S_{\mathcal{A}}, I_{\mathcal{A}}, U, O)$ for $\mathcal{P}(n)$, where $n > k$.

**Heuristic 1.** *For ruleset $\widehat{r}_j$ tagged with* AEG *and with abstracted ruleset parameter $i$, suppose that $\widehat{r}_0$ is UA. If $\mathcal{A}_{\ell(1)} \models \mathsf{AG}\,(\mathit{relax}(\widehat{\rho}_0, i)|_{i=1} \rightarrow \mathsf{EF}\,(\widehat{\rho}_0|_{i=1}))$ then tag* AEG *is discharged for showing $\widehat{r}_j$ to be UA.*

*Proof. (Sketch)*

This Heuristic handles the case where we aim to show that an abstracted node eventually performs some action $\widehat{r}_j$. The idea is to consider any concrete state that when abstracted, satisfies the guard of $\widehat{r}_0$. Some of these concrete states may not satisfy the guard of the concrete rule $r_v$ due to the values of *local* variables of the abstracted node. We use a permutation to "swap" the abstracted node with the non-abstracted node 1. Because the values of local variables of the abstracted node are unknown (other than that they satisfy any safety properties that have been shown by suitable methods), we get that the state reached by the permutation satisfies $\mathit{relax}(\widehat{\rho}_0, i)|_{i=1}$. We construct a local path to a state that satisfies $\widehat{\rho}_0|_{i=1}$ and then perform action $\widehat{r}_j$. Details of the proof are given in Appendix B. □

**Heuristic 2.** *For ruleset* $\widehat{r}_j$ *tagged with* $\mathsf{AEG}$ *and/or* $\mathsf{AEC}$ *with abstracted ruleset parameter* $i$*, suppose that* $\widehat{r}_0$ *is UA. If* $\mathcal{A}_{\ell(1)} \models \mathsf{AG}\left((\mathit{relax}(\widehat{\rho}_0, i)|_{i=1} \wedge L[1]) \rightarrow \mathsf{EF}_{\widehat{r}_0}\left(L[1]\right)\right)$ *then* $\mathsf{AEG}$ *and* $\mathsf{AEC}$ *are discharged for showing* $\widehat{r}_j$ *to be* $L$*-preserving.*

**Heuristic 3.** *For ruleset* $\widehat{r}_j$ *tagged with* $\mathsf{AUG}$ *and not* $\mathsf{AEG}$*, let* $\forall i \in \mathsf{P}^n.\ C[i]$ *be the forall condition of* $\rho$*. If* $\mathcal{A}_{\ell(1)} \models \mathsf{AG}\left(\Gamma(\widehat{\rho}_j) \rightarrow \mathsf{EF}\left(C[1]\right)\right)$*, then tag* $\mathsf{AUG}$ *is discharged for showing* $\widehat{r}_j$ *to be UA.*

**Heuristic 4.** *For ruleset* $\widehat{r}_j$ *tagged with* $\mathsf{AUG}$ *and/or* $\mathsf{AUC}$*, let* $\widehat{r}_{2^m-1} \in \psi(r)$ *be the abstract ruleset where all ruleset parameters of* $r$ *are abstracted. If* $\mathcal{A}_{\ell(1)} \models \mathsf{AG}\left((\Gamma(\widehat{\rho}_j) \wedge L[1]) \rightarrow \mathsf{EF}_{\widehat{r}_{2^m-1}}\left(L[1]\right)\right)$*, then tags* $\mathsf{AUG}$ *and* $\mathsf{AUC}$ *are discharged for showing* $\widehat{r}_j$ *to be* $L$*-preserving.*

We apply each of these heuristics by performing model checking using a mixed abstraction that uses *only* local rules for $U$. As local rules are identified entirely by syntax, they are known *a priori*; therefore, we could take a brute force approach that attempts to use our heuristics to prove every abstract rule is UA and $L$-preserving. However, we prefer to take a counter-example driven approach, as there are two distinct situations in which our heuristics may not suffice that arose in our case studies. Firstly, additional auxiliary variables may be needed to capture the system state with a slightly finer-grained abstraction. Secondly, if the ruleset is not underapproximate, manual guard strengthening or splitting into multiple rulesets may help. These are illustrated with examples in Section 5.5. See Appendix C for detailed examples of applying Heuristics 1 and 3.

## 5.5 Case Studies

Mixed abstractions are expressed as Mur$\varphi$ models. The OA rulesets are borrowed from Chou *et al.* [40] and the (initial) UA transitions are derived manually according to tags – those rules with no guard tags. Thus, the UA rulesets are maintained as a subset of the OA rulesets. Rulesets with no tags at all are identified as $L$-preserving, and the relevant subset of these are identified as local.

We used PReach [22] for the mixed abstraction checks. As described in Chapter 3, PReach was originally designed to check state-invariants. We added a feature to check CTL properties of the form $\mathsf{AG}\,(p \rightarrow \mathsf{EF}\,q)$. The search algorithm is simple: for every $(p \wedge \neg q)$-state $s$ visited during the forward reachability computation, choose an enabled rule of $U$ and fire it to reach a new state. Firing rules of $U$ continues until one of the following occurs.

1. a $q$-state is found,
2. a $U$-dead-end state is found, or
3. a cycle is detected.

In the first case, a path from $s$ to a $q$-state exists and we proceed with the forward reachability computation. In the second case, there *might* not exist such a path (although we believe that in practice this is strong evidence that no path exists). If a cycle is found, this is usually an indication that $U$ contains rules that do not help us reach $q$-states, so we might as well exclude them and try again[8]. For example, there are several easily identifiable rules in both German and FLASH that initiate requests by injecting messages, and are not useful transitions in finding a quiescent state where all messages are consumed. Notice that deadlock freedom properties can be verified by a CTL model checker, but for our case studies we chose PReach because it was straightforward to implement the notion of UA rulesets and counterexample generation.

This section contains a brief overview of the case studies. For a more detailed report, the reader may refer to supplementary material [21] including the Mur$\varphi$ sources.

### 5.5.1 Automatic Deadlock Freedom Predicates

As mentioned above, it is common when checking antecedent 1 of Theorem 1 to reach a $U$-dead-end state $\check{s}$ where no further progress can be made toward the goal. When this

---

[8] Clearly, if $U$ is an underapproximate transition relation and $U' \subset U$ then $U'$ is underapproximate as well. Accordingly, removing transitions from $U$ produces a mixed abstraction.

occurs, the model checker reports a failure and prints the rules of $O$ that are enabled in $\check{s}$, as a guide to the user of which rules could be useful to prove UA and add to $U$. These enabled rulesets necessarily have tags AEG or AUG or both. We have written a simple tool that, given a particular rule/ruleset name, will determine the tags and generate the model checking obligation to prove it is UA through Heuristics 1 and 3.

**Example:** Suppose we seek to show ruleset $\widehat{r}_2 =$ `ABS_SendGntE2` is UA, and suppose it is already known by Heuristic 3 that associated $\widehat{r}_1 = \widehat{r}_0 =$ `ABS_SendGntE1` is UA. Ruleset `ABS_SendGntE2` is tagged with AEG because ruleset parameter $i$ is abstracted. The guard $\widehat{\rho}_0|_{i=1}$ is

```
CurCmd = ReqE ∧ CurPtr = 1 ∧ Chan2[1].Cmd = Empty ∧ ¬ExGntd
  ∧ forall j : NODE do ¬ShrSet[j] end
```

and $relax(\widehat{\rho}_0, i)|_{i=1}$ is

```
CurCmd = ReqE ∧ CurPtr = 1 ∧ ¬ExGntd ∧ forall j : NODE do ¬ShrSet[j] end
```

As implemented, our tool does not support automatic generation of the properties to check for Heuristics 2 and 4. However, this is generally straightforward to do by hand, and could be automated as well. In cases when the deadlock freedom property for some heuristic when applied to ruleset $\widehat{r}_j$ fails the verification attempt, the user may use the counterexample trace as a guide for strengthening $\widehat{\rho}_j$ manually. Any ruleset of $O$ may be duplicated and strengthened with some predicate, which is trivially sound because the $O$ transitions are not changed. The resulting strengthened ruleset might satisfy the heuristic deadlock freedom property and be proven UA or $L$-preserving. Such manual strengthening is required in the verification of both German and FLASH.

### 5.5.2 The German Protocol

The system used for $O$ is the abstract Mur$\varphi$ model for German of Chou *et al.*, instantiated with a single non-abstracted node ($k = 1$). The initial set of UA transitions $U_0$ includes all rulesets with no guard tags and the local subset of these are also identified.

The property we verify is (5.4), where $G$ states that that the directory is not currently processing a transaction (`CurCmd = Empty`) and $L[i]$ states that all communication channels associated with the $i^{th}$ cache are empty:

`Chan1[i].Cmd = Empty` $\wedge$ `Chan2[i].Cmd = Empty` $\wedge$ `Chan3[i].Cmd = Empty`

Antecedent 1 of Theorem 1 requires $\mathcal{A} \models \mathsf{AG}\,\mathsf{EF}\,(G)$ for a mixed abstraction $\mathcal{A}$. Checking this property, the model-checker gets stuck at a $U$-dead-end state where the rule `ABS_SendGntE1` is enabled (see Section 5.3.2). Our tool recognizes this as an AUG-tagged rule and generates the obligations to according to Heuristic 1 so the rule can be soundly added to $U$. The model checker discharges the obligation, and `ABS_SendGntE1` is added to $U$.

Checking Antecedent 1 is repeated with the weakened $U$ and gets stuck three more times: once where `ABS_SendGntE2` is enabled (tagged with AEG *and* AUG), and twice where other AEG-tagged rulesets are enabled. The Heuristic 3 obligation for `ABS_SendGntE2` is identical to the one previously shown for `ABS_SendGntE1`, so there is no need to repeat its verification. The tool generates the Heuristic 1 obligation and it is discharged by model checking. In the other two, AEG cases, the corresponding rulesets $\widehat{r}_0$ are already known to be in $U$, so we proceed directly with the tool, and obligations for Heuristic 1 are generated. One is discharged automatically; the other requires human guidance because the generated deadlock freedom property fails to verify. An examination of the counterexample reveals that when exclusive access has been granted to an abstracted node, there is no pointer indicating which node has been granted (only a flag to indicate that it has indeed been granted, `ExGntd`). Without this pointer, the permutation of Heuristic 1 is not applied to the proper abstract node actually holding exclusive access. Although manual, the solution is straightforward: add a new system variable `EPtr` of type P that points to the node holding exclusive access; and strengthen the guard of the ruleset. This is done in a sound manner where only the ruleset version we prove is UA is strengthened in this way; the original ruleset belonging to $O$ is not modified. After this modification, the relevant property is verified.

Having added these four rules to $U$ of mixed abstraction $\mathcal{A}$, Antecedent 1 of Theorem 1 is established by model checking. We now describe the procedure to show Antecedent 2. Initially, all rulesets that have no tags are known to be $L$-preserving, and these are added to $U$ for mixed abstraction $\mathcal{B}$. Model checking then reveals that two additional rules are needed to establish the Antecedent: `ABS_SendGntE1` (tagged AUG) and `ABS_RecvInvAck2` (tagged AEG and AEC). These tags are discharged by automatically generating and checking the obligations of Heuristics 4 and 2, respectively. Adding these two rules to $U$ for mixed abstraction $\mathcal{B}$ allows Antecedent 2 to hold and completes the verification of the German protocol.

### 5.5.3 The FLASH Protocol

The quiescence property verified of FLASH is of the same form as (5.4), and states that all channels are clear and the directory is not waiting to perform a write-back[9]. Antecedent 1 of Theorem 1 holds immediately using the initial set of UA rulesets having no guard tags.

To show Antecedent 2 of Theorem 1, we start with the set $U$ of $L$-preserving states provided by tag examination and use model checking as with the German protocol. Four rules, each tagged with AEG and AEC, must be shown $L$-preserving. We first show that they are UA, by applying Heuristic 1. For two of these rulesets, model checking the obligations for Heuristic 1 succeeds. For the other two, model checking fails upon reaching a dead-end state $\check{s}'$ where no local rules fpr node 1 are enabled. The manual strengthening needed for these two rulesets is identical. Without loss of generality let the ruleset be $\widehat{r}_j$. Inspecting the counter example reveals that the state $s \in \mathit{relax}(\widehat{r}_0, i)|_{i=1}$ that led to $\check{s}'$ has different values for some B-type variables than those in $\check{s}$, the original dead-end

---

[9] Although the Mur$\varphi$ system for the mixed abstraction of FLASH contains rules where two index variables have been instantiated as *Other*, none of these must be shown UA/$L$-preserving to prove our example property. Some such rules are needed to be shown UA if the conjunct $\neg$`Pending` is added to the quiescent property. We omit these from this paper for ease of presentation, but note that similar reasoning to Heuristic 1, which assumes only one such index variable, is sufficient.

state revealed when checking Antecedent 2, where $\widehat{r}_j$ is enabled. This indicates that the guard $\widehat{r}_j$ is too weak and must be strengthened with a predicate on these variables. We duplicated the ruleset for the aforementioned reasons of soundness, and strengthened the guard with a predicate requiring these variables to match their value in $\check{s}$. Then, the automated procedure completed successfully and the four rules are established as UA. To show they are $L$-preserving, Heuristic 2 is applied to each ruleset and the obligations are discharged automatically; this establishes the quiescence property by Theorem 1. With regard to the manual strengthening step, we note that in principle the model checker could classify the reachable states of $\mathit{relax}(\widehat{r}_0, i)|_{i=1}$ for which a path to $\widehat{r}_0|_{i=1}$ is found versus those where no such path is found. Thus, the strengthening predicate could be generated automatically.

## 5.6 Discussion

Much of the theory is restricted for ease of presentation and because it is sufficient for the case studies. However, there are a number of generalizations and further directions to consider. As this work has been guided by example protocols, in the future we may apply these methods to a real-world protocol that requires one or more of the following extensions.

### 5.6.1 Permutations on More than One Abstracted Node

As mentioned, the current method for dealing with the AEG tag focuses on permuting at most a single abstracted node with some non-abstracted node. The same principle may be applied to multiple abstracted nodes, where they are each swapped with a distinct non-abstracted node. The local state of such nodes are unknown, and furthermore it is unknown if they are equal or not. Regardless, the reachability of the OA transitions may be strong enough to show they must be equal or unequal, and may have highly constrained local states[10]. This technique is also applicable to array variables. Suppose the guard of some concrete ruleset with parameter `i` contains the condition `i = Ptr1 & MyArray[i] = Ptr2`, where `MyArray` is of type `array [P] of P`, and `Ptr1` and `Ptr2` are of type `P`. An abstract ruleset where `i` is abstracted will overapproximate this condition as `Other = Ptr1`, but a permutation can be designed that swaps in *both* `i` and `MyArray[i]`, with `isdefined(Ptr1) & Ptr1 != Other & (isdefined(Ptr2) -> Ptr2 != Other)`. This is the start predicate of the DF property to check for proving the abstract ruleset is UA (the end predicate is, `exists i : NODE do i = Ptr1 & MyArray[i] = Ptr2 end`).

### 5.6.2 Local Rule Generalizations

Our approach has a restricted view of what constitutes a "local" rule; the local state of at most one node may change in the update, and the *global state*, the variables of type B or P, may not change. Indeed, if a protocol description contains *no such rules*, then the heuristic methods cannot be applied to find "hidden paths", the paths found by model checking that establish a DF property and composed of local rules. One could imagine generalizing the definition of local rules to depend on and/or update the local state of (say) two nodes $i_1$ and $i_2$. However, this complicates the reasoning about hidden paths because it may be necessary to show that for some local state of $i_1$, there always exists a node $i_2$ such that a local rule depending on $i_1$ and $i_2$ can fire.

An orthogonal generalization is to allow local rules to change the global state. If every hidden path changes the global state in an identical manner, then the ruleset that is shown to be underapproximate must respect these changes in it's update. Such *side-effects* of these hidden paths could easily be incorporated to our method, but this was not

---

[10] As a side note, notice that if $m$ is the number of nodes that are swapped from the set of abstracted nodes to the set of non-abstracted nodes, then the number of equivalence class realizations is exactly the number of partitions of a set with $m$ elements. This is known as the $m^{th}$ Bell number $B_m$, which can be expressed as a sum of Stirling numbers of the second kind: $\sum_{i=0}^{m} \left\{ {m \atop i} \right\}$. The values of $B_m$'s for $1 \leq m \leq 10$ are $1, 2, 5, 15, 52, 203, 877, 4140, 21147, 115975$. Thus, this generalization to more than one abstracted node seems reasonable for small $m$ and intractable for $m \geq 10$.

necessary for our case studies.

### 5.6.3 Automatic Strengthening

In the case study of the FLASH protocol, one of the abstract rulesets of $O$ that we wanted to prove was UA had a guard that was too weak for the heuristic method to handle. Instead, we were able to show that a strengthened version of this rule was UA. The strengthening predicate, expressed in terms of variables of type $\mathsf{B}$, was deduced by comparing the global state of $\tilde{s}$ (the dead-end state found when checking an antecedent of Theorem 1) with $\tilde{s}'$ (the dead-end state found when checking the DF property of the heuristic). In the current implementation, PREACH will halt as soon as the constructed path from a state satisfying the heuristic start predicate $p_{start}$ reaches a dead-end. Instead, we could categorize all reachable $p_{start}$-states for which a path is found, $p_{path}$, and those for which a path is not found, $p_{no\text{-}path}$. Then, then a strengthening predicate over $\mathsf{B}$-typed variables could be determined that includes every $p_{path}$-state but none of the $p_{no\text{-}path}$-states. With this approach, human intervention is only necessary when key auxiliary variables are absent from the system.

| Symbol | Meaning |
|---|---|
| $\mathcal{P}$ | parameterized system |
| $n$ | natural number |
| $k$ | small fixed natural number |
| $\mathsf{P}$ | parametric type |
| $\mathcal{S}$ | a system $(S, I, T)$ |
| $S$ | set of system states |
| $I$ | system initial states |
| $T$ | system transition relation |
| $\mathcal{A}, \mathcal{B}$ | mixed abstraction |
| $\theta, \psi$ | abstraction relation |
| $S_{\mathcal{A}}$ | set of abstract states |
| $I_{\mathcal{A}}$ | set of abstract initial states |
| $w, w'$ | a concrete state |
| $s, s'$ | an abstract state |
| $w \rightsquigarrow_T w'$ | $T$-path from $w$ to $w'$ |
| $\mathsf{B}$ | boolean type |
| $\mathsf{P}^n$ | parametric type with $n$ values |
| $\pi$ | permutation on $\mathsf{P}^n$ |
| $r$ | a rule |
| $\rho$ | a rule guard |
| $\boldsymbol{a}$ | a rule update |
| $[\![r]\!]$ | transitions corresponding to $r$ |
| $L$ | a local boolean predicate |
| $\psi(r)$ | abstract rulesets corresponding to $r$ |
| $\widehat{r}_0$ | abstract ruleset with all ruleset parameters non-abstracted |
| $\widehat{r}_{2^m-1}$ | abstract ruleset all ruleset parameters abstracted |
| $Q_n$ | parameterized quiescent property; see (5.4) |
| $G$ | global boolean predicate |
| $\mathsf{P}_i^j$ | $\{h \in \mathsf{P}^n : i \leq h \leq j\}$ |
| $\pi_{j \leftrightarrow h}$ | permutation that exchanges $j$ and $h$; see (5.5) |
| $\widehat{r}_j$ | an element of $\psi(r)$ |
| $\mathcal{A}_{\ell(i)}$ | mixed abstraction with all UA transitions local to node $i$ |
| $\mathsf{EF}_{\widehat{r}}$ | exists a path along which $\widehat{r}$ fires |
| $\widehat{r}\vert_{i=1}$ | $\widehat{r}$ with instances of ruleset parameter $i$ replaced with 1 |
| $\mathit{relax}(\widehat{\rho}, i)$ | guard $\widehat{\rho}$ with constraints local to $i$ removed |
| $\Gamma(A)$ | strongest boolean predicate implied by $A$ |

Table 5.3: List of chapter symbols

# Chapter 6

# Distributed Response Property Checking

A response property is a simple liveness property that, given state predicates $p$ and $q$, asserts “whenever a $p$-state is visited, a $q$-state will be visited in the future”. This chapter presents an efficient and scalable implementation for explicit-state model checking of response properties on systems with strongly- and weakly-fair actions, using a network of machines. Our approach is a novel twist on the One-Way-Catch-Them-Young (OWCTY) algorithm. Although OWCTY has a worst-case time complexity of $O(n^2m)$ where $n$ is the number of states of the model, and $m$ is the number of fair actions, we show that in practice, the run-time is a very small multiple of $n$. This allows our approach to handle large models with a large number of fairness constraints. Implemented with the PReach distributed, explicit-state model checker introduced in Chapter 3, we demonstrate the effectiveness of our approach by applying it to several standard benchmarks and on some real-world, proprietary, architectural models.[1]

---

[1] This chapter is based on [27].

## 6.1 Introduction

Response properties are liveness properties of the form "From any state in which proposition $p$ is satisfied, execution will eventually reach a state in which proposition $q$ is satisfied." In LTL such properties are expressed as $\Box(p \rightarrow \Diamond q)$; the corresponding CTL specification is $\mathsf{AG}\,(p \rightarrow \mathsf{AF}\,q)$. Specifications of cache protocols and high-level architectural models often include response properties — *e.g.* if a processor attempts to write to a memory location, the processor will eventually have an exclusive copy of that location in its cache; or, if an instruction is fetched, eventually either it will be executed and committed or that (speculative) path will be aborted. In comparison with the previous two chapters, response properties are stronger than the corresponding deadlock-freedom property $\mathsf{AG}\,(p \rightarrow \mathsf{EF}\,q)$, but verification is harder. By providing methods to verify both properties, we allow the user a choice in trading property strength versus model accuracy.

The standard approach to explicit state model checking of LTL properties involves constructing a product automaton. This automaton is the synchronous product of the Büchi automaton for the specification, and the Büchi automaton for the system itself [110, 59]. The specification automaton accepts the *negation* (*i.e.* violations) of the property, while the automaton with for system accepts traces that the system allows. By this construction, if the language accepted by the product automaton is empty, then the LTL property holds; otherwise, a counterexample trace is found. Counterexamples correspond to reachable cycles of the product automaton that do not include an accepting state. All model checking approaches are vulnerable to state-explosion problems, and the product-automaton construction for LTL model checking exacerbates this problem. If the original system has $n$ reachble states, and the LTL specification, $\phi$, consists of $|\phi|$ symbols and operators, then constructing the product automaton takes $\mathcal{O}(n2^{|\phi|})$ time and space.

Response properties for models without fairness assumptions can be expressed with a Büchi automaton with only 2 states [110], and thus the blowup from the formula size is curbed. Essentially, this Büchi automaton has one state where the system is waiting for a response, and the other state where no request is pending. Unfortunately, only contrived systems that contain no cycles along any path from a $p$-state to a $q$-state will satisfy such response properties. In practice, response is verified subject to *fairness assumptions* that attempt to characterize realistic traces. Response may be verified under those fairness assumptions that can be written as the LTL formula *Fair*, by using LTL model checking to verify the formula $\mathit{Fair} \rightarrow \Box(p \rightarrow \Diamond q)$. The Büchi automaton for this formula grows exponentially in $|\mathit{Fair}|$, which in turn causes the number of states of the product automaton to explode.

Instead of expressing fairness as an antecedent to the LTL property of interest, fairness can be expressed in terms of how the original system is defined or as a specially handled input to the model checking algorithm. Kesten *et al.* [74] compare expressing fairness as a property antecedent with a "fair-aware" approach and show that latter achieves better performance. Manna and Pnueli [88, 89] present a model-checking algorithm property checking for response properties that takes advantage of two notions of action-based fairness. We build upon these ideas and implemented a fair-aware version of OWCTY for response properties in the PReach model checker. Our approach uses Manna and Pneuli's notions of strong and weak fairness. In the worst-case, the algorithm could perform $O(n^2|\mathit{Fair}|)$ state expansions, where $n$ is the number of reachable system states. In the typical scenario where $|\mathit{Fair}|$ is much smaller than $\log(n)$, this far exceeds the number of worst-case expansions of the Büchi automaton approach which is $O(n2^{|\mathit{Fair}|})$. However, our results for benchmark models vastly outperform the worst-case, even though the worst case is achievable on a contrived example (see Section 6.3.1). In contrast, we also report results in Section 6.7 for a tool that implements the Büchi automaton approach and uses time and memory as one would expect from the worst-case analysis.

Our contributions are as follows.

1. We present a novel, efficient, parallel approach for model checking response properties.

2. The algorithm is implemented as an extension of the PReach [22, 30] model checker.

3. Demonstration that verifying liveness for large, realistic systems augmented with both strong and weak fairness is tractable using a modest network of machines.

4. We show that the time requirements for One-Way-Catch-Them-Young style algorithms are far better in practice than would be expected from the worst case analysis. In practice, we observe that each state is visited a small number of times (typically less than 30).

## 6.2 Overview

To check response properties, we implemented an algorithm inspired by the set-based *One-Way-Catch-Them-Young* algorithm described in [73, 38]. We focus on systems with both *strongly fair actions* (a.k.a. compassion), denoted $\mathcal{C}$ and *weakly fair actions* (a.k.a. justice), denoted $\mathcal{J}$.

### 6.2.1 Preliminaries

A fair transition system (FTS) is a tuple $(\mathcal{S}, I, T, \mathcal{J}, \mathcal{C})$ where

- $\mathcal{S}$ is a finite set of states;
- $I \subseteq \mathcal{S}$ is the set of initial states;
- transition relation $T \subseteq \mathcal{S} \times \mathcal{S}$;
- weakly fair actions $\mathcal{J} \subseteq 2^T$;
- strongly fair actions $\mathcal{C} \subseteq 2^T$.

An *action* is a subset of $T$. For example, the set of transitions corresponding to a Mur$\varphi$ rule could be an action. The rule *guard* $\multimap$ *action* describes the set of transitions,

$(s, s')$ where $s$ satisfies the state predicate *guard* and $s'$ is the state reached by performing update *action* starting from state $s$. Our implementation of action-based fairness associates weak- and strong-fairness constraints with Mur$\varphi$ rules.

Function $\mathit{En} : \mathcal{S} \to 2^{\mathcal{C} \cup \mathcal{J}}$ gives the set of actions enabled at state $s$, i.e. $\mathit{En}(s) = \{a \in \mathcal{C} \cup \mathcal{J} : \exists s'.\ (s, s') \in a\}$. State $s$ enables action $a$ if $a \in \mathit{En}(s)$. Given state $s$ we use the shorthand notations $\mathcal{C}_s$ and $\mathcal{J}_s$ to refer to the sets of enabled actions that are strongly and weakly fair, respectively. Formally, $\mathcal{J}_s = \mathcal{J} \cap \mathit{En}(s)$ and $\mathcal{C}_s = \mathcal{C} \cap \mathit{En}(s)$. For convenience we assume transitions that are not members of any element of $\mathcal{J} \cup \mathcal{C}$ are members of the *non-fair* set, i.e. $\mathcal{NF} = T \setminus \left(\bigcup_{a \in \mathcal{J} \cup \mathcal{C}} a\right)$. For $A \subseteq \mathcal{S}$, $\langle A \rangle$ denotes the subgraph of the digraph $(\mathcal{S}, T)$ induced by $A$.

A *trace* is a finite sequence of states $s_0 \circ s_1 \circ \ldots \circ s_\ell$ where $s_o \in I$, and $(s_i, s_{i+1}) \in T$ for $0 \leq i < \ell$. A *predecessor trace* for state $s$ is any trace where $s_\ell = s$.

An *execution* is an infinite sequence of states, $s_0 \circ s_1 \circ \ldots$, where $s_0 \in I$, and $\forall i \geq 0.\ (s_i, s_{i+1}) \in T$. For a given trace, action $a$ satisfies

- $\mathit{InfOftenTaken}(a)$, if $\forall i \geq 0.\ \exists k \geq i.\ (s_k, s_{k+1}) \in a$,
- $\mathit{InfOftenEn}(a)$, if $\forall i \geq 0.\ \exists k \geq i.\ a \in \mathit{En}(s_k)$, and
- $\mathit{InfOftenDisabled}(a)$, if $\forall i \geq 0.\ \exists k \geq i.\ a \notin \mathit{En}(s_k)$.

An execution is called *fair* if

$$\begin{array}{rl} & \forall a \in \mathcal{C}.\ \mathit{InfOftenEn}(a) \Rightarrow \mathit{InfOftenTaken}(a) \\ \wedge & \forall a \in \mathcal{J}.\ \mathit{InfOftenTaken}(a) \vee \mathit{InfOftenDisabled}(a). \end{array}$$

In other words, an execution is fair if both of the following hold of any suffix of the execution.

1. If action $a \in \mathcal{C}$ is enabled in an infinite number of suffix states then a transition in $a$ must eventually occur.

2. If action $a \in \mathcal{J}$ is enabled in *all* suffix states then a transition in $a$ must eventually occur.

A strongly connected component (SCC) is called *fair* (a FSCC) if all strongly fair actions enabled within the SCC are taken within the SCC, and all weakly fair action enabled within in the SCC are either taken within the SCC or disabled at some state within the SCC. Section 6.3 presents an algorithm that detects FSCCs within the subgraph of reachable states that can be reached on a path from some $p$-state without visiting a $q$-state along the way (this subset is referred to as *pending*; see Figure 6.1). Such FSCCs are counterexamples to the response property $\Box(p \rightarrow \Diamond q)$, as traces are infinite and the system is finite state. Furthermore, every counterexample execution has an infinite suffix that only visits states in a FSCC. Note that $p$ is a subset of *pending*, and $q$ is disjoint with *pending*. The initial states are usually disjoint from both $p$ and *pending*, but this need not be the case.

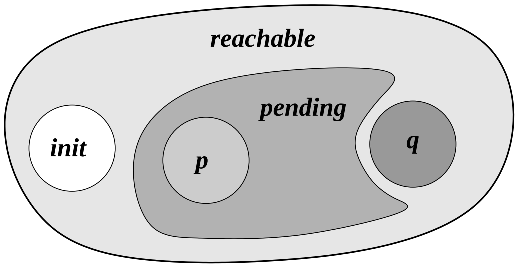


Figure 6.1: Sets of interest when checking a system adheres to $\Box(p \rightarrow \Diamond q)$.

### 6.2.2 A Note about Stuttering

We note that fair systems may be defined with or without inherent stuttering, the former assuming that every state has a transition to itself and the latter does not. For simplicity in the following presentation, we assume that stuttering is possible in all states, thereby requiring a fair "reason" why indefinite stuttering cannot occur. This assumption implies that $T$ is reflexive. Including stuttering simplifies the presentation; for example, it ensures that all traces can be extended to infinite executions.

**Algorithm 6.1** High level algorithm

```
function FindFairCycle($I \subseteq \mathcal{S}, T \subseteq \mathcal{S} \times \mathcal{S}, \mathcal{C} \subseteq \mathcal{A}, \mathcal{J} \subseteq \mathcal{A}, p \subseteq \mathcal{S}, q \subseteq \mathcal{S}$)
    ▷ $\mathcal{A}$ is the power set of the power set of $\mathcal{S} \times \mathcal{S}$
    $pending, MaybeFair, Prev, ToExpand \subseteq \mathcal{S}$
    ▷ Compute the pending states
    $pending \leftarrow$ Reachability($\mathcal{S}, I, T, p, q$)
    $ptfa \leftarrow$ new $bit[pending][\mathcal{J} \cup \mathcal{C}]$                ▷ array of bit-strings
    Clear($ptfa$)                                                     ▷ initialize to all 0s
    $MaybeFair \leftarrow pending$
    $Prev \leftarrow \emptyset$
    while $MaybeFair \neq Prev$ do
        $Prev \leftarrow MaybeFair$
        $ToExpand \leftarrow MaybeFair$
        while $ToExpand \neq \emptyset$ do
            $s \leftarrow$ RemoveSomeElement($ToExpand$)
            for all $a \in \mathcal{J} \setminus \mathcal{J}_s$ do              ▷ Weakly fair actions not enabled at $s$
                $ptfa[s][a] \leftarrow 1$
            end for
            $Next \leftarrow$ Successors($s$) $\setminus q$       ▷ Ignore $q$-states to remain within pending
            for all $s' \in Next$ do
                $OldActions \leftarrow ptfa[s']$
                for all $a \in$ WhatActionsTaken($s, s'$) do
                    if $a \in \mathcal{J} \cup \mathcal{C}$ then
                        $ptfa[s'][a] \leftarrow 1$                          ▷ Record action taken
                    end if
                end for
                $ptfa[s'] \leftarrow$ BitwiseOr($ptfa[s], ptfa[s']$)        ▷ Actions preceeding $s$ also preceed $s'$
                if ($ptfa[s'] \neq OldActions$) then
                    $ToExpand \leftarrow ToExpand \cup \{s'\}$
                end if
            end for
        end while
        for all $s \in MaybeFair$ do
            if $\exists a \in \mathcal{J}_s \cup \mathcal{C}_s : ptfa[s][a] = 0$ then
                $MaybeFair \leftarrow MaybeFair \setminus \{s\}$
            end if
        end for
        Clear($ptfa$)
    end while
    return $MaybeFair \neq \emptyset$
end function
```

## 6.3 Algorithm

Our distributed response checking algorithm is based on the One-Way-Catch-Them-Young (OWCTY) [73] approach. The key idea of the algorithm is to eliminate all states that do not belong to an FSCC and are unreachable from an FSCC. If all reachable states are eliminated, then no counterexample exists; otherwise an FSCC is contained within the remaining states. Observe that if a state is part of some FSCC, then any fairness constraints associated with that state must be satisfied by an (infinite length) path within the FSCC. In particular, there will be an *incoming* path to the state that satisfies the fairness constraints of the state. OWCTY propagates the satisfaction of fairness constraints forward, thus marking each state with the fairness constraints that are satisfied by incoming paths. When this process reaches a fixpoint, states whose fairness constraints are not satisfied cannot be part of a FSCC. These states are eliminated. This process is repeated until a fixpoint is reached. If all reachable states are eliminated, then no counterexample exists; otherwise an FSCC is contained within the remaining states.

This computation is implemented by first initializing a set *MaybeFair* with the *pending* states, and then iteratively removing states from *MaybeFair* that cannot belong to a FSCC. State $s$ is removed when it is discovered that there is no predecessor trace of $s$ in $\langle \mathit{MaybeFair} \rangle$ along which action $a \in \mathcal{C}$ is taken, where $a \in \mathcal{C}_s$. Similarly, $s$ is removed if it is found that there is no predecessor trace of $s$ in $\langle \mathit{MaybeFair} \rangle$ along which action $a \in \mathcal{J}_s$ is either taken *or* disabled at some state $s'$ of the trace, where $a \in \mathcal{J}_s$. The response property holds iff *MaybeFair* is empty when the algorithm terminates. To see this, note that any state that is removed from *MaybeFair* cannot belong to a FSCC; thus, $\langle \mathit{MaybeFair} \rangle$ contains all of the FSCCs of $\langle \mathit{pending} \rangle$.

Furthermore, the SCCs of $\langle \mathit{pending} \rangle$ form a DAG. Consider an SCC $C$ that does not have any incoming edges. If $C$ is not an FSCC, then it contains state $s$ with some fairness constraint that is not satisfied by any transition within $C$. Because there are no incoming edges to $C$ from other SCCs, there is no incoming path to $s$ that satisfies the fairness constraint under consideration, and $s$ will be eliminated. Thus, if $\langle pending \rangle$ contains no FSCCs, then OWCTY will eventually eliminate all of its states.

The FSCCs of $\langle MaybeFair \rangle$ form a DAG. Let $F$ be any FSCC of $\langle MaybeFair \rangle$ that has no predecessor FSCCs. It is straightforward to construct a cycle in $F$ that satisfies all fairness constraints. By construction, this cycle is reachable from some initial state.

The description of OWCTY from [38] for model checking LTL formulas with strong and weak state-based fairness operates on sets of states performing union and disjunction operations, as well as deleting all members from a set which have no predecessor within the set until a fixed point is reached[2]. As described in Section 3.1, PReach uses lossy compression when hashing states; thus, we cannot reconstruct states from hashtable entries. To retain the efficiency advantages of the Mur$\varphi$ hashtables, we avoid the explicit representation of large sets of states, and replace the union and intersection operations of OWCTY with tag bit manipulations, where each hash table entry includes one such tag bit per fair action. In Algorithm 6.1, these bits are stored in *ptfa (predecessor trace fair actions)* table, which is a two-dimensional array of bits (line 6) initialized to all 0s (line 7. The two indices correspond to the elements of *pending* (*i.e.* the hash table index), and the fair actions (*i.e.* which tag bit in the hash table entry). For any action $a \in \mathcal{J} \cup \mathcal{C}$ and state $s \in MaybeFair$, bit $ptfa[s][a]$ is set if $a$ is taken in some predecessor trace of $s$ in $\langle MaybeFair \rangle$, or if $b \in \mathcal{J}$ is disabled at some state of a predecessor trace of $s$ in $\langle MaybeFair \rangle$. The set *pending* stores the states of interest for response, those that can be reached on a path from a $p$-state without visiting a $q$-state.

Each iteration of the outer while-loop (lines 10–38) is called a *round*, and involves two *phases.*

*Action Propagation Phase (AP)*:

This step is the while-loop from lines 13 to 31. Some state $s$ is removed from *ToExpand* and the tag bits are set for each weakly fair action that is disabled at $s$; this is because

---

[2] To the best of our knowledge, the algorithm from [38] not been implemented.

any eventual successor of $s$ within $\langle pending \rangle$ *may* be part of an SCC with $s$. If so, this SCC is fair with respect to these weakly fair actions. Then, the successors of $s$ within $\langle pending \rangle$ are computed. For each of these, the current tag bits are saved in *OldActions*. If the transition that is taken from $s$ to reach a successor $s'$ is a member of some $a \in \mathcal{J} \cup \mathcal{C}$, the *ptfa*$[s'][a]$ is set (line 23). Then, the bit-string *ptfa*$[s]$ is ORed with the *ptfa*$[s']$, as any predecessor trace $\rho$ for $s$ implies a predecessor trace for $s'$, namely $\rho \circ s'$. If any of these operations have set new bits for $s'$, it must be added to *ToExpand* so the bits are propagated along. Otherwise, the $s'$ is discarded. This loop continues until a fixed point is reached for the contents of *ptfa*.

Figure 6.2 illustrates some operations of AP with an example. For this example, $\mathcal{J} = \{a_0, a_1, a_2, a_3\}$ and $\mathcal{C} = \{a_4, a_5, a_6, a_7\}$, and PTFAs are represented as $a_7 \ldots a_0$, as seen below each state. Assume that $En(b) = \{a_0, a_2, a_3, a_4\}$, $En(c) = \{a_0, a_1, a_7\}$, and $En(d) = \{a_0, a_1, a_3, a_5\}$. When $b$ is expanded, the PTFA on the arc is passed to state $e$ which changes the PTFA for $e$ and requires $e$ to be expanded. Subsequently, $c$ is expanded and the PTFA for $e$ is again updated and another $e$ expansion is needed to communication the new PTFA to successors. Finally, when $d$ is expanded the PTFA sent to $e$ contains no new actions, so $e$ does not need another expansion.

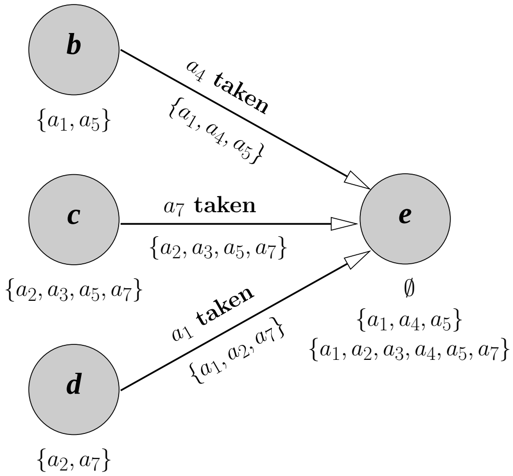


Figure 6.2: Example of PTFA updates as states are expanded

*State Deletion Phase (SD)*:
This phase appears from lines 32 to 37. Any states that enabled a fair action $a$ but with the corresponding tag bit cleared cannot be part of a FSCC and are removed from *MaybeFair*.

Soundness for the algorithm was described at the beginning of this section. To see that the algorithm terminates, we first note that the while-loop at lines 13–31 must terminate because the flag bits in *ptfa* are strictly increasing with successive iterations of the loop. The while-loop at lines 10–38 terminates because the loop body adds no new elements to *MaybeFair*. Thus the number of elements in *MaybeFair* strictly decreases with successive executions of the loop body.

### 6.3.1 Worst-Case Time Complexity for OWCTY

Here we present worst-case time-complexity bounds for the One-Way-Catch-Them-Young algorithm. The analysis is reasonably straightforward, but we are not aware of previously published bounds for the algorithm. We include the analysis here for completeness and to show that the run-times for OWCTY in practice are *much* smaller than the worst-case values.

First, consider the problem of finding all states reachable from some set of initial states, $V_0$. Let $V_R$ denote the set of reachable states (graph vertices), $n = |V_R|$, and $E_R$ the reachable transitions (graph edges). An explicit-state model checker must visit all edges, which provides an $\Omega(|E_R|)$ time bound for a sequential algorithm. Standard graph-traversal algorithms achieve this bound. Thus, the time complexity for finding the set of reachable states is $\Theta(|E_R|)$. For a dense graph, $|E_R| \in \Theta(n^2)$, and the time complexity for reachability is $\mathcal{O}(n^2)$. In practice, reachability graphs are often quite sparse, with an average in-degree (or out-degree) per state less than 10. If we assume $|E_R| \in \Theta(dn)$ where $d$ is the average in- (or out-) degree for a state, then the time complexity for such sparse models is $O(dn)$.

Now consider OWCTY with $m$ fairness constraints. The worst-case would be if each round eliminated one node, there were $\Omega(n)$ such rounds, and each round required $m$ passes to propagate fairness bits to $\Omega(n)$ vertices over $\Omega(n^2)$ edges (again, a dense reachable state graph). We now describe such a graph. Typically, the number of fairness constraints will be much smaller than the number of reachable stages, so we assume $m \ll n$. Let $n_X$ and $n_Z$ be $\Theta(n)$, such that $n_X + (m+1) + n_Z = n$. The vertices are partitioned into three sets as described below:

$$\begin{aligned}
X &= \{x_i \mid 0 \leq i < n_X\} \\
Y &= \{y_i \mid 0 \leq i \leq m\} \\
Z &= \{z_i \mid 0 \leq i < n_Z\} \\
V_R &= X \cup Y \cup Z \\
E_{X1} &= \{(x_i, x_i) \mid 0 \leq i < n_X\} \\
E_{X2} &= \{(x_i, x_{i+1}) \mid 0 \leq i < n_X - 1\} \\
E_X &= E_{X1} \cup E_{X2} \cup \{(x_{n_X-1}, y_0)\} \\
E_{Y1} &= \{(y_i, y_{i+1}) \mid 0 \leq i < m\} \\
E_{Y2} &= \{(y_m, z_i) \mid 0 \leq i < n_Z\} \\
E_Y &= E_{Y1} \cup E_{Y2} \\
E_Z &= \{(z_i, z_j) \mid 0 \leq i, j < |Z|\} \\
E_R &= E_X \cup E_Y \cup E_Z
\end{aligned}$$

The graph vertices are $V_R$; transitions are $E_R$; and the initial states are $\{x_0\} \cup Y$. There are $m$ strongly fair actions $a_0, ..., a_{m-1}$ with

$$\begin{aligned}
a_j = {} & \{(x_i, x_i) \mid x_i \in X, i + 1 \ (\mathrm{mod} m) = j\} \cup \\
& \{(x_i, x_{i+1} \mid 0 \leq i \leq n_X - 1, i \ (\mathrm{mod} m) = j\} \cup \\
& \{(y_j, y_{j+1})\}
\end{aligned}$$

Number iterations of the while-loop at lines 10–38 of Algorithm 6.1 starting at 0. In round 0, there is no path into $x_0$ satisfying the fairness constraint for $a_0$, so $x_0$ is deleted. In round 1, there is no path into $x_1$ satisfying the fairness constraint for $a_1$, so

$x_1$ is deleted. For round $0 \leq i < n_X$, vertex $x_i$ is deleted. Thus, OWCTY requires $\Omega(n)$ rounds for this example. Now, suppose that in each round the initial states are processed in order according to $y_m, y_{m-1}, ..., y_0, x_0$. In each round, each $z \in Z$ will be expanded $m$ times, and all outgoing edges from $z$ will be explored. Thus, the total time for each round is $\Omega(m|E_Z|) = \Omega(mn^2)$. There are at least $|X| \in \Omega(n)$ rounds, and the total runtime is $\Omega(mn^3)$. It is straightforward to show an upper-bound of $O(mn^3)$. Thus, the worst-case runtime for OWCTY with dense reachability graphs is $\Theta(mn^3)$.

Typical models do not have the dense reachability graphs described above. For example, if a Mur$\varphi$ model had $d$ rules, then each state has at most $d$ successors. Typically, $d \ll n$. If the reachability graph is sparse with maximum out-degree $d$, we can partition the vertices of $Z$ into $\Theta(n/d)$ separate cliques of $d$ states each, reserving $\Theta(n/d)$ vertices to build a fan-out tree from vertex $y_{m-1}$ to the $Z$ vertices. This shows a worst-case runtime in $\Theta(mdn^2)$.

Comparing these bounds with those for reachability, we find that OWCTY has a *worst-case* run-time that is $\Theta(mn)$ times greater than that of reachability alone. If the run-times for real problems resembled these bounds, then OWCTY would be unusable. As seen in Table 6.1, the actual number of rounds of the algorithm appears to be quite small for realistic examples. For our examples, at most 23 rounds were needed, and the "average" state was visited even fewer times. This enormous gap between the worst-case performance and the observed performance makes the OWCTY algorithm useful in practice.

## 6.4 Distributed Implementation

The distributed version of this algorithm starts with a Stern-Dill style reachability computation that identifies all *p*- and *pending*-states. Each worker process stores its *p*-states on disk, and *pending*-states are marked with tag bits in the hash-table. Initially, the PTFA for *pending*-states are set to all fair actions, $\mathcal{J} \cup \mathcal{C}$. The distributed algorithm then performs rounds that correspond to those of the sequential version, Algorithm 6.1. As described in more detail below, each round propagates PTFA tags according to the next state relation until a fix point is reached. At the boundary between rounds, states are identified whose PTFAs do not satisfy the fairness constraints for the state. Such states cannot be part of an FSCC and are marked as “dead” (*i.e.*, removed from *MaybeFair*). The number of live, *MaybeFair* states is non-increasing. The algorithm terminates when this number no-longer decreases. If at this point, all *MaybeFair* states have been eliminated, then the response property is satisfied. Otherwise, a counter-example is generated. The remainder of this section describes this algorithm in more detail.

**Algorithm 6.2** Root Process

```
function RootStart($I, p, q$)
    ▷ Tags for initial states
    for all $s \in I$ do
        SendState($(s, \emptyset)$)
    end for
    $CurMaybeFairCount \leftarrow$ Tally(nstates)
    $PrevMaybeFairCount \leftarrow 0$
    while $CurMaybeFairCount \neq PrevMaybeFairCount$ do
        Broadcast(doRound)
        $PrevMaybeFairCount \leftarrow CurMaybeFairCount$
        $CurMaybeFairCount \leftarrow$ Tally(nstates)
    end while
    Broadcast(stop)
    if $CurMaybeFairCount > 0$ then
        return GenerateCounterexampleTrace(...)
    else
        return verified
    end if
end function
```

Algorithm 6.2 shows pseudo-code for the root process. It initiates the initial reachability computation to identify *p*- and *pending*-states, by sending each initial state to their owners via SendState. It then initiates rounds of propagating PTFA tags and eliminating *pending*-states until no further states can be eliminated. The termination detection algorithm from the original Stern and Dill approach is used to identify the end of each round and compute the total number of *MaybeFair*-states. Function Tally sums the *MaybeFair*-states owned by each worker. This provides a barrier separating the computations of successive rounds. After the final round, the root process notifies the workers that the computation is complete and reports either that the response property has been verified or provides a counter-example.

Algorithm 6.3 shows pseudo-code for the worker processes. Like the reachability computation, each worker has two main activities: receiving incoming states and checking if they have been "seen" previously, and expanding states to send their successors to their owners. Algorithm 6.3 augments each of these activities to maintain the tags for PTFA. First, the set of owned and reachable $p$-states is found and stored via ComputePStates. This is similar to Stern-Dill reachability. Then, at the beginning of each round, each process checks its subset of the $p$-states to determine which ones satisfied their associated fairness constraints in the previous round. Those that do not are marked as dead. All $p$-states are added to the work-queue, *ToExpand*, even if they are dead to ensure that their successors are examined in this round. When a state is received, the algorithm first checks to see if this is the first time the state has been seen for the current round. If so, the state's PTFA is checked to see if the state should be marked as dead, and all states are entered into *ToExpand* the first time they are visited in each round. If the state has been seen before, then if the new PTFA indicates incoming paths for fairness constraints that haven't already been satisfied, these constraints are added to the state's PTFA, and the state is enqueued in *ToExpand* to propagate this information to its successors. Notice that each state may be expanded multiple times in a given round; at most $m$ times.

When a worker removes a state from its work queue, *ToExpand*, it computes all successor states as in the original reachability algorithm. Because the incoming paths to this state are prefixes of incoming paths for its successors, the PTFA of the successor must contain the PTFA for this state. Furthermore, if the transition to the new state correspond to a fair action, then this action is added to the PTFA. These updates are made to the

**Algorithm 6.3** Worker Process

```
function WORKER($\mathcal{S}, I, T, \mathcal{J}, \mathcal{C}, p, q, rootPid$)
    PS ← COMPUTEPSTATES($\mathcal{S}, I, T, \mathcal{J}, \mathcal{C}, p, q$)    ▷ Global queue for $p$-states
    $RoundCount \leftarrow 0$
    while true do
        case RECEIVE() of    ▷ Blocking receive
            doRound → ok
            stop → break while loop
        end case
        $RoundCount \leftarrow RoundCount + 1$
        for all $s \in$ PS do
            WQ ← INITSTATEFORROUND($s, \emptyset, RoundCount$)
        end for
        while round not terminated do    ▷ Stern and Dill's termination alg
            while ($s, thisPTFA$) ← RECEIVE() do    ▷ Nonblocking receive
                $T \leftarrow$ HT.GETTAGS($s$)
                if $T.round \neq RoundCount$ then
                    INITSTATEFORROUND($s, thisPTFA, RoundCount$)
                else if $\neg T.dead \wedge (thisPTFA \nsubseteq T.PTFA)$ then
                    $T.PTFA \leftarrow T.PTFA \cup thisPTFA$
                    WQ.INSERT(($s, T$))
                    HT.UPDATETAGS($s, T$)
                end if
            end while
            EXPANDANDSEND($\mathcal{J}, \mathcal{C}$)    ▷ See Algorithm 6.4
        end while
        send (nstates, $MyMaybeFairCount$) to $rootPid$
    end while
end function

function INITSTATEFORROUND($S, thisPTFA, RoundCount$)
    $T \leftarrow$ HT.GETTAGS($S$)
    if ENABLED($S$) $\nsubseteq T.PTFA$ then
        $T.dead \leftarrow$ true
        $thisPTFA \leftarrow \emptyset$
    end if
    $T.round \leftarrow RoundCount$
    $T.PTFA \leftarrow thisPTFA$
    WQ.INSERT(($S, T$))
    HT.UPDATETAGS($S, T$)
end function
```

**Algorithm 6.4** Dequeues a **WQ** state and sends next states with tags to their owners.

```
function ExpandAndSend($\mathcal{J}, \mathcal{C}$)
    if IsEmpty(WQ) then
        return done
    end if
    $(X, Tags) \leftarrow$ Dequeue(WQ)
    $NextStates \leftarrow$ ComputeSuccessors($X$)
    if $Tags.dead$ then
        for all $s' \in NextStates$ do
            SendState($(s', \emptyset)$)
        end for
        return
    end if
    $PTFA \leftarrow Tags.PTFA$
    $PTFA \leftarrow PTFA \cup (\mathcal{J} -$ Enabled($X$)$)$
    for all $s' \in NextStates$ do
        $ActionTaken \leftarrow$ whatActionTaken($X, s'$)
        ▷ Successor PTFA is current state PTFA with the fair action taken
        if $ActionTaken \in \mathcal{NF}$ then
            $NextPTFA \leftarrow PTFA$
        else
            $NextPTFA \leftarrow PTFA \cup ActionTaken$
        end if
        SendState($(s', NextPTFA)$)
    end for
    return
end function
```

PTFA for the successor, and the successor with this PTFA set is sent to the successor's owner.

Every operation either marks a state as dead or adds a fair action to some state's PTFA. Thus, the activities for updating fairness information eventually reach a fixpoint and the round terminates. Many optimizations are possible to improve the performance of this algorithm. These are described in the next section.

## 6.5 Optimizations

Early experiments with a prototype implementation revealed several opportunities to improve performance. We aim to address the average number of state expansions during a phase, the number of states visited during a phase, the number of rounds. A key observation is that for many examples, the number of states in the *pending* set decreases rapidly with successive rounds. Thus, it is important to avoid touching "dead" states so that the work done in later rounds decreases with the smaller size of *pending*. This also means that typically most of the time is spent in the initial reachability computation and the first two or three rounds of the liveness computation. Thus, optimizations should focus on these early rounds. Furthermore, the same state can be updated several times during a single round. Consolidating these updates was simple and led to significant performance gains. The remainder of this section presents three methods of reducing each of these metrics in turn. In addition, various optimizations are inherited from PReach's state space exploration technique. Namely, load balancing of states offers modest speedups even in a homogeneous network of machines. Batching of states into messages containing hundreds or thousands is also of benefit. See Chapter 3 for details regarding these optimizations.

### 6.5.1 Saved Expansions

The description in the algorithms and implementations presented so far have states paired with their tags, including PTFA, when enqueued to the **WQ**. When the **WQ** grows large, state $s$ may arrive tagged with PTFA $b_2$ while the same state is waiting for expansion in the **WQ** while paired with PTFA $b_1$, which matches the PTFA at the **HT** entry for $s$. When $b_2$ has at least one bit set that $b_1$ does not, $s$ is enqueued for expansion in **WQ** paired with PTFA $b_1 \cup b_2$. This renders the earlier **WQ** entry of $(s, b_1)$ redundant and unnecessary.

To avoid this scenario, the **HT** is used to maintain PTFA information, and **WQ** entries do not contain a PTFA. When a state $s$ is enqueued, a new **HT** tag bit $InWQ$ is set; when $s$ is dequeued, $InWQ$ is cleared and the current **HT** value for PTFA is used when computing the PTFA for $s$'s successors. If state $s$ with PTFA $b_2$ arrives when the **HT** entry has $InWQ$ set, then **HT** PTFA $b_{\mathbf{HT}}$ is set to $b_{\mathbf{HT}} \cup b_2$ and the just-arrived state $s$ is discarded. This approach reduces the number of state expansions at the cost of an additional bit in **HT** per state, and one additional **HT** lookup.

### 6.5.2 Dynamic Kernel

The algorithm implementation above uses the reachable $p$-states as the *kernel*, defined as follows.

**Definition 3.** *Given a FTS, $K \subseteq \mathcal{S}$ is a* kernel *for $A \subseteq \mathcal{S}$ if $A$ is a subset of the reachable states from $K$ in the digraph $(\mathcal{S}, T)$.*

Note that the set of initial states $I$ is a kernel for any subset of the reachable states. In the code presented in Section 6.4, we used the reachable $p$-states $K_p$ as a kernel for *MaybeFair* to initiate each phase because $K_p$ is a kernel for every subset of *pending*. Our experiments showed that for typical examples, the number of states in *MaybeFair* drops rapidly with each SD phase. The expansion of such deleted states can be avoided by modifying $K$ after each SD phase, using an extra **HT** tag bit *InK* and additional disk space.

During the initial phase, only the $p$-states have *InK* set to true, and these states are saved to disk in the kernel-queue. When a state $s$ is removed from *MaybeFair* during SD that has *InK* set, this flag is cleared. When a process receives state $s'$ tagged with mode `delete_pred` (signaling that a predecessor of $s'$ has just been removed from *MaybeFair*), then if $s'$ has its *InK* flag cleared, it is set to true and $s'$ is added to the kernel-queue. Finally, at the start of an AP phase the kernel-queue is copied to the **WQ** to serve as the set of initial states, but any state encountered that has its *InK* flag cleared is ignored and removed from the kernel-queue.

While this approach does not necessarily maintain the smallest possible kernel for *MaybeFair*, its simple implementation and low overhead lead to large performance gains.

### 6.5.3 Deletion by Predecessor Counting

There are performance advantages when storing the number of predecessors each state has in $\langle \mathit{MaybeFair} \rangle$. Under the assumption of stuttering and ensuring the safety property that every state $s \in \mathit{pending}$ has $|\mathcal{J}_s \cup \mathcal{C}_s| \geq 1$, any state with 0 predecessors in $\langle \mathit{MaybeFair} \rangle$ will be deleted from *MaybeFair* in the next SD phase. However, storing the number of predecessors in **HT** allows detection of this case in order to preemptively remove such states. We choose to add 8 bits to the **HT** tags to store the predecessor count. This additional bookkeeping complicates Algorithms 6.3 and 6.4 somewhat (details omitted). In particular, a state may be expanded more than once during an SD. This occurs when the first time a state is visited the condition on line 32 of Algorithm 6.3 holds, but subsequently all of its predecessors are deleted. However, this turns out to be a rare occurrence in the benchmarks, and this strategy can reduce the number of phases. Note that the impact of this optimization is omitted from the Results section as it was inherent to our early implementation versions.

## 6.6 Results

We ran PReach on a variety of combinations of Mur$\varphi$ models with all optimizations of section 6.5 enabled, summarized in Table 6.1. For each, we chose a suitable response property such as “requests for exclusive access to a cache line are eventually granted”, or “processes waiting to enter the critical section will eventually do so”. The Mur$\varphi$ models used are the German cache coherence protocol, the Peterson mutual exclusion algorithm, the MCS lock mutual exclusion algorithm, a snoopy protocol used as a benchmark in previous verification work [39] and an Intel proprietary protocol. `GermanX` is the German model with `X` caches; `petersonY` is Peterson’s algorithm with `Y` processes and `mcslock5` is the MCS Lock algorithm with 5 processes; `snoop2` is the snoopy protocol with 2 L1 caches and 2 clusters. Models `saw`, `gbn` and `swp` are various sliding window communication protocols, with the response property that the sender can always eventually accept new data to transmit. All models and the PReach code is provided online [24]. Each Mur$\varphi$ “rule” (a.k.a. guarded command) is considered a separate action; we attached suitable fairness assumptions specific to the model. The network of machines used for experiments are as follows:

- UBC cluster: 40 PReach processes on a homogeneous cluster of 20 Intel Core i7-2600K at 3.40 GHz with 8 GB of memory (non-`intel_*` models).
- Intel cluster: 16 PReach processes on a heterogeneous network of contemporary Intel® Xeon® machines, each with at least 8 GB of memory (`intel_*` models).

Not included in the table, but worth noting, is an Intel proprietary sliding window protocol model. With over 450 million states and tens of fairness (both strong and weak), we were able to verify response in about 5 and a half hours using 32 cores.

| model | runtime | states | $p$-states | *pending*-states | $q$-states | rounds | exp/state | no `-ko` | no `-se` | no opt. |
|---|---|---|---|---|---|---|---|---|---|---|
| `German5_sf` | 189 | 15,836,445 | 3,699,486 | 4,858,596 | 5,103 | 1 | 3.48 | 0.98 | 2.42 | 2.86 |
| `German6_sf` | 4,253 | 316,542,087 | 74,465,244 | 95,266,520 | 18,225 | 1 | 3.33 | 1.01 | 3.30 | 3.52 |
| `peterson6_wf` | 820 | 13,817,679 | 2,947,800 | 12,111,713 | 45,209 | 14 | 12.91 | 1.65 | 1.30 | 1.95 |
| `peterson6_sf` | 423 | 13,817,679 | 2,947,800 | 12,111,713 | 45,209 | 5 | 9.03 | 1.36 | 1.73 | 2.12 |
| `peterson7_wf` | 26,957 | 380,268,668 | 79,029,594 | 340,549,743 | 775,138 | 17 | 14.19 | 1.65 | 1.66 | 2.16 |
| `peterson7_sf` | 14,613 | 380,268,668 | 79,029,594 | 340,549,743 | 775,138 | 6 | 10.11 | 1.27 | 2.26 | - |
| `mcslock5_wf` | 1415 | 59,318,541 | 27,785,789 | 51,474,427 | 2,780,517 | 3 | 5.09 | 1.17 | 1.10 | 1.25 |
| `snoop2_sf` | 160 | 2,648,763 | 670,689 | 1,313,100 | 1,335,663 | 3 | 12.71 | 1.07 | 4.57 | 5.00 |
| `saw20_sf` | 323 | 314,183 | 309,140 | 309,140 | 5,043 | 23 | 44.06 | 1.04 | 1.09 | 1.15 |
| `gbn3_2_sf` | 369 | 12,753,395 | 7,859,200 | 7,859,200 | 4894195 | 6 | 6.44 | 1.60 | 1.95 | 2.56 |
| `swp4_2_sf` | 503 | 18,595,425 | 11,715,440 | 11,715,440 | 6,879,985 | 6 | 6.58 | 1.59 | 1.63 | 2.22 |
| `intel_small_sf` | 285 | 476,778 | 268,078 | 268,078 | 164,057 | 4 | 6.36 | - | - | - |
| `intel_med_sf` | 1,015 | 2,696,059 | 1,944,360 | 1,944,360 | 635,672 | 4 | 8.59 | - | - | - |
| `intel_big_sf` | 13,872 | 51,791,350 | 29,899,694 | 29,899,694 | 19,855,989 | 8 | 11.92 | - | - | - |

Table 6.1: PReach-Resp benchmark results. Column “runtime” is given in seconds; “exp/state” is the average number of times each *pending*-state was expanded. Model `peterson6_sf` is `peterson6` with all actions strongly fair, as opposed to `peterson6_wf` where some rules were weakly fair and the rest as not fair (for example, the rule that initiates the move from the noncritical section to requesting to enter the critical section needs no fairness assumption). These two models have the same number of states of each type but perform a different number of expansions, and illustrate the benefit of only using more fairness than required for the response property to hold. All other models require strong fairness.

The rightmost three columns of Table 6.1 show the slowdown when benchmarks are run without the kernel optimization, without the saved expansions optimization and without either, respectively. The kernel optimization is of most benefit when the number of rounds is large[3]. In particular, it is of no benefit for those benchmarks that only require a single round, as the kernel states are only used during subsequent rounds. The saved expansion optimization offers large performance gains in many cases. Typically, only 5 to 10% of the total state expansions are explicitly avoided by detecting that a just-received state state is present in the **WQ**. However, avoiding these redundant expansions can in turn save many expansions of successor states which in turn saves expansions of states that are two transitions away. This cascading effect decreases the total number of expansions by a significant factor.

A few modifications were required when checking the `snoop` protocol. This model was created to represent a cache-coherence protocol in a realistic processor. The protocol appears to have been designed with an emphasis on safety, and liveness does not appear to have been primary concern. For example, requests for cache lines are clearly not responsive as they may be *negatively acknowledged* (Nackd) an arbitrary number of times. To avoid this, we changed the protocol so that Nacks of this type are simply ignored, and the request persists. This turned out to also not be responsive, although less obviously so – the counterexample trace included 72 transitions. Therefore, not all of the pending states were deleted. Figure 6.5 shows that about 65% of the the *pending*-states remained in the *MaybeFair* set when the algorithm terminated.

Figures 6.3 through 6.5 show the length of the work queues and the size of *MaybeFair* for each process during model checking of some of the benchmarks. The work queue plots clearly demonstrate the effective load balancing used by PReach. The difference between the shortest and longest work queue is at most a factor of 5. Notice that the **WQ** plots clearly indicate the start and end of phases. Recall that the start of each phase, except for

---

[3] One exception is `saw_20_sf` where a large proportion of the runtime is spent coordinating threads between rounds.

the first one, initializes the work queue to be the owned $p$-states for that process, causing the rapid uptick at the start of each phase. Also recall that even numbered phases are SD – comparing the two plots for the same model shows the decline of *MaybeFair*-states during SD and plateaus during AP, as well as the initial discovery of the pending states during the first phase.

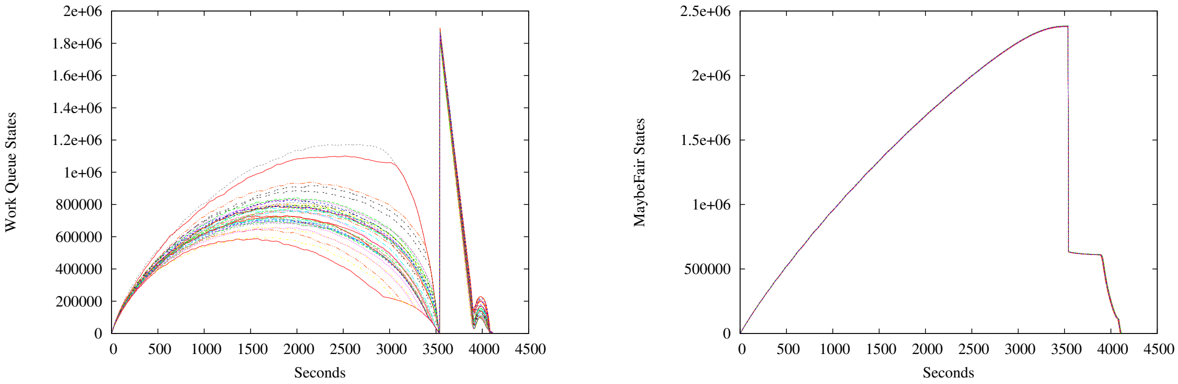


Figure 6.3: Response property model checking plots for `German6`: **WQ** length (left) and $|MaybeFair|$ (right) for each process

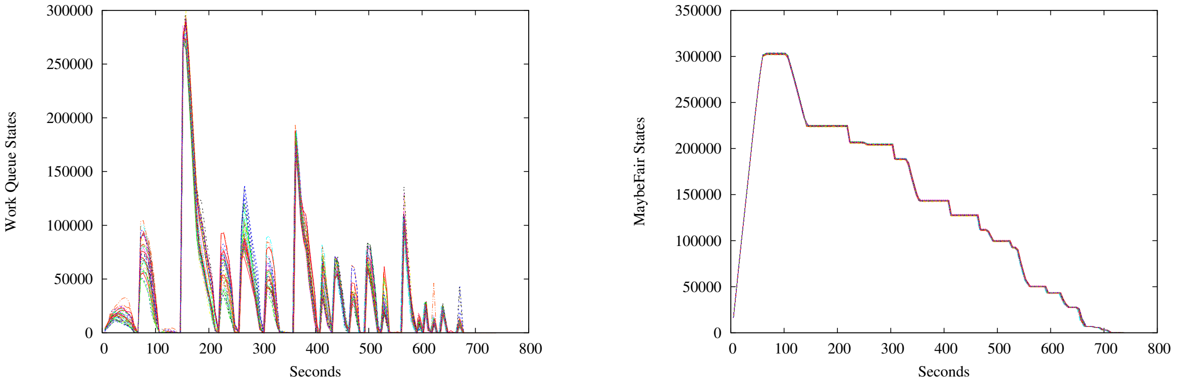


Figure 6.4: Response property model checking plots for `peterson6_wf`: **WQ** length (left) and $|MaybeFair|$ (right) for each process

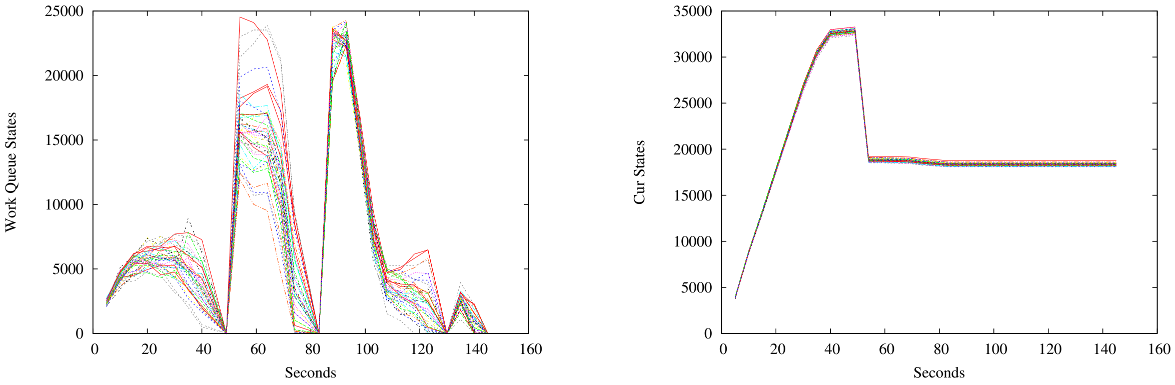


Figure 6.5: Response property model checking plots for `snoop2`: **WQ** length (left) and $|MaybeFair|$ (right) for each process.

## 6.7 Comparison with DiVinE

DiVinE is a parallel and distributed LTL model checker that is the closest tool to ours [7]. DiVinE constructs a product Büchi automaton to check liveness properties; thus, we expect DiVinE's space requirement to grow as the product of the number of states in the system model and the number in the system automaton. Applying DiVinE to the examples from Section 6.6, we observed that it ran out of memory for all examples except for those with no or a small number of strong fairness constraints. DiVinE provides a mode for models where all transitions are weakly fair. Using this feature, DiVinE performed well for the Peterson example for which weak fairness constraints are sufficient to ensure responsiveness. Note that the error found in the Mur$\varphi$ model for Peterson is not detected when all transitions are assumed to be weakly fair. The error arises because there are *no* fairness assumptions for the action where clients make requests, *i.e.* clients are not *required* to make requests. Furthermore, many problems require strong fairness; for example cache coherence protocols often include states where taking one action disables another. We found that for an encoding of the German protocol with 4 caches, the reachable state space of DiVinE's product automaton doubled with *each* additional fair action included. For only 6 fair rules, DiVinE on a multicore machine took 17 minutes to construct the system automata, 13 minutes to perform the model checking task and used over 16 GB of main memory. In our experiments, adding one more fair rule exhausted the main memory of our 32 GB machine and rendered the computation time infeasible. The protocol in question has on the order of 20 strongly fair actions.

Our algorithm has a worst-case time complexity that is at least $O(mn^3)$ where $n$ is the number of reachable states and $m$ is the number of fair actions. Section 6.3.1 gives an example where the transition relation has $O(n^2)$ edges, and for which Algorithm 6.1 removes one state per iteration of the outer while-loop. In practice, we observe that the transition relation is sparse and Algorithm 6.1 converges in far fewer than $n$ rounds – the most extreme case in Table 6.1 has 23 rounds. If $|E|$ is the number of reachable transitions, then the number of computed sucessor states is $\mathcal{O}(r(n+|E|))$. The *worst-case* time complexity of DiVinE is better, $\mathcal{O}(n2^{|\phi|})$ — DiVinE replaces a factor of the system model size with the number of states for the checking Büchi automaton. However, our experiments show that the actual time and memory requirements for DiVinE's algorithm are fairly close to what one would expect from the worst-case bounds, while our approach, in practice, scales much more efficiently. We see this gap between worst-case and actual performance as a promising area for further investigation.

## 6.8 Conclusions and Future Work

We have extended the PREACH explicit-state, distributed model-checking tool to support verification of response properties under both strong and weak fairness of actions. Our approach uses multiple rounds of reachability computation to implement a variation of the OWCTY algorithm. For a model with $n$ states, $m$ fairness constraints, OWCTY could expand states $O(nm)$ times on average. This would be prohibitively expensive. Our implementation shows that for practical examples, the number of rounds is small — typically less than 10, with a maximum of 23. Thus, OWCTY appears to provide a practical approach to checking response properties for real-world problems. For these examples, liveness checking is slower than safety checking, but not prohibitively so.

Implementing our algorithm on top of the PReach distributed model checker allows it to exploit the aggregate memory of large compute clusters. This enabled verification of response properties for a sliding-window protocol with over 450 million states in about $5\frac{1}{2}$ hours.

We compared our approach with a tool that uses the standard product-automaton formulation, with one automaton for the system model, and the other for the LTL liveness property. As predicted by the worst-case analysis, we observed that the size of the property automaton grew exponentially with the number of fairness constraints. The product-automaton approach was significantly faster than PReach for the problems that it could complete. However, it ran out of memory for all but the smallest examples.

This approach can be generalized in a number of directions. One is to handle other simple liveness properties such as *reactivity*, expressed in LTL as $\Box\Diamond p \vee \Diamond\Box q$, where $p$ and $q$ are state predicates. This says that either $p$ holds infinitely often, or eventually $q$ holds forever. Reactivity can express, for example, a strong-fairness condition. Futhermore, a conjunction of reactivity properties is as expressive as full LTL [84, 88], and so would enable model checking of a much wider class of properties. We hope to combine these model checking methods with the decompositional inference rules of Manna and Pnueli [88, 89]. Such decompositions establish that a response property is implied by a handful of safety properties and "smaller" response properties, *i.e.* depending on a smaller fraction of the state space. Adapting our algorithm to verify multiple such response properties in the same model checking run would leverage human insight to improve scalability to enable verification of more detailed models with more expressive specifications.

# Chapter 7

# Conclusions

This thesis work successfully demonstrates that explicit-state model checking can be leveraged for *practical* verification of liveness of hardware protocol models. With PReach, we were able to scale safety verification to new levels of model sizes. Our two extentions of PReach to liveness checking utilized this achievement to verify more interesting properties of like-sized models. All three of these tools have been applied to real, industrial examples. Our approach for parameterized proofs of deadlock freedom was not applied to an industrial example; this could require significant human effort. For the simple examples we considered, German and FLASH, very little additional effort was required to show deadlock-freedom given the non-interference lemmas inherited from the CMP method for safety verification. This is in stark contrast to previous approaches for parameterized liveness verification, which are quite technical and likely require an expert user. We expect that we would observe a similar outcome if our approach were applied to real, industrial problems following the CMP method, but this is a topic for future work.

The PReach model checker operates on Mur$\varphi$ models, a widely adopted format for hardware protocols. Our liveness extensions have the benefit of *straightforward* use and interpretation to verification engineers and hardware designers alike. PReach-DF requires the user to partition actions (*i.e.*, Mur$\varphi$ rules) into those that progress some transaction and those that do not, and give a predicate describing "no transactions in flight". PReach-Resp calls for the specification of weakly- and strongly-fair actions, notions that are often intuitively clear to architects. While the distributed scalability of PReach was not needed for our parameterized deadlock freedom work, we do not view parameterized methods and scalable model checking tools as mutually exclusive. Indeed, during a recent internship with Intel, a parameterized abstraction model of an industrial sliding window protocol was on the order of hundreds of millions of states — large enough that distributed model checking was necessary for this abstraction.

The contributions of this thesis touch on different points in the trade-off between effort and strength of the result in verification of protocols. PReach can automatically check safety properties in Mur$\varphi$ models, and can scale to tens of billions of states. With some human guidance in the form of helpful rules and a quiescent state predicate, PReach-DF can automatically check a DF property of like-sized models at the cost of a small time overhead (about 30%). If the user is willing to provide adequate fairness assumptions, PReach-Resp will verify the much stronger response properties, but with a memory overhead of roughly a factor of 5 and a time overhead that is typically a factor of 30 when compared with safety verification. where $n$ is the number of reachable states and $m$ the number of fair actions. Finally, our contributions in parameterized DF require moderate human effort, but we expect it is proportional to the effort needed to apply the CMP method for parameterized safety, thus establishing a parameterized liveness-like result.

Distributed explicit-state model checking allows one to exploit the aggregate memory of many machines. PReach provides a platform for developing various practical DEMC based methods. We have used this to address some liveness properties that were chosen based on their usefulness and significance in practice. These are deadlock-freedom, parameterized deadlock-freedom and response properties. This thesis enables the verification of a new class of properties for real, industrial problems — especially for hardware protocols.

## 7.1 Contributions Recap

Restating my thesis statement Chapter 1:

> *This thesis develops and demonstrates tractable, practical and scalable distributed explicit-state model checking methods for establishing liveness properties of practical importance for large-scale models of hardware protocols.*

I have demonstrated this with the following, concrete contributions (also listed in Chapter 1:

1. PREACH is an industrial strength distributed explicit-state model checker that has verified properties of larger Mur$\varphi$ models than any other published explicit-state methods by distributing the computation across hundreds of machines. The PREACH software architecture achieves efficiency by building on the C++ code base from the Mur$\varphi$ model checker and simplicity by using a layer of Erlang code to implement the communication aspects. We have found this approach to be robust and extensible. PREACH has been used extensively by our collaborators at Intel. Detailed in Chapter 3.

2. Chapter 4 presented two algorithms for verifying *deadlock-freedom* using DEMC. This property ensures that from all reachable states, the system has a path to some *quiescent* state, which is a state with no pending transactions. DF has the advantage that it is easily understood by computer architects and hardware protocol designers. Our algorithms are computationally efficient, typically requiring a factor of 1.2 to 2 of the time taken by safety property checking. We implemented these algorithms by extending PREACH, and found an error in the Peterson model included in the Mur$\varphi$ distribution, which had persisted for about twenty years.

3. Chapter 5 extended our DF checks to include *symmetric, parameterized systems.* We showed how a *mixed abstraction* can be used, and sound techniques for weak-

ening the underapproximate transitions. This complements the CMP method [40] for strengthening overapproximate abstractions in an effort to prove parameterized safety properties. In practice, our approach appears to require only small additional human effort once the CMP method has been used.

4. PReach-Resp is our explicit-state approach to verifying response properties. These properties are much stronger than DF, as they show that all requests are eventually granted. Similar to DF, they have the advantage of being intuitive to the non-expert. In practice, response property verification requires *fairness assumptions* that ignore unrealistic counterexamples. We demonstrated that integrating fairness assumptions into the state space exploration of PReach produces a practical tool. In particular, our twist on the OWCTY algorithm [73, 38] takes only a modest constant number of expansions per state — far less than the theoretical worst-case. Detailed in Chapter 6.

## 7.2 Future Work

This work has opened the door for a number of future directions. The choice of Erlang as the chief programming language for PReach allows rapid development and testing of new approaches in the context of DEMC. While other tools have a codebase that would be vexing for a new researcher to digest, PReach and its derivatives require only a familiarity with Erlang, and the time to read 1000 to 1500 lines of code[1]. For example, other researchers have proposed heuristics for partial order reduction in DEMC [10, 78], but to the best of our knowledge, these methods have neither been implementation nor experimentally evaluated. Such experiments ought to be easy in PReach.

As performance has not been a primary concern, I have not spent very much time searching for bottlenecks. However, we suspect that as the number of threads assigned to a single multi-core machine exceeds 2 or 3, the disk IO becomes a limiting factor. This is

---

[1] Certain modifications may also require tweaks to the Mur$\varphi$ engine or the Mur$\varphi$ compiler, but these changes are typically local to a few C/C++ functions.

likely due to the fact that each worker reads and writes states from a different large disk file. In many of our DEMC algorithms, the worker process that expands a state need not be the state's owner, thus we are eager to experiment with a dedicated Erlang process to manage disk accesses and coalesce states into a single file. Another PReach limitation is crash recovery; if a single worker crashes for any reason (such as Mur$\varphi$ errors, running out of disk space, Erlang runtime errors) then all workers will die and model checking progress is lost. A recent master's thesis [71] considered using a database to store the Stern-Dill algorithm data structures, but it is too slow to be practical. We believe that snapshots of these data structures taken at infrequent intervals will offer minimal performance impact while providing a rollback point in case of a crash. Another idea in this space is to store states redundantly, say by assigning *multiple* owners to each state. This would allow crash recovery even if a snapshot for a given worker is lost.

The modular design of the Erlang communication layer and Mur$\varphi$ computation engines opens the door for PReach-like DEMC applied to other modelling languages. The Erlang code is agnostic to the specifics of a "state"; all queries regarding if a state adheres to a predicate or which states are successors are calls through an interface to the Mur$\varphi$ Engine. Indeed, during a recent Intel internship I was able to replace the Mur$\varphi$ Engine with a custom C++ based modelling language and use the Erlang layer for model checking. A potential drawback of using the Erlang runtime is its inherent distribution limits. I have not tested PReach using more than 200 workers and typically use about 50. It would be interesting to try a very large PReach experiment using hundreds of workers using WestGrid [112] or a similar large-scale cluster and see if and where things break down.

There are a handful of technical aspects of PReach-DF and PReach-Resp that should be addressed. Unlike the original PReach, which has active users and a group of maintainers, PReach-Resp and PReach-DF have been implemented as research prototypes and they have not been made as easily available to a wider community. First and foremost, the bitbucket online repositories for PReach, PReach-DF and PReach-Resp should be merged. The latter two were branches of PReach for research purposes, but clearly we shouldn't expect users to manage three different installations. PReach-Resp could easily be extended to enforce state-based (as opposed to action-based) fairness assumptions. Some users may have Mur$\varphi$ models that more naturally fit with this kind of fairness. Finally, the action-based fairness of PReach-Resp should be generalized to possibly range over an entire Mur$\varphi$ ruleset, rather than considering each rule as a distinct action. One case where these missing features cause problems occurs with rulesets whose parameters do not appear in the rule's guard, but only in the command. This is sometimes done to express update assignments of nondeterministically chosen values. Taking each rule as a different action often leads to excessively long counterexamples that are confusing to interpret and time consuming for the tool to generate.

There are a number of directions for longer term future work. The usefulness of model checking response properties with fairness would be expanded if we could also show refinement from a high level Mur$\varphi$ model to an RTL implementation. In particular, we would need to show that

1. the RTL *is* a refinement of the high level model, and
2. the RTL adheres to the fairness assumptions of the high level model

While there is a number of researchers that have considered the first problem [109, 29, 49], I am unaware of work that deals with the second problem. To expand on the kinds of LTL properties that PReach-Resp can handle, I am looking forward to considering *reactivity* properties, which are a generalization of response properties and are expressive enough that any LTL formula can be written as reactivity properties with conjunction and disjunction. Finally, the originally proposed idea of using Manna and Pnueli's inference rules [88, 89] for decomposing response properties into simpler ones and dispatching to PReach for model checking would increase our model checking performance through human insight.

Currently PReach-Resp does not support the checking of multiple response properties at once, which needs to be implemented before this approach would see performance gains.

We expect that checking multiple response properties within a single PReach-Resp run would be a straightforward feature to add, especially if we assume that the multiple response properties take a special form. Suppose the response properties to show are $r_0, ..., r_{n-1}$, where $r_i \equiv \Box(\phi_i \to \Diamond \bigvee_{j=i+1}^{n} \phi_j)$. If each $r_i$ holds, then it is trivial to show that $\Box(\phi_0 \to \Diamond\phi_n)$ also holds. The extra bookkeeping would require an additional $\lceil \log{(n-1)} \rceil$ bits per state, to tag each *pending*-state with an index value. This index for states in *MaybeFair* is the greatest $i < n$ such that some $\phi_i$-state lies on a predecessor path within $\langle$*MaybeFair*$\rangle$. This approach does not exploit the full generality afforded by Manna and Pnueli's inference rules, but I hypothesize it will reduce the runtime of PReach-Resp if the user is willing to provide the $\phi_i$ predicates.

As an example of a decomposed response property, consider German's simple cache coherence protocol, appearing in Appendix A. Suppose we aim to verify a response property (subject to fairness) where $p$ is " cache 1 has requested exclusive access", and $q$ is "cache 1 has been granted exclusive access". One way to decompose this property is to let $\phi_0 = p$, $\phi_2 = q$ and $\phi_1$ be "the request for exclusive access has been received by the directory". The response property $\Box(\phi_0 \to \Diamond\phi_1 \vee \phi_2)$ is nontrivial to verify, as the directory cannot accept the request for exclusive access from cache 1 until it is finished servicing other requests. Likewise, the response property $\Box(\phi_1 \to \Diamond\phi_2)$ is nontrivial as requests for exclusive access may involve invalidating the line in other caches. Splitting the original response property in this way results in a partitioned *pending* set, and only some of the fairness assumptions are relevant to each sub-response property. This ought to lead to fewer state expansions for our OWCTY adaptation, which is the chief contributor of runtime.

Our work in parameterized DF in symmetric protocols could be extended to show parameterized response properties. Consider a parameterized system that has fairness attached to parameterized actions. In the standard CMP-style overapproximate abstraction, fairness assumptions on concrete actions will soundly persist, but it is unsound to assume fairness on actions that undergo abstraction and have weakened guards. Model checking a response property on the abstract system will almost certainly fail because such abstract rules are not obligated to fire. However, I believe that similar permutation/model checking methods as established in this thesis may be applied to prove that abstract traces where a given abstract rule is persistently enabled *have no concretization*. Thus, the abstraction may be strengthened by attaching a fairness assumption to this abstract rule that avoids such executions. As with the parameterized DF work, this method could be applied after the standard CMP method has been used and strengthening steps already applied to the abstraction.

This thesis has shown that tractable verification of interesting liveness properties in large protocol models is possible. The proposed extensions above build upon this result and give different ways of widening the applicability and scalability further. While model checking of arbitrary LTL liveness properties is difficult, this thesis lends itself to techniques that are tailored to specific properties and fairness assumptions. Proceeding with future work with this idea in mind will lend itself to tractable verification of stronger results. In particular, using inference rules to decompose response properties will allow modest human insight to dramatically reduce the time needed for model checking and proving responsiveness.

# Bibliography

[1] K. Apt and D. Kozen. Limits for automatic verification of finite-state concurrent systems. *Information Processing Letters*, 15:307–309, 1986.

[2] Joe Armstrong. *Programming Erlang: software for a concurrent world.* Pragmatic Bookshelf, 2007.

[3] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, Katherine A. Yelick, Meetings Jim Demmel, William Plishker, John Shalf, Samuel Williams, and Katherine Yelick. The landscape of parallel computing research: A view from berkeley. Technical report, TECHNICAL REPORT, UC BERKELEY, 2006.

[4] J. Barnat, P. Bauch, L. Brim, and M. Češka. Employing Multiple CUDA Devices to Accelerate LTL Model Checking. In *16th International Conference on Parallel and Distributed Systems (ICPADS 2010)*, pages 259–266. IEEE Computer Society, 2010.

[5] J. Barnat, L. Brim, I. Cerna, P. Moravec, P. Rockai, and P. Simecek. DiVinE – a tool for distributed verification. In *Computer Aided Verification*, pages 278–281, 2006.

[6] J. Barnat, L. Brim, S. Edelkamp, D. Sulewski, and P. Šimeček. Can Flash Memory Help in Model Checking. In *Formal Methods for Industrial Critical Systems (FMICS 2008)*, volume 5596 of *LNCS*, pages 150–165. Springer-Verlag, 2008.

[7] J. Barnat, L. Brim, V. Havel, J. Havlíček, J. Kriho, M. Lenčo, P. Ročkai, Vladimír Štill, and J. Weiser. DiVinE 3.0 – An Explicit-State Model Checker for Multithreaded C & C++ Programs. In *Computer Aided Verification (CAV 2013)*, volume 8044 of *LNCS*, pages 863–868. Springer, 2013.

[8] J. Barnat, L. Brim, and P. Ročkai. A Time-Optimal On-the-Fly Parallel Algorithm for Model Checking of Weak LTL Properties. In *Formal Methods and Software Engineering (ICFEM 2009)*, volume 5885 of *LNCS*, pages 407–425. Springer, 2009.

[9] J. Barnat, L. Brim, and P. Ročkai. A Time-Optimal On-the-Fly Parallel Algorithm for Model Checking of Weak LTL Properties. In *Formal Methods and Software Engineering (ICFEM 2009)*, volume 5885 of *LNCS*, pages 407–425. Springer, 2009.

[10] J. Barnat, L. Brim, and P. Ročkai. Parallel Partial Order Reduction with Topological Sort Proviso. In *Software Engineering and Formal Methods (SEFM 2010)*, pages 222–231. IEEE Computer Society Press, 2010.

[11] J. Barnat, L. Brim, P. Simecek, and M. Weber. Revisiting resistance speeds up I/O-efficient LTL model checking. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 48–62. Springer, 2008.

[12] J. Barnat, L. Brim, M. Češka, and T. Lamr. CUDA accelerated LTL model checking. In *ICPADS*. IEEE, 2009.

[13] J. Barnat, L. Brim, M. Češka, and P. Ročkai. DiVinE: Parallel Distributed Model Checker (Tool paper). In *Parallel and Distributed Methods in Verification and High Performance Computational Systems Biology (HiBi/PDMC 2010)*, pages 4–7. IEEE, 2010.

[14] J. Barnat, J. Chaloupka, and J. Van De Pol. Distributed Algorithms for SCC Decomposition. *Journal of Logic and Computation*, 21(1):23–44, 2011.

[15] Jiri Barnat. *Distributed Memory LTL Model Checking.* PhD thesis, Masaryk University Brno, Faculty of Informatics, 2004.

[16] Jiří Barnat, Jan Havlíček, and Petr Ročkai. Distributed LTL Model Checking with Hash Compaction. *Electr. Notes Theor. Comput. Sci.*, 296:79–93, 2013.

[17] K. Baukus, S. Bensalem, Y. Lakhnech, and K. Stahl. Abstracting WS1S systems to verify parameterized networks. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, pages 188–203, 2000.

[18] K. Baukus, Y. Lakhnech, and K. Stahl. Parameterized verification of a cache coherence protocol: Safety and liveness. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 317–330, 2002.

[19] G. Behrmann. A performance study of distributed timed automata reachability analysis. *Electr. Notes Theor. Comput. Sci.*, 68(4), 2002.

[20] Alexander Bell, Er Bell, and Boudewijn R. Haverkort. Sequential and distributed model checking of petri net specifications. *STTT*, 7:43–60, 2002.

[21] B. Bingham. Murphi sources for this chapter. `http://www.cs.ubc.ca/~binghamb/cav2011.html`. Accessed: July 13, 2013.

[22] B. Bingham, J. Bingham, F. M. de Paula, J. Erickson, G. Singh, and M. Reitblatt. Industrial strength distributed explicit state model checking. In *Parallel and Distributed Model Checking*, 2010.

[23] B. Bingham, M. Greenstreet, and J. Bingham. Parameterized verification of deadlock freedom in symmetric cache coherence protocols. In *Formal Methods in Computer-Aided Design (FMCAD), 2011*, pages 186–195, Oct 2011.

[24] Brad Bingham. Preach response. `https://bitbucket.org/binghamb/preach-response`. Accessed: October 21, 2014.

[25] Brad Bingham, Jesse Bingham, and John Erickson. Preach-df online. `https://bitbucket.org/binghamb/preach-brads-fork`. Accessed: July 13, 2013.

[26] Brad Bingham, Jesse Bingham, John Erickson, and Mark Greenstreet. Distributed explicit state model checking of deadlock freedom. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification*, volume 8044 of *Lecture Notes in Computer Science*, pages 235–241. Springer Berlin Heidelberg, 2013.

[27] Brad Bingham and Mark Greenstreet. Response property checking via distributed state space exploration. *Formal Methods in Computer Aided Design (FMCAD)*, 2014.

[28] J. Bingham. Automatic non-interference lemmas for parameterized model checking. In *Formal Methods in Computer Aided Design (FMCAD)*, 2008.

[29] J. Bingham, J. Erickson, G. Singh, and F. Andersen. Industrial strength refinement checking. In *Formal Methods in Computer-Aided Design, 2009. FMCAD 2009*, pages 180 –183, nov. 2009.

[30] Jesse Bingham, John Erickson, Brad Bingham, and Flavio M. de Paula. Open-source PReach. `http://bitbucket.org/jderick/preach`. Accessed: July 14, 2011.

[31] Stefan Blom, Bert Lisser, Jaco Van De Pol, and Michael Weber. A database approach to distributed state-space generation. *Journal of Logic and Computation*, 21(1):45–62, 2011.

[32] Stefan Blom, Jaco van de Pol, and Michael Weber. LTSmin: Distributed and symbolic reachability. In *CAV'10*, pages 354–359, 2010.

[33] AaronR. Bradley. Sat-based model checking without unrolling. In Ranjit Jhala and David Schmidt, editors, *Verification, Model Checking, and Abstract Interpretation*, volume 6538 of *Lecture Notes in Computer Science*, pages 70–87. Springer Berlin Heidelberg, 2011.

[34] A.R. Bradley, F. Somenzi, Z. Hassan, and Yan Zhang. An incremental approach to model checking progress properties. In *Formal Methods in Computer-Aided Design (FMCAD), 2011*, pages 144–153, 2011.

[35] Randal E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Comput. Surv.*, 24(3):293–318, September 1992.

[36] J. R. Burch, E. M. Clarke, K. L. Mcmillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: $10^{20}$ states and beyond, 1990.

[37] Jerry Burch and David Dill. Automatic verification of pipelined microprocessor control. pages 68–80. Springer-Verlag, 1994.

[38] I. Černá and R. Pelánek. Distributed explicit fair cycle detection. In *Proc. SPIN workshop*, volume 2648 of *LNCS*, pages 49–74. Springer, 2003.

[39] Xiaofang Chen, Yu Yang, M. Delisi, G. Gopalakrishnan, and Ching-Tsun Chou. Hierarchical cache coherence protocol verification one level at a time through assume guarantee. In *High Level Design Validation and Test Workshop, 2007. HLVDT 2007. IEEE International*, pages 107–114, 2007.

[40] C.-T. Chou, P. K. Mannava, and S. Park. A simple method for parameterized verification of cache coherence protocols. In *FMCAD*, pages 382–398, 2004.

[41] A. Cimatti, E.M. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: a new symbolic model verifier. In N. Halbwachs and D. Peled, editors, *Proceedings Eleventh Conference on Computer-Aided Verification (CAV'99)*, number 1633 in Lecture Notes in Computer Science, pages 495–499, Trento, Italy, July 1999. Springer.

[42] E. M. Clarke, O. Grumberg, and M. C. Browne. Reasoning about networks with many identical finite-state processes. In *Proceedings of the fifth annual ACM symposium*

*on Principles of distributed computing*, PODC '86, pages 240–248, New York, NY, USA, 1986. ACM.

[43] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *ACM Trans. Program. Lang. Syst.*, 16(5):1512–1542, 1994.

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

[45] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *Computer aided verification*, pages 154–169. Springer, 2000.

[46] R. Cleaveland, S. P. Iyer, and D. Yankelevich. Abstractions for preserving all CTL* formulae. Technical report, 1994. Tech. Rep. 9403, Dept. of Comp. Sc., North Carolina State University, Raleigh, NC.

[47] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms.* McGraw-Hill Higher Education, 2nd edition, 2001.

[48] D. Dams, R. Gerth, and O. Grumberg. Abstract interpretation of reactive systems. *ACM Trans. Program. Lang. Syst.*, 19(2):253–291, 1997.

[49] N. Dave, M. C. Ng, and Arvind. Automatic synthesis of cache-coherence protocol processors using bluespec. In *Proceedings of the 2Nd ACM/IEEE International Conference on Formal Methods and Models for Co-Design*, MEMOCODE '05, pages 25–34, Washington, DC, USA, 2005. IEEE Computer Society.

[50] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, August 1975.

[51] D. L. Dill, A. J. Drexler, A. J. Hu, and C. H. Yang. Protocol verification as a hardware design aid. In *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 522–525, 1992.

[52] David L. Dill. The murphi verification system. In *International Conference on Computer Aided Verification*, pages 390–393, London, UK, 1996.

[53] Peter C Dillinger and Panagiotis Manolios. Bloom filters in probabilistic verification. In *Formal Methods in Computer-Aided Design*, pages 367–381. Springer, 2004.

[54] John Erickson. Personal communication, 2010.

[55] S. Evangelista, A. W. Laarman, L. Petrucci, and J. C. van de Pol. Improved multi-core nested depth-first search. In S. Ramesh, editor, *Proceedings of the 10th International Symposium on Automated Technology for Verification and Analysis, ATVA 2012, Thiruvananthapuram (Trivandrum), Kerala*, Lecture Notes in Computer Science, pages 269–283, London, October 2012. Springer Verlag.

[56] Y. Fang, N. Piterman, A. Pnueli, and L. Zuck. Liveness with invisible ranking. *Int. J. Software Tools for Technology Transfer*, 8(3):261–279, June 2006.

[57] Kathi Fisler, Ranan Fraer, Gila Kamhi, Moshe Y. Vardi, and Zijiang Yang. Is there a best symbolic cycle-detection algorithm? In *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS 2001, pages 420–434, London, UK, 2001. Springer-Verlag.

[58] Steven German. Personal communication, 2000.

[59] Rob Gerth, Doron Peled, Moshe Y. Vardi, R. Gerth, Den Dolech Eindhoven, D. Peled, M. Y. Vardi, and Pierre Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *In Protocol Specification Testing and Verification*, pages 3–18. Chapman & Hall, 1995.

[60] Orna Grumberg, Tamir Heyman, Nili Ifergan, and Assaf Schuster. Achieving speedups in distributed symbolic reachability analysis through asynchronous computation. In *In CHARME*, pages 129–145. Springer, 2005.

[61] Orna Grumberg, Tamir Heyman, and Assaf Schuster. A work-efficient distributed algorithm for reachability analysis. In Jr. Hunt, WarrenA. and Fabio Somenzi, editors, *Computer Aided Verification*, volume 2725 of *Lecture Notes in Computer Science*, pages 54–66. Springer Berlin Heidelberg, 2003.

[62] Z. Hassan, A. R. Bradley, and F. Somenzi. Incremental, inductive CTL model checking. In *Proc. of Int'l. Conf. on Computer Aided Verification*, pages 532–547, 2012.

[63] Klaus Havelund and Thomas Pressburger. Model checking JAVA programs using JAVA PathFinder. In *International Journal on Software Tools for Technology Transfer (STTT)*, volume 2(4), pages 366–381. Springer-Verlag, March 2000.

[64] R. C. Holt. Some deadlock properties of computer systems. *ACM Computing Surveys*, 4(3):179–196, 1972.

[65] G. J. Holzmann. The model checker SPIN. *IEEE Trans. Softw. Eng.*, 23(5):279–295, 1997.

[66] Gerard J. Holzmann. Parallelizing the spin model checker. In *Proceedings of the 19th international conference on Model Checking Software*, SPIN'12, pages 155–171, Berlin, Heidelberg, 2012. Springer-Verlag.

[67] A. Hu. *Techniques for Efficient Formal Verification Using Binary Decision Diagrams.* PhD thesis, Stanford University, 1995.

[68] Michael Huth and Mark Ryan. *Logic in Computer Science: Modelling and Reasoning About Systems.* Cambridge University Press, New York, NY, USA, 2004.

[69] C. N. Ip and D. L. Dill. Better verification through symmetry. In *11th IFIP WG10.2 International Conference on Computer Hardware Description Languages and their Applications*, pages 97–111, 1993.

[70] C. N. Ip and D. L. Dill. Better verification through symmetry. *Formal Methods in System Design*, 9(1/2):41–75, 1996.

[71] Valerie L. Ishida. Fault tolerance for distributed explicit state model checking. Master's thesis, University of British Columbia, Department of Computer Science, 2014.

[72] Hiroaki Iwashita, Tsuneo Nakata, and Fumiyasu Hirose. CTL model checking based on forward state traversal. In *Proc. Int'l. Conf. Computer-Aided Design*, pages 82–87, 1996.

[73] Yonit Kesten, Amir Pnueli, and Li on Raviv. Algorithmic verification of linear temporal logic specifications. In *Proc. 25th Int. Colloq. Aut. Lang. Prog., volume 1443 of Lect. Notes in Com p. Sci*, pages 1–16. Springer-Verlag, 1998.

[74] Yonit Kesten, Amir Pnueli, Li-On Raviv, and Elad Shahar. Model checking with strong fairness. *Form. Methods Syst. Des.*, 28:57–84, January 2006.

[75] W. J. Knottenbelt, P. G. Harrison, M. A. Mestern, and P. S. Kritzinger. A probabilistic dynamic technique for the distributed generation of very large state spaces. *Perform. Eval.*, 39(1-4):127–148, 2000.

[76] S. Krstić. Parameterized system verification with guard strengthening and parameter abstraction. In *Automated Verification of Infinite-State Systems*, 2005.

[77] R. Kumar and E. G. Mercer. Load balancing parallel explicit state model checking. In *Parallel and Distributed Model Checking*, 2004.

[78] A. W. Laarman and D. Faragó. Improved on-the-fly livelock detection: Combining partial order reduction and parallelism for $\text{dfs}_{FIFO}$. In G. P. Brat, N. Rungta, and A. J. Venet , editors, *Proceedings of the Fifth NASA Formal Methods Symposium, NFM 2013, Moffett Field, CA, USA*, Lecture Notes in Computer Science, London, May 2013. Springer Verlag.

[79] A. W. Laarman, R. Langerak, J. C. van de Pol, M. Weber, and A. Wijs. Multi-core nested depth-first search. In T. Bultan and P-A. Hsiung, editors, *Proceedings of the 9th International Symposium on Automated Technology for Verification and Analysis, ATVA 2011, Tapei, Taiwan*, volume 6996 of *Lecture Notes in Computer Science*, pages 321–335, London, July 2011. Springer Verlag.

[80] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Trans. Softw. Eng.*, 3:125–143, March 1977.

[81] K. G. Larsen and B. Thomsen. A modal process logic. In *Third Annual Symposium on Logic in Computer Science (LICS)*, pages 203–210, 1988.

[82] F. Lerda and R. Sisto. Distributed-memory model checking with SPIN. In *Proc. of SPIN 1999, volume 1680 of LNCS.*, pages 22–39. Springer-Verlag, 1999.

[83] Y. Li. Mechanized proofs for the parameter abstraction and guard strengthening principle in parameterized verification of cache coherence protocols. In *Proceedings of the 2007 ACM symposium on Applied computing*, pages 1534–1535, 2007.

[84] Orna Lichtenstein, Amir Pnueli, and Lenore Zuck. The glory of the past. In Rohit Parikh, editor, *Logics of Programs*, volume 193 of *Lecture Notes in Computer Science*, pages 196–218. Springer Berlin Heidelberg, 1985.

[85] LTSmin. Ltsmin related papers. `fmt.cs.utwente.nl/tools/ltsmin/#sec2`. Accessed: December 1, 2015.

[86] Y. Lv, H. Lin, and H. Pan. Computing invariants for parameter abstraction. In *MEMOCODE '07: Proceedings of the 5th IEEE/ACM International Conference on Formal Methods and Models for Codesign*, pages 29–38, 2007.

[87] N. Lynch and F. Vaandrager. Forward and backward simulations – part I: Untimed systems. *Information and Computation*, 121(2):214–233, 1995.

[88] Zohar Manna and Amir Pnueli. Completing the temporal picture. *Theor. Comput. Sci.*, 83:97–130, June 1991.

[89] Zohar Manna and Amir Pnueli. Temporal verification of reactive systems: Progress (draft). `http://theory.stanford.edu/~zm/tvors3.html`, 1996.

[90] K. L. McMillan. Symbolic model checking. In *PhD thesis, Carnegie Mellon University, Pittsburgh*, May 1992.

[91] K. L. McMillan. Circular compositional reasoning about liveness. In *Correct Hardware Design and Verification Methods (CHARME)*, pages 342–345, 1999. An extended version appeared as a Cadence technical report.

[92] K. L. McMillan. Verification of infinite state systems by compositional model checking. In *in CHARME*, pages 219–233. Springer, 1999.

[93] K. L. McMillan. Parameterized verification of the FLASH cache coherence protocol by compositional model checking. In *Correct Hardware Design and Verification Methods (CHARME)*, pages 179–195, 2001.

[94] Ken McMillan. Personal communication, 2011.

[95] K.L. McMillan. Interpolation and sat-based model checking. In Jr. Hunt, WarrenA. and Fabio Somenzi, editors, *Computer Aided Verification*, volume 2725 of *Lecture Notes in Computer Science*, pages 1–13. Springer Berlin Heidelberg, 2003.

[96] I. Melatti, R. Palmer, G. Sawaya, Y. Yang, R. M. Kirby, and G. Gopalakrishnan. Parallel and distributed model checking in eddy. *Int. J. Softw. Tools Technol. Transf.*, 11(1):13–25, 2009.

[97] S. Naffziger, J. Warnock, and H. Knapp. Se2 when processors hit the power wall (or ”when the cpu hits the fan”). In *Solid-State Circuits Conference, 2005. Digest of Technical Papers. ISSCC. 2005 IEEE International*, pages 16–17, Feb 2005.

[98] University of Utah School of Computing. Murphi model checker web page. `http://www.cs.utah.edu/formal_verification/Murphi/`. Accessed: December 1, 2014.

[99] J. O'Leary, M. Talupur, and M. R. Tuttle. Parameterized verification using message flows: An industrial experience. In *International Conference on Formal Methods in Computer Aided Design (FMCAD)*, 2009.

[100] R. Palmer and G. Gopalakrishnan. Partial order reduction assisted parallel model checking. In *Proc. Parallel and Distributed Model Checking (PDMC) Workshop*, 2002.

[101] Seungjoon Park and David L. Dill. Protocol verification by aggregation of distributed transactions. In *Proc. $8^{th}$ Int'l. Conf. on Computer Aided Verification*, pages 300–310, 1996.

[102] A. Pnueli, J. Xu, and L. D. Zuck. Liveness with (0, 1, infinity)-counter abstraction. In *Proceedings of the 14th International Conference on Computer Aided Verification (CAV)*, pages 107–122, 2002.

[103] U. Stern and D. L. Dill. Improved probabilistic verification by hash compaction. In *Correct Hardware Design and Verification Methods, IFIP WG 10.5 Advanced Research Working Conference, CHARME '95*, pages 206–224, 1995.

[104] U. Stern and D. L. Dill. Parallelizing the murphi verifier. In *International Conference on Computer Aided Verification*, pages 256–278, 1997.

[105] U. Stern and D. L. Dill. Using magnetic disk instead of main memory in the murphi verifier. In *10th International Conference on Computer Aided Verification (CAV)*, 1998.

[106] U. Stern and D. L. Dill. Parallelizing the murphi verifier. *Formal Methods in System Design*, 18(2):117–129, 2001.

[107] Murali Talupur and Mark R. Tuttle. Going with the flow: parameterized verification using message flows. In *Proceedings of the 2008 International Conference on Formal Methods in Computer-Aided Design*, FMCAD '08, pages 10:1–10:8, Piscataway, NJ, USA, 2008. IEEE Press.

[108] Robert Endre Tarjan. Depth-first search and linear graph algorithms. *Siam Journal on Computing*, 1:146–160, 1972.

[109] S. Tasiran, Yuan Yu, and B. Batson. Linking simulation with formal verification at a higher level. *Design Test of Computers, IEEE*, 21(6):472–482, Nov 2004.

[110] Moshe Y. Vardi and Pierre Wolper. An automata-theoretic approach to automatic program verification. In *Proceedings of the 1st Annual Symposium on Logic in Computer Science (LICS'86)*, pages 332–344. IEEE Comp. Soc. Press, June 1986.

[111] K. Verstoep, H. Bal, J. Barnat, and L. Brim. Efficient large-scale model checking. In *International Parallel and Distributed Processing Symposium*, 2009.

[112] WestGrid. Western canada research grid. `http://www.westgrid.ca`. Accessed: January 28, 2015.

[113] Pierre Wolper and Denis Leroy. Reliable hashing without collision detection. In *IN COMPUTER AIDED VERIFICATION. 5TH INTERNATIONAL CONFERENCE*, pages 59–70. Springer-Verlag, 1993.

[114] Y. Yu, P. Manolios, and L. Lamport. Model checking TLA+ specifications. In *Correct Hardware Design and Verification Methods (CHARME)*, pages 54–66, 1999.

# Appendix A

# German's Protocol

```
--
--  German's Protocol
--
--  A caching protocol proposed by Steven German of IBM as a challenge for
--  parameterized verification.
--

const  ---- Configuration parameters ----

  NODE_NUM : 2;
  DATA_NUM : 2;

type   ---- Type declarations ----

  NODE : scalarset(NODE_NUM);
  DATA : scalarset(DATA_NUM);

  CACHE_STATE : enum I, S, E;
  CACHE : record State : CACHE_STATE; Data : DATA; end;
  MSG_CMD : enum Empty, ReqS, ReqE, Inv, InvAck, GntS, GntE;
  MSG : record Cmd : MSG_CMD; Data : DATA; end;

var   ---- State variables ----

  Cache : array [NODE] of CACHE;      -- Caches
  Chan1 : array [NODE] of MSG;        -- Channels for Req*
  Chan2 : array [NODE] of MSG;        -- Channels for Gnt* and Inv
  Chan3 : array [NODE] of MSG;        -- Channels for InvAck
  InvSet : array [NODE] of boolean;   -- Set of nodes to be invalidated
  ShrSet : array [NODE] of boolean;   -- Set of nodes having S or E copies
  ExGntd : boolean;                   -- E copy has been granted
  CurCmd : MSG_CMD;                   -- Current request command
  CurPtr : NODE;                      -- Current request node
  MemData : DATA;                     -- Memory data
  AuxData : DATA;                     -- Auxiliary variable for latest data

---- Initial states ----

ruleset d : DATA do startstate "Init"
  for i : NODE do
    Chan1[i].Cmd := Empty; Chan2[i].Cmd := Empty; Chan3[i].Cmd := Empty;
    Cache[i].State := I; InvSet[i] := false; ShrSet[i] := false;
  end;
  ExGntd := false; CurCmd := Empty; MemData := d; AuxData := d;
end end;

---- State transitions ----

ruleset i : NODE; d : DATA do rule "Store"
```

```
  Cache[i].State = E
==>
  Cache[i].Data := d; AuxData := d;
end end;

ruleset i : NODE do rule "SendReqS"
  Chan1[i].Cmd = Empty & Cache[i].State = I
==>
  Chan1[i].Cmd := ReqS;
end end;

ruleset i : NODE do rule "SendReqE"
  Chan1[i].Cmd = Empty & (Cache[i].State = I | Cache[i].State = S)
==>
  Chan1[i].Cmd := ReqE;
end end;

ruleset i : NODE do rule "RecvReqS"
  CurCmd = Empty & Chan1[i].Cmd = ReqS
==>
  CurCmd := ReqS; CurPtr := i; Chan1[i].Cmd := Empty;
  for j : NODE do InvSet[j] := ShrSet[j] end;
end end;

ruleset i : NODE do rule "RecvReqE"
  CurCmd = Empty & Chan1[i].Cmd = ReqE
==>
  CurCmd := ReqE; CurPtr := i; Chan1[i].Cmd := Empty;
  for j : NODE do InvSet[j] := ShrSet[j] end;
end end;
```

```
ruleset i : NODE do rule "SendInv"
  Chan2[i].Cmd = Empty & InvSet[i] = true &
  ( CurCmd = ReqE | CurCmd = ReqS & ExGntd = true )
==>
  Chan2[i].Cmd := Inv; InvSet[i] := false;
end end;

ruleset i : NODE do rule "SendInvAck"
  Chan2[i].Cmd = Inv & Chan3[i].Cmd = Empty
==>
  Chan2[i].Cmd := Empty; Chan3[i].Cmd := InvAck;
  if (Cache[i].State = E) then Chan3[i].Data := Cache[i].Data end;
  Cache[i].State := I; undefine Cache[i].Data;
end end;

ruleset i : NODE do rule "RecvInvAck"
  Chan3[i].Cmd = InvAck & CurCmd != Empty
==>
  Chan3[i].Cmd := Empty; ShrSet[i] := false;
  if (ExGntd = true)
  then ExGntd := false; MemData := Chan3[i].Data; undefine Chan3[i].Data end;
end end;

ruleset i : NODE do rule "SendGntS"
  CurCmd = ReqS & CurPtr = i & Chan2[i].Cmd = Empty & ExGntd = false
==>
  Chan2[i].Cmd := GntS; Chan2[i].Data := MemData; ShrSet[i] := true;
  CurCmd := Empty; undefine CurPtr;
end end;

ruleset i : NODE do rule "SendGntE"
```

```
  CurCmd = ReqE & CurPtr = i & Chan2[i].Cmd = Empty & ExGntd = false &
  forall j : NODE do ShrSet[j] = false end
==>
  Chan2[i].Cmd := GntE; Chan2[i].Data := MemData; ShrSet[i] := true;
  ExGntd := true; CurCmd := Empty; undefine CurPtr;
end end;

ruleset i : NODE do rule "RecvGntS"
  Chan2[i].Cmd = GntS
==>
  Cache[i].State := S; Cache[i].Data := Chan2[i].Data;
  Chan2[i].Cmd := Empty; undefine Chan2[i].Data;
end end;

ruleset i : NODE do rule "RecvGntE"
  Chan2[i].Cmd = GntE
==>
  Cache[i].State := E; Cache[i].Data := Chan2[i].Data;
  Chan2[i].Cmd := Empty; undefine Chan2[i].Data;
end end;
```

# Appendix B

# HIR Proofs

Refer to equation (5.5) for the definition of $\pi_{j\leftrightarrow h}$ and let $T$ be shorthand for $T(n)$.

**Heuristic 1.** *For ruleset* $\widehat{r}_j$ *tagged with* $\mathsf{AEG}$ *and with abstracted ruleset parameter* $i$*, suppose that* $\widehat{r}_0$ *is UA. If* $\mathcal{A}_{\ell(1)} \models \mathsf{AG}\,(\mathit{relax}(\widehat{\rho}_0, i)|_{i=1} \rightarrow \mathsf{EF}\,(\widehat{\rho}_0|_{i=1}))$ *then tag* $\mathsf{AEG}$ *is discharged for showing* $\widehat{r}_j$ *to be UA.*

*Proof.* Without loss of generality, consider only states of $\psi^{-1}(\widehat{\rho}_j)$ where the ruleset parameters different from $i$ have values in $\{2, ..., k\}$; denote the set of such states that are $O$-reachable with $A$. We claim that for any $w \in \psi^{-1}(A)$, there exists some $h \in \mathsf{P}^n \setminus \mathsf{P}^k$ and state $w'$ where $\pi_{h\leftrightarrow 1}(w') \in \psi^{-1}(\mathit{relax}(\widehat{\rho}_0, i)|_{i=1})$ and either

1. $w = w'$, or
2. $w \rightsquigarrow_T w'$ and $w'$ only differs from $w$ in the local state of node 1.

Assume first that $\widehat{\rho}_0$ contains no forall condition, i.e., $\widehat{r}_0$ has no guard tags. Comparing $\psi^{-1}(\mathit{relax}(\widehat{\rho}_0, i)|_{i=1})$ with $\rho$, comparisons depending on $i$ syntactically replace $i$ with 1, and comparisons depending on array variables indexed by $i$, $a[i]$, are removed. If $\rho$ contains the $\mathsf{P}$-comparison $i = e_{\mathsf{P}}$ (at most one such condition exists by definition of admissible guard), then $\widehat{\rho}_j$ contains the condition $\mathit{Other} = e_{\mathsf{P}}$ if $e_{\mathsf{P}}$ is not abstracted away, and thus there is some $h \in \mathsf{P}^n \setminus \mathsf{P}^k$ such that $h = e_{\mathsf{P}}$ in $w$. If $\rho$ does not contain $i = e_{\mathsf{P}}$, then choose any $h \in \mathsf{P}^n \setminus \mathsf{P}^k$. In either case, $\pi_{h\leftrightarrow 1}(w) \in \psi^{-1}(\mathit{relax}(\widehat{\rho}_0, i)|_{i=1})$, i.e. $w = w'$.

On the other hand, suppose $\widehat{r}$ is tagged with $\mathsf{AUG}$. Since $\widehat{r}_0$ is UA, it must have been established UA through Heuristic 3, i.e., $\mathcal{A}_{\ell(1)} \models \mathsf{AG}\,(\Gamma(\widehat{\rho}_0) \rightarrow \mathsf{EF}\,C[1])$. Thus, although $\pi_{h\leftrightarrow 1}(w)$ may not satisfy $C[1]$, there exists a path to state $\pi_{h\leftrightarrow 1}(w') \in C[1]$ that only differs from $\pi_{h\leftrightarrow 1}(w)$ in the local state of node 1. Now, $\pi_{h\leftrightarrow 1}(w') \in \psi^{-1}(\mathit{relax}(\widehat{\rho}_0, i)|_{i=1})$ by a similar argument as above, and the claim is established.

The mixed abstraction property implies that $\pi_{h\leftrightarrow 1}(w') \rightsquigarrow_T \pi_{h\leftrightarrow 1}(w'')$, where $\pi_{h\leftrightarrow 1}(w')$ and $\pi_{h\leftrightarrow 1}(w'')$ only differ in the local state of node 1, and $\pi_{h\leftrightarrow 1}(w'') \in \psi^{-1}(\widehat{\rho}_0|_{i=1})$. Case split on whether or not $\widehat{r}_0$ is tagged with $\mathsf{AUG}$.

Case no $\mathsf{AUG}$: This implies that $\pi_{h\leftrightarrow 1}(w'') \in \rho$. Fire concrete rule $r$ to transition from $\pi_{h\leftrightarrow 1}(w'')$ to $\pi_{h\leftrightarrow 1}(w''')$. Notice that the local state of $h$ does not change between $\pi_{h\leftrightarrow 1}(w)$ and $\pi_{h\leftrightarrow 1}(w''')$ if $\widehat{r}_0$ has no command tags. Otherwise, if it is tagged with $\mathsf{AUC}$, then some update is performed on the local state of $h$. Thus, in the path from $w$ to $w'''$ the local state of 1 is updated identically, and this update is performed by $\widehat{\boldsymbol{a}}_j$. This implies state $w'''$ is in the preimage of the post-state of $\widehat{r}_j$, i.e., $\{\psi(w''')\} = \widehat{\boldsymbol{a}} \circ \{\psi(w)\}$ (the state obtained by applying update $\widehat{\boldsymbol{a}}$ to $\psi(w)$).

Case $\mathsf{AUG}$: Similar to the previous case, with the difference that the local state of nodes $\mathsf{P}^n \setminus (\mathsf{P}^k \cup \{h\})$ must satisfy $C$ before $r$ fires. Again, since $\widehat{r}_0$ is UA, it must have been established UA through Heuristic 3, i.e., $\mathcal{A}_{\ell(1)} \models \mathsf{AG}\,(\Gamma(\widehat{\rho}_0) \rightarrow \mathsf{EF}\,C[1])$. Since $\pi_{h\leftrightarrow 1}(w'') \in \Gamma(\widehat{\rho}_0)$, path symmetry implies the path $\pi_{h\leftrightarrow 1}(w'') \rightsquigarrow_T \pi_{h\leftrightarrow 1}(w''')$ where $\pi_{h\leftrightarrow 1}(w''') \in \rho$. Fire concrete rule $r$ to transition from $\pi_{h\leftrightarrow 1}(w''')$ to $\pi_{h\leftrightarrow 1}(w^{(4)})$. Using a similar case split on $\widehat{r}_0$ no command tags or $\mathsf{AUC}$ as above, state $w^{(4)}$ is in the preimage of the post-state of $\widehat{r}_j$. In either case, by path symmetry and noting that any changes to the local state of node $h$ between $w$ and $w'$ are not observable in the abstract system ($\psi(w) = \psi(w')$), the proof is completed. □ □

**Heuristic 2.** *For ruleset $\widehat{r}_j$ tagged with* $\mathsf{AEG}$ *and/or* $\mathsf{AEC}$ *with abstracted ruleset parameter* *i, suppose that* $\widehat{r}_0$ *is UA. If* $\mathcal{A}_{\ell(1)} \models \mathsf{AG}\,((\mathit{relax}(\widehat{\rho}_0, i)|_{i=1} \wedge L[1]) \rightarrow \mathsf{EF}_{\widehat{r}_0}\,(L[1]))$ *then* AEG *and* AEC *are discharged for showing* $\widehat{r}_j$ *to be L-preserving.*

*Proof.* To show the claim, suffices to show that for each reachable $(s, s') \in \widehat{r}_j$,

$$\forall w \in \psi^{-1}(s).\ \exists w' \in \psi^{-1}(s').\ \forall j \in \mathsf{P}^n_{k+1}.\ w \rightsquigarrow_T w' \ \wedge\ L[j] \Rightarrow w' \in L[j] \tag{B.1}$$

Let $h \in \mathsf{P}^n_{k+1}$ be the concrete ruleset parameter in these $w$. We first show the weaker property that

$$\forall w \in \psi^{-1}(s).\ \exists w' \in \psi^{-1}(s').\ w \rightsquigarrow_T w' \ \wedge\ L[h] \Rightarrow w' \in L[h] \tag{B.2}$$

noting that $h$ here depends on $w$. Let $A_j = \mathit{relax}(\widehat{\rho}_0, i)|_{i=j} \wedge L[j] \wedge \mathit{Reach}$. By the mixed abstraction property,

$$\forall w \in \psi^{-1}(A_1).\ \exists w' \in \psi^{-1}(L[1]).\ \ w \rightsquigarrow_T w'$$

where some transition from $r$ with ruleset parameter value 1 is taken along the $w \rightsquigarrow_T w'$ path. Applying permutation $\pi 1 \leftrightarrow h$ to these path for each $h \in \mathsf{P}^n_{k+1}$ gives

$$\forall w \in \psi^{-1}(A_h).\ \exists w' \in \psi^{-1}(L[h]).\ \ w \rightsquigarrow_T w'$$

where some transition from $r$ with ruleset parameter value $h$ is taken along the $w \rightsquigarrow_T w'$ path. This implies the property (B.2). To see that the strongly property (B.1) also holds, observe that if the ruleset is tagged with AUG or AUC then Heuristic 4 will imply the locality property for nodes indexed with $\mathsf{P}^n_{k+1} \setminus \{h\}$. On the other hand, if the rulset is not tagged in this way then the local state of such nodes is unchanged because only local rules fired on the $w \rightsquigarrow_T w'$ paths. □

**Heuristic 3.** *For ruleset* $\widehat{r}_j$ *tagged with* AUG *and not* AEG*, let* $\forall i \in \mathsf{P}^n.\ C[i]$ *be the forall condition of* $\rho$*. If* $\mathcal{A}_{\ell(1)} \models \mathsf{AG}\,(\Gamma(\widehat{\rho}_j) \rightarrow \mathsf{EF}\,(C[1]))$*, then tag* AUG *is discharged for showing* $\widehat{r}_j$ *to be UA.*

*Proof.* Let $A = \Gamma(\widehat{\rho}_j) \wedge \mathit{Reach}$. By the mixed abstraction property,

$$\forall w \in A.\, \exists w' \in C[1].\, w \rightsquigarrow_T w' \tag{B.3}$$

where $w$ and $w'$ only differ in the local state of node 1. For $k+1 \leq h \leq n$, let $I_h$ denote the property

$$\forall w \in A.\, \exists w' \in \left( \bigwedge_{i \in \mathsf{P}^h_{k+1}} C[i] \right).\, w \rightsquigarrow_T w'$$

where $w$ and $w'$ only differ in the local state of $i \in \mathsf{P}^h_{k+1}$. Applying permutation $\pi_{1\leftrightarrow k+1}$ to (B.3) gives property $I_{k+1}$. This is the base case for induction on $h$, assuming $I_h$ holds. Apply permutation $\pi_{1\leftrightarrow h+1}$ to (B.3) and use transitivity to establish property $I_{h+1}$. Thus, $I_n$ holds which implies the UA property. □

**Heuristic 4.** *For ruleset $\widehat{r}_j$ tagged with* AUG *and/or* AUC*, let $\widehat{r}_{2^m-1} \in \psi(r)$ be the abstract ruleset where all ruleset parameters of $r$ are abstracted. If $\mathcal{A}_{\ell(1)} \models \mathsf{AG}\left((\Gamma(\widehat{\rho}_j) \wedge L[1]) \rightarrow \mathsf{EF}_{\widehat{r}_{2^m-1}}(L[1])\right)$, then tags* AUG *and* AUC *are discharged for showing $\widehat{r}_j$ to be L-preserving.*

*Proof.* Let $A_h = \Gamma(\widehat{\rho}_j) \wedge L[h] \wedge \mathit{Reach}$. By the mixed abstraction property,

$$\forall w \in A_1.\, \exists w' \in L[1].\, w \rightsquigarrow_T w' \tag{B.4}$$

where each path $w \rightsquigarrow_T w'$ includes a transition from ruleset $r$ where 1 is nota ruleset parameter value. For $k+1 \leq h \leq n$, let $I_h$ denote the property

$$\forall w \in A_h.\, \exists w' \in L[h].\, w \rightsquigarrow_T w'$$

where each path $w \rightsquigarrow_T w'$ includes a transition from ruleset $r$ where $i$ is not a rulset parameter value. Applying permutation $\pi_{1\leftrightarrow h}$ to (B.4) gives property $I_h$. The set of properties $I_h$ imply the claim, because each path is independent with respect to local variables of $h$, with the exception of the transition $t$ from ruleset $r$. For a given $h$, transition $t$ is agnostic to the local variables of other nodes, thus a $t$ exists that satisfies all paths for each property $I_h$. □

# Appendix C

# Heuristic Examples

## C.1 Discharging AEG

When the only guard tag is AEG, at least one ruleset parameter has been abstracted. Our presentation makes the simplifying assumption that there is at most one such ruleset parameter when AEG is discharged, but the same approach generalizes to the case of more than one. If an automatically generated DF property on the mixed abstraction holds, this tag is *discharged*, and the ruleset is proven UA, and thus may be added to the set of $U$ transitions. This property is based on a permutation of the guard predicate of the ruleset in question, and also the guard of the related ruleset where none of the ruleset parameters are abstracted.

Consider the following concrete ruleset from the German protocol:

```
ruleset i : NODE do rule "SendGntS"
  CurCmd = ReqS & CurPtr = i & Chan2[i].Cmd = Empty & ExGntd = false
   ==>
  Chan2[i].Cmd := GntS; Chan2[i].Data := MemData; ShrSet[i] := true;
  CurCmd := Empty; undefine CurPtr;
end end;
```

This ruleset abstracts to two rulesets in the mixed abstraction:

```
ruleset i : NODE do rule "ABS_SendGntS1"
  CurCmd = ReqS & CurPtr = i & Chan2[i].Cmd = Empty & ExGntd = false
   ==>
  Chan2[i].Cmd := GntS; Chan2[i].Data := MemData; ShrSet[i] := true;
  CurCmd := Empty; undefine CurPtr;
end end;

rule "ABS_SendGntS2"
  CurCmd = ReqS & CurPtr = Other & ExGntd = false
   ==>
  CurCmd := Empty; undefine CurPtr;
end;
```

In `ABS_SendGntS1`, the ruleset parameter `i` is non-abstracted, and the concrete rule `SendGntS` contains no forall conditions in the guard. Thus, `ABS_SendGntS1` has no guard tags. Ruleset `ABS_SendGntS2`, on the other hand, has `i` abstracted and is therefore tagged with AEG. The generated DF property considers all reachable states satisfying the guard of `ABS_SendGntS2`, but with `CurPtr` permuted to be some non-abstracted node. From these states, we seek a path composed of only local transitions to the value of `CurPtr`[1] to a state satisfying the guard of `ABS_SendGntS1`. This is expressed in Mur$\varphi$ as

```
-- using only rules local to 1
liveness "H1"
  -- the "start predicate"
  CurCmd = ReqS & (isdefined(CurPtr) & CurPtr != Other) & ExGntd = false
CANGETTO
  -- the "end predicate"
  CurCmd = ReqS & (exists i : NODE do CurPtr = i & Chan2[i].Cmd = Empty end)
  & ExGntd = false
end;
```

---

[1] Due to symmetry we can assume this value is 1 without loss of generality.

It turns out that every reachable state satisfying the start predicate either

1. also satisfies the end predicate, or
2. requires local rule `RecvGntS` to fire, which clears a "grant shared access" message from channel 2, and the resulting state satisfies the end predicate.

This proves `ABS_SendGntS2` is UA. To understand why, consider the proof schema in Figure C.1. In this example, $\widehat{r}_2$ is the rule `ABS_SendGntS2`, and $\widehat{\rho}_2$ is its guard predicate. Permutation $\pi$ is specific to the given state $w$; it exchanges the value of `CurPtr` in $w$ with 1. Ruleset $r$ refers to the concrete ruleset `SendGntS`, where $r|_{\texttt{CurPtr}=1}$ is the rule when the ruleset parameter `i` is 1, and $r|_{\texttt{CurPtr}>1}$ is the set of rules when `i` is some other value.

It is unknown if `Chan2[CurPtr].Cmd = Empty` for an arbitrary, reachable $w \in \psi^{-1}(s)$ (i.e., a reachable concretization of $s$) because `CurPtr = Other` in $s$. Since `CurPtr` $>$ 1 in $w$, there exists some permutation $\pi$ such that `CurPtr = 1` in $\pi(w)$. The output printed by PReach when DF property `H1` is model checked indicates that in $\pi(w)$ either $r|_{\texttt{CurPtr}=1}$ is enabled or the concrete rule `RecvGntS` is enabled, and when fired reaches a state where $r|_{\texttt{CurPtr}=1}$ is enabled. In either case, there is a path from $\pi(w)$ to some state $\pi(w')$ that enables $r|_{\texttt{CurPtr}=1}$, and when this rule fires reaches state $\pi(w'')$; by symmetry, there is an analogous path from $w$ to some state $w''$ resulting from some rule of $r|_{\texttt{CurPtr}>1}$ firing. Since $w$ was chosen arbitrarily, it follows that `ABS_SendGntS2` is UA.

## C.2 Discharging AUG

Referring to the rulesets of the example in Section 5.3.2, suppose we wish to show `ABS_SendGntE1` is UA. Notice that although its guard is syntactically identical to that of the concrete ruleset `SendGntE`, it is not assumed underapproximate because the forall condition weakens the guard. Thus, this rule is tagged with AUG. To discharge the tag, we show that in the concrete system each abstracted node $j$ can take a local path to some $C[j]$-state, where $C[j]$ is `!ShrSet[j]`. That is, we check

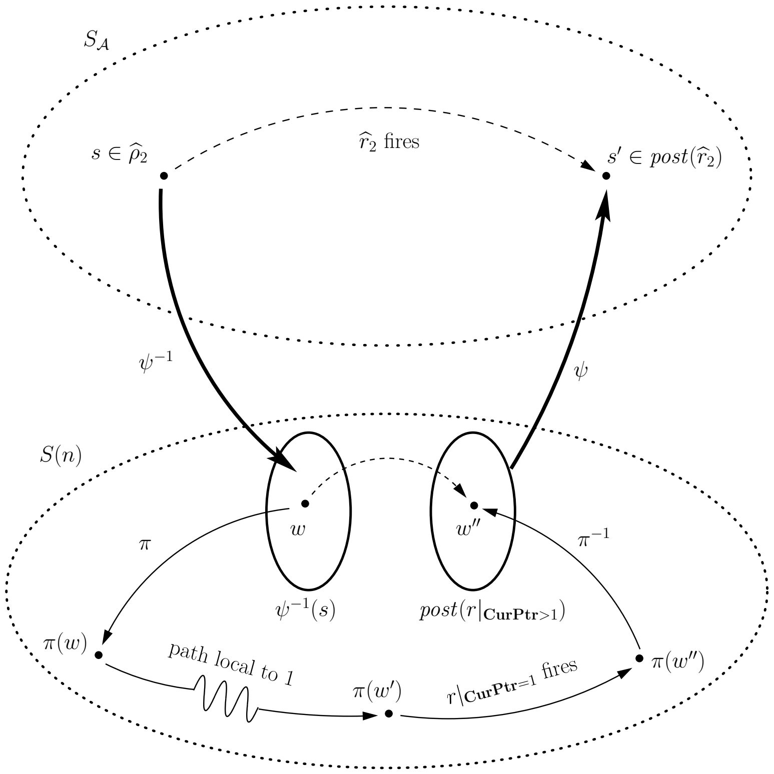


Figure C.1: Schema for proving `ABS_SendGntS2` is UA, which follows a similar flow for handling any rule tagged with AEG. For arbitrary reachable $w \in \psi^{-1}(s)$, the path from $\pi(w)$ to $\pi(w'')$, by symmetry, implies a path exists from $w$ to some $w'' \in \psi^{-1}(s')$. This satisfies the definition for the transition $(s, s')$ to be UA. The path from $\pi(w)$ to $\pi(w')$ is implied by model checking the DF property `H1` in the mixed abstraction.

```
liveness "H2" -- using only rules local to 1

   CurCmd = ReqE & isdefined(CurPtr) & !ExGntd -- the "start predicate"

CANGETTO

   (forall j : NODE do !ShrSet[j] end) -- the "end predicate" (j = 1)

end;
```

This DF property holds in the mixed abstraction. For every reachable state $s$ satisfying the start predicate, one of the following cases applies.

1. State $s$ satisfies the end predicate.

2. A sequence of 4 local rules fire, starting from $s$, to reach some state satisfying the end predicate. This sequence begins at $s$ where a pending message in channel 2 of cache 1 that grants shared access; the first rule consumes this message. The second rule sends invalidate message to cache 1 from the directory. The third rule consumes this message and response with an invalidation acknowledgment message on channel 3, and the fourth rule consumes this message at the directory and sets the sharer flag of cache 1 to false.

3. Some post-fix of this sequence of rules is taken, starting from $s$ to reach some state satisfying the end predicate. That is, either the latter 3 rules above fires, or the latter 2 rules, or the final rule.

This establishes that for any local state of some cache in a reachable state satisfying the start predicate, there is a local path to some state where the sharer flag is false. Intuitively, although the local state of abstracted nodes are unknown when the guard of `ABS_SendGntE1` is enabled, the above check verifies the existence of an abstracted path to a state where the sharer flag is cleared for all abstracted caches. See Figure C.2.

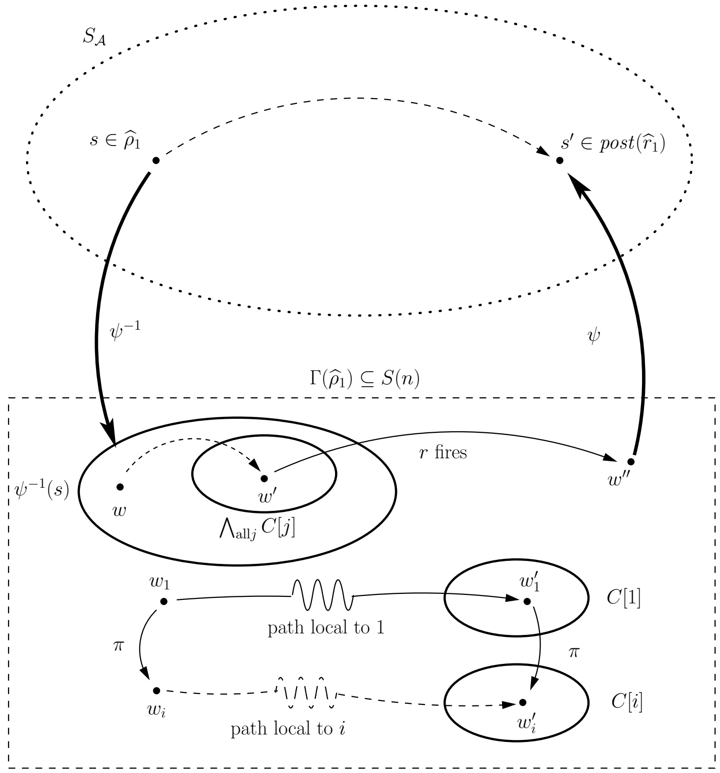


Figure C.2: Schema for proving ruleset `ABS_SendGntE1` is UA. For any reachable concrete state $w_1$ satisfying the boolean predicate implied by $\widehat{\rho}_1$, there is a path to $w_1' \in C[1]$ that only changes the local state of 1. By symmetry, there is a path from any reachable $w_i \in \Gamma(\widehat{\rho}_1)$ to $w_i' \in C[i]$ that only changes the local state of $i$, for $1 \leq i \leq n$. Concatenating these paths together over all $i$, there is a path from any $w \in \Gamma(\widehat{\rho}_1)$ to $w'$ satisfying $\bigwedge_{1 \leq i \leq n} C[i]$. In particular, if $w \in \psi^{-1}(s)$, then $w'$ satisfies the guard of $r$, and this rule fires to reach $w''$. The path from $w$ to $w''$ satisfies the definition of the transition $(s, s')$ to be UA. The path from $w_1$ to $w_1'$ is implied by model checking a DF property in the mixed abstraction.