Directed and Distributed Model Checking of B-Specifications

Inaugural-Dissertation

zur Erlangung des Doktorgrades
der Mathematisch-Naturwissenschaftlichen Fakultät
der Heinrich-Heine-Universität Düsseldorf

vorgelegt von

Jens Marco Bendisposto

aus Wuppertal

Düsseldorf, Dezember 2014

Aus dem Institut für Informatik
der Heinrich-Heine-Universität Düsseldorf

Gedruckt mit der Genehmigung der
Mathematisch-Naturwissenschaftlichen Fakultät der
Heinrich-Heine-Universität Düsseldorf

Referent: Prof. Dr. Michael Leuschel
Heinrich-Heine-Universität Düsseldorf

Koreferent: Prof. Dr. Stefan Hallerstede
Aarhus University

Tag der mündlichen Prüfung: 23. Januar 2015

Abstract

This is the age of software. Almost every modern electronic device has some parts implemented in software. Compared to hardware solutions, software is much cheaper and more flexible. But there is a downside to this: Software errors can kill. A concurrency bug in the software of the THERAC-25 system for radiation therapy killed at least three people. Others were seriously injured [1].

These safety critical systems require a different approach than ordinary business applications. Testing, which is the dominant method in quality management is not sufficient. Development of safety critical systems requires formal methods, i.e., a more rigorous mathematical approach.

This thesis deals with the verification of B and Event-B models using the ProB model checker. The goal was to significantly improve the performance exploiting the high abstraction level of B. We achieved this goal. Using the results of this thesis it is possible to verify models that could not previously be checked within reasonable time constraints.

In particular the distributed version of ProB improved the model checking run time by orders of magnitude. Using the prototype developed in this thesis, we think it is possible to check state spaces with billions of states. Previously it was not possible to check more than a couple of 10 million states, depending on the model.

We also integrated proof information into the model checking process to reduce the costs of checking each individual state. Using our improvement, we could halve the run time for some of the benchmarks.

Finally, we developed a method to extract flow information from Event-B models. The results can be used to reduce the memory footprint as well as the run time of model checking. It can also be used to gain better understanding of the algorithmic structure of a model and for proving deadlock freedom and some other liveness properties.

We evaluated the implementations using several models from academia and industry.

Zusammenfassung

Wir leben im Zeitalter der Software. Es gibt kaum ein technisches Gerät, das heute ohne Komponenten auskommt, die in Software realisiert sind. Software ist kostengünstig zu entwickeln und viel flexibler als Hardware. Die Kehrseite der Medaille ist jedoch: Softwarefehler können töten. Ein Fehler in der Synchronisation der nebenläufigen Software des THERAC-25 Linearbeschleunigers für die Strahlentherapie hat mindestens drei Menschen das Leben gekostet und weitere schwer verletzt [1].

In solchen sicherheitskritischen Systemen kann man nicht die im Umfeld der typischen Anwendungsentwicklung dominante Methode zur Qualitätssicherung — das Testing — einsetzen. Solche Software erfordert striktere, mathematische Methoden, die sogenannten formalen Methoden.

In dieser Arbeit beschäftigen wir uns mit der Verifikation von formalen Modellen in B und Event-B mit Hilfe des Model Checkers ProB.

Das Ziel ist, den hohen Abstraktionsgrad der Modellierungssprachen auszunutzen, um signifikante Verbesserungen der Laufzeit zu erreichen. Dieses Ziel wurde erreicht. Es lassen sich mit den in dieser Arbeit vorgestellten Verbesserungen Modelle prüfen, deren Verifikation vorher nicht praktikabel war.

Insbesondere die verteilte Version von ProB hat eine Verbesserung der Laufzeit von teilweise zwei Grössenordnungen bewirkt. Mit Hilfe des in dieser Arbeit vorgestellten Prototypen können potentiell Modelle mit Milliarden von Zuständen verifiziert werden. Bisher war die praktikable Grenze im Bereich von einigen 10 Millionen Zuständen, abhängig vom Model.

Desweiteren haben wir eine Integration von Beweisinformationen ausgenutzt um die Kosten für die Berechnung einzelner Zustände zu reduzieren. Mit Hilfe dieser Methode konnten wir bei einem Teil der Beispiele eine Reduktion auf die halbe Laufzeit erreichen.

Ausserdem haben wir eine Methode entwickelt, die zur Extraktion von Programmfluss-Informationen aus Event-B Modellen dienen kann. Diese Methode kann sowohl zur Reduktion des Speicherverbrauchs und der Laufzeit als auch zum Verständnis der algorithmischen Struktur des Modells beitragen. Ausserdem können mit Hilfe der Analyse die Abwesenheit von Deadlocks und andere Aussagen über die Lebendigkeit eines Systems bewiesen werden.

Die Evaluierung der in dieser Arbeit entwickelten Implementierungen erfolgte mit Hilfe verschiedener akademischer und industrieller Beispiele.

Acknowledgments

This thesis has been influenced by the contributions of so many people to whom I owe my deepest gratitude.

First of all, I want to thank Michael Leuschel for supporting me in so many ways. Almost from the very beginning of my studies at the HHU, he opened so many doors for me. He always encouraged me to pursue my interests and learn as much as I can. I am deeply grateful that I had the chance to work with Michael, one of the smartest and kindest persons I have ever met. I also want to thank Stefan Hallerstede. Stefan not only took the time to read and review my dissertation, but he also shaped my thinking about formal modeling.

I also want to thank Claudia Kiometzis for her support dealing with the bureaucracy of a big organization.

When I started studying computer science, I had more than a decade of experience in developing software. I thought that there wasn't much more to learn about programming. I couldn't have been more wrong. I thank all the teachers who opened my eyes and changed my point of view. I am grateful that they shared their knowledge and also for their welcoming atmosphere. I always felt welcome when I needed help or had questions.

I want to thank all my colleagues, co-authors, students and friends at the STUPS group for their contribution to my research. In alphabetical order: Rene Bartelmus, Michael Birkhoff, Carl Friedrich Bolz, Markus Borgermans, Marc Büngener, Joy Clark, Ivalyo Dobrikov, Nadine Elbeshausen, Marc Fontaine, Fabian Fritz, René Goebbels, Dominik Hansen, Michael Jastram, Sebastian Krings, Philipp Körner, Lukas Ladenberger, Daniel Plagge, Matthias Radig, Mireille Samia, David Schneider, Corinna Spermann, Ingo Weigelt, Harald Wiegard and John Witulski.

I want to especially highlight the tremendous work of Philipp Körner who did an excellent job of implementing my ideas about the distribution framework in C.

And I also want to highlight the valuable feedback I got from David Schneider, Sebastian Krings and Joy Clark. They proofread my thesis and helped me to improve the clarity of the text and to fix my goofy punctuation.

The research that led to this thesis was funded by the Deutsche Forschungsgemeinschaft (DFG) in the GEPAVAS projects.

Computational support and infrastructure was provided by the “Centre for Information and Media Technology” (ZIM) at the University of Düsseldorf (Germany).

I would also thank all the people who provided me with adminstrative and technical support over the years: Janus Tomaschewski, Angela Rennwanz, Guido Königstein, Thomas Spitzelei, Christian Knieling, Stephan Raub, Philipp Helo Rehs and Angelika Simons.

Finally my gratitude goes to out my friends and family. I could not have done this without your love and support.

Contents

1 Introduction **1**
1.1 The B-Method . . . 2
1.2 Event-B and Rodin . . . 5
1.3 ProB . . . 11
1.3.1 Animation . . . 11
1.3.2 Visualization . . . 12
1.3.3 Model Checking . . . 12
1.3.4 Other animators and model checkers for B . . . 13
1.4 Explicit Model Checking . . . 14

2 Proof supported and directed model checking **19**
2.1 The basic idea of proof supported model checking . . . 19
2.2 Formal Verification . . . 22
2.2.1 Basic contexts . . . 22
2.2.2 Abstract Model Checker . . . 24
2.2.3 First Refinement: Introducing sequential state processing . . . 25
2.2.4 Second Refinement: Model Checker . . . 26
2.2.5 Third Refinement: Abstract Invariant Processing . . . 27
2.2.6 Regular and Proof Supported Model Checking . . . 29
2.2.7 Theorems and Well-Definedness . . . 32
2.3 Implementation of Proof Supported Model Checking . . . 34
2.4 Missing Proof Obligations . . . 35
2.5 Application in classical B . . . 36
2.6 Experimental results . . . 36
2.6.1 Mondex . . . 37
2.6.2 Siemens Deploy Mini Pilot . . . 37
2.6.3 Scheduler . . . 38
2.6.4 Earley Parser . . . 38
2.6.5 SAP DEPLOY Mini Pilot . . . 38
2.6.6 SSF DEPLOY Mini Pilot . . . 39
2.6.7 Cooperative Crosslayer Congestion Control CXCC . . . 39
2.6.8 Constructed Example . . . 39
2.7 Using proof information for directed model checking . . . 41
2.7.1 Proof Mode . . . 41
2.7.2 Debug Mode . . . 42
2.8 Related work . . . 42

3 Automatic Flow Analysis **45**
3.1 Nondeterministic Assignments 45
3.2 Total substitution lifting 47
3.3 Independence of Events 48
3.3.1 Trivial Independence 49
3.3.2 Non-Trivial Independence 50
3.3.3 Application: Guard Evaluation 53
3.3.4 TLF versus DIF 55
3.3.5 Application: State patching 56
3.3.6 Nontrivial Independence and missing Abstractions 57
3.4 Enabling and disabling Predicates 58
3.4.1 Computing the enabling predicates 61
3.4.2 Simplification 63
3.5 Enable Graph 66
3.6 Example: Extended GCD Algorithm 67
3.7 Application: Guard Evaluation 69
3.8 Weak versus Strong Independence 71
3.9 Flow Analysis 72
3.10 Feasibility of Flow Graph Construction 74
3.11 Application: Proving Deadlock Freedom 75
3.12 Example: Extended GCD Algorithm 78
3.13 Future work 79
3.14 Related work 80

4 Distributed Model Checking **83**
4.1 Motivation . . . 83
4.2 Parallel Model Checking . . . 85
4.3 When does distributed model checking (not) work? . . . 88
4.4 The parB Model Checker . . . 89
4.4.1 First Attempts . . . 89
4.4.2 Architecture . . . 90
4.4.3 Work-Items . . . 91
4.4.4 Shared Data . . . 92
4.4.5 Storing checked states . . . 95
4.4.6 Communication . . . 100
4.4.7 Counterexample Generation . . . 105
4.4.8 Symmetry Reduction . . . 106
4.4.9 Proof Support . . . 107
4.4.10 Control UI . . . 108
4.4.11 Assertion checking and Data validation . . . 108
4.5 Empirical Evaluation . . . 111
4.5.1 Overhead . . . 114
4.5.2 Benchmarks with limited scaling. . . . 116
4.5.3 Benchmarks with (almost) linear scaling . . . 117
4.5.4 High performance cluster . . . 123
4.6 Future Work . . . 124
4.7 Related Work . . . 125

5 Conclusion **131**
5.1 Contributions . . . 131
5.1.1 Proof supported and directed model checking . . . 131
5.1.2 Flow Analysis . . . 132
5.1.3 Distributed model checking . . . 133

A Formal Model of Proof Supported Model Checking **137**

B Publications **165**

Bibliography **183**

List of Figures **191**

List of Tables **193**

Chapter 1

Introduction

In his essay "Why Software Is Eating The World" the Netscape co-founder Marc Andreessen writes how software impacts practically every business and every area of our daily life.

> "In today's cars, software runs the engines, controls safety features, entertains passengers, guides drivers to destinations and connects each car to mobile, satellite and GPS networks. The days when a car aficionado could repair his or her own car are long past, due primarily to the high software content. The trend toward hybrid and electric vehicles will only accelerate the software shift — electric cars are completely computer controlled. And the creation of software-powered driverless cars is already under way at Google and the major car companies." [2]

Although implementing functionality in software instead of hardware helps to reduce costs and time to market, there is a big catch. We have to come up with processes that allow us to reliably produce high-quality software. Producing correct software is a problem that has been tackled from different angles. Most "Business software" is developed using testing as the tool to ensure quality. In contrast, safety-critical software is often developed using formal methods. It is hard to give a precise definition of formal methods because there are many different approaches and verification techniques, but the recurring theme is that stronger use of mathematics helps to ensure reliability of the software. Typical areas where formal methods are applied include aerospace and railway. In the car industry, we can also observe more interest in formal verification of software. The most common verification techniques are proof, model checking and abstract interpretation. Proof is often used in a "correct-by-construction" approach, where a software system is specified abstractly using a mathematical notation and then transformed into an implementation. Proofs are used to guarantee that the implementation respects

the specification. The other two techniques can be used with a correct-by-construction approach, but they also can be used to verify implementations at the code level. In particular, abstract interpretation is often used on implementations that have not been developed using a correct-by-construction approach. For abstract interpretation and model checking, we sometimes don't even require an explicit specification. Instead the tools look for standard problems, e.g., buffer overflows or accessing null pointers.

However, because software has become more and more business-critical, there is a growing number of applications of formal methods outside the domain of safety critical systems. For instance, Amazon used TLA$^+$ to verify parts of the Amazon Web Services [3, 4], Microsoft uses their SLAM tool to verify Windows device drivers [5], and SAP used a domain specific language on top of Event-B to ensure correctness in the communication between services [6].

We believe that it will be unavoidable for the IT industry to adopt more rigorous standards in order to produce quality software. However, the broad adaption of formal methods in the software engineering community requires tools that are reliable and fast. In this thesis we propose improvements to the ProB model checker to extend its applicability. In particular, our distributed version of ProB allows us to verify models that cannot be checked within reasonable time constraints using the regular version of ProB.

In the rest of this chapter we will give an overview of the B-Method and its successor Event-B. We give a short overview of the Rodin toolset and explain what ProB is.

1.1 The B-Method

The B-Method (also referred to as classical B) was developed by Jean-Raymond Abrial in the 1980s [7]. It is a development method and notation for software systems that are mathematically proven. It is mainly used to develop safety critical systems, most notably in the railway domain.

The notation is based on set theory and first order logic. The development method makes heavy use of refinement and proof. A software system is specified in an abstract machine and then gradually refined towards a concrete system. The artifact of a classical B development is a machine, which consists of sets, machine parameters, constants, variables, and operations.

A set in B is basically a type: it is a non-empty set of elements that is disjoint from any other set. This means within the sets declaration it is not possible to declare a set $S = \{1, 2, 3\}$ because S would not be disjoint from the integer numbers. We can introduce S as a constant of type $\mathbb{P}(\mathbb{Z})$, which could indeed be equal to $\{1, 2, 3\}$. A machine can be parameterized; this is useful to construct generic machines, for instance, we could implement a generic stack that can be later parameterized to be a stack of integer numbers or a stack of strings. Parameters can be either sets or constants. All these elements describe the static properties of the machine, and they can be restricted using some predicates. The predicates that restrict the parameters are called constraints. The predicates that restrict the sets and constants are called properties.

The B-Method requires a developer to discharge so called proof obligations (PO) to verify the correctness of a model. For instance, it requires to prove that the constraints C are not self contradictory, i.e., there are some parameters p that fulfill the constraints.

$$\exists p \cdot C(p)$$

It is also required to prove that if the constraints can be satisfied there are sets s and constants c that satisfy the properties P.

$$C(p) \Rightarrow \exists s, c \cdot P(p, s, c)$$

For instance, if we model a cruise control system, and we want to use parameters to describe physical properties of the car, we might restrict the range of possible values in order to guarantee correctness. In our example we may add something that keeps the top speed below some reasonable limit.

Machines also describe the dynamic behavior using variables. Variables are initialized in the so called INITIALISATION and modified by operations. The variables form the state of the model and their values are restricted in the invariant. In the B-Method, the invariant is very important because it is used to specify the safety properties of the system. In order to prove correctness, we have to prove that the initialization *init* establishes the invariant I for state v.

$$C(p) \wedge P(p, s, c) \Rightarrow ([init]I)(p, s, c)$$

Additionally, we have to prove that each operation preserves the invariant. We use S to describe the substitution of an operation. v denotes the state before applying the

operation, and v' denotes the state after applying the operation. Pre is the precondition of the operation. We use $[S]I$ to denote the weakest predicate that describes the largest set V of states for which $\forall v \in V \Rightarrow (([S]I)(v) \Rightarrow I(v'))$ holds. This weakest predicate, called the weakest precondition, can be computed mechanically$[S]I$ [8, p. 25ff]. We need to prove for all operations that the following holds:

$$C(p) \wedge P(p, s, c) \wedge I(p, s, c, v) \wedge Pre(p, s, c, v) \Rightarrow ([S]I)(p, s, c, v)$$

Furthermore, a machine can be refined. A refined machine typically introduces more detail for an algorithm, i.e., it reduces the non-determinism or introduces more concrete steps or data structures.

As an example for the introduction of a more concrete implementation detail we will use swapping the content of two variables (the example is taken from [8, p. 174]). The abstract substitution is $x, y := y, x$. In principle there are many possible implementations. For instance, if we assume that x and y contain numbers in a bit-encoding we could use bitwise XOR (denoted by $\oplus$) to swap the variables using the following sequence of substitutions $x := x \oplus y; y := x \oplus y; x := x \oplus y$. But more likely, we would introduce a temporary variable, yielding the sequence of statements $t := x; x := y; y := t$.

If we use different variables in the refinement, we specify a so called gluing invariant that relates the concrete and abstract variables.There are also proof obligations that deal with the correctness of the refinement [8, p. 219ff], but they are not relevant in the context of this thesis.

Once the model has reached a certain level of detail, called B_0, automatic tools can generate executable code from the specification. A B_0 specification is also called implementation and it imposes constraints on the model [8, p. 258]. For instance, while an abstract machine can use nondeterministic assignment like $x :\in \{1, 2, 3\}$, an implementation can only use deterministic assignments.

Nowadays, the proof part of the B-Method is mainly supported by the AtelierB [9] tool, which is developed by the company ClearSy. A free alternative for academic purposes called B4Free [10] was also offered by ClearSy. Both tools used the same decision procedures. Since 2009 there has been a free AtelierB community edition and the development of B4Free has been stopped.

The second important tool for the B-Method is ProB, which is developed at the University of Düsseldorf and described in more detail in Section 1.3.

AtelierB and ProB are largely orthogonal; AtelierB is used to prove correctness of a model while ProB is used for animation and visualization. ProB also supports model checking. ProB can be started from within AtelierB.

1.2 Event-B and Rodin

Event-B [11] is a successor of classical B created to allow specifications on a system level rather than on the level of software. Rodin [12] is an Eclipse based tool set that supports modeling in Event-B. The Rodin core tool consists of components for editing, static checking and proof management. It was developed in the EU funded project RODIN and successively improved in the projects DEPLOY and ADVANCE.

A drawback of classical B is that it sometimes results in complex and complicated proof obligations that are hard to discharge, in particular when it comes to refinement proof obligations. Event-B was designed to produce simpler proof obligations. Its notion of refinement is also a bit more liberal than refinement in classical B. For instance, in Event-B we can introduce new events, while in classical B all events have to be present in the abstract model. In Event-B there are two different artifacts: machines and contexts. Contexts are used to specify the static parts of the model, i.e., sets, constants and the corresponding axioms and theorems. Axioms are similar to the constraints and properties in classical B. The dynamic part, i.e., the variables, the invariant, and the operations is modeled in a machine. Contexts can extend each other and they are seen by machines. The only relationship between machines supported by the Rodin core is refinement. There are plug-ins for Rodin that add new relationships between Event-B artifacts. An example of such a plug-in is the composition [13] plug-in, which allows Event-B machines to be to decomposed and composed. The mathematical language of Event-B used in the Rodin core is — with few exceptions — a subset of the classical B notation. An example for an exception is an additional syntax for set comprehensions. In classical B a set comprehension has the form $\{v \mid v \in T \land P(v)\}$ while in Event-B it has the form $\{v \cdot P(v) \mid E(v)\}$. The Event-B notation is a bit more convenient and concise, for instance, if we compare how we could specify the set of square numbers in classical B ($\{x \mid x \in \mathbb{Z} \land (\exists y \cdot y \in \mathbb{Z} \land x = y * y)\}$) with the specification in Event-B ($\{x, y \cdot x = y * y \mid x\}$).

Event-B was designed to be rather minimalistic. Many constructs that are present in classical B are missing in Event-B, for instance, sequences. However, Event-B can be

extended and in Rodin there is the theory plug-in [14] that allows defining these kind of language extensions.

As with classical B, the correctness of a development requires proving several obligations. The Rodin tool uses an incremental and interactive approach to modeling, i.e., each time a model is changed by the user, Rodin will run a pipeline of tools and update the proof obligations and the proofs. This is basically done in three steps. The first step is to run a static checker, which is essentially the Rodin parser and type checker. The static checker will infer the type for each constant and variable. It will produce an error annotation if there are syntax or type errors. The static checker produces a new version of the model that only contains the correct parts of the original input. This static checked model is then fed into the proof obligation generator (POG), which will generate obligations that we have to discharge to prove correctness of our development. Finally, Rodin applies several automatic reasoners to the obligations. We will come back to the automatic reasoners and other Rodin extensions after discussing the proof obligations that the POG produces.

We will only cover the proof obligations that are important for this thesis, namely well-definedness (WD), theorem (THM) and invariant preservation (INV). The most important proof obligation is INV, but the other two have to be considered because they have an impact on the soundness of the proof supported model checking, which we will discuss in Chapter 2 of the thesis. There are also other proof obligations that are not considered in our work, namely:

- Feasibility of nondeterministic assignment (FIS)
- Refinement proof obligations
  - Guard strengthening (GRD)
  - Guard merging (MRG)
  - Simulation (SIM)
- Proof obligations for termination of convergent events
  - Numeric variant is a natural number (NAT)
  - Finite set variants (FIN)
  - Decreasing of the variant (VAR)
- Witness feasibility (WFIS)

The WD proof obligation

The WD proof obligation prevents us from specifying ill-defined expressions. If we write, for instance, the axiom $card(S) < 2$, the expression $card(S)$ might be ill-defined if S is not a finite set. Therefore, we need to prove *finite*(S). We get similar proof obligations for several other expressions as listed in [11, p. 202].

The THM proof obligation

The THM proof obligation is generated for each theorem. Theorems can be specified in the axioms clause of a context. In a machine, we can write theorems in the invariant and also within the guards of an event. Typically, theorems are used to support the automatic prover by introducing a lemma. We get the theorem as the goal of the proof obligation and we are allowed to use all axioms/invariants/guards/theorems that occur before the theorem. For instance, if we specify an axiom $\exists p \cdot p \in 1..4 \rightarrowtail\!\!\!\!\twoheadrightarrow c$ and a theorem $finite(c)$, we can use the axiom to prove the theorem (which can be done). If we flip the order, we cannot use the axiom to prove the theorem (which cannot be done). The ordering prevents circular reasoning.

The INV proof obligation

The INV proof obligation [11, p. 188f] is probably the most prominent proof obligation. As in classical B, it states that we have to prove that the invariant holds after observing an event given that it was true before the event occurred. As hypotheses, we can use the event's guard, the invariant in the predecessor state, as well as axioms and theorems from seen contexts and theorems from the machine. The rule is:

$$I(s, c, v) \wedge G(s, c, v, x) \wedge BA(s, c, v, x, v') \wedge A(s, c) \vdash inv(s, c, v') \tag{1.1}$$

In this rule, s and c denote the sets and constants introduced by some contexts. v and v' denote the machine's variable that form the state, and x are some variables that are local to the event. I are the invariants and theorems of the machine, G is the guard of the event, BA is the before-after-predicate for the event's substitution, and A are the axioms and theorems from some contexts. Finally, inv is one conjunct of the invariant of the machine.

The invariant establishment rule is very similar, it states that we have to prove that the initialization of the machine establishes the invariant:

$$K(s, c, v') \land A(s, c) \vdash inv(s, c, v')$$

Because the system does not have a state before the initialization, we cannot use the invariant in the proof and the substitution is expressed in the form of an after-predicate K instead of a before-after predicate. Also the initialization cannot have a guard.

The complete invariant is split into its conjuncts because it is much easier to reason about a single conjunct at a time. This is equivalent to proving the conjunction because if we can prove that each conjunct is preserved by an event, the conjunction is preserved as well ([7, p. 11] and [11, p. 311]).

For our purposes, we do not have to distinguish between the invariant establishment and preservation proof obligations. From now on, when we talk about the invariant preservation proof obligation, we will always also mean the invariant establishment obligation.

As an example for the INV proof obligation rule, let us look at an example. The machine shown in Figure 1.1 has an error that will be exposed by the proof obligations.

In theory, this machine should require proving four proof obligations:

1. **INITIALISATION/type/INV**
   Invariant establishment of the invariant labeled with *type*.

2. **add/type/INV**
   Invariant preservation of the invariant labeled with *type* by the event *add*.

3. **INITIALISATION/positive/INV**
   Invariant establishment of the invariant labeled with *positive*.

4. **add/positive/INV**
   Invariant preservation of the invariant labeled with *positive* by the event *add*.

In Rodin we only get the proof obligations that are concerned with the *positive* invariant. The reason is that since the *type* invariant can be verified by the type checker, the tool can always automatically discharge the obligation if the machine is well typed. The tool therefore does not even create these proof obligations.

MACHINE m0
VARIABLES
n

INVARIANTS
`type` : $n \in \mathbb{Z}$
`positive` : $n > 0$

EVENTS
Initialisation
begin
`init` : $n := 1$
end
add $\widehat{=}$
any
x
where
`type` : $x \in \mathbb{Z}$
then
`add` : $n := n + x$
end
END

Figure 1.1: Example erroneous Event-B machine

The INITIALISATION/positive/INV obligation boils down to proving $1 > 0$ which can trivially be discharged. The second proof obligation cannot be discharged because from $n > 0$ and $x \in \mathbb{Z}$ we cannot prove that $n + x > 0$. In fact, if x is negative and $|x| > n$, n will be negative.

There are two reasons why a proof obligation is not discharged. The first reason is that the machine contains a mistake. In our example, we may realize that we originally wanted to add the square of x instead of x.

add $\widehat{=}$
any
x
where
`type` : $x \in \mathbb{Z}$
then
`add` : $n := n + x * x$
end

This yields a different proof obligation $n > 0 \land x \in \mathbb{Z} \vdash n + x * x > 0$. With only the automatic provers, Rodin still cannot discharge the obligation. But now the reason is

different. The model is consistent, but the automatic provers are not powerful enough to discharge the obligation. They require a little hint from the user, namely the lemma $x * x \geq 0$. We can either do this using the interactive prover or by adding the lemma as a theorem inside the guard. In either case, after adding the lemma, Rodin is able to discharge the proof obligation.

add $\widehat{=}$

any
x
where
`type` : $x \in \mathbb{Z}$
`thm` : $x * x \geq 0$ `(theorem)`
then
`add` : $n := n + x * x$
end

The Rodin tool is delivered with a number of automatic tactics that can be applied to transform a proof obligation or to discharge simple obligations. For instance, if the goal of the proof obligation is identical to one of the hypotheses, one of the tactics will automatically discharge the obligation. There is also a decision procedure called newPP delivered with Rodin. However, most of the time, we do not use newPP but the battle tested AtelierB provers that are delivered as a Rodin plug-in. More recently, a plug-in for SMT solvers like veriT [15] and CVC3 [16] has been developed [17]. The plug-in translates a proof obligation into the SMTLib format and runs one or multiple solver backends on the SMT formula. ProB can also be used to discharge proof obligations [18]. Given a sequent $H_1, H_2, ..., H_n \vdash G$, we ask ProB to find a solution for $H_1, H_2, ..., H_n \wedge \neg G$. By a solution, we mean a model of the predicate. If we can find a model, this model is a counterexample for the proof obligation. If we cannot find a counterexample this either means that the solver wasn't powerful enough or that there is no counterexample. Previous incarnation of the tool made no further distinction, but more recently we have introduced a method to refine the process. Now ProB can — if no enumeration of a potential infinite set occurred during solving — deduce that no counterexample can exist and therefore discharge the proof obligation. There is also a plug-in to use Isabelle/HOL as a prover [19].

There are several other third-party plugins including requirements engineering [20], modeling using an UML like graphical notation [21, 22], editing [23], pretty printing using LaTeX [22], animation [22, 24, 25], and model checking [26].

1.3 ProB

ProB [27, 28] is a model checker and animator for a number of different formal specification languages. Among the supported languages are classical B and Event-B, which we use in this thesis.

The main goals of ProB are to:

- Increase the confidence that a specification meets the requirements. This is the scope of the animator part of ProB. Using the animator, we can convince ourselves that a specific feature is present in the model.
- Help to detect counterexamples for the correctness conditions. This is the domain of the various model checkers in ProB. Using model checking we can detect errors without wasting time interactively trying to prove a goal that cannot be proven.
- Improve the understanding of a formal specification by visualizing important information.

1.3.1 Animation

Animation in this context means the stepwise execution of a formal model making ProB effectively a debugger. We can step through a model and inspect the values of each variable at each state. Using the animator it is possible to manually inspect the happy-paths[1] of a model, i.e., validate that the model behaves as expected. The animator is handy because it catches mistakes at a very early stage in the development process. For instance, when we developed the formal model described in Section 2.2, we alternated between proof and animation. After each change we ran the animator to convince ourselves that the model behaved as expected. Only then did we attempt to discharge the proof obligations. Animation alone is clearly not sufficient, but in combination with proof it is a very effective development method.

ProB also has several tools dedicated to debugging, for instance, for analysis of a predicate. If a predicate (e.g. the guard of an event) is false when we expect it to be true, we can run an analysis to shrink the predicate down to a smaller predicate that is also false. We can then inspect why the predicate is false and fix the formal model or adapt our own mental model.

[1] "In the context of software or information modeling, a happy path is a default scenario featuring no exceptional or error conditions, and comprises the sequence of activities executed if everything goes as expected." (`http://en.wikipedia.org/w/index.php?title=Happy_path&oldid=615829885`)

We can also open an evaluation shell and evaluate expressions in the current state, which can be handy if the types are very complex and we only care for a projection of the original data. The Rodin plug-in version of ProB can add expressions and predicates to the state view. This means we can add a customized derived value to the user interface and observe the value over time.

1.3.2 Visualization

The person who develops a model usually has a good understanding of what the data means, but it can become very hard to communicate the meaning to another person. This is especially the case if the person is a domain expert rather than an expert in formal modeling. However, the developer of a model can also get into problems when inspecting a large dataset such as the state space of a machine. ProB has a rich set of visualizations to help users with these kind of problems. For instance, it is possible to apply reduction techniques to the state space, i.e., projection on a single expression. This can help the person who writes a model to verify the behavior at a more abstract level.

ProB can also help communicating with domain experts using a graphical representation of the states.

1.3.3 Model Checking

ProB has several different ways to perform model checking. It supports LTL model checking [29] using an extension written in C. It has an explicit consistency and deadlock checker written in Prolog, and it also does constraint based checking [30]. This thesis is mostly concerned with the explicit consistency and deadlock checking. We will therefore explain explicit model checking in greater detail in Section 1.4 and discuss the other kinds of model checking only very briefly.

In constraint based model checking (CBC), ProB tries to find a pair (v, v') of states such that there is an event that leads from v to v', and the invariant I holds in v but not in v'. If ProB finds a solution this means that we cannot prove the model. The solution is a counterexample for the invariant preservation proof obligation; we have to prove invariant preservation for any event and any state that satisfies the invariant. However, this does not mean that the states are reachable from the initialization. Sometimes, finding a counterexample using CBC only means that our invariant is not strong enough.

However, as mentioned before, this is a weakness in the model that prevents us from proving that it is correct.

Linear temporal logic (LTL) is a modal temporal logic that can be used to formulate properties about traces in a model. For instance, we can specify that a system infinitely often reaches a state where a predicate ϕ holds using the formula $GF\{\phi\}$. The curly braces are part of ProB's LTL syntax. Inside the curly braces we can use predicates of the modeling language. G and F are temporal operators; G stands for globally and F for finally. The formula means that we can select any state (globally) and in the future (finally) the property has to be true. In other words, there cannot be a last time where the property holds, therefore it must hold infinitely often.

1.3.4 Other animators and model checkers for B

For classical B there are only few other animators, namely BZTT [31] and the B-Toolkit [32], none of which provide automatic animation. This means that the user has to manually provide solutions for the input parameters which can be difficult. For instance, take the operation:

```
foo(aa,bb) = PRE
               −aa³+ 3aa² + aa = 3 &
               card(bb) = aa &
               bb ⊆ y
             THEN
               x,y := aa,bb
             END
```

ProB is able to find values for *aa* and *bb* automatically, while the other tools would require that the user solves the constraints, i.e., the user has to compute that the three solutions to $-aa^3 + 3aa^2 + aa = 3$ are -1, 1 and 3. Because $card(bb) = aa$ the value of aa has to be positive. Finally, the user has to select a subset of y with 1 or 3 elements. As we can see, having an automatic solver is a huge benefit for a user of the tool. In particular if the solutions are big sets it becomes a very cumbersome task to manually find them. Another issue is that it is easy to overlook unexpected corner cases. Both other tools are no longer being developed or supported.

For Event-B there are several other tools. Within Rodin there are AnimB [24] and BRama [25], both are animators that also have some capabilities for graphical visualization. Both tools have been successfully applied to formal models. Outside of Rodin

there is B2Express [33] which translates Event-B specifications into the EXPRESS language [34]. However, like the tools for B, all of these tools do not find solutions for parameters automatically.

On the model checking side is Eboc [35] a lazy unbounded model checker written in Racket. EBoc uses a diagonalization technique to enumerate unbounded sets instead of using a fixed bound like ProB does.

1.4 Explicit Model Checking

In this section we will explain ProB's model checking algorithm in a bit more detail. In principle, we want to find out if the following predicates are true.

$$\forall s \cdot s \in \mathit{reachable\ States} \Rightarrow \mathit{Invariant}(s) \tag{1.2}$$

$$\forall s \cdot s \in \mathit{reachable\ States} \Rightarrow (\exists e \cdot e \in \mathit{Events} \wedge \mathit{Guard}(e)) \tag{1.3}$$

Predicate 1.2 describes consistency checking, i.e., the system cannot violate the invariant, while Predicate 1.3 describes one specific deadlock freedom condition. In this case, each state has at least one successor state, and we don't require that the successor of a state is a different state, i.e., a self loop is not a deadlock. In Predicate 1.3, an event e refers to a solution found by ProB, i.e., the event's name, a solution for its parameters and a decision for any nondeterministic assignment inside the event. In other words, e corresponds to an instance of an event as shown in the operations view of ProB.

It is worth noting that the deadlock freedom condition described in Predicate 1.3 is only one of many possible options. For instance, we could demand that the successor state of e is different from s. This would eliminate self loops and rule out query events in classical B. This could be a reasonable restriction, because a system that can only be observed cannot be distinguished from a deadlocked system.

We also may want to rule out systems that contain a cycle between the same k states. For instance, if we have a system that implements a mutual exclusion protocol using locks and we end up in a situation where the only action the system can perform is to acquire and release a lock we may consider the system to be in a quasi deadlock often called a livelock.

Finally, we may also have a correct system that is supposed to contain a deadlock, e.g., because it is a model of an algorithm that terminates.

While ProB supports many different notions of deadlocking through LTL model checking, the consistency checker that we describe here only[2] allows us to check for deadlocks in the sense of Predicate 1.3.

If we knew all reachable states, we could simply check the properties using a loop, but computing all reachable states in advance is not reasonable when we look for counterexamples. In cases where we want to verify the model, we could in principle compute the reachable states in advance and then check them.

However, we prefer an online algorithm, i.e., exploration and checking of reachable states is done in a single pass. The advantage is that we can stop as soon as we find a counterexample, and we can drastically reduce the memory consumption, if we use hash codes for states that have been checked. This introduces the possibility of errors, but we can keep the probability of an error very low as we will see in Chapter 4.

The online algorithm ProB uses is shown in Figure 1.2. It is based on algorithm 5.1 from [28]. We abstracted away the details how the processing queue works; in the context of this thesis it is not important. We also added deadlock checking to the algorithm.

The algorithm starts from a set of initial states. These states are computed in a special way. If the machine contains static information, e.g., axioms in Event-B, ProB tries to find a solution for the static setup. After that, it will execute the initialization event resulting in proper states. In Figure 1.2 this is done in the *initialize* function.

ProB then enters its model checking loop, where it selects an unprocessed state and checks the invariant. If the check fails, it has found an invariant violation and the algorithm terminates. Otherwise, ProB computes the successor states. If there are no successors, we have found a deadlock in the model and the algorithm terminates. Finally, it will iterate over the successors and if they are not yet known, they will be put into the processing queue. When *queue* is empty, the algorithm terminates and *known* contains all reachable states.

An example run of the algorithm is shown in Figure 1.3. Initially, as shown in Figure 1.3(a), only state a is known, but its invariant has not yet been checked. Then we dequeue a, we check its invariant of a compute and enqueue the successors b and c. This situation is shown in Figure 1.3(b). We then dequeue b resulting in the situation shown in Figure 1.3(c). Finally, we process c resulting in the situation shown in Figure 1.3(d).

[2]This is not entirely true, ProB can also check custom predicates that can be used to encode some of the other possible deadlock conditions.

```
// Initialization
queue := initialize()
known := queue

// Model Checking Loop
while queue ≠ ∅ do
    s := chose_from(queue)
    queue := queue \ {s}

    if ¬invariant(s)
       then return counter-example(s);

    succ := {s' | ∃ e · e ∈ Events ∧ s →e s'}

    if succ = ∅
       then return deadlock(s);

    foreach s' ∈ succ do
       if s' ∉ known then do
             queue := queue ∪ {s'}
             known := known ∪ {s'}
      od
    od
od

return no-counter-example
```

Figure 1.2: Explicit Consistency and Deadlock Checking

Table 1.1 shows the states that are in the sets *known* and *queue*. Note that b is not enqueued again in Figure 1.3(d) although it is a successor of the state that was processed. It is already in the set *known* and therefore ignored. This ensures termination of the algorithm for finite state spaces.

It is worth noting that a state is explored in two phases. First, it is "discovered" as the successor of some state. When discovered, ProB will compute the values of the state variables and store them, but it will not yet check the invariant and compute the successors. This is done in the second step. When a state is taken from the processing

Figure	*known*	*queue*
1.3(a)	a	a
1.3(b)	a,b,c	b,c
1.3(c)	a,b,c,e	c,e
1.3(d)	a,b,c,d,e	d,e

Table 1.1: Exploring the state space in Figure 1.3

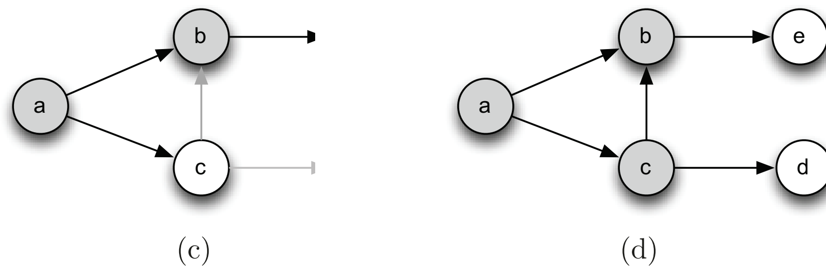


Figure 1.3: Exploration of a part of the state space

queue, its invariant is checked and the successors are computed, i.e., the successor states are now in phase 1. This is also shown in Figure 1.3. Fully unknown states are outlined with a light gray color, black colored states are in phase 1, and the gray filled states are fully processed.

Chapter 2

Proof supported and directed model checking

This chapter describes work published in [36] on exploiting information about discharged proofs to improve ProB's performance. The status of a proof obligation carries valuable information for a model checker. As described in Section 1.3, ProB's model checker performs an exhaustive search. It traverses the state space and verifies that the invariant is preserved for each state. This section describes how we incorporate proof information from Rodin into the ProB core. Section 2.1 will explain how we can use the information about discharged proof obligations to our advantage when doing model checking, and we will formally show that our approach is sound in Section 2.2. We will then demonstrate in Section 2.6 that our approach is actually beneficial in terms of model checking runtime using experimental results. Finally, we will explain how we could use proof information to create strategies for state space exploration, i.e., use it as a heuristic for best first search.

2.1 The basic idea of proof supported model checking

Assuming we have a model that contains the invariants I_1, I_2, and I_3. During model checking we observe an event *evt* resulting in a new state. For instance, if we have proven that *evt* preserves I_1 and I_3, there is no need to actually perform the check for these invariants. This kind of knowledge is precisely what we get from the prover and it can potentially reduce the cost of invariant verification during model checking if the benefit we gain from not checking the proven invariants outweighs the cost of filtering the invariants.

The ProB plug-in translates a Rodin project, consisting of the model itself, its abstractions, and all necessary contexts, into a representation used by ProB. We evolved this translation process to also incorporate proof information, i.e., our representation contains a list of tuples (E_i, I_j) of all discharged POs. This means that we have a proof that event E_i preserves invariant I_j.

Using this information, we can determine an individual pattern of invariants we have to check for each event that is defined in the machine. Instead of checking all conjuncts, we can restrict the checking to only those that are not proven to be correct.

Even better, if multiple events $e_1, e_2, \ldots, e_k$ lead to a state, it is sufficient to only check the invariants that are not proven for all events $e_1, e_2, \ldots, e_k$. If we keep sets $unproven(e_i)$ of invariants that are not proven for an event e_i, we would only have to check the intersection of these sets, i.e., $unproven(e_1) \cap unproven(e_2) \cap \ldots \cap unproven(e_k)$.

As an example we can use the Event-B model shown in Figure 2.1. The full state space of this model and the proof status delivered by the automatic provers of the Rodin tool are shown in Figure 2.2.

The proof status shows that Rodin is able to discharge the proof obligations $a/inv1$ and $b/inv2$ but not $a/inv2$ and $b/inv1$. This means that if a occurs, we can be sure that $f \in \mathbb{N} \nrightarrow \mathbb{N}$ holds in the successor state if it holds in the predecessor state. Analogously, we know that if b occurs, we are sure that $x > 3$ holds in the successor state if it holds in the predecessor state.

Consider a situation, where we already verified that all invariants hold for *S1* and we are about to check that *S2* is consistent. We discovered two incoming transitions corresponding to the events a and b. From a, we can deduce that $f \in \mathbb{N} \nrightarrow \mathbb{N}$ holds. From b, we know that $x > 3$ holds. To verify *S2*, we need to check the intersection of unproven invariants, i.e., $\{f \in \mathbb{N} \nrightarrow \mathbb{N}\} \cap \{x > 3\} = \varnothing$, thus we already know that all invariants hold for *S2*.

This is of course only a tiny example, but it demonstrates that using proof information we are able to reduce the number of invariants for each event significantly, and sometimes by combining proof information from different events, we do not have to check any invariants. If taken to the extreme, it is in principle possible to check invariant preservation of a partially proven model by traversing the state space without ever checking any invariant at all.

MACHINE m0
VARIABLES
f
x

INVARIANTS
`inv1` : $f \in \mathbb{N} \nrightarrow \mathbb{N}$
`inv2` : $x > 3$

EVENTS
Initialisation
begin
`act1` : $f := \{1 \mapsto 100\}$
`act2` : $x := 10$
end

a $\widehat{=}$
begin
`act1` : $f := \{1 \mapsto 100\}$
`act2` : $x := f(1)$
end

b $\widehat{=}$
begin
`act1` : $f := f \cup \{1 \mapsto 100\}$
`act2` : $x := 100$
end

END

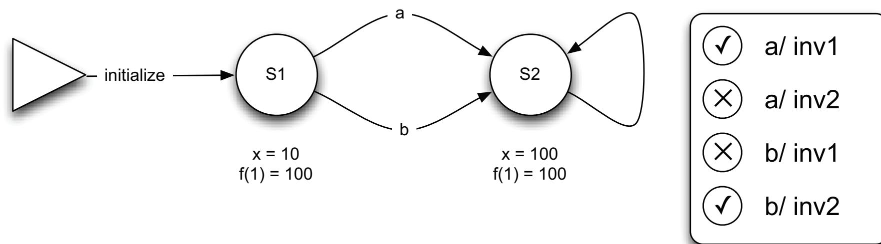


Figure 2.2: State space of the model in Figure 2.1

2.2 Formal Verification

To show that our approach is indeed correct, we have developed a formal model of proof supported model checking. We omitted a few technical details, such as the way the state space is traversed by the actual implementation, and we also omitted the fact that our implementation always uses all available information. Instead, we have proven correctness for any traversal and any subset of the available information. Our model was developed using Event-B and has been fully proven in Rodin. The Event-B contexts and machines are available in Appendix A.

The formal model is structured as shown in Figure 2.3. We artificially linearized the contexts to avoid warnings in the Rodin tool. For instance, from the modeling point of view, c3 does not have to extend c2. Both c3 and c2 only have to extend c1. But if we would combine c2 and c3 in a machine we would get a "Redundant seen context" warning.

2.2.1 Basic contexts

The c1 context contains the basic carrier sets *STATES*, *INVARIANTS* and *EVENTS*. By introducing the sets as deferred sets, we make the implicit assumption that all models contain at least one state, one invariant, and one operation (or event). We think this is reasonable because otherwise there would be no point in model checking the system, i.e., a model where one of these assumptions does not hold is trivially correct.

In c1 we also introduce *truth*, a set containing tuples of states and invariants. An invariant i is true in the state s if and only if the tuple $(s \mapsto i)$ is an element of *truth*. Additionally, we introduce two sets *preserve* and *violate* that partition the *STATES* type into states where each invariant is true and states where at least one invariant is false. The following axioms are used to tie together the *preserve*, *violate* and *truth* sets:

$$preserve \cap violate = \varnothing \tag{2.1}$$

$$preserve \cup violate = STATES \tag{2.2}$$

$$preserve = \{s \mid \{s\} \times INVARIANTS \subseteq truth\} \tag{2.3}$$

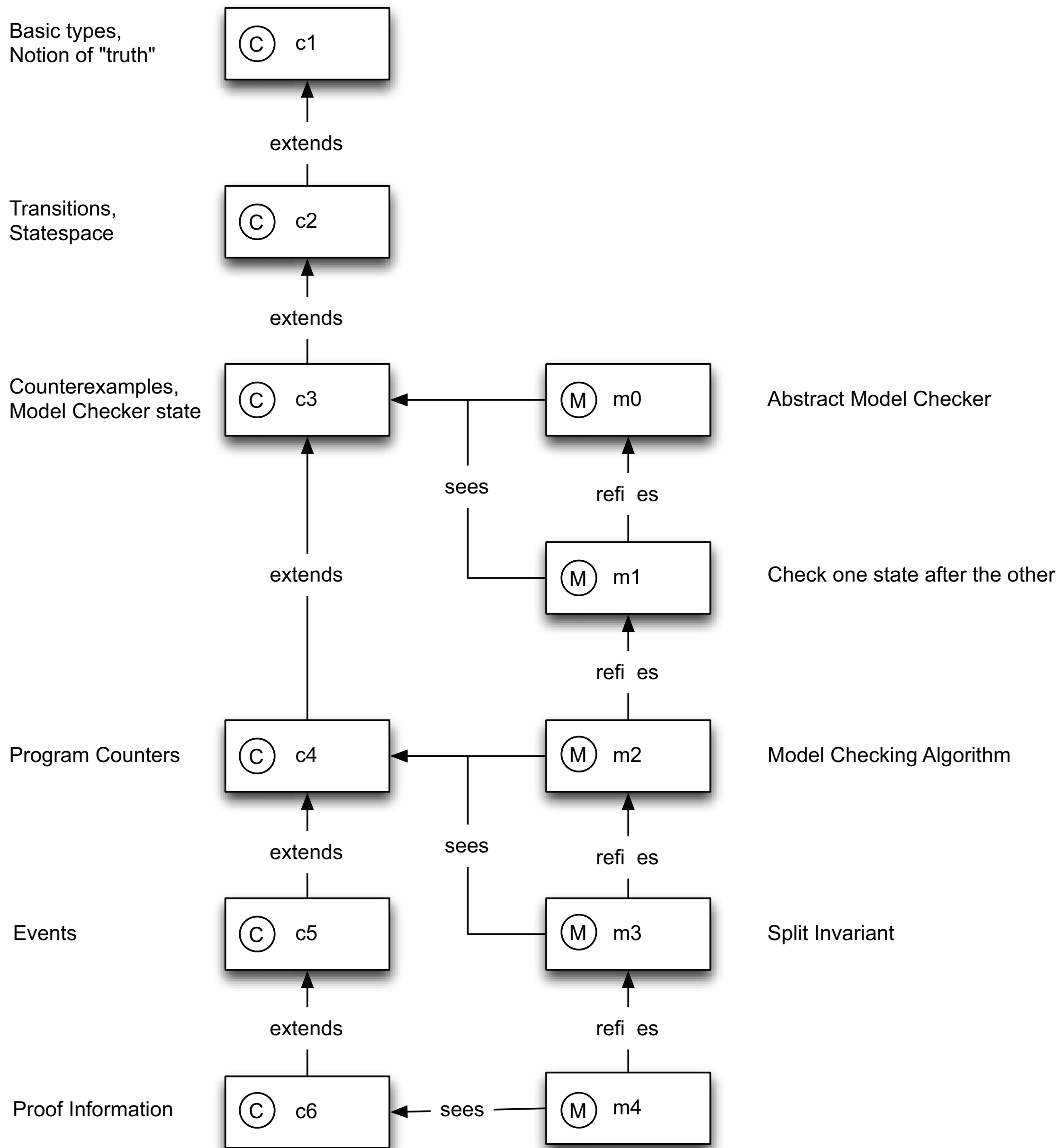


Figure 2.3: Organization of the formal model

Using these definitions, we have proven a number of lemmas that are handy when proving certain properties in the later development, for instance,

$$\forall t \cdot t \in \mathit{violate} \Rightarrow (\exists i \cdot t \mapsto i \notin \mathit{truth}) \tag{2.4}$$

In context c2, we introduce the notion of transitions between states and an initial state called *root*. We also introduce an axiom that ensures that all other states are reachable from *root*.

$$\mathit{STATES} \subseteq \mathit{cls}(\mathit{transitions})[\{\mathit{root}\}] \cup \{\mathit{root}\} \tag{2.5}$$

We use the transitive closure *cls* from the theory plug-in's standard library. Restricting the states to the subset of states that are reachable is not a problem in explicit model checking. We may exclude problematic states that would be detected by constraint based methods but explicit model checking can only deal with reachable states. The *root* state is an artificially introduced state that is supposed to represent the uninitialized system. For now, we will treat it as a state that preserves all invariants. At a later point in the development, we add some more restrictions to *root*.

Context c3 introduces the state of a model checker, i.e., the model checker is either running or it has terminated with or without reporting an invariant violation. We capture this in three constants: *running*, *terminated_ok* and *terminated_ce*.

2.2.2 Abstract Model Checker

Using c1 to c3 we can now specify an abstract model checker. The machine m0 "cheats" in its specification. It uses global knowledge about the *preserve* and *violate* sets. We use this technique, which is often used in Event-B, to establish the correctness of the result at a high level of abstraction. We then lose the usage of global knowledge in further refinement steps. The model consists of two events: *terminate_ok* which can be observed if the invariant holds in every state, i.e, $\mathit{violate} = \varnothing$ and *terminate_ce*, if there is at least one state that violates the invariant, i.e., $\mathit{violate} \neq \varnothing$. Both events set some state variables: *result* and *counterexample*. The full state space of m0 is shown in Figure 2.4.

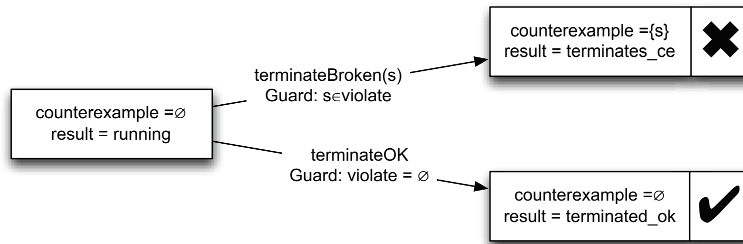


Figure 2.4: State space of the m0 machine

2.2.3 First Refinement: Introducing sequential state processing

In the next refinement m1, we partially introduce the algorithm. We introduce the sets *ok*, *unchecked* and *broken*. The *broken* and *ok* sets are initialized as the empty set, the *unchecked* set will initially contain all elements from *STATES*. We then move each state s using freshly introduced events *checkOK* and *checkBroken*. We move s to the set *ok* if $s \in preserve$. Otherwise, if $s \in violate$, the state is moved to the set *broken*. After we copied either all states to *ok* or at least one state to *broken*, we can observe one of the terminate events which now use *broken* and *ok* in their guards rather than *preserve* and *violate*. Figure 2.5 shows how the effect of *terminateOK* in the abstraction is implemented in the refinement.

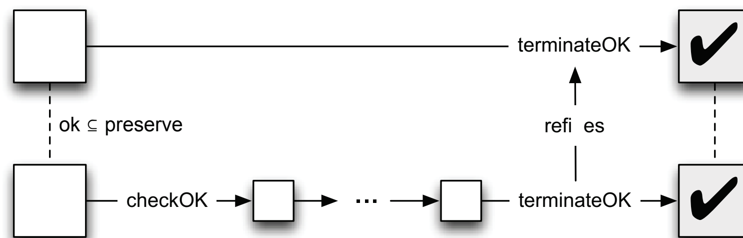


Figure 2.5: Introduce sequential processing

We still require global knowledge for *checkOK* and *checkBroken*. We have to remove this knowledge in the further development, but we already have a basic algorithmic structure.

2.2.4 Second Refinement: Model Checker

In this refinement step, we introduce the model checking algorithm. The c4 context introduces two program counters for the concrete model checking algorithm. One counter is used for the model checking loop, i.e., select a state, check the invariant and compute the successor states. The second counter is used for processing the successors.

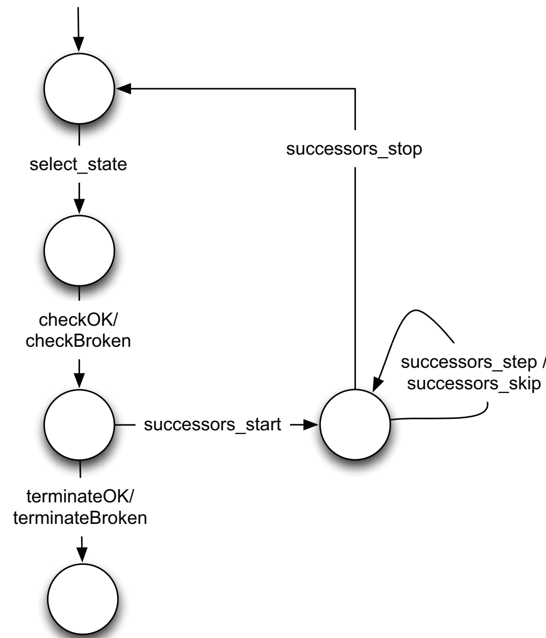


Figure 2.6: Algorithm of m2

An overview of the algorithm is shown in Figure 2.6. The algorithm selects an unprocessed state from a set called *queue* and checks its invariant. Although the set is called *queue*, we do not impose any search strategy, i.e., we non-deterministically choose one state. We can then decide to terminate if one of the terminate events is enabled. Otherwise we process the successor states. The *successors_start* event calculates the set of successors. The events *successors_step* and *successors_skip* process each single successor state. The state is discarded by *successors_skip* if it is either the current state or if it is already in the processing queue. Otherwise the successor state is added to the queue by *successors_step*. If all successor states have been processed, we select the next state from *queue*.

This refinement step is rather big, and we introduce a lot of events that may seem to be unnecessary. We could have introduced the two loops in two refinement steps, but we

have decided to introduce almost the full algorithm in a single step because it allows us to use the same variables in the whole development. If we were to introduce the algorithm in two steps, we would have had to introduce new variables in the refinement. This is a trade off and we decided that the advantage of using the same variables outweighs the complexity of this refinement step.

2.2.5 Third Refinement: Abstract Invariant Processing

In machine m3 we refine the *checkOK* and *checkBroken* events in order to remove the global knowledge from their guards. We store the information about invariant conjuncts in some relations between *STATES* and *INVARIANTS* called *invs*, *checked* and *unchecked*. The *unchecked* relation stores all tuples of state and invariant that we have to check, and the *checked* relation stores all the tuples that we have already found to be true. Any tuple can be at most in one of these two sets. The *invs* relation is the union of the two relations, i.e., $invs = checked \cup unchecked$.

In this step, we refine the successor_step and successor_skip events to put the state-invariant tuples into the sets. The specification of successor_step is the following:

successors_step $\widehat{=}$
refines *successors_step*

any
s
is
where
`pc` : $mcpc2 = step_successors$
`grd1` : $s \in succs$
`grd2` : $s \notin queue$
`grd3` : $s \neq current$
`grd4` : $\forall i \cdot i \in is \Rightarrow s \mapsto i \in truth$
`grd5` : $s \notin dom(invs)$
then
`act1` : $succs := succs \setminus \{s\}$
`act2` : $queue := queue \cup \{s\}$
`act3` : $checked := checked \cup (\{s\} \times is)$
`act4` : $unchecked := unchecked \cup (\{s\} \times (INVARIANTS \setminus is))$
`act5` : $invs := invs \cup (\{s\} \times INVARIANTS)$
end

The actions act1 and act2 and the guards pc, grd1, grd2, and grd3 are exactly the same as in the abstract event. We have added a parameter *is* that is a set of invariants that are true in the state s. In regular model checking, this is always the empty set. However, when we do proof supported model checking we want to remove some of the invariants that are known to be true. We will refine this model into two different versions in the next step, one version where $is = \varnothing$ that represents regular model checking and one version that uses proof to determine the set. We create state-invariant tuples and store them in the corresponding relation. Those tuples that contain an invariant from the set *is* are stored in *checked*; all others are stored in *unchecked*.

The *successor_skip* event is defined in a very similar way. The difference is that we have already observed the *successor_step* event previously and therefore the *unchecked* and *checked* relations already contain the tuples. The *successor_skip* event is allowed to move some tuples from *unchecked* to *checked* if their invariant is true. Again, in the case of regular model checking no tuples are moved. In proof supported model checking this event means that a new incoming event e for the successor state was discovered and the algorithm can discard all invariants that are proven for e, i.e., we compute the intersection of the invariants that are unproven until now and the unproven invariants of e as explained in Section 2.1.

successors_skip $\widehat{=}$
refines *successors_skip*

any
s
is
where
`pc` : $mcpc2 = step_successors$
`grd1` : $s \in succs$
`grd2` : $s \in queue \lor s = current$
`grd3` : $\forall i \cdot i \in is \Rightarrow s \mapsto i \in truth$
`grd4` : $(\{s\} \times INVARIANTS) \subseteq invs$
then
`act1` : $succs := succs \setminus \{s\}$
`act2` : $unchecked := unchecked \setminus (\{s\} \times is)$
`act3` : $checked := checked \cup (\{s\} \times is)$
end

The *checkOk* event is also refined in this step. We replace the guard $current \in preserve$ from the abstraction by $checked[current] = INVARIANTS$, i.e., we only use information that the machine has discovered. Note that we do not check the invariant in this step.

This is done in a new event *check_true_inv* that moves one state-invariant tuple to the *checked* relation if the invariant holds. The guard $current \mapsto i \in truth$ represents the potentially expensive operation to check the invariant in the current state.

check_true_inv $\widehat{=}$

any
  i
where
  `grd2` : $i \in unchecked[\{current\}]$
  `grd3` : $current \mapsto i \in truth$
  `grd4` : $mcpc1 = check_invariant$
then
  `remove1` : $checked := checked \cup (\{current \mapsto i\})$
  `remove2` : $unchecked := unchecked \setminus (\{current \mapsto i\})$
end

Finally, we refined the *checkBroken* event. The abstract event checked if $current \in violate$ whereas the concrete version chooses an invariant such that $\neg(current \mapsto i \in truth)$.

In an implementation, the *check_true_inv* and *checkBroken* events could be combined into an if-then-else statement in order to avoid duplicate computation of the predicate $current \mapsto i \in truth$.

2.2.6 Regular and Proof Supported Model Checking

Regular model checking, i.e., checking each invariant for each state, is a very simple refinement of m3. We only have to refine *successor_step* and *successor_skip*. We add the guard $is = \varnothing$ to both events. The result is that we always put all state invariant tuples into the *unchecked* set, and if we encounter the same state again, we will not change the *checked* and *unchecked* relations. As a result, the only way for a tuple to be moved from the *unchecked* relation to the *checked* relation is by actually checking the truth value using the *check_true_inv* event.

Proof supported model checking was modeled similarly to regular model checking. It is a refinement of m3 where we specify how we calculate the parameter *is* within the guards of the *successor_step* and *successor_skip* events.

The first thing we need to model is the information about discharged proof obligations. For this we have to introduce a labeling function that associates transitions in

the state space with events. The reason is that the information we get from Rodin are tuples of invariant and event that we have to map to states. In context c5 we introduce labels as a total function from *transitions* to $EVENTS$. Furthermore, we need the information about discharged proof obligations. We store the information about discharged proof obligations in a total function $discharged \in EVENTS \rightarrow \mathbb{P}(INVARIANTS)$. Invariants in the set $discharged(e)$ are proven to be preserved by the event e. We also define a complementary function $single_specialized_invariant$ of the same type. The set $single_specialized_invariant(e)$ is the set of invariants that are not known to preserve the invariant, i.e., for any event e the predicate $partition(INVARIANTS, discharged(e), single_specialized_invariant(e))$ holds.

Proof supported model checking is captured in the following axiom. If there is an event e that leads from state s to state t, s preserves the invariant, and we have discharged the invariant preservation proof obligation for an invariant i, then i also holds in state t.

$$\forall s, t, e, i \cdot (s \mapsto t) \mapsto e \in labels \land i \in discharged(e) \land s \in preserve \Rightarrow t \mapsto i \in truth$$

Membership of an invariant in $discharged(e)$ means that the proof obligation 1.1 has been discharged, i.e. $I(s, c, v) \land G(s, c, v, x) \land BA(s, c, v, x, v') \land A(s, c) \Rightarrow inv(s, c, v')$ is a tautology. If we change the notation to fit our model, we get $s \in preserve \land s \mapsto t \in transitions \land (s \mapsto t) \mapsto e \in labels \land A(s, c) \Rightarrow t \mapsto i \in truth$.

The axiom we proposed captures almost the entire meaning of a discharged proof obligation with the exception that one necessary condition is missing. In order to use the proof information, we have to be sure that the axioms and theorems from the contexts hold, i.e., there must be some s,c for which $A(s, c)$ is true.

We cannot deduce this from our formal model. However, in the implementation this will not be a problem because ProB tries to find a model for the axioms. If it can find a model, we have a proof for $\exists s, c \cdot A(s, c)$. In Section 2.2.7, we will discuss some more details that are missing in our formal model, and we will show why the formal model is still reasonable.

Another important thing that we have to consider before we can model proof supported checking is the root state. Previously we only cared about a single state at a time when we computed if the invariants hold in that state. That means the root state could not influence the outcome of the model checking. We only required that root itself is not a counter-example. Proof supported model checking also has to take the predecessor state into consideration. This means a direct successor of root could potentially be influenced

by the root state. In order to avoid these kind of problems, we require that all outgoing transitions of the *root* state are labeled with the event *initialization*, i.e.:

$$labels[\{root\} \lhd transitions] = \{initialization\}$$

The *initialization* event cannot depend on the previous state so if an *initialization* proof obligation is discharged, we can safely use it without having to care about the predecessor state.

We can now specify a refinement that uses the information about discharged proof obligations. We need to refine *successor_step* and *successor_skip* again. We remove the abstract parameter *is* and add the event e as a parameter. The witness in both cases is $discharged(e) = is$.

The *successor_step* event only adds those invariants to the *unchecked* relation that are not known to be true. Invariants that are known to be true should directly be stored in the *checked* relation avoiding the potentially expensive check in the *check_true_inv* and *checkBroken* events.

successors_step $\widehat{=}$
refines *successors_step*

any

s

e

where

`pc` : $mcpc2 = step_successors$

`grd1` : $s \in succs$

`grd2` : $s \notin queue$

`grd3` : $s \neq current$

`grd5` : $s \notin dom(invs)$

`grd6` : $current \in ok$

`grd7` : $(current \mapsto s) \mapsto e \in labels$

with

`is` : $discharged(e) = is$

then

`act1` : $succs := succs \setminus \{s\}$

`act2` : $queue := queue \cup \{s\}$

`act3` : $checked := checked \cup (\{s\} \times discharged(e))$

`act4` : $unchecked := unchecked \cup (\{s\} \times single_specialized_invariant(e))$

`act5` : $invs := invs \cup (\{s\} \times INVARIANTS)$

end

2.2.7 Theorems and Well-Definedness

The models m3 and m4 are abstractions of the model checking process. They contain the parts that are important in the context of this thesis. However, they abstract away some parts that we have to take into account when implementing the approach. In particular, they do not have a notion of theorems and well-definedness.

It is possible in ProB to switch off theorem checking because the theorems are supposed to be implied by the invariants. In the context of proof supported model checking, turning theorem checking off can lead to incorrect results. An example is the following machine

MACHINE m0
VARIABLES
    x
INVARIANTS
    thm1 : $x > 10$
    `inv` : $x > 9$
EVENTS
Initialisation
    begin
        `act` : $x := 20$
    end
dec $\widehat{=}$
    begin
        `act1` : $x := x - 1$
    end
END

The theorem thm1 cannot be proven, yet it can be used to prove invariant preservation for the *dec* event. If $x > 10$ was true then $x - 1 > 9$ would also be true after executing *dec*. If we do not check the theorem, ProB would eliminate all checks.

A solution is to automatically check theorems if proof supported model checking is enabled. However, if theorems are proven they can be eliminated.

The second thing that is not modeled is well-definedness. We can produce a model where all invariant preservation proof obligations are discharged, but a well-definedness (WD) proof obligation cannot be proven.

MACHINE wd
VARIABLES
x
INVARIANTS
`inv1` : $x \in \{0, 1\}$
`inv2` : $1/x \neq 0 \lor x = 0$
EVENTS
Initialisation
begin
`act1` : $x := 1$
end
set $\widehat{=}$
any
y
where
`grd1` : $y \in \{0, 1\}$
then
`act1` : $x := y$
end
END

However, this is not a problem because ProB checks well-definedness separately; it will detect if there are WD problems in the model.

2.3 Implementation of Proof Supported Model Checking

Using the formal model, we can specify an algorithm for proof supported model checking as shown in Figure 2.7. We tried to be close to the formal model, but we changed some of the data structures. For instance, some of the bookkeeping data structures like *invs* or *checked* are not required.

```
queue := {root};
known := ∅;
ok := ∅;
unchecked = {root ↦ ∅}

// Precomputing the specialized invariants
for evt ∈ events do
    specialized_inv(evt) := {inv_i | inv_i not proven for evt};

// Model Checking Loop
while queue ≠ ∅ do
    current := take_from(queue) // select_state
    queue := queue \ {current}

    for i ∈ unchecked(current)
        if i is false //checkBroken
          then return counter-example(current);  // terminateBroken
          else skip //   check_true_inv
    // checkOK
    ok := ok ∪ {current} // mark as processed

    foreach succ, evt s.t. current →evt succ do
        if succ ∈ ok then continue; // already processed
        if succ ∉ known then // successor_step
            queue := queue ∪ {succ}
            known := known ∪ {succ}
            unchecked(succ) := single_specialized_inv(evt)
        else   // successor_skip
            unchecked(succ) := unchecked(succ) ∩ specialized_inv(evt)
od

return no-counter-example    // terminateOk
```

Figure 2.7: Proof Supported Model Checking

The structure of the algorithm is very similar to ProB's model checking algorithm published in [28, Algorithm 5.1]. Our algorithm does not specify how successor states are selected, but we are more specific about how the check for errors works while the original algorithm is very abstract with respect to error checking.

The algorithm precomputes the subset of invariants that are not proven to be preserved

for each event of the model. Then, while there are unprocessed states we select one of them, check the invariant which is associated with that state, and terminate if we find an invariant violation.

The loop containing the if-then statement combines *checkBroken*, *check_true_inv*, *terminateBroken* and *checkOK* from the formal model. The test in this if-then statement is basically *checkBroken*. If the test succeeds, we know that we have a counterexample and we will observe *terminateBroken*. The actual implementation would return a trace to the state that violates the invariant. This is slightly different to the formal model, which will not terminate and will even allow all the other events to occur except for *terminateOK*.

If the test fails, we move on to the next invariant. This is the *check_true_inv* event from the formal model. Finally, when the loop terminates we know that the complete invariant holds in this state. This corresponds to *checkOK* in the formal model; we put the state into the set *ok* which contains all verified states.

After the invariant checking, we process each successor state. First, we check if the successor has already been processed. If this is the case, we skip it. In the formal model removing the checked states is done in the *successors_start* event. The set of successors is $transitions[current] \cap unknown$, i.e., it can only contain unchecked states.

We then test if the state has not yet been seen before. In that case, we mark the state as seen, add it to the processing queue and assert the specialized invariant for the event that led to the state. In the formal model this is done in the *successors_step* event. Otherwise, if the state has been seen before, there must be another incoming edge that we have processed before. We replace the stored information with the intersection of the stored set of invariants and the specialized invariants for the current event. This is *successors_skip* in the formal model.

When all states have been processed without an invariant violation, we report that no error has been found and terminate. This is equivalent to *terminateOK* in the formal model.

2.4 Missing Proof Obligations

When we implemented the translation for ProB, we noticed that Rodin does not generate all required proof obligations. Some obligations, i.e., those related to typing invariants, are considered trivial by the proof obligation generator. The POG does not

generate these obligations. This is sound because passing the static checker already is a proof for these proof obligations. However, because ProB only removes proven invariants, the typing invariants were always kept. We decided to reconstruct the proof obligations within ProB and automatically mark them as discharged.

2.5 Application in classical B

In this work, we have focused on Event-B. This is because the interaction with the Rodin prover is very easy. However, in principle the method can be applied to other formal specification languages with similar proof obligations in the same way. The obvious candidate is classical B. The current implementation does not incorporate a prover for classical B, but the automatically reconstructed proof obligations are used even in classical B.

2.6 Experimental results

The following section is copied almost verbatim from the original paper [36]. The version of ProB used for the experiments in Table 2.1, 2.2 and 2.4 is rather old (ProB 1.3.0-final.4, Subversion revision 2874). The mesurements were performed on an Apple Mac Book Pro, 2.4 GHz Intel Core 2 Duo Computer with 4 GB RAM running Mac OS X 10.5. We carried out single refinement level and multiple refinement level checks. For the single level animation, we collected 40 samples for each model and calculated the average and standard deviation of the times measured in milliseconds. For the multi level animation, we collected 5 samples for each model. Unless explicitly stated, we removed all interactive proofs and only used the default autoprovers on each model.

Table 2.3 shows some benchmarks performed with a more recent version (ProB 1.5.0-beta2, git tag bm_pomc). The experiment was carried out on an Apple Mac Book Pro, 2.5 GHz Intel Core i5 Computer with 8 GB RAM running Mac OS 10.9.5. The option to check a single refinement level has been removed in more recent versions of ProB, so we only provide results for multiple refinement level checks. The difference between Table 2.2 and 2.3 are due to both, a faster computer and better performance of ProB.

To verify that the combination of proving and model checking results in a considerable reduction of model checking effort, we prepared an experiment consisting of specifications we got from academia and industry. In addition, we constructed an example for a

case where the prover has a very high impact on the performance of the model checker. The Model Checker model in Table 2.3 is the formal model of proof supported model checking from Section 2.2.

The rest of this section describes how we carried out the measurement. We will also briefly introduce the models and discuss the result for each of them. The experiment contains models where we expected to have a reasonable reduction and models where we expected to have only a minor or no impact.

2.6.1 Mondex

The mechanical verification of the Mondex Electronic Purse was proposed for the repository of the verification grand challenge in 2006. We use an Event-B model developed at the University of Southampton [37]. We have chosen two refinements from the model, m2 and m3. The refinement m2 is a rather big development step while the second refinement m3 was used to prove convergence of some events introduced in m2. The machine m3 only contains gluing invariants.

In case of single refinement level checking, it is obvious that it is not possible to further simplify the invariant of m3 but we noticed that we do not even lose performance caused by the additional specialization of the invariants. This is important because it is evidence that our implementation's performance is in the order of the standard deviation in our measurement. For the case of m2, where we have machine invariants, we measured a reduction of about 12%.

In case of multiple refinement level checking, we have a rare case where we lost a bit of performance for m2. However, the absolute value is in the order of the standard deviation. For m3 we also did not get significant improvements of performance, most likely because the gluing invariant is very simple and only contains simple equalities.

2.6.2 Siemens Deploy Mini Pilot

The Siemens Mini Pilot was developed within the DEPLOY Project. It is a specification of a fault-tolerant automatic train protection system, that ensures that only one train is allowed on a part of a track at a time. The Siemens model shows a very good reduction because the invariants are rather complex. This model only contains a single machine, thus multi level refinement checking does not affect the speedup.

2.6.3 Scheduler

This model is an Event-B translation of the scheduler from [38]. The model describes a typical scheduler that allows a number of processes to enter a critical section. The experiment has shown that the improvement using proof information is rather small, which was not a surprise. The model has a state space that grows exponentially when increasing the number of processes. It is rather cheap to check the invariant

$$ready \cap waiting = \varnothing \wedge active \cap (ready \cup waiting) = \varnothing \wedge active = \varnothing \Rightarrow ready = \varnothing$$

because the number of processes is small compared to the number of states. Nevertheless, we save a small amount of time in each state and these savings can add up to a reasonable speedup. The scheduler also contains only a single level of refinement.

2.6.4 Earley Parser

The model of the Earley parsing algorithm was developed and proven by Abrial. As with the mondex example, we used two refinement steps that have different purposes. The second refinement step m2 contains a lot of invariants, while the m3 contains only very few of them. This is reflected in the savings we gained from using the proof information in the case of single refinement level checking. While m3 showed practically no improvement, in the m2 model the savings add up to a reasonable amount of time. In the case of multiple refinement level checking, the result are very different. While m2 is not affected, the m3 model benefits a lot. The reason is that it contains several automatically proven gluing invariants.

2.6.5 SAP DEPLOY Mini Pilot

As with the Siemens model, this is a DEPLOY pilot project. It is a model of a system that coordinates transactions between seller and buyer agents. In the case of single refinement level, we gain a very good speedup from using proof information, i.e., model checking takes less than half of the time. As in the Siemens example, the model contains rather complicated invariants. In case of the multi refinement level checking, the speedup is still good, but it is not as impressive as in single refinement level checking.

2.6.6 SSF DEPLOY Mini Pilot

The Space Systems Finland example is a model of a subsystem used for the ESA Bepi-Colombo mission to Mercury. The start of the BepiColombo spacecraft is planned for 2016. The model is a specification of the parts of the BepiColombo On-Board software that contains a core software and two subsystems used for tele command and telemetry of the scientific experiments, the Solar Intensity X-ray and particle Spectrometer (SIXS) and the Mercury Imaging X-ray Spectrometer (MIXS). The time for model checking was reduced by 7% for a single refinement level and by 16% for multiple refinement checking.

2.6.7 Cooperative Crosslayer Congestion Control CXCC

CXCC [39] is a cross-layer approach to prevent congestion in wireless networks. The key concept is that, for each end-to-end connection, an intermediate node may only forward a packet towards the destination after its successor along the route has forwarded the previous one. The information that the successor node has successfully retrieved a package is gained by active listening. The model is described in [40]. The invariants used in the model are rather complex and thus we get a good improvement by using the proof information in both cases.

2.6.8 Constructed Example

The constructed example is mainly to show a case where we get a huge saving from using the proofs. It basically contains an event that increments a number x and has an an invariant $\forall a, b, c \cdot a \in \mathbb{N} \wedge b \in \mathbb{N} \wedge c \in \mathbb{N} \Rightarrow (a = a \wedge b = b \wedge c = c \wedge x = x)$. Because the invariant contains the variable modified by the event, we cannot simply remove it. However, Rodin can automatically prove that the event preserves the invariant, thus our tool is able to remove the whole invariant. Without proof information, ProB needs to enumerate all possible values for a,b and c which results in an expensive calculation.

	w/o proof information [ms]	**using proof information [ms]**	**Speedup Factor**
Mondex m3	1454 ± 5	1453 ± 5	1.00
Earley Parser m3	2803 ± 8	2776 ± 7	1.01
Earley Parser m2	140310 ± 93	131045 ± 86	1.07
SSF	31242 ± 64	29304 ± 44	1.07
Scheduler	9039 ± 15	8341 ± 14	1.08
Mondex m2	1863 ± 7	1665 ± 6	1.12
Siemens (auto proof)	54153 ± 50	25243 ± 22	2.15
Siemens	56541 ± 57	26230 ± 28	2.16
SAP	18126 ± 18	8280 ± 14	2.19
CXCC	18198 ± 21	6874 ± 12	2.65
Constructed Example	18396 ± 26	923 ± 8	19.93

Table 2.1: Experimental results (single refinement level check)

	w/o proof information [ms]	**using proof information [ms]**	**Speedup Factor**
Mondex m2	1747 ± 21	1767 ± 38	0.99
Mondex m3	1910 ± 20	1893 ± 6	1.01
Earley Parser m2	309810 ± 938	292093 ± 1076	1.06
Scheduler	9387 ± 124	8167 ± 45	1.15
SSF	35447 ± 285	30590 ± 110	1.16
SAP	50783 ± 232	34927 ± 114	1.45
Earley Parser m3	7713 ± 40	5047 ± 15	1.53
Siemens (auto proof)	51560 ± 254	24127 ± 93	2.14
Siemens	51533 ± 297	23677 ± 117	2.18
CXCC	18470 ± 151	6700 ± 36	2.76
Constructed Example	18963 ± 31	967 ± 6	19.61

Table 2.2: Experimental results (multiple refinement level check)

	w/o proof information [ms]	**using proof information [ms]**	**Speedup Factor**
Earley Parser m2	51154 ± 151	50712 ± 408	1.01
Model Checker (full)	96686 ± 197	64678 ± 39	1.49
Model Checker (auto)	96864 ± 225	86620 ± 524	1.12
Siemens	17340 ± 406	11244 ± 45	1.54
Scheduler	6752 ± 39	6546 ± 16	1.03

Table 2.3: Experimental results (more recent ProB)

	w/o Proof [#]	w Proof [#]	Savings [%]
Earley Parser m2	–	–	-
Mondex m3	440	440	0
Earley Parser m3	540	271	50
Constructed Example	42	22	50
SAP	48672	16392	66
Scheduler	20924	5231	75
Mondex m2	6600	1560	76
SSF	24985	5009	80
CXCC	88480	15368	83
Siemens	280000	10000	96
Siemens (auto proof)	280000	10000	96

Table 2.4: Number of invariants evaluated (single refinement level check).

2.7 Using proof information for directed model checking

Proof information can in principle be used to direct the model checker in order to minimize its work when verifying a model or in order to optimize the search when trying to find an invariant violation. We define two modes of operation, a *proof mode* where we want to prove that the invariant holds in a finite set of states and a *debug mode* where we try to find a counterexample. We have not implemented these modes in ProB, but in principle implementing them seems to be doable with a reasonable amount of work.

2.7.1 Proof Mode

If we want to verify the correctness of a model, we want to reduce the total amount of work as much as possible. For this reason, we want to defer the checking of each state until we have a good knowledge about incoming edges. We use the observation that if we have multiple events leading to a state, we only need to check the intersection of the sets of unproven invariants for these events. Instead of ProB's mixed depth and breadth first search, we can do a best first search using the number of remaining invariants as the heuristic. We choose the state with the minimal number of unproven invariants. This approach does not reduce the number of states but can significantly reduce the number of invariant evaluations.

2.7.2 Debug Mode

If we are trying to find counterexamples, the proof mode is only of little help because it prefers states that are almost (or even fully) proven. Instead, we want to select states that are most likely broken. The heuristic for debug mode is therefore the exact opposite of proof mode. We select the states with the most unproven invariants and hope that we hit a state that violates the invariant. This strategy does not necessary improve the result, but it might improve performance in some cases and is probably not worse than random search.

2.8 Related work

There have been several approaches to combine model checking and proof most of which are far more ambitious than our approach. In previous work [41] we have already shown how a model checker can be used to complement the proving environment by acting as a disprover. In [41] it was also shown that sometimes the model checker can be used as a prover, namely when the underlying sets of the proof obligation are finite. More recently we have changed the disprover implementation to use constraint solving rather than model checking. ProB can track enumeration of infinite sets during constraint solving and is therefore able to tell if the absence of a counterexample is a proof [18]. An experimental plug-in for Rodin exists that is surprisingly efficient. In particular in cases where a manual proof is very repetitive, e.g., if a manual proof would contain many case distinctions.

In [42] ProB was used to prove some theorems for the model of an Hamming encode/decoder that were difficult to prove using the normal provers. Also in this case ProB worked well with a large number of case distinctions. The authors applied ProB manually, today they could probably use the ProB prover instead.

Outside the B community there are also several approaches that combine model checking and theorem proving. For instance, in [43] Müller and Nipkow used theorem proving to reduce infinite or large state spaces to small finite cases that could then be verified using model checking. This combines the strengths of model checking and theorem proving: “model checking is automatic but limited to finite state processes, theorem proving requires user interaction but can deal with arbitrary processes” [43]. The authors apply the method to the Alternating Bit Protocol [44, 45] with an unbounded channel. The unbounded channel is different from model checking based approaches which typically

use a bounded channel with a small capacity. The authors use the theorem prover to create a finite abstraction for an infinite implementation and then they use model checking to verify that this abstraction is correct with respect to a specification, i.e., they show that all traces that are possible in the implementation are also possible in the specification.

Shankar argues "for a specific combination where theorem proving is used to reduce verification problems to finite-state form, and model checking is used to explore properties of these reductions" [46]. The Symblic Analysis Laboratory (SAL) is based on that approach.

In [47] Pnueli and Shahar used deduction in combination with model checking. The system proposed by the authors works on top of the symbolic model checker CMU SMV. T Deduction is used to derive invariants which are used "to restrict the range of the transition function in computing the backwards closure, usually employed in model checking for invariance properties" [47]. The method was successfully applied to partially verify the IEEE Futurebus+ system [48] and the authors found a bug during their work that was not discovered previously.

Prioni [49] is a tool that integrates the Alloy Analyzer (AA) and the theorem prover Athena. The authors explain the use case as follows.

> "The user starts from an Alloy specification, model-checks it and potentially changes it until it holds for as big a scope as AA can handle. After eliminating the most obvious errors in this manner, the user may proceed to prove the specification. This attempt may introduce new proof obligations, such as an inductive step. The user can then again use AA to model-check these new formulas to be proved. This way, model checking aids proof engineering. But proving can also help model checking" [49]

The approach is very similar to the way we use the combination of proof and model checking within Rodin. However, it seems to be more a tool to support a user switching back and forth between model checking and proving than a deep integration of the verification methods.

There are several other approaches that loosely combine multiple verification techniques including model checking and proving to verify systems [50, 51].

Chapter 3

Automatic Flow Analysis

In this part of the thesis, we define a theoretical framework that allows us to analyze a model to find an answer to questions like "Can event h take place after event g was observed? If so, under which circumstances?".

We derive the notion of event independence, of the enable graph, and of the flow graph. We propose some applications in model checking. Section 3.7 describes how we can exploit the flow analysis to reduce the effort of model checking by reducing the number of guards that need to be checked. In Section 3.13 we describe another potential application of the flow analysis to direct the model checker and to generate code. This work has been published in [52]

3.1 Nondeterministic Assignments

In the rest of the chapter, we will always assume that the model might contain non-determinism in the parameters but does not contain any non-deterministic assignments. This is necessary because we need to have control over execution in order to derive precise information. This does not impede the generality of the approach because we can mechanically transform an event containing a non-deterministic assignment into an event where the non-determinism has been lifted into the parameters.

For the non-deterministic assignment from a set of values, $x :\in S$, we introduce a fresh variable $freshx$ and add $freshx \in S$ to the guards. The assignment becomes $x := freshx$. An example result of the transformation of an event with no guards and $x :\in \mathbb{N}$ as the only action is shown in Figure 3.1.

A Becomes-such-that assignment $v_1, ..., v_k :| P(v_1, ..., v_k, v'_1, ..., v'_k)$ can also be lifted. For each variable that is assigned, we introduce a fresh variable and add P to the guard

g $\widehat{=}$

any
$freshx$
where
`grd1` : $freshx \in \mathbb{N}$
then
`act1` : $x := freshx$
end

Figure 3.1: Lifting $x :\in \mathbb{N}$

replacing the primed variables by the fresh variables. The actions assign each variable to their fresh counterpart. An example with two variables is shown in Figure 3.2.

g $\widehat{=}$

any
$freshx$
$freshy$
where
`grd1` : $freshx = y + x \land freshy = y - x$
then
`act1` : $x := freshx$
`act1` : $y := freshy$
end

Figure 3.2: Lifting $x, y :| x' = y + x \land y' = y - x$

3.2 Total substitution lifting

We can define another transformation that practically moves all information the event contains into its guard leaving only trivial substituitions:

Definition 3.1 (Totally Lifted Form)**.** *An event is in the totally lifted form (TLF), if it only contains actions of the form $v := x$, where v is a global variable and x is a local variable of the event introduced using* **any***.*

A deterministic assignment $v := Expr(...)$ is transformed similar to Figure 3.1 by introducing a fresh variable $freshv$ and adding $freshv = Expr(...)$ to the guard. The assignment becomes $v := freshv$. Note that in some cases we have to rewrite the expression. For instance, if the assignment is $f(x) := f(x) + 1$, we have to rewrite it to $f := f \Leftarrow \{x \mapsto f(x) + 1\}$ before we can move it into a guard. This is shown in Figure 3.3.

g $\widehat{=}$

    any
        $freshf,\ freshx$
    where
        `xform1` : $freshf = f \Leftarrow \{x \mapsto f(x) + 1\}$
        `xform2` : $freshx = x + 1$
    then
        `act1` : $f := freshf$
        `act2` : $x := freshx$
    end

Figure 3.3: Lifting $f(x) := f(x) + 1 \parallel x = x + 1$

We can also define a normal form that is the opposite of the totally lifted form. Intuitively, we inline as many parameters as possible without introducing non-deterministic assignments.

Definition 3.2 (Deterministic Inlined Form)**.** *An event is in the deterministic inlined form (DIF), if it only contains deterministic assignments and it is not possible for any parameter to be inlined into the actions.*

The deterministic inlined form of event g from Figure 3.3 is the original assigment, maybe in a slightly different form, e.g. we would get $f := f \Leftarrow \{x \mapsto f(x)+1\}$ instead of $f(x) := f(x)+1$. Another example is shown in Figure 3.4. In this example, the parameter x can

g $\widehat{=}$

any
$x,\ y$
where
`grd1` : $x = q + 1$
`grd2` : $y \in \mathbb{N}$
then
`act1` : $q := x + y$
end

g' $\widehat{=}$

any
y
where
`grd2` : $y \in \mathbb{N}$
then
`act1` : $q := (q + 1) + y$
end

Figure 3.4: Example transformation into the deterministic inlined form

be inlined without introducing a non-deterministic assignment. However, for parameter y this is not possible.

All the transformations can be applied automatically and transparently by a tool, i.e., the user can continue to model in the style he or she prefers, but a tool such as the flow analysis can transform the events and use the transformed version.

Both normal forms will be used in this thesis. We will discuss the differences between the results we get from using DIF versus TLF in Section 3.3.4.

3.3 Independence of Events

As an example, let's say we have an event g that has an action $x, y := (x + 1), 0$:

1. If the guard of an event h is $y = 1$ then executing h after g was observed is impossible because we know that y will always be 0.
2. If the guard of h is $y = 0$ then h can always be executed after g was observed for the same reason.
3. If the guard of h is $x = 12$ then we can sometimes execute h after g was observed, i.e., only if x was 11 in the state before g happened.

4. Finally, if the guard of h is $z = 42$ executing g has no influence at all because g does not change the value of z.

In the last case, we say that h is independent from g. Informally said, if h is independent from g this means that whatever g's action does, it cannot have any influence on the truth value of h's guard. We distinguish two kinds of independence: trivial and non-trivial independence.

3.3.1 Trivial Independence

Trivial independence is a syntactic property of a model. We can compute trivial independence by analyzing the read and write sets of the events.

Definition 3.3 (read/write sets)**.** *Let g be an event in deterministic inlined form with guard G and action S. Let V be the set of state variables. The read set of an event g, noted $read(g)$, is the set of global variables such that*

$$read(g) = \{v \mid v \in V \wedge v \textit{ is free in } G\}$$

The write set of an event, noted $write(g)$, is the set set of global variables such that

$$write(g) = \{v \mid v \in V \wedge S \textit{ assigns a value to } v\}$$

If the event does not have a guard, then $read(g) = \varnothing$. If it does not have an action, then $write(g) = \varnothing$. The event can have parameters, but they are not considered in the read/write sets.

Note that we do not include the variables that are read by the action of the event because for now, we are only concerned whether an event can be executed or not. This means that we currently only care about the truth value of the guards of an event.

Also note that the event is in deterministic inlined form, this means we minimize the number of variables in *read* while still only using deterministic assignment. We will discuss the reason for this choice in Section 3.3.4.

Using the information about variables used by the events, we can define a relation describing which event can possibly have an effect on another event.

Definition 3.4 (Effect relation)**.** *The **effect relation** η between events g and h is defined as*

$$\eta = \{init\} \times Events \cup \{g \mapsto h \mid g, h \in Events \land read(h) \cap write(g) \neq \varnothing\}$$

where Events is the set of events defined in the model including the initialization, init is the initialization, and read and write are the read/write sets.

Although this is different from [11, p. 43], we assume that initialization is executed only once. We can easily regain the behavior from [11, p. 43] by introducing a second, regular event that has the same action as the initialization and no guard.

For every event, we define that that the initialization always has an effect on it; this will simplify the construction of flow graphs in Section 3.9.

Note that we do not require $g \neq h$. An event may or may not have an effect on itself.

Using η, we can define the trivial independence. It is just the complement of η.

Definition 3.5 (Trivial Independence)**.** *If $(g, h) \notin \eta$ we say that h is trivially independent from g.*

We have already seen an example of trivial independence in the beginning of this section. If event g has an action $x, y := (x+1), 0$ and event h's guard is $z = 42$ then h is trivially independent from g. The read set of h is $\{z\}$, the write set of g is $\{x, y\}$, therefore $read(h) \cap write(g) = \varnothing$ and $g \mapsto h \notin \eta$.

3.3.2 Non-Trivial Independence

While $(g, h) \notin \eta$ is sufficient for independence, it is not necessary. Take for instance the events as shown in Figure 3.5. If we compute the read and write sets for g and h, we get

$$\begin{aligned} read(g) &= \varnothing \\ write(g) &= \{x, y, z, q\} \\ read(h) &= \{x, y\} \\ write(h) &= \{z\} \end{aligned}$$

Event g clearly modifies variables that are read by h. That means $(g, h) \in \eta$ but g cannot enable or disable h because the guard of h is invariant under g's substitution. Obviously we can prove $x + y > 5 \vdash (x + 1) + (y - 1) > 5$.

We can use this observation to define a more general notion of independent events.

Initialisation

begin
`init` : $x, y, z, q := 2, 2, 1, 1$
end

g $\widehat{=}$

begin
`act1` : $x := x + 1$
`act2` : $y := y - 1$
`act3` : $z := z * 2$
`act4` : $q := 2$
end

h $\widehat{=}$

when
`grd1` : $x + y > 5$
then
`act1` : $z := z/q$
end

Figure 3.5: Independent events where $(g, h) \notin \eta$

Definition 3.6 (Weak Independence of events). *Let g and h be events in deterministic inlined form, furthermore let $h \neq init$ and $g \neq init$. We say that h is weakly independent from g — denoted by $g \not\rightsquigarrow_w h$ — if the guard of h is invariant under the substitution of g, i.e., iff the following holds:*

$$I(s, c, v) \wedge G(s, c, v, x) \wedge BA(s, c, v, x, v') \wedge A(s, c) \Rightarrow \exists y' \cdot (H(s, c, v, y) \Leftrightarrow H(s, c, v', y'))$$

where H denotes the guard of event h. y and y' are the local parameters of h.

In this definition we only care about the truth value of H, for now, we do not require that H is true using the same local parameters. Later, we will refine this definition to also consider the local parameters.

Lemma 3.7. *If an event h is trivially independent from an event then $g \not\rightsquigarrow_w h$ holds.*

Proof Let h be trivially independent from g. This means $read(h) \cap write(g) = \varnothing$. Thus for all variables $v_i \in read(h)$ the before-after predicate is $v_i' = v_i$ because they

cannot be changed by g. Without loss of generality, let $read(h) = \{v_i \mid 0 < i \leq k\}$, for some $0 \leq k$. We sort the variables and start with those that are used in the guard of h. We start from Definition 3.6, and we have to prove that there are some parameters y' that make the following goal true

$$true \vdash I(s,c,v) \land G(s,c,v,x) \land BA(s,c,v,x,v') \land A(s,c) \Rightarrow \exists y' \cdot (H(s,c,v,y) \Leftrightarrow H(s,c,v',y'))$$

Deduction yields

$$I(s,c,v) \land G(s,c,v,x) \land BA(s,c,v,x,v') \land A(s,c) \vdash \exists y' \cdot (H(s,c,v,y) \Leftrightarrow H(s,c,v',y'))$$

The guard $H(s,c,v,y)$ cannot contain variables v_i where $i > k$, so we can write $H(s,c,v_1,...,v_k,y)$ instead.

$$\begin{aligned} &I(s,c,v) \land G(s,c,v,x) \land BA(s,c,v,x,v') \land A(s,c) \vdash \\ &\quad \exists y' \cdot (H(s,c,v_1,...,v_k,y) \Leftrightarrow H(s,c,v'_1,...,v'_k,y')) \end{aligned}$$

We can now split the before-after predicate S. We have simple equivalences for variables that are not modified by g, i.e., $v'_1 = v_1 \land \ldots \land v'_k = v_k$ and one predicate $S_{>k}$ that contains the rest of the before-after predicate.

$$\begin{aligned} &I(s,c,v) \land G(s,c,v,x) \land S_{>k}(s,c,v,x,v') \land v'_1 = v_1 \land \ldots \land v'_k = v_k \land A(s,c) \vdash \\ &\qquad \exists y' \cdot (H(s,c,v_1,...,v_k,y) \Leftrightarrow H(s,c,v'_1,...,v'_k,y')) \end{aligned}$$

We can use the hypothesis $v'_1 = v_1 \land \ldots \land v'_k = v_k$ in the goal, yielding

$$\begin{aligned} &I(s,c,v) \land G(s,c,v,x) \land S_{>k}(s,c,v,x,v') \land v'_1 = v_1 \land \ldots \land v'_k = v_k \land A(s,c) \vdash \\ &\qquad \exists y' \cdot (H(s,c,v_1,...,v_k,y) \Leftrightarrow H(s,c,v_1,...,v_k,y')) \end{aligned}$$

If we choose $y' = y$, the goal is a tautology.

The proof for the negation of the guard follows the same schema: replace H by $\neg H$. □

The trivial independence, i.e., $(g,h) \notin \eta$, can be decided by simple static analysis, i.e., by checking if $read(h) \cap write(g) = \varnothing$. Non-trivial independence is a much harder problem. In general, proving independence is undecidable, but in real world cases it is still a good idea to try to prove independence using automatic provers. If a proof can be found we increased our knowledge about the system. If h is independent from g, but

we cannot find a proof, only reduces precision of our analysis but does not impede the soundness of the methods we present in the rest of the chapter.

3.3.3 Application: Guard Evaluation

The exploration of a state during model checking in ProB happens in two phases. First, the state is discovered as the result of an event occurring. In this first step, ProB computes the state variables, but it neither checks if the invariant holds nor does it compute the events that can be observed. This is done in a second step, which also discovers the successor states. In Chapter 2, we have shown how the costs of invariant checking can be reduced. In this section, we describe how we can reduce the costs of guard evaluation for independent events.

Assuming we have an event g that led us from state v to v', and we have an event h that is independent from g, then we can skip guard evaluation for h in v' because we already know the result. The event g cannot change the truth value of h's guard, therefore the guard of h is true in state v' if and only if it is true in state v.

Here we need the fact that in Definition 3.4 we included $\{init\} \times Events$ in η. This is the base case for the inductive argument. We need to enforce at least one guard evaluation in the beginning.

The problem is that Definition 3.6 allows using different local variables, which means that we still have to evaluate the guard in order to find a solution for y' although we already know that the guard is true.

Therefore, we need a stronger notion of independence than Definition 3.6. If we demand that the local variables cannot change, i.e., $y' = y$, we can reuse the solution from v for v'.

Definition 3.8 (Strong Independence of events)**.** *Let g and h be events in totally lifted form, furthermore let $h \neq init$ and $g \neq init$. We say that h is strongly independent from g — denoted by $g \not\leftrightarrow_s h$ — if the following holds:*

$$I(s,c,v) \wedge G(s,c,v,x) \wedge BA(s,c,v,x,v') \wedge A(s,c) \Rightarrow (H(s,c,v,y) \Leftrightarrow H(s,c,v',y))$$

Note that strong independence implies weak independence.

```
g ≙
    begin
        act1 : x := x + 1
    end
h ≙
    any
        y
    where
        grd1 : y = x
    then
        skip
    end
```

```
g ≙
    any
        fx
    where
        grd1 : fx = x + 1
    then
        act1 : x := fx
    end
h ≙
    any
        y
    where
        grd1 : y = x
    then
        skip
    end
```

Figure 3.6: Example demonstrating the difference between Definition 3.6 and 3.8.

This is a stricter notion of independence than the notion in Definition 3.6. Here we not only care if the truth value of the guard is the same. We also demand that the truth value of the guard is the same using the same values for the parameters.

An example that shows that Definition 3.6 and 3.8 yield different results is shown in Figure 3.6. On the left hand side the event g is in DIF, on the right hand side g is in TLF. Let us first examine if h is trivially independent from g. In both cases, we can see that $write(g) = x$ and $read(h) = x$ and therefore $write(g) = x \cap read(h) \neq \varnothing$. In other words, h is not trivially independent from g.

If we use Definition 3.6, we use the inlined from. Substituting the information from the events into the condition for weak independence yields:

$$true \wedge x' = x + 1 \Rightarrow \exists y' \cdot (y = x \Leftrightarrow y' = x')$$

The guard of g is true in the DIF. We assume that the invariant and axioms do not matter, e.g., they only contain typing or irrelevant information.

The condition holds, because there is always a y' that makes the existential quantification true, namely $y' = x + 1$.

This means that h is weakly independent from g with respect to Definition 3.6.

If we use Definition 3.8, we get a different result. in this case, we use the totally lifted form.

$$fx = x + 1 \wedge x' = fx \Rightarrow (y = x \Leftrightarrow y = x')$$

This is a contradiction. The left hand side of the implication is true, because we investigate a situation where g happened. This means we can assume is that g's guard and before-after-predicate are true. The left hand side is false, it would require that $y = x \Leftrightarrow y = x + 1$, i.e. $x = x + 1$ hold.

This means that h is **not** non-trivially strongly independent from g with respect to Definition 3.8.

This is a case where h is weakly independent from g but it is not strongly independent.

3.3.4 TLF versus DIF

It is worth noting that the read set of an event written in DIF is different from the read set of the event written in TLF.

In the context of weak independence, the choice of the form does not matter. The guard of an event in TLF compared to the event in DIF only contains some additional conjuncts of the form $x = Expression$, where x is a local parameter. This is essentially a *let* that introduces new names for expressions. All these new names are existentially quantified on the right hand side of the implication in the condition for weak independece, hence they cannot be the cause if the right hand side of the implication is false. But we potentially move variables from the action into the guard. This means the read set of an event written in TLF may be larger than the read set of the event written in DIF. Trivial independence is directly related to the read set. The larger the read set is, the more likely it is to have a nonempty intersection with the write set of some event. In other words, events that are trivially weakly independent in DIF can become non-trivially independent in TLF. Because for weak independence the choice of the normal form does not matter, we choose DIF as it has potentially more trivial independent events that are cheaper to find than non-trivial events.

However, for Definition 3.8 the situation is different. We still introduce fresh names for expression in TLF but they are not existentially quantified. This means they contribute to the truth value of the right hand side of the implication of the condition for strong independence.

g $\widehat{=}$ **begin** act1 : $y := y + 1$ **end** **h** $\widehat{=}$ **begin** act1 : $x := y$ **end**	**g** $\widehat{=}$ **any** fy **where** grd1 : $fy = y + 1$ **then** act1 : $y := fy$ **end** **h** $\widehat{=}$ **any** fx **where** grd1 : $fx = y$ **then** $x := fx$ **end**

Figure 3.7: Example demonstrating the difference between TLF and DIF in Definition 3.8.

We can demonstrate this using the example in Figure 3.7. If we would use DIF in Definition 3.8 we would conclude, that h is independent from g, the condition for strong independence would be:

$$true \wedge y' = y + 1 \Rightarrow (true \Leftrightarrow true)$$

Actually in DIF h is trivially independent from g, because $read(h) = \varnothing$.

However, in TLF the situation is different. The condition is:

$$fy = y + 1 \wedge y' = fy \Rightarrow (fx = y \Leftrightarrow fx = y')$$

This is a contradiction which means that h is not strongly independent from g.

We can conclude that while the choice of the normal form does not matter for weak independence it does matter for strong independence.

3.3.5 Application: State patching

Besides reduction of the effort to compute guards, strong independence allows us to use patching to compute successor states instead of computing them by constraint solving. The situation where state patching can be applied is shown in Figure 3.8. State s has been fully explored, i.e., its invariant has been checked and the successor states have

been computed by evaluating the guards of g and h and applying their actions to s. The event g yields sta l that $g \not\rightsquigarrow_s h$ and the model ch

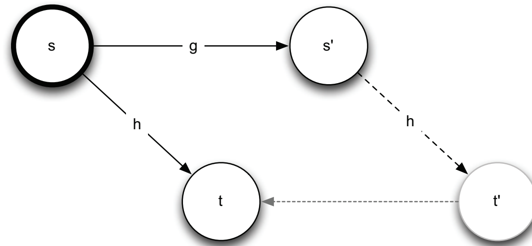


Figure 3.8: Application of effect independence: State patching.

Because $g \not\rightsquigarrow_s h$, we know that the solution computed for the guard of h in state s is also a solution for the guard of h in state s'. We also know that the substitution is deterministic because the event is in TLF. Thus, it is possible to compute a “diff” between t and s and simply apply the “diff” to patch s' in order to compute t'.

3.3.6 Nontrivial Independence and missing Abstractions

If a model contains non-trivially independent events, it is an indication that there might exist a useful abstraction where the independence does not exist. We surmise that constructing this abstraction, if it does not yet exist in the refinement hierarchy, could help to better understand a model.

In our toy example from Figure 3.5, we could introduce an abstraction that contains a variable v and the invariant $v = x + y$. In this abstraction, the independence of h from g is trivial and we made explicit that $x + y$ is invariant under execution of g. This taught us something new about the model.

3.4 Enabling and disabling Predicates

For dependent events we now introduce the notion of enabling and disabling predicates. These predicates describe the circumstances under which an event will enable or disable another event. We have the freedom to choose one of the notions of independence. We will use strong independence in the rest of this chapter. We will discuss the implications of choosing weak independence in Section 3.8.

Definition 3.9 (Enabling predicate)**.** *Let g and h be events excluding the initialization. Furthermore, let h depend on g. The predicate P_E is called an enabling predicate for an event h after an event g denoted by $g \curvearrowright_{P_E} h$, if and only if the following holds:*

$$I(s,c,v) \wedge G(s,c,v,x) \wedge BA(s,c,v,x,v') \wedge A(s,c) \Rightarrow (P_E(s,c,v,x,y) \Leftrightarrow H(s,c,v',y))$$

where $I(s,c,v)$ is the invariant of the machine, $G(s,c,v,x)$ is the guard of g with parameters x and $BA(s,c,v,x,v')$ the before-after predicate of g's action, and where $H(s,c,v',y)$ is the guard of h with parameters y.

Intuitively, an enable predicate is equivalent to the guard of an event h in the context that event g will happen. Figure 3.9 shall illustrate the situation, if the enable predicate P holds in state v, then the guard of h will be true in state v'.

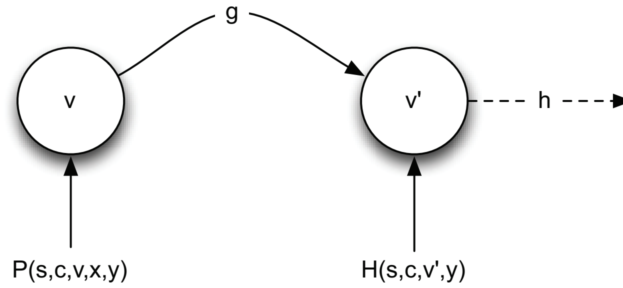


Figure 3.9: Explanation of the enable predicate

Let us take Figure 3.10 as an example. The event depends on itself, we can easily disprove $x < 10 \vdash x + 1 < 10$ using $x = 9$ as a counterexample. We assume that the invariant is $x \in \mathbb{Z}$ and there are no axioms.

We have to find a P_E such that the condition from Definition 3.9 holds.

$$x \in \mathbb{Z} \wedge x < 10 \wedge x' = x + 1 \Rightarrow (P_E(x) \Leftrightarrow x' < 10)$$

```
count ≙
    when
        grd1 : x < 10
    then
        act1 : x := x + 1
    end
```

Figure 3.10: Counter event

A possible solution for the enabling predicate is $x < 9$. There are many other enabling predicates, for instance $x \leq 8$. If the enabling predicte $x \leq 8$ holds in state v then it is guaranteed that the guard of h is true in v'. It is important to note that the enabling predicate and the guard of h use different states as their input. P_E uses the state before g happens, h's guard uses the state after g happened.

Analogously to the enabling predicate, we can define a disabling predicate, that ensures that an event h is not enabled after we observed the occurrence of g.

Definition 3.10 (Disabling predicate)**.** *The predicate P_D is called disabling predicate for an event h after an event g excluding the initialization, if and only if the following holds:*

$$I(s,c,v) \land G(s,c,v,x) \land BA(s,c,v,x,v') \land A(s,c) \Rightarrow (P_D(s,c,v,x,y) \Leftrightarrow \neg H(s,c,v',y))$$

where $I(s,c,v)$ is the invariant of the machine, $G(s,c,v,x)$ is the guard of g with parameters x and $BA(s,c,v,x,v')$ the before-after predicate of g's action, and where $H(s,c,v',y)$ is the guard of h with parameters y.

In the case of the *count* example from Figure 3.10, a candidate for the disabling predicate is $x \geq 9$:

$$x \in \mathbb{Z} \land x < 10 \land x' = x + 1 \Rightarrow (x \geq 9 \Leftrightarrow x' < 10)$$

Definition 3.11 (Enabling predicate for initialization)**.** *The enabling predicate for $(init \mapsto init) \in \eta$ is $false$ and the disabling predicate is true.*

We have to specify the special case for the initialization to enforce our chosen semantic, i.e., the initialization is only observed once in the beginning.

Definition 3.12 (Validity of P_E)**.** *An enabling predicate P_E is valid if P_E is not a disabling predicate. Conversely, a disabling predicate P_D is valid if it is not an enabling predicate.*

Note that if there are inconsistencies in $I(s,c,v) \wedge G(s,c,v,x) \wedge BA(s,c,v,x,v') \wedge A(s,c)$, e.g., if the invariant is violated, there is no valid enabling predicate because any arbitrary predicate is both an enabling and a disabling predicate.

As an illustration, we will compute the predicates for a small example. Take for instance a model of a for-loop that iterates over an array and increments each value by one. Assuming the array is modeled as a function $f : 0..n \to \mathbb{N}$ and we have a global counter $i : 0..(n+1)$, we can model the loop (at some refinement level) using two events *terminate* and *loop*.

terminate $\widehat{=}$
    when
        `grd1` : $i > n$
    then
        skip
    end
loop $\widehat{=}$
    when
        `grd1` : $i \leq n$
    then
        `act1 2:` $f(i), i := f(i) + 1, i + 1$
    end

Figure 3.11: Incrementing values in an array

First we compute the read and write sets for each event as shown in Table 3.1. The effect relation is $\eta = \{loop \mapsto loop, loop \mapsto terminate\}$.

Event	read set	write set
terminate	$\{i, n\}$	$\varnothing$
loop	$\{i, n\}$	$\{i, f\}$

Table 3.1: Read/write sets for the model in Figure 3.11

Event Pairs	$G \Rightarrow (P \Leftrightarrow S[H])$	Simplified solution for P_E
terminate $\rightsquigarrow_P$ terminate	$i > n \Rightarrow (P_E \Leftrightarrow i > n)$	true
loop $\rightsquigarrow_P$ loop	$i \leq n \Rightarrow (P_E \Leftrightarrow (i+1) \leq n)$	$(i+1) \leq n$
loop $\rightsquigarrow_P$ terminate	$i \leq n \Rightarrow (P_E \Leftrightarrow (i+1) > n)$	$(i+1) > n$
terminate $\rightsquigarrow_P$ loop	$i > n \Rightarrow (P_E \Leftrightarrow i \leq n)$	false

Table 3.2: Enable Predicates for the model in Figure 3.11

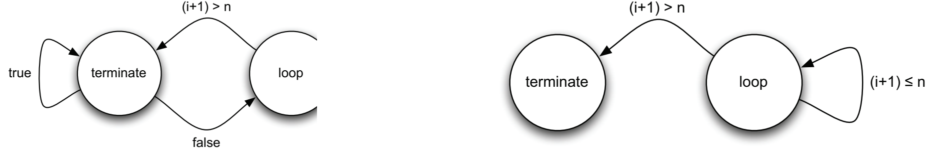


Figure 3.12: Graph Representations of Dependence for a Simple Model

We can now try to find enabling predicates for each possible combination of events. Table 3.2 shows the conditions for P_E from Definition 3.9 and a simplified solution for P_E which satisfy it. We left out any context except for the guard of the preceeding event. We can also see that the solutions are indeed valid enabling predicates. Take, for instance, the last entry: if the solution for P_E were also a disabling predicate, then $i > n \Rightarrow (false \Leftrightarrow i > n)$, or equivalently, $i > n \Rightarrow \neg(i > n)$ would have to hold, which is obviously not the case.

The directed graph on the left in Figure 3.12 is a graphical representation of Table 3.2. Every event is represented by a node and for every enabling predicate $first \leadsto_P second$ from Table 3.2 there is an edge between the corresponding nodes.

The right picture shows the same graph if we take strong independence of events into account, i.e., if $g \not\leftrightarrow_s h$, we do not insert an edge between g and h. In particular, as *terminate* does not modify any variables, it cannot modify the truth value of any guard.

On first sight it seems as if we may have also lost some information, namely that after the execution of *terminate* the event *loop* is certainly disabled. However, we have already seen in Section 3.3.3 that we can eliminate the guard evaluation of strongly independent events, thus this information loss has no effect on model checking.

3.4.1 Computing the enabling predicates

In absence of non-deterministic assignment and given that the invariant and axioms are not contradictory, the weakest precondition $[S(s, c, v, x, v')]H(s, c, v', y)$ is a valid enabling predicate, i.e., it is a solution for P_E in

$$I(s, c, v) \land G(s, c, v, x) \land A(s, c) \Rightarrow (P_E(s, c, v, x, y) \Leftrightarrow H(s, c, v', y))$$

It is important that we demand that all non-deterministic assignments were removed. In the presence of non-deterministic assignments the weakest precondition calculus would not produce valid enabling predicates. Take the two events shown in Figure 3.13 as an example.

g $\widehat{=}$

  then
    `act1` : $x :\in \mathbb{N}$
  end

h $\widehat{=}$

  when
    `grd1` : $x = 5$
  then
    skip
  end

Figure 3.13: No enabling predicate in presence of non-deterministic assigment

g $\widehat{=}$

  any
    fx
  where
    `grd1` : $fx \in \mathbb{N}$
  then
    `act1` : $x := fx$
  end

Figure 3.14: Transformed event allows definition of enabling predicate

The weakest precondition $[x :\in \mathbb{N}](x = 5)$ is $false$; we cannot guarantee that h is enabled after g. The weakest precondition for the negation of h's guard $[x :\in \mathbb{N}](x \neq 5)$ is also false. This means that the enable predicate is invalid.

However, using the event transformation for non-deterministic assignment on g yields the event shown in Figure 3.14. It is possible to compute the enabling predicate for this event. The weakest precondition $[x := fx](x = 5)$ is $fx = 5$ and the disable predicate is the negation $fx \neq 5$.

3.4.2 Simplification

We need to simplify the solution, otherwise it is likely as complicated as the original guard and we will probably gain no benefit from using the enable predicate instead of the original guard. We therefore use the additional information that the guard of the preceding event g is true and the invariant holds before g occurs.

Consider the model shown in Figure 3.15. The weakest precondition for g preceding h is $[x := x + 2]x = 1$ which yields $x = -1$. This contradicts the invariant $x > 0$ and thus h can never be executed after g took place. In the context of the invariant, $x = -1$ is equivalent to $false$.

INVARIANTS
  `positive` : $x > 0$
g $\widehat{=}$
  then
    $x := x + 2$
  end
h $\widehat{=}$
  when
    `grd1` : $x = 1$
  then
    $skip$
  end

Figure 3.15: Simplification of P_E

A very important requirement in our setting is that the simplifier does not increase the number of conjuncts. We have to keep the input small to prevent exponential blowup in the following steps. The optimal situation is when the simplifier can figure out that a conjunct is equivalent to true or false. If it is true then it can be removed from the conjunction; if it is false, the whole conjunction is false.

We have implemented a prototype in Prolog, which was used to carry out our case studies. The method does not rely on this particular implementation. We can replace it by more powerful simplification tools in the future.

The prototype simplifier uses a relatively simple approach as shown in Figure 3.16. After normalizing the fomulas, we successively try to add the conjuncts of the formulas to a set of current conjuncts K. If we try to add the conjunct c we can observe two special cases:

```
K_I := closure(conjuncts(I(v))) // precomputed
K_g := closure(conjuncts(G(v,t))) // precomputed for every event g

K := closure(K_I ∪ K_g)
WP := conjuncts([S(t,v)]H(v,s))
P := ∅
while WP ≠ emptyset do
    choose c and remove from WP
    if c ∉ K then
       K := closure(K ∪ {c})
       if inconsistent(K) then
          P := {false};
          WP := ∅
        else P := P ∪ {c}
       fi
    fi
od
P is the simplified version of [S(t,v)]H(v,s)
```

Figure 3.16: Algorithm for simplifying the weakest precondition in the context of the invariant and guard of g

1. c is already a member of K: we then skip c because we know that it is $true$ in the context of K.
2. c is inconsistent with a member of K: the enabling predicate is then $false$.

The second special case is detected using a small number of Prolog clauses such as the following (where, the first argument is the binary operator followed by the arguments to the operator):

```
inconsistent_fact(not_equal,X,X).
inconsistent_fact(less,X,X).
inconsistent_fact(less,X,Y) :- value(X), value(Y), X >= Y.
inconsistent_fact(equal,X,Y) :-  value(X), value(Y), X \= Y.
inconsistent_fact(member,_,empty_set).
```

The first rule states that $x \neq x$ is inconsistent, the second one that $x < x$ is inconsistent, the third one that $x < y$ is inconsistent if x and y are known values with $x \geq y$, the fourth rule states that $x = y$ is inconsistent if x and y are known values with $x \neq y$, and the fifth rule states that membership in the empty set is inconsistent. We use normalization to keep the number of rules low. For instance, there is no rule for greater than. We simply rewrite $x \geq y$ to $y < x$ and use the rule for less.

In any other case we add the conjunct and calculate the closure of K using some rules that combine two formulas, computing new logical consequences, e.g., using the transi-

tivity of $<$ and $\leq$ we compute $x < z$ as a consequence of $x \leq y$ and $y < z$. This step is important, without computing consequences we would only be able to remove conjuncts within the enabling predicate that are exact duplicates of conjuncts in the guard or the invariant or to detect inconsistencies if there is an exact rule for the situation.

For example, assume we have built the set of conjuncts $K = \{x < 2, y = 5\}$, and we now try to add $x = y$. Assuming we have only the rules mentioned above. Note that there is no rule that directly detects the contradiction.

When adding the new fact, we have to derive possible consequences, i.e., we combine the facts until we reach a fixpoint. Combining $x = y$ with $y = 5$ yields $x = 5$ which we add to K. Combining the new fact $x = 5$ and $x < 2$, yields $5 < 2$. This triggers the third Prolog rule and we detect an inconsistency.

The actual implementation does not always recompute everything from scratch. We precompute a closure K_I from the conjuncts of the invariant. This set can be reused for the whole model. For each event g we precompute a closure K_g from the guards. This set can be reused for all enabling predicates where g is the first event.

If the algorithm has not stopped because of a contradiction the enabling predicate is the conjunction of the formulas stored in P.

3.5 Enable Graph

We can now finally define the enable graph that captures the effects between events. Actually, we have already informally introduced an enable graph in Figure 3.12. The graph on the right side is the enable graph for the example.

Definition 3.13 (Enable Graph)**.** *An* ***Enable Graph*** *of a model M consists of a directed Graph $G = (N, E)$ and an edge labeling function. The vertices N of the graph are the events of M. The edges are all pairs of dependent events, i.e., $E = \{(g, h) \mid g, h \in N \wedge g \rightsquigarrow_y h\}$. The labeling $\lambda : E \rightarrow Pred$ function assigns an enabling predicate P_E to each edge of G, i.e., $\lambda = \{(e, P) | e = (g, h) \wedge e \in E \wedge g \curvearrowright_P h\}$*

Definition 3.13 actually defines a family of graphs because the enabling predicate is not unique and the information about independence between events may be incomplete, i.e., the enable graph can contain additional edges. In the following, when we talk about the enable graph, we actually mean the graph that is computed by some fixed procedure.

We can also visually represent an enable graph as a forest, where the root of each tree is an event of the model and the tree contains each event that depends on the root event. The representation as a forest might be easier for a human to comprehend.

Constructing the enable graph is relatively efficient; it requires the calculation of $O(card(Events)^2)$ enabling predicates. From our experience with invariant reduction using proof supported model checking, we think that in the case of software specifications, generating the enable graph and using the information gained for guard reduction can yield in a reduction of the model checking time. However, we have no experimental data yet.

We can also influence the graph “interactively”. For instance, take the graph shown in Figure 3.19. The edges (*upini*,*up*), (*upini*,*dn*), and (*upini*,*gcd*) all contain one conjunct referring to the variable *dn*. By inspecting the model, we realize that the initialization sets dn to $false$. We also realize that *upini* does not refer to dn. This means that the information $dn = false$ is not available for the simplification. Adding a theorem $dn = false$ to the guard of *upini* enables the simplifier to remove $dn = false$ from the enabling predicates of (*upini*,*up*), (*upini*,*dn*). And more importantly, it can infer that the enabling predicate of (*upini*,*gcd*) is $false$.

We believe that expressing these theorems does not only improve the graph but also our understanding of a model because we explicitly formalize properties of the model that are not obvious, at least not for the automatic simplifier.

3.6 Example: Extended GCD Algorithm

To demonstrate the process, we use the extended GCD algorithm taken from [53]. We will only use one part, the first loop, of the last level of refinement as shown in Figure 3.17. We have also added a guard $dn = FALSE$ to the guard of the *upini* event. Adding the guard will not make any difference to the algorithm. The variable dn is set to false in the initialization and *upini* is supposed to be the first event after the initialization.

EVENTS

upini $\widehat{=}$

  when

    `grd1` : $up = FALSE$
    `grd2` : $dn = FALSE$

  then

    `act1` : $up := TRUE$
    `act2` : $f := 0$
    `act6` : $r := \{0 \mapsto a \ mod \ b\}$

  end

up $\widehat{=}$

  when

    `grd1` : $up = TRUE$
    `grd2` : $r(f) \neq 0$
    `grd3` : $dn = FALSE$

  then

    `act1` : $f := f + 1$
    `act5` : $r(f + 1) := t(f) \ mod \ r(f)$

  end

dnini $\widehat{=}$

  when

    `grd1` : $up = TRUE$
    `grd2` : $r(f) = 0$
    `grd3` : $dn = FALSE$

  then

    `act1` : $dn := TRUE$

  end

END

Figure 3.17: Example Model for automatic flow analysis

First, we need the read and write sets of the events, as shown in Table 3.4. Because we want to use strong independence, we must transform the events into the totally lifted

e	**read(e)**	**write(e)**
upini	{up,dn}	{up,f,r}
up	{up,r,f,dn}	{f,r}
dnini	{up,r,f,dn}	{dn}

Table 3.3: Read/Write sets

form. We have omitted the TLF here.

From the sets, we can easily calculate the effect relation.

$$\begin{aligned}\eta = \{&upini \mapsto upini, upini \mapsto up, upini \mapsto dnini,\\ &up \mapsto up, up \mapsto dnini,\\ &dnini \mapsto dnini, dnini \mapsto up, dnini \mapsto upini\}\end{aligned}$$

For each pair of events we can now calculate P_E. By definition the enable graph contains only edges between dependent events, therefore we can ignore all pairs that are not element of η. In this example, almost all events are dependent except for *upini* which is trivially independent from *up*.

(upini,upini)	$\bot$
(upini,up)	$a \ mod \ b \neq 0$
(upini,switch)	$a \ mod \ b = 0$
(up,up)	$t(f) \ mod \ r(f) \neq 0$
(up,dnini)	$t(f) \ mod \ r(f) = 0$
(dnini,up)	$\bot$
(dnini,dnini)	$\bot$
(dnini,upini)	$\bot$

Table 3.4: Enable predicates

The resulting graph is shown in Figure 3.18. For better readablity, we split the graph into subgraphs for each event.

We also applied the process to the full model. This is described in [52]; the full enable graph is shown in Figure 3.19. We have also explained how the enable graph can reduce the number of guard evaluations. For one particular run of the algorithm with fixed input numbers, the run will start with *init* and *upini* then contain a certain number n of *up* events. This will be followed by *dnini* and then exactly n *dn* events and will

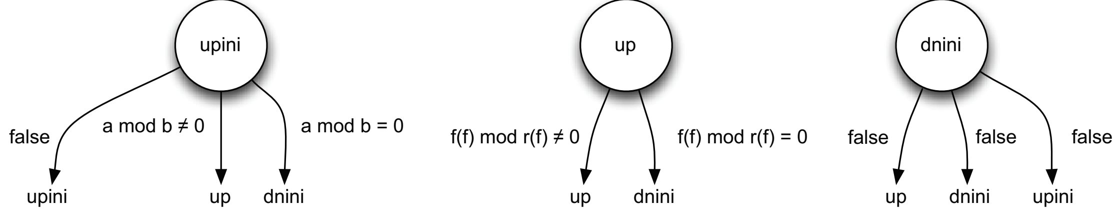


Figure 3.18: Enable graph for example from Figure 3.17

finish with one *gcd* event. All in all, the calculation takes $2n+4$ steps. After each step, the model checker needs to evaluate 5 event guards (one for each event, except for the guard of the initialization which does not need to be evaluated) yielding $10n+20$ guard evaluations in total. Using the information of the enable graph we only need a total of $4n+4$ guard evaluations. For example, after observing *up*, we only need to check the guards of *up* and *dnini*: they are the only outgoing edges of *up* in Figure 3.19 that are not labeled with $false$.

3.7 Application: Guard Evaluation

In Section 3.3.3, we have shown how we can use information about strong independence between events to avoid guard evaluation. In this section, we will use the enabling information to reduce or even avoid the guard evaluation when the events are not independent.

When we check a model for consistency, we know at least one predecessor state and one event that led us into the state we want to explore. For example, we might know that g led us into the state under investigation and furthermore we have computed an enabling predicate P_E for some event h. This information can be exploited to reduce the guard evaluation for h. We can replace h's guard by the enabling predicate. In particular, the special case $P_E = false$ is interesting. We can skip the predicate evaluation for every event where the enabling predicate is false.

Lemma 3.14. *Let g and h be events. If the enabling predicate P_E is false, then the guard of h is false after g occurred, assuming that the invariant and the axioms hold.*

Proof We start with the definition of the enabling predicate

$$I(s,c,v) \wedge G(s,c,v,x) \wedge BA(s,c,v,x,v') \wedge A(s,c) \Rightarrow (P_E(s,c,v,x,y) \Leftrightarrow H(s,c,v',y))$$

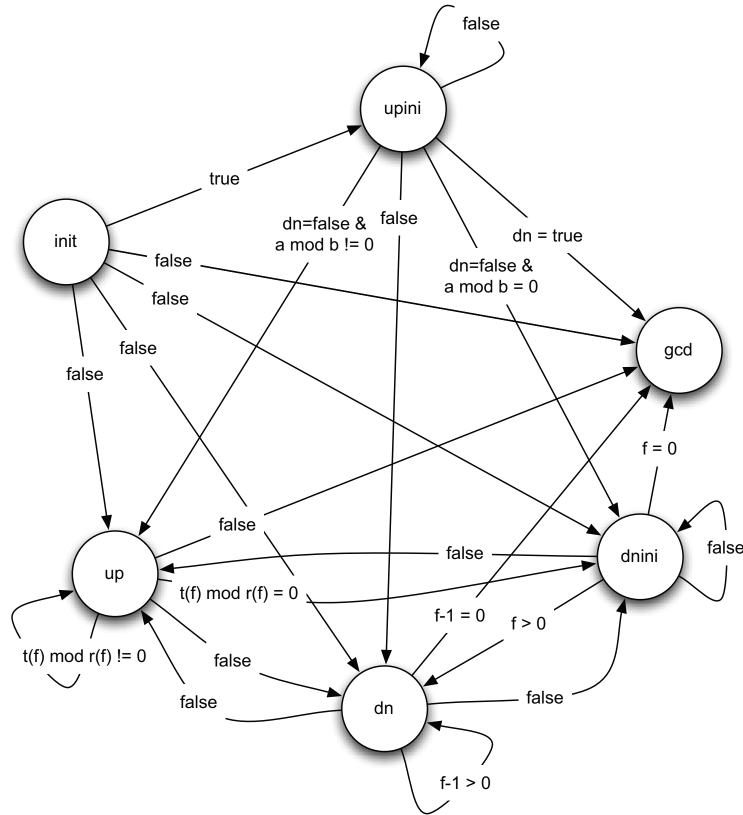


Figure 3.19: Enable graph for example from [53]

We know that g occurred, i.e., $G(s, c, v, x)$ and $BA(s, c, v, x, v')$ are true. Also by assumption $I(s, c, v)$ and $A(s, c)$ are true and the enabling predicate is false.

$$false \Leftrightarrow H(s, c, v', y) \equiv \neg H(s, c, v', y)$$

□

Because we use the invariant when simplifying the enabling predicate, the invariant must hold in the previous state in order to use the flow information. However, we believe this is reasonable because most of the time we are hunting bugs and thus we stop at a state that violates the invariant. The implementation must take this into account, and in the case of an invariant violation it must not use the information gained by flow analysis. Also it needs to check not only the invariant but also the theorems and axioms if they are used in the simplifier.

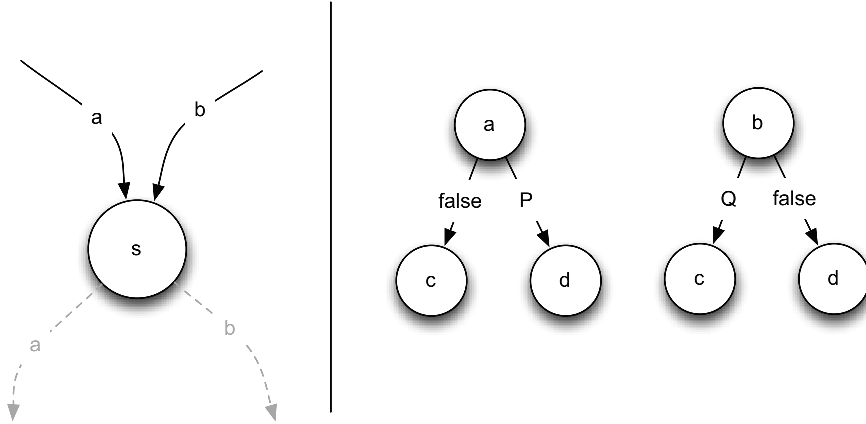


Figure 3.20: Example for combining information

Note that we cannot combine the information from all incoming transitions like we did for invariant preservation checking to further simplify P_E.

However, we can freely chose which enabling predicate we select for the guard evaluation. For instance, say we have four events a, b, c and d, and we know that a disables c and b disables d. Furthermore, we encounter a state s via a but do not yet calculate the successors. Later we encounter s again, this time via b. This is shown in Figure 3.20. On the left is the situation that occured during model checking. On the right hand side is the partial enable graph. When calculating the successors of s we can skip both c and d. The reason is that we have a proof for *c is disabled* because the state was reachable using event a and a proof that *d is disabled* because the state was reachable using event b. Thus the conjunction *c and d are disabled* is also true.

Because the enable graph does not contain edges (a, a), (a, b), (b, a) and (b, b) we can assume, that a and b are independent from itself and each other. This means that we can also deduce that a and b are enabled in state s.

3.8 Weak versus Strong Independence

If we use weak independence instead of strong independence, we can potentially eliminate more edges from the enable graph because it is more likely that two events are weakly independent than strongly independent. On one hand, this means that maybe we cannot use information about true conjuncts in the enabling predicate for the same reason why weak independence does not work in Section 3.3.3, i.e., we must keep the

information about solutions for local parameters. On the other hand, we can use information about conjuncts that are false (because we then do not have to compute the guard at all) and the computation of independence and the enable graph might be cheaper.

In situations where we only care about completely avoiding guard evaluations, for example in a deterministic system, it is maybe sufficient to use the cheaper solution. The price is that we can only use the enabling predicate instead of the guard if the enabling predicate is false.

3.9 Flow Analysis

Given an enable graph we now want to construct a flow graph. A flow graph is an abstraction of the model's state space where an abstract state represents a set of concrete states. Each abstract state is characterized by a set of events, representing all those concrete states for which those (and only those) events are enabled.

The flow graph is constructed by an abstract model checking process as shown in Figure 3.21 and 3.22. We start with a processing list that contains the state *init* which represents the uninitialized state, i.e., only the initialization can be executed. A state is represented by the events that are enabled in the state. Treating the set of enabled events as the state in the flow graph allows us to use the state directly as an entity in the algorithm.

While there are still unprocessed states in the todo list, we take one state and iterate over its members. For each enabled event $e \in state$, we compute the set *keep* which contains all enabled events that are independent from e. Because we defined all events to be dependent on the initialization, we know that $keep(init) = \varnothing$. Therefore we evaluate all guards at least once.

The next step is to compute an expansion which is shown in Figure 3.22. After the expansion, the variable *atoms* contains a mapping from predicates to a state of the flow graph, i.e., a set of events. The flow states, i.e., the range of *atoms* are put into the processing list and the flow graph is extended. Finally we clean up the processing list, removing all states that have been already processed.

After the process list is empty, the flow graph is stored in the variable $flow$.

```
todo := {{init}}
done := ∅
flow := ∅

while todo ≠ ∅ do
    choose state from todo
    foreach e ∈ state do
      keep := state ∩ independent(e)
      atoms := expand(e, keep)
      todo := (todo ∪ ran(atoms))
      flow := flow ∪ {state ↦ atoms}
     od
    done := done ∪ {state}
    todo := todo \ done
 od
```

Figure 3.21: Algorithm for constructing a Flow Graph

The expansion function shown in Figure 3.22 combines enable predicates to create a new predicate P_F for the flow graph transition. The basic idea is that we chose some of the enable predicates that are used in their original form and for the rest of them we use their negation.

```
Given the enable graph as EG : (Events × Events) ⇸ Predicate
def expand(e, keep) =
    true_pred := {f ↦ true|(e ↦ f) ∈ dom(EG) ∧ EG(e ↦ f) = true} // precomputed
    maybe_pred := {f ↦ p|(e ↦ f) ∈ dom(EG) ∧ EG(e ↦ f) = p ∧ p ≠ false}// precomputed
    result := ∅
    foreach s ⊆ maybe_pred do
      targets := dom(true_pred) ∪ keep ∪ dom(s)
      predicate := ⋀ran(s) ∧ ¬(⋁ran(s ⩤ maybe_pred)
      result := result ∪ {predicate ↦ targets}
    od
    return result
end def
```

Figure 3.22: Algorithm for expanding the Enable Graph (i.e., computing successor configurations)

Figure 3.23 shows a simple flow graph construction. Take for instance the vertex labeled with $\{a\}$. In this case we do not have a choice; we must execute a. From the enable graph on the left hand side we know that if P is true then a will be enbled afterwards and in the same way if Q holds then b will be enabled. Combining all combinations of P and Q and their negations, we get the new states $\{\}, \{b\}$ and $\{a, b\}$. If we continue, we finally get the graph shown on the right hand side. If more than one event is enabled, we add edges for each event separately. We can combine edges by the disjunction of the

predicates. In our case, we did that for the transition from $\{a, b\}$ to $\{a\}$ which can be used by either executing b or a.

We immediately remove predicate combinations that are contradictory. For instance, from state $\{b\}$ we have only one outgoing edge $(b, true)$; the other edge $(b, false)$ is self-contradictory and was removed. If we can, for example, prove that $\neg P \wedge Q$ is a contradiction, we would also remove the edge between $\{a\}$ and $\{b\}$. We also remove an edge if the combination contradicts the guard of the event or the invariant. Removing infeasible paths as early as possible helps to reduce the flow graph.

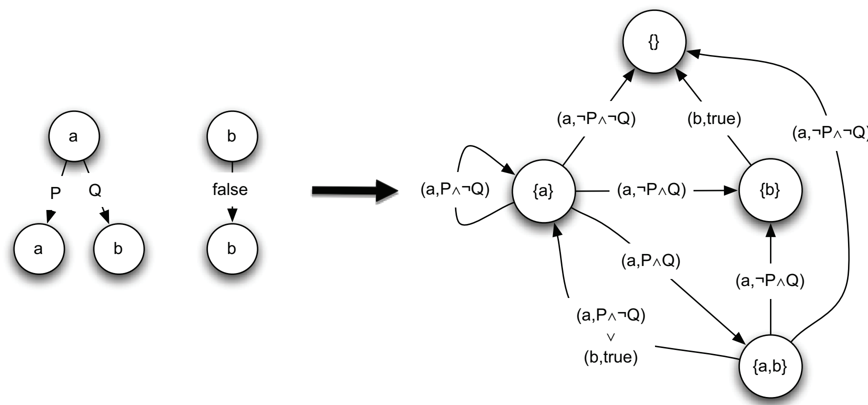


Figure 3.23: Simple Flow Graph Construction

3.10 Feasibility of Flow Graph Construction

Constructing the flow graph can be infeasible because the flow graph can blow up exponentially. It is clear that flow analysis is probably not applicable if the model does not contain any algorithmic structure. In the worst case any combination of events can be enabled in some state, leading to $2^{card(Events)}$ states, where $card(Events)$ is the number of events and it is even worse when it comes to the edges because we have to combine the guards and the negation of the guards as shown in Figure 3.23. However, in the case of software development, it is very likely that the model will contain groups of events, each group implementing a specific functionality of the model. In other words, at each point during the computation only a hopefully small set of events is enabled.

We surmise that the closer a model is to a low-level implementation, the better the results from simplification will be. Our experience is that Event-B models of software

at a sufficiently low level of refinement typically have some notion of an abstract program counter that implicitly controls the flow in a model. These abstract program counters are typically used in predicates that only use equivalence or set membership. Therefore it is very likely that the simplifier can use these predicates for simplification. In many cases, computing the weakest predicate yields either *true* or *false*.

In cases where constructing the flow graph is feasible, we gain a lot of information. A flow describes the implicit algorithmic structure of an Event-B model. This information is valuable for a number of different applications, such as code generation, test-case generation, model comprehension and also model checking.

3.11 Application: Proving Deadlock Freedom

When we build the flow graph, we perform an abstract model checking yielding only states that are potentially reachable. Conversely, if a particular state does not appear in the graph, this means the system cannot reach that state. The flow graph may contain a special state $\varnothing$ which represents all states of the original model where no event is enabled, i.e., a deadlock state. If we construct the flow graph, and we do not encounter this state, we have a proof that is not possible to reach a deadlock state. The presence of the $\varnothing$-state is not a proof that the system deadlocks. It can also mean that the simplifier was not powerful enough to find out that some of the enabling predicates are false. However, beside the information that a deadlock might occur, we also get information how the deadlocking state can be reached. We can extract scenarios from the flow graph, i.e. traces leading from the initial state to the deadlock. Then we can verify if these states are feasible using ProB's constraint solver.

The example in Figure 3.24 can be used to demonstrate deadlock finding. For simplicity, we introduced a very obvious deadlock into the system.

The enable graph for the model is shown in Figure 3.25. The flow graph is shown in Figure 3.26. The flow graph contains the $\varnothing$-state, so we cannot prove deadlock freedom. If we fix the deadlock, for instance by removing the guard of event f, we get the flow graph shown in Figure 3.27. Because the $\varnothing$-state is not in the flow graph, we have proven deadlock freedom of the fixed model.

The flow graph can also be used for similar tasks. For instance, we can prove that the system is deterministic, i.e., there is no state containing more than one event.

MACHINE m0
VARIABLES
x
y
$deadlock$
INVARIANTS
`inv1` : $x \in \mathbb{Z}$
`inv2` : $y \in \mathbb{Z}$
`inv3` : $deadlock \in BOOL$
EVENTS
Initialisation
begin
`act1` : $x := 2$
`act2` : $y := 3$
`act3` : $deadlock := FALSE$
end
f $\widehat{=}$
when
`grd1` : $deadlock = FALSE$
then
`act1` : $y := y + 1$
end
g $\widehat{=}$
when
`grd1` : $deadlock = FALSE$
then
`act1` : $x := y - 4$
end
h $\widehat{=}$
when
`grd1` : $x = 5$
`grd2` : $deadlock = FALSE$
then
`act1` : $deadlock := TRUE$
end
END

Figure 3.24: Deadlock finding using flow analysis

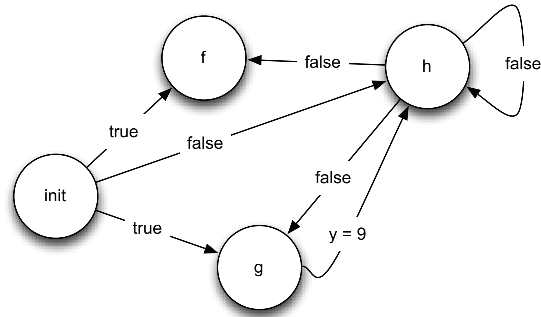


Figure 3.25: Enable graph for deadlock model

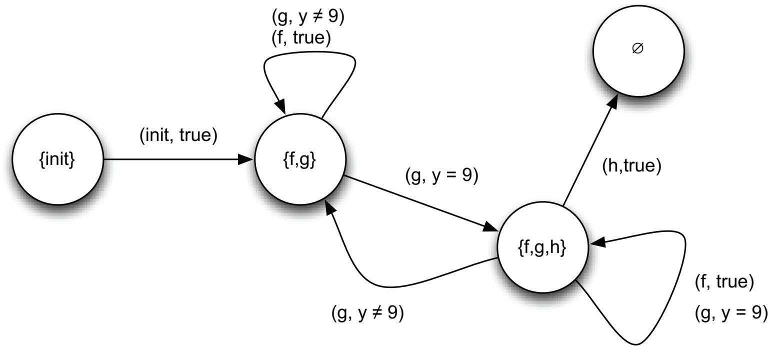


Figure 3.26: Flow graph for deadlock model

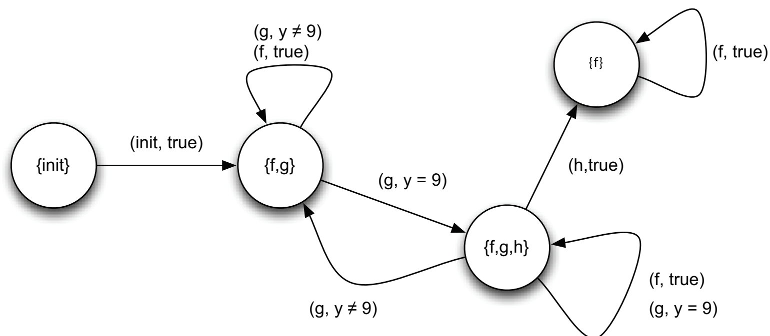


Figure 3.27: Flow graph for deadlock model (after fix)

A commonly used modeling-style in Event-B is to model the system that is developed together with its environment. For instance, if we want to model a cruise controller it can be helpful to also include parts of its environment into the model. In the case of the cruise controller this could be, for example, the break pedal. These environment events often can occur at any time, e.g., the driver can apply the breaks. Environment events typically have very permissive guards or even no guard. This means that they often do not have a deadlock because one of the environment events can be observed.

We can use the flow graph to check if the system can be in such a quasi-deadlock by searching for states in the flow graph that only contain environment events. We can also check that the model is determinstic with respect to the controller events, i.e., no state contains more than one controller event.

3.12 Example: Extended GCD Algorithm

In [53] Hallerstede proposes a structural notation for modeling and proving of Event-B models. The paper contains a graphical notation of the extended GCD algorithm, as shown in Figure 3.28. If we compare this to the result of our flow construction shown if Figure 3.29, we see that our graph indeed resembles the algorithmic structure of the model.

However, the automatic flow analysis helped us to discover an interesting property. The flow graph contains a state that corresponds to concrete states where no event is enabled, i.e., states where the system deadlocks. Thus the model contains a potential deadlock. Inspectio
graph co
and *gcd* (
deadlocki

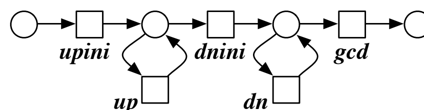


Figure 3.28: Structural model from [53]

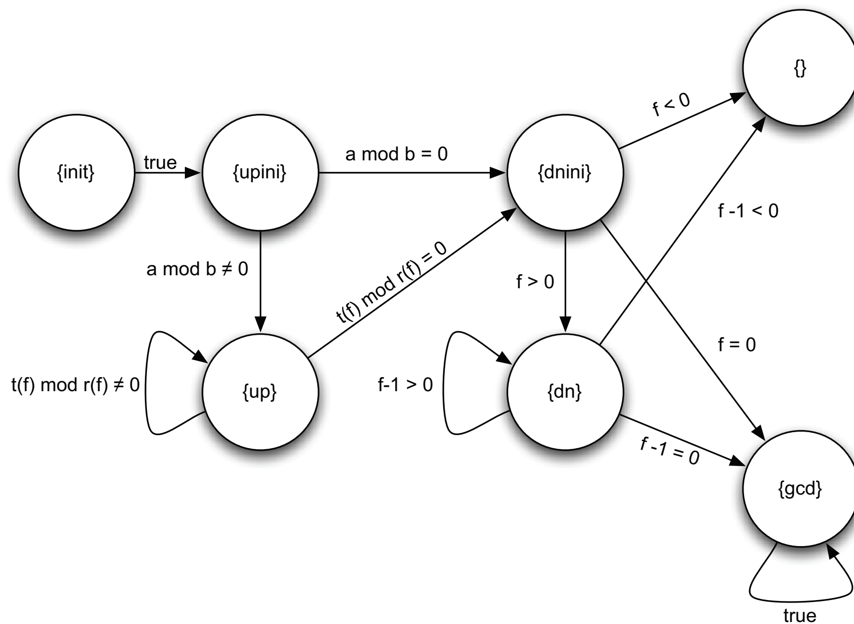


Figure 3.29: Flow Graph

3.13 Future work

Directed Model Checking and Testcase Generation

The flow graph can be used as a guide for model checking. Assume, that we want to find a deadlock in our model and we have the flow graph from Figure 3.26. We can use the graph as a guide to find our target state. Instead of performing a random search, we could use the information from the flow graph to prune the search tree. We could also use the distance to the deadlock state as a heuristic for a best first search. If we think of the flow graph as an automaton where $\{init\}$ is the initial state and our target is the accepting state, a trace leading to the target must be a word that the automaton accepts. As with directed model checking, test case generation can benefit from a more informed search. The flow graph can be used to find traces that cover specific events.

Code Generation

Another potential application for the flow graph is code generation. The flow graph is very similar to a control flow graph used in compilers. We have to verify that the graph

is deterministic. As already mentioned in Section 3.11, we can verify this by looking at the event set of each state. However, we also have to be sure that the substitutions can be translated into code, i.e., we need an equivalent to B_0.

3.14 Related work

Inferring Flow Information Another possible solution for extracting flow information is reduction or projection of the original state space, i.e., we could use ProB's reduction algorithms introduced in [54] to produce abstractions of the state space. In particular, the signature merge algorithm that merges all states which have the same outgoing edges produces a graph that is similar to a flow graph. The problem is that we need to explore the complete state space which can be very time and memory consuming and maybe even impossible because of an infinite state space. Also, creating the signature merged graph can become infeasible for extremely large state spaces. The flow graph can be constructed for infinite state spaces but it potentially suffers from combinatoric explosion.

The GénéSyst approach [55] infers a symbolic labeled transitions system (SLTS) from an Event-B model. The paper applies the technique to a specification of an electronic purse called Demoney [56]. The SLTS is like the flow graph an abstraction of the state space of an Event-B model. GénéSyst uses a set of predicates $P_1, ..., P_n$ for which the condition $\mathit{Invariant} \Rightarrow P_1 \vee ... \vee P_n$ must hold. This set of predicates is used to create an abstraction of the state space. In contrast to the flow graph this method is focused on the state of the B model rather than the interplay of events.

Specifying Flow Information Explicit specification of the flow is more common than automatic extraction. For classical B there has been a lot of work on using CSP [57, 58, 59] to specify the controller of a B model.

In the context of Event-B, there are mainly four other approaches that are related to our flow analysis. Hallerstede introduced a new approach in [53] to support refinement in Event-B that contains information about the structure of a component. Also, Butler showed in [60] how structural information can be kept during the refinement of a component. The state diagrams of UML-B are also used to specify a flow. All these approaches have the advantage of incorporating the information about structure into the method, resulting in better precision. However, the methods require the developer to use the methods and corresponding tools from the beginning while automatic flow

analysis can be applied to existing projects. In particular automatic flow analysis can actually be used to discover properties of a model such as liveness and feasibility of events.

Hallerstede's structural refinement approach is probably the closest to our analysis, but it does not fully replace our automatic flow analysis. Both methods overlap to some extent, i.e., they have the similar purpose to enrich Event-B with explicit information about the algorithmic structure. We think that both approaches can be combined, such that the automatic flow analysis uses structural information to ease the generation of the flow graph. In return, our method can suggest candidates for the intermediate predicates used during structural refinement.

In [61] Cansell, Méry and Merz proposed predicate diagrams to specify reactive systems. The diagrams start at a high level of abstraction and are refined to add more details. The diagrams can be model checked using linear temporal logic. Interestingly "all proof obligations concerning temporal properties are stated in terms of finite transition systems and can therefore be discharged by model checking". The paper uses a mutual exclusion protocol as an example. The top level specification and the first refinement are shown in Figure 3.30. In the paper the authors experimented with several different refinements of the abstract mutual exclusion protocol. The authors conclude that their " definitions are flexible enough to support nontrivial refinements of fairness conditions by a mix of low-level fairness assumptions and arguments based on well-founded orderings." [61].

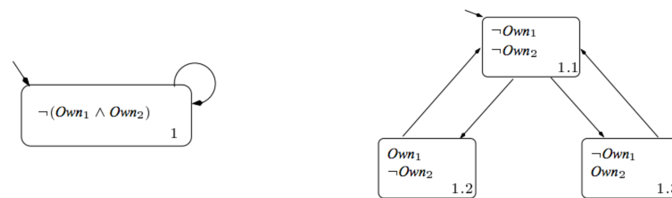


Figure 3.30: Top level specification and first refinement. Taken from [61].

Another approach is the Flow plug-in for Rodin [26]. The Flow plug-in allows the developer to express flow properties for a model and to verify them using proofs. However, the tool has not been updated and we assume that it has been abandoned.

Finally, based on the theoretical results of this thesis, Dobrikov and Leuschel [62, 63] have developed another way to generate enable graphs using ProB's constraint solver

instead of a prover/simplifier. This analysis cannot produce enable predicates, but it returns whether an edge in the graph is always true, always false, or undecided. This analysis was improved with the same technique used for the ProB prover [18]; previously the analysis could be wrong in the presence of infinite sets.

Chapter 4

Distributed Model Checking

This chapter describes the results of a distributed version of the ProB model checker. We will demonstrate that our new tool is able to verify models where model checking using the regular version of ProB is impossible within reasonable time constraints. This chapter is based on a not yet published paper and an extended abstract published in [64]. The full text of the unpublished paper can be found in Appendix B.

4.1 Motivation

When we compare ProB with other tools like Spin [65], we may come to the — from our point of view false — conclusion that ProB is useless as a model checker. While Spin can deal with billions of states, ProB cannot cope with models that consist of more than a couple of million states. However, in [66], Leuschel has argued that these numbers are really difficult to compare due to the different level of abstractions. While the input language for Spin is almost C like, the classical B language is almost pure mathematics. The high level of abstraction in B can lead to a significant reduction in the number of states because a single state at a high level of abstraction can represent hundreds or even thousand states in a low level language. Leuschel concludes that "due to the inherent exponential blow-up of the state space, it is often not that relevant whether a model checking tool can treat 100,000 or 10,000,000 states" [66].

However, we still want to extend the capabilities of ProB to support larger state spaces, maybe even state spaces that consist of billions of states.

Twenty years ago, the easiest way to double the speed of a program was to wait about eighteen months and buy a new computer. This is not the case anymore. CPUs are not getting exponentially faster anymore. Instead, the number of cores in a computer

increases. Another development is the rise of cloud computing that allows renting computation power at a reasonable price.

While there is, without doubt, potential for ProB to improve its speed by using new techniques such as partial evaluation, we have to switch to parallel or distributed computation to make use of new hardware. We think that the tool we developed in this thesis has the potential to lift ProB to a new level.

Case Study: Interlocking

We start by giving an example of a model where model checking became feasible with our distributed version of ProB. It is a classical B version of an interlocking system model from [11] that we studied and optimized for model checking in [67]. In the paper, we reduced the state space by changing the model using partial order reduction resulting in a model that can be checked using the normal version of ProB in about two hours. We use this optimized version as a benchmark for the distributed version of ProB in Section 4.5, but we also want to be able to check the original version. Its state space consists of slightly over 61 million states. Our first attempt to model check the system using the regular ProB version failed. ProB crashed after about 4 days because the computer ran out of swap space. In a second attempt, we allowed ProB to assume that there are no hash collisions and to drop states after exploration. This yields a much smaller memory footprint. We aborted this attempt after ProB had checked 26628001 states in 7 days 11 hours and 41 minutes. We think if this run would have succeeded, it would have taken over 17 days. This prognosis assumes that ProB would indeed check all states, which may not be the case because we might miss a part of the state space due to hash collisions. Furthermore, it assumes that the speed does not degrade over time, which is also not entirely true.

The new distributed version was able to model check the full state space in about 30 hours on a six core 3.33 GHz Mac Pro with 16 GB of RAM using eleven worker processes. We could check the specification on the HILBERT high performance cluster at the University of Düsseldorf in about five hours (295 minutes) using 44 workers spread over 15 computers. Using 104 worker processes spread over 15 computers, the check took slightly more than two hours (126 minutes).

To get a better impression, Figure 4.1 shows the model checking time for each experiment visually. The value for one core is our prognosis of 17 days because we have not actually finished model checking using the regular ProB version.

This example shows that — given the model is suitable for parallel checking — we can extend the applicability of ProB to models where model checking with ProB was not feasible.

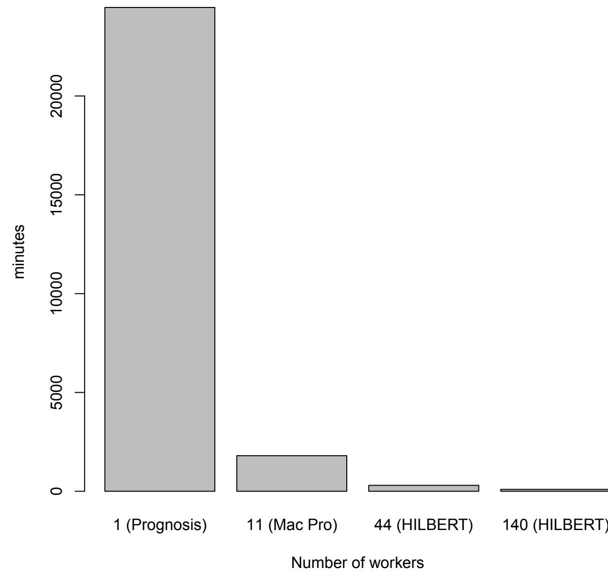


Figure 4.1: Visualization of the model checking speedup for an interlocking system

4.2 Parallel Model Checking

Some questions about a formal model that arise in the B-Method can be answered for each state individually, that is, without knowledge of the path that led to a particular state. Answering these kinds of questions can easily be parallelized.

When we want to verify a property P that does not depend on path information we need to check formulas of the type $\forall s.s \in States \Rightarrow P(s)$. In the B-Method a typical case for such a property is the invariant. Deadlocking can be checked in the same way using the disjunction of the guards.

We want to define a family of sets of states $S_1, ..., S_k$ such that $States = S_1 \cup S_2 \cup ... \cup S_k$. For parallel property checking we want k to be in the order of the number of CPUs we want to use.

By construction the following sequent can be proven:

$$States = S_1 \cup ... \cup S_k, (\forall s.s \in S_1 \Rightarrow P(s)), ..., (\forall s.s \in S_k \Rightarrow P(s)) \vdash (\forall s.s \in States \Rightarrow P(s))$$

This means that we can perform the check for the full set using k different processes in parallel with each checking the states in one index set.

Let D be a relation that associates states with processes, i.e., $D : States \leftrightarrow 1..k$. If $(s_i, j) \in D$ then process j will explore the state s_i.

We can now define some criteria that describe the quality of a family of sets and therefore the relation. Firstly, we do not want the processes to check the same element multiple times, thus the optimal solution would be a family of disjoint sets and the relation D should actually be a function. Secondly, we want a distribution of the work among all processes such that every process gets approximately the same amount of work.

If we knew a cost function C that maps each state to a cost value, e.g., the time required to check the state, we could compute a solution that minimizes the maximal costs of a single index set, i.e., we would try to minimize $max(\sum_{s \in S_1} C(s), \sum_{s \in S_2} C(s), ..., \sum_{s \in S_k} C(s))$. This is the optimization variant of the k-PARTITION problem, often called multiprocessor scheduling problem. It is known to be NP hard [68], but there are efficient approximation algorithms such as the LPT (longest processing time) algorithm which sorts the states by their costs. Then it assigns them to the index set with the lowest costs so far, i.e., it is a greedy approach.

However, this approach is not feasible because we have no knowledge about the costs for a given state before we actually check the state. Currently there is no known method to produce an approximation of the cost function for a given model.

But even if we could produce a cost function, generating all states in advance in order to feed the LPT algorithm is itself not feasible. We want to generate the states in parallel, i.e., we require an online algorithm.

The state space explorations adds another important requirement. If a state is explored, ProB will return a set of possibly yet unexplored successor states. For each of the states we have to decide which process should explore the state. Each transfer between two processes is associated with some additional costs. We want the solution to require as few state transfers between two processes as possible.

Choosing a process for each state is a trade-off between transfer of states and load balancing. An approach that puts emphasis on the load balancing side of the tradeoff is

to produce a hash code $H(s_i)$ from a state s_i and then compute the index of the process that is responsible for the state using $H(s_i)\ mod\ k$ where k is the number of processes. This approach basically ignores any cost associated with state transfer, i.e., if we have n processes, the probability that a state is processed by the same process that discovers the state is only n^{-1}. This approach works very well if the costs for state transfer is low, for instance, if the model checker uses multiple threads within a single process.

In a distributed model checker that runs on different computers the costs for state transfer might become a non-negligible factor. This depends on the computer system and network that is used. On a high performance cluster with InfiniBand connection, the approach may still work. On a Beowulf cluster with regular Ethernet MBit connection, it is likely that the approach would not work well.

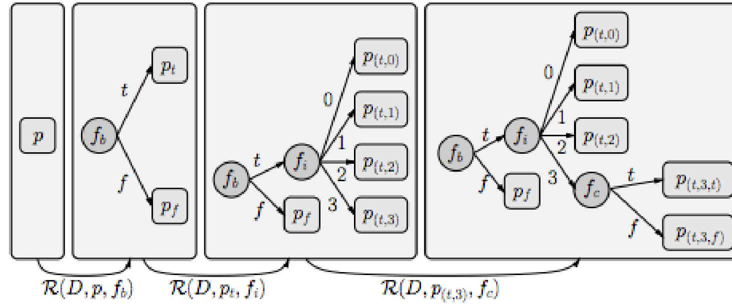


Figure 4.2: Dynamic state space partitioning example from [69]

Another approach is dynamic state partitioning as proposed in [69]. The idea is to use state variables to partition the state space on the fly. A partitioning function is computed on the fly and can be extended later if necessary. Initially the function assigns all states to the same partition. If a split of a partition is required, a state variable is chosen and used to split the original partition. An example taken from [69] is shown in Figure 4.2. In a first step the partition p is split using a boolean state variable into two partitions p_t and p_f. Then the partition p_t is split using a numeric state variable into $p_{(t,0)}$,$p_{(t,1)}$,$p_{(t,2)}$ and $p_{(t,3)}$. Finally, $p_{(t,3)}$ is split using a boolean variable into $p_{(t,3,t)}$ and $p_{(t,3,f)}$. This method tries to take locality into account; the assumption is that states that are linked are similar.

We have chosen to use a different dynamic approach that does not approximate the cost function. Instead, we reorganize the partitions through work stealing while model checking. Each successor state is kept by the worker that discovers that state. This

means, we try to keep everything local. Only when one worker becomes idle do we have to transfer some states. However, this means that we have to compromise on load balancing.

4.3 When does distributed model checking (not) work?

The worst case for our distributed model checker is the model of a counter. Every state has exactly one successor state and we explore the state space while checking the invariant, we can only consider a single state of the system at any time. But a state is the granularity of our implementation. In other words, regardless of how many workers we add, we will never get any speedup. Adding more processes to the system is associated with some performance costs. Therefore the model checker can become slower if we add too many workers. We will examine how big the performance loss is in Section 4.5.1.

If we produce enough successor states to keep the workers busy, we can expect to get a good speedup using distributed computation. This means that we require a certain amount of non-determinism in our model. Fortunately, this is very often the case in real world models. Even if they implement a deterministic controller, the environment often has non-determinism, i.e., during model checking the controller is checked for different inputs. Even if the whole dynamic behavior is deterministic, we often have non-determinism in the setup of the constants and sets.

A technique that can be applied to find the appropriate number of workers is rather simple using a scan technique. We start the model checker with a single worker for a fixed amount of time — say a minute — and measure how many states have been checked. We keep doubling the number of workers and repeating the procedure until we do not get a good improvement anymore. The improvement can be judged by a simple heuristic. For instance, if we double from k workers to $2 \times k$ workers we demand to have checked at least $0.25 \times k$ more states. Once we have an interval, we can narrow the search down to a reasonable number of workers, e.g. using binary search. It is not important to be very precise, but we should not use an extreme excess of workers for two reasons:

a) It blocks valuable resources.

b) The algorithm that controls work stealing is linear in the number of workers, so we waste a bit of performance for each idling worker.

Finding the right amount of workers using this scanning approach could also be implemented as an automatic tool. This would be especially valuable in combination with a tool that automatically reserves resources from a cloud. Implementing such a tool would likely be a straightforward task.

We can also think about changing the granularity of our work items, i.e., we could split the checking of the invariant for a state into multiple work items, each dealing with only a part of the invariant. We use this technique for assertion checking as explained in Section 4.4.11, but assertion checking uses only a single state. When model checking, splitting the invariant would require duplicating the states in memory because each work item has to be self contained in order to be transferable to another process. We think it would not work well in most cases, but for small state spaces with very difficult invariants it could improve scaling.

4.4 The parB Model Checker

The core of ProB has been developed in SICStus Prolog. Unfortunately, SICStus Prolog does not support parallel execution through multi threading, so we could not implement parallel model checking within a single process. We have to run multiple Prolog processes instead, i.e., we have to run multiple instances of the ProB binary. In this section, we will introduce the basic concepts of our distributed version of ProB which we call parB as well as some of the implementation details.

4.4.1 First Attempts

We have tried a number of different approaches. Initially, we experimented with the Linda library provided by SICStus. The library implements a tuple space [70], i.e., data is stored in a central repository and can be retrieved by worker processes. In the case of ProB, the state space as well as the states that haven't yet been explored were stored in the repository.

A worker would retrieve a state, check the invariant, and send all successor states to the repository. The big disadvantage of the library was that we always had to move states around; this is an expensive operation. The approach was beneficial for only very few models where the checking time for each state was very high. This was because of the

rather big performance overhead caused by the framework. At that time ProB was much slower. This means that the ratio

$$\frac{\textit{Overhead of Linda}}{\textit{Computation Time}}$$

was smaller than it is today assuming that the Linda library has not become much faster. With a more recent version of ProB we can therefore expect that the Linda approach would perform even worse.

We also implemented a Java based framework that was intended to be a platform for experimentation with various state distribution methods. In the bachelor thesis that implemented the approach, Radig [71] found that the "system distributes the workload uniformly and scales sufficiently but requires about eight times as much computing power as the single-threaded model checker. [...] there is significant overhead introduced when exchanging states between the Java and Prolog processes."

We considered the penalty factor of 8 to be too high to use this approach, so we abandoned this implementation and switched to the more rigid low level approach that is described in this thesis.

However, the Java based approach had some very interesting aspects. It used a peer to peer network to locate workers, and it was able to easily distribute code among the servers. This is very handy when we want to operate a model checking cluster. We hope that we can at some point in the future regain some of the properties of this approach in the new version.

Finally, we tried a version that shared data between processes using Berkeley DB — a high-performance embedded key/value store — which can be accessed directly using a library that is provided by SICStus Prolog. Our hope was that the direct integration would be fast enough to share states between worker processes. It turned out that this was not the case. In [72] we experimented with a prototype and found that the performance dropped from 7 seconds to write 100000 facts to over 2 minutes as soon as we added another process that concurrently read from the database. The approach did not scale so we decided to not further pursue it.

4.4.2 Architecture

As already mentioned, we cannot use multi-threading; our distributed version of ProB therefore runs multiple instances of the Prolog binary in parallel. Some of them run

on the same physical computer which we will call a node. Each node has at least two processes running. Exactly one of them is an instance of a hashcode manager. Besides the manager, each node runs one or more worker processes. Globally, one of the nodes runs exactly one instance of a master process. The architecture of our distributed version of ProB thus consists of three components:

1. A globally unique master that coordinates the computation.
2. One hashcode manager process on each physical computer.
3. Multiple instances of a worker process that performs the actual model checking.

The master and worker components are implemented as C libraries but they also contain a small part written in Prolog. We have chosen to use C because it allows a very tight integration into ProB. Also, the infrastructure for the distributed model checking should be fast. Data is transferred very efficiently between Prolog and C.

Figure 4.3 shows an example setup of our architecture on three Quad-Core nodes. Node B co

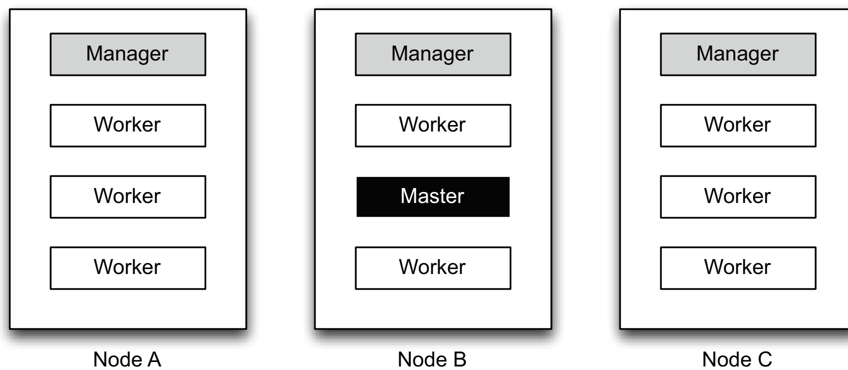


Figure 4.3: Running the distributed model checker on three nodes

4.4.3 Work-Items

Our tool is organized around the notion of a work item, which is a task that should be executed on some worker process. A work item can be any Prolog term. The job of our distribution framework is to organize the work items, i.e., store them in queues, balance the workload, and keep a record of what has been done. For the distribution

framework, the content of a work item is irrelevant; it only handles binary large objects (BLOBs) and hash codes of these BLOBs.

Tasks are executed by the Prolog part of a worker. When the distribution framework needs to process a work item, it calls the process/3 predicate as shown in Figure 4.4. The predicates takes a work item as its input and produces a result and a list of new work items. The result is specific for the work item. In the case of model checking, for instance, it will contain the information if the invariant is broken in that state. The result and successor tasks are sent to the distribution framework for further processing, i.e., the result is aggregated and potentially displayed to the user and the successor tasks are enqueued locally.

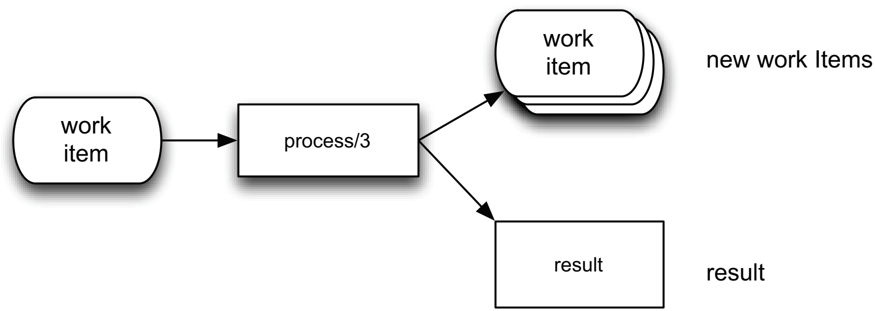


Figure 4.4: Prolog interface of the distribution framework

This works well for model checking, but the distribution framework could also be applied to many more areas. We think that the implementation of new kinds of tasks is relatively simple because the distribution framework does not care about the actual content of the work items.

4.4.4 Shared Data

The distribution framework has to consider two pieces of data: a processing queue that contains all states that are considered for processing and a set of already processed states. Both have to work in a distributed environment.

We actually do not have a global queue but one local queue for each worker. The queues are independent, we do not force them to be disjoint. That means a state can be stored in multiple local queues.

The second piece of data is the set of states that have been processed or are already in a queue. A bit flag distinguishes states that have been fully checked from states that are only enqueued. When a state is dequeued from a worker's local queue, the worker checks if it has been marked as fully processed. If the set of known states contains the state and it has been marked as processed, it is dropped. If a successor state is enqueued, we check if it is present in the set of known states. If it is in the set, either fully processed or only enqueued, we drop it because we know that it either has been processed or it will be processed in the future. Figure 4.5 illustrates the decision a worker has to make if a state should be enqueued or dequeued. As mentioned, the difference is that if we dequeue, we have to be sure the state has been processed before. The set containing all known states has to be shared among the workers in order to avoid duplicate calculations.

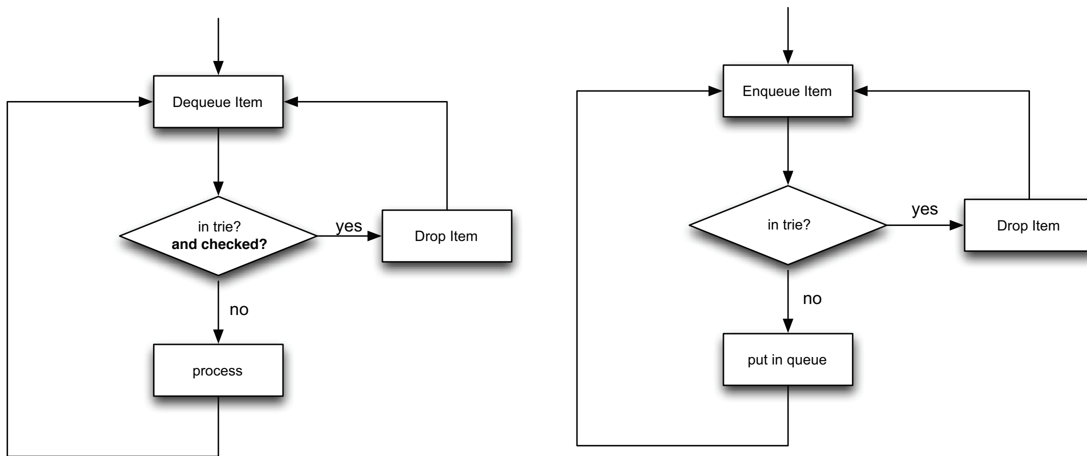


Figure 4.5: Droping work items

A strictly consistent view, i.e., all components have the exact same information about processed states, would require a locking mechanism or a centralized service. This would have a big impact on the performance of the system. However, we know that the set of processed states is monotonically growing. No state that has been added to the set is ever removed. This means, we can use an eventually consistent approach [73]. We replicate the set and we accept that the local information of a worker may differ from the global view as shown in Figure 4.6. Globally the set of known states consists of the states 1 to 8, but each worker only has a partial knowledge. In the example it can happen, that Worker A enqueues State1 for further processing although it can be dropped. However, the distribution framework will at some point in the future provide Worker A with the information that State1 has been processed.

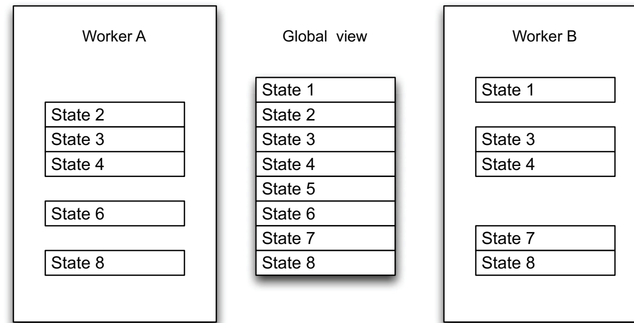


Figure 4.6: Two workers with a different view on the set of known states

As a consequence, a worker might falsely assume that a state has not yet been checked and unnecessarily re-process the state. However, if we can keep the number of duplicate calculations reasonably low, the impact on performance is negligible. Our experiments have shown that this is indeed the case.

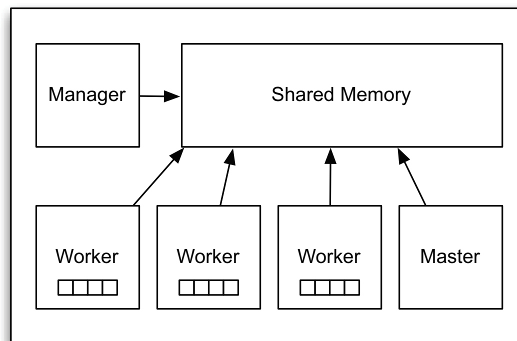


Figure 4.7: Single Node

Each worker having its own copy of the set would result in bad memory usage. We would end up with many copies of the same data on the same computer. Instead, we decided to use a single copy of the set of processed states in shared memory for each physical computer [74]. Initializing and freeing the shared memory as well as adding states that have been checked remotely is the job of the hashcode manager component. Adding states that have been checked by one of the workers on the same machine is done directly, i.e., the worker directly writes into the shared memory. To avoid race conditions we use semaphores as described in [74]. Figure 4.7 shows a single node. The shared memory block is used by all components. In addition to that each worker has a

local queue stored in regular memory. In the following section we discuss the memory management for the hash codes.

4.4.5 Storing checked states

While the states that are queued for later processing are stored in a binary representation, the states that have been processed are only stored as hash codes. The implementation currently uses SHA-1, a cryptographic hash function specified in RFC 3174, to compute the hash codes.

If we encounter states with the same hash code, we assume that the states are the same state. This can lead to unsoundness of the model checking if two different states produce the same hash value. However, the probability of hash collisions is very low in the case of SHA-1. An approximation for the probability p of a hash collision, given the number of possible keys d, and the number of stored keys n is [75]

$$p \approx 1 - e^{-n\left(\frac{n-1}{2d}\right)} \approx 1 - e^{-\left(\frac{n^2}{2d}\right)}$$

SHA-1 produces 160 Bit hash values, therefore the approximate collision probability for a billion elements is less than 2^{-100}. For a trillion states it is less than 2^{-80}.

$$p \approx 1 - e^{-\left(\frac{(10^9)^2}{2 \times 2^{160}}\right)} \approx 6.8 \times 10^{-31} < 2^{-100}$$

The hash function can be replaced in the future if we want to further reduce the collision probability.

The reason why we use a cryptographic hash function is that it produces values that seem to be uniformly distributed. That means if we look at the output of the SHA-1 function, it looks like random data. The reason we want to have such a hash function is that we use a search tree to store the codes. The costs for searching a value linearly depends on the height of the tree. We do not use a self-balancing tree like an AVL tree [76]. Instead we rely on the fact that if we sequentially add random values to a search tree, the order of the values is random and the resulting tree will most likely be balanced [77]. Our search tree has a high branching factor of 32, and the resulting tree is therefore rather shallow. While this is not guaranteed, we have never found a tree

with a height of more than 10 in our experiments with about 100 million states; the theoretically minimal depth of a balanced tree containing 100 million states is six.

We use a Trie (also referred to as a prefix tree) to store the states. We will refer to our implementation as the Hash-Trie. Knuth [78] defines tries as follows:

> "A trie is essentially an M-ary tree, whose nodes are M-place vectors with components corresponding to digits or characters. Each node on level ℓ represents the set of all keys that begin with a certain sequence of ℓ characters; the node specifies an M-way branch, depending on the $(\ell{+}1)$st character."

An example of a trie that stores regular strings is shown in Figure 4.8. The trie on the left contains the words "slay", "slayer" and "say". If we want to store the word "sad", we have to find the leaf node that shares the longest prefix and re-organize it.

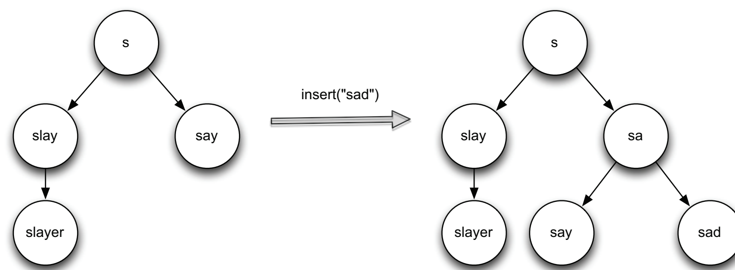


Figure 4.8: Storing words in a trie

In our case, we want to store binary strings, i.e., the hash codes of a state. We therefore use a special variant of the trie data structure inspired by Bagwell [79] and Hickey [80]. Figure 4.9 illustrates how our trie structure is organized. We use arrays as inner nodes of the trie. The length n of the array is chosen to be a power of 2. This means we can use $log_2(n)$ bits of the input string to identify an index in the array. We use the hash code of a work item as the key, so we know that all binary strings we want to add have the same length. This means that we do not have to care about cases like "slay" and "slayer" from Figure 4.8 where one input string is the prefix of another string.

In Figure 4.9. We have stored two values that share the common prefix '0110'. The next two bits of the values are different, and the values of the other bits of the input do not matter. Using chunks of 2 bits we can navigate to a value. Our actual trie has a branching factor of 32, therefore each five bit chunk of the 160 bit input is represented by one level in the trie.

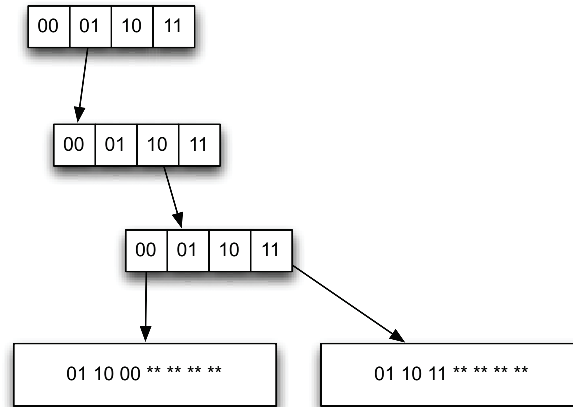


Figure 4.9: Example of a Hash Trie

The prefix for each hash code is in principle encoded in the path throughout the trie, i.e., in Figure 4.9, it would be sufficient to only store the suffix (denoted '** ** ** **') part in the leaf nodes in order to be able to reconstruct a value. We have decided to store the full hash codes instead. This requires more memory, but it is very important when we add a new computer to a running model checking job. We then need to provide the new computer with a copy of the hash codes. Because we store the full hash codes in linear blocks of memory, we can put all hash codes into a single message using `memcpy` without traversing the trie structure in the sending process. The receiving hashcode manager inserts the SHA-1 values into its own trie. In other words, the price for synchronization is almost fully paid by the new node, which is not yet involved in the model checking process. In other words, the running model checking jobs are not disrupted.

Like the regular trie, we extend the depth of the trie structure if necessary. The algorithm shown in Figure 4.11 explains how the trie is extended. We compute the index position for a specific level, i.e., we extract the chunk of bits that correspond to the level. Then we retrieve the content of the array at that index. The content could be one of four things. It can be the hash code itself; in that case we are done. It can be a null pointer; in that case we can simply store the hash code. It can be an inner node, which means we have not yet reached a leaf, therefore we recursively call the function for the next level. Finally, it can be a different hash code. In that case we create a new inner node, push down the originally stored hash code and call the function again. In the next call the index position will be different or we introduce another inner node. We know that the process will always terminate because the depth of the trie is limited by the length of the bit string.

As a simplified example with a branching factor of four, take Figure 4.10. At the bottom we have chunks of linear memory in which we store the hash codes together with some meta-data sequentially. On the top of the left side we have a node in our trie structure. The small squares represent bits that allow us to distinguish pointers to linear memory from pointers to trie nodes. In the figure we see how the trie is changed if we add a hash that is supposed to be stored in a slot that is already occupied. We create a new internal node, we move the pointer to the old content into the new node using the next five bits of the hash code as the index. We add the new content, point from the original trie node to the new node and flip the bit to represent a pointer to an internal node.

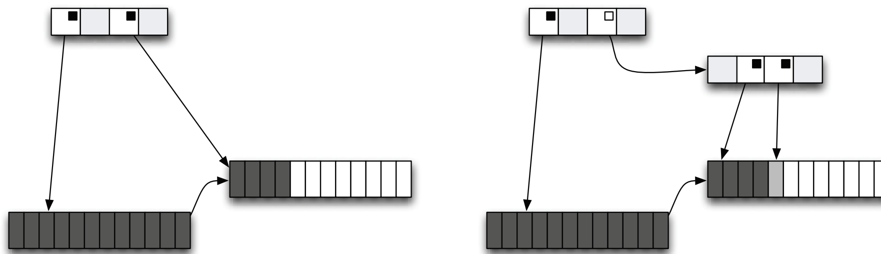


Figure 4.10: Adding a new layer to the hashtrie

```
insert(sha) = insert(sha,root,0)

insert(sha,arr,level) =
    pos = bit_chunk_for(sha,level)
    switch arr[pos]
     case sha:
       return // already present
     case nil:
       arr[pos] := sha
     case inner-node:
       insert(sha,arr[pos],level+1)
     case leaf-node:
       arr' = new array()
       pos' = bit_chunk_for(arr[pos],level+1)
       arr'[pos'] = arr[pos]
       arr[pos] = arr'
       insert(sha, arr, level)
```

Figure 4.11: Inserting a hashcode into the trie

The layout of the data that is stored in the linear memory is shown in Figure 4.12. A single state consists of the hash code, a single bit to signal if the state has been checked and optionally additional data, or seven filling-bits if no additional data is required. The

additional data could be used to store information about the event in the formal model that led to the state. This information could then be used to compute a specialized invariant in order to do proof supported model checking as described in Section 2.1 in the distributed version of ProB. We discuss this application in Section 4.4.9.

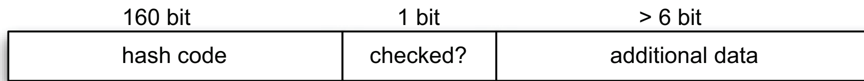


Figure 4.12: Memory layout of an hashtrie entry

It is worth noting that the decision to keep leaf nodes apart from internal trie nodes together with the decision to put the structure into the shared memory has an important implication on the implementation. We do not have a single big sequential memory block. Instead we have multiple segments. Each segment contains either leaf nodes or internal nodes of the trie. Each process maps the shared memory segments to potentially different addresses in its virtual address space as shown in Figure 4.13. This means we cannot have pointers between the memory segments. Instead, we use segment identifiers and a position index as explained in [74]. However, conceptually we can pretend to have pointers.

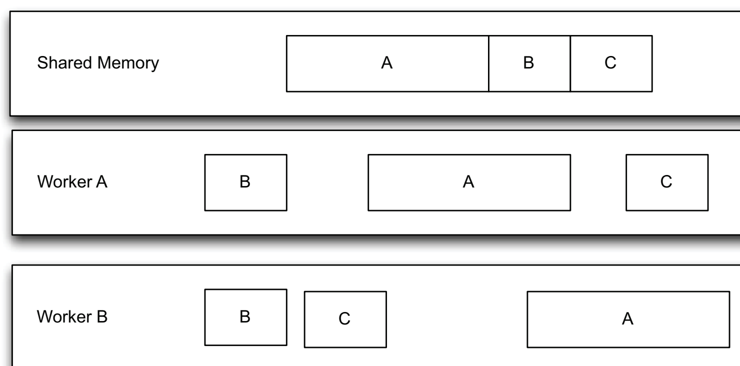


Figure 4.13: Mapping of shared memory segments into different addresses in the virtual address space

4.4.6 Communication

The communication between components is implemented using a library called ØMQ (pronounced Zero MQ) [81]. ØMQ is oriented around message queues that can be used to implement typical communication patterns, such as direct messaging or publish-subscribe. We use it for both communication and as a concurrency framework.

> "ØMQ [...] looks like an embeddable networking library but acts like a concurrency framework. It gives you sockets that carry atomic messages across various transports [...]. It's fast enough to be the fabric for clustered products. Its asynchronous I/O model gives you scalable multicore applications, built as asynchronous message-processing tasks." [81]

We use the ØMQ reactor pattern [81], a loop that queries a set of message queues in a round robin fashion and calls a specific function if a queue contains a value. We use a timeout of 1 ms per queue. While this did not cause problems in our initial experiments on a local computer and on the Amazon cloud, we recently discovered in our experiments on a high performance cluster that it is necessary to further optimize the queue handling. The reactor is part of the ØMQ library. Without the reactor loop we probably would have used threads that concurrently modify the state of the component. For instance, one thread would receive information about the currently known state space, while another thread would check a state. Both threads would have had to write to the same state, which would have been hard to implement correctly. The reactor pattern, on the other hand, allows us to decompose the concurrent algorithm into simple functions. Each of these functions has exclusive access to the state of the component because only one function is run at a time. Using a single queue and dispatching on the message could also be an option, however we have decided to use multiple queues because using different communication patterns seems to be simpler. For instance, the part in the master that is responsible for assigning ids to components uses direct messaging while the part that sends hash codes uses a publish-subscribe pattern. Fitting every communication into a single pattern might be possible, but it is clearly harder to accomplish. Also we can change priorities of certain messages if necessary.

In the following section, we describe each component in more detail. In particular, we explain the queues that are used for communication.

The hashcode manager

At the communication level, the hashcode manager is extremely simple. It has three queues: one queue to receive control information, i.e., the signal that the computation is completed and the process should terminate, one queue to receive hash codes from the master, and one queue to synchronize the initialization on a node. Both master and worker have to register with the hashcode manager and potentially wait until the manager has run the initialization. In a reactor loop, we try to receive new hash codes. If we receive hash codes, they are stored in the trie. Then we try to receive a termination signal and terminate if we receive it.

While the communication of the hashcode manager is ridiculously simple, managing the memory is rather tricky. The manager is responsible for initializing the trie data structure for a node and to setup semaphores to control the locking of parts of the trie. The different locking strategies have been discussed in detail in [74].

The master

The master component has the responsibility of coordinating the work load, managing and broadcasting the information about known states of the model, collecting data about the progress, and terminating the workers upon completion of the task. Each time a worker processes a state, it sends the result (i.e., whether the invariant holds and whether the state contains a deadlock) together with some metadata to the master. The metadata includes the SHA-1 hashes of each successor state. In addition, the worker sends statistical information containing its queue length.

The master forwards all SHA-1 hashes to the hashcode manager components, which will store them in their local trie replica. The master detects if a state has been checked independently by two workers. This can happen because the tries are updated asynchronously, i.e., one worker may use an outdated data structure. This double checking of states cannot be completely avoided, but in most of our experiments the number was negligible. The introduction of the hashcode manager component reduced the number of clashes. For instance, in the experiments on the HILBERT cluster for Section 4.1 we had four collisions for 44 workers on 15 nodes and sixteen collisions in the experiment using 104 workers on 15 nodes. Previously, we typically had collisions for less than 1% of the states. In very few exceptional cases we had up to about 10% collisions, which clearly had an effect on the results.

The master also gets statistical information. The most important bit is the size of the worker's queues. If the master detects that worker W_1 has an empty queue while worker W_2 has a certain amount of elements in its queue, the master will initiate work stealing. It sends a message to W_2 who will transmit some of the elements in its queue to the master. The master forwards the package to W_1. We use a parameter *min_queuesize* to control the minimal amount of elements in the queue and the maximal number of states a worker sends. The master will only ask a worker to send elements from its queue, if the queue size is greater than $2 \times min_queuesize$. The worker sends half of its queue but at most $5 \times min_queuesize$ states.

After sending, the worker's queue will have more then *min_queuesize* states left. There is no hard guarantee because the information about the queue size is not synchronized, i.e., the master could have made its decision based on an outdated value. However, it will prevent unnecessary transmissions sending states back and forth. In particular, if we were close to the end of a model checking job, the system would start to "jitter" if we did not use a threshold. In most cases, we can simply use the default value 100, which seems to be a reasonable choice.

One could argue that it might be more efficient to directly send the work packages from one worker to the other without the master. This is true, but it also comes at the price of complicating the setup of the processes. Changing the implementation in the future to use direct communication should be fairly simple.

The master's reactor loop consists of four queues:

1. **Receive join request.** If a worker connects to the master, it sends a join request. The master answers the request with a unique worker ID and the Prolog term that represents the initialization code. The term is then evaluated by each worker process. Typically this means the worker loads a model. The term can also contain additional information such as settings.

2. **Receive hashes request.** If a new hashcode manager joins, it will ask the master to send the hash codes that have already been broadcast. The master answers this request with all hash codes that are known.

3. **Receive statistics.** After checking a state, the worker sends a message containing statistical information such as the time spent to check the state and, more importantly, the current queue size of the worker. Based on the queue size, the master may initiate a work sharing request.

4. **Receive results.** Also after checking a state, the worker will send a message containing the result of the check (i.e., if the invariant was violated or a deadlock was found) and the hash codes of the checked state and all successors. The master extracts the hash codes and forwards them to all workers. For performance reasons, the master automatically combines information from multiple workers.

The worker

The worker components perform the actual computations. The sequence diagram shown in Figure 4.14 shows an example of a computation. The distribution framework is shown twice to emphasize that it is a nested call. We start with a work item in the queue of a worker. The worker dequeues the item (1), which consists of a Prolog term and a SHA-1 hash code. The worker retrieves the information from the hash trie using the SHA-1 value as the key. We know that the information must exists because it was stored there when the item was discovered previously. The worker checks, if the "checked?" bit is set. If it is, the worker drops the item. Otherwise, it queries Prolog using the term as the parameter (2). Prolog will process the term, i.e., it will call process/3 to produce a result and a list of successors. It will then transfer the result to the memory that is handled by the distribution framework by calling put_result and put_successor. The successor work items are enqueued in the local work queue if they have not yet been seen, i.e., if the hash code is not stored in the hash trie. The other work items are dropped. In the context of model checking, this leads to a loss of potentially useful information, namely that there is a transition from the currently checked state to that particular state. In the future, we will consider keeping and using this information, e.g., to exploit proof information as described in Section 4.4.9.

After all successor work items have been submitted, the initial call into Prolog finishes, and the framework sends a status update containing the hash code of the processed state, the hash codes of the new successors, the queue size, and some statistical information to the master process (3). This sequence is repeated until there are no more items in the worker's queue. Note that the sequence might restart if new items are put into the queue.

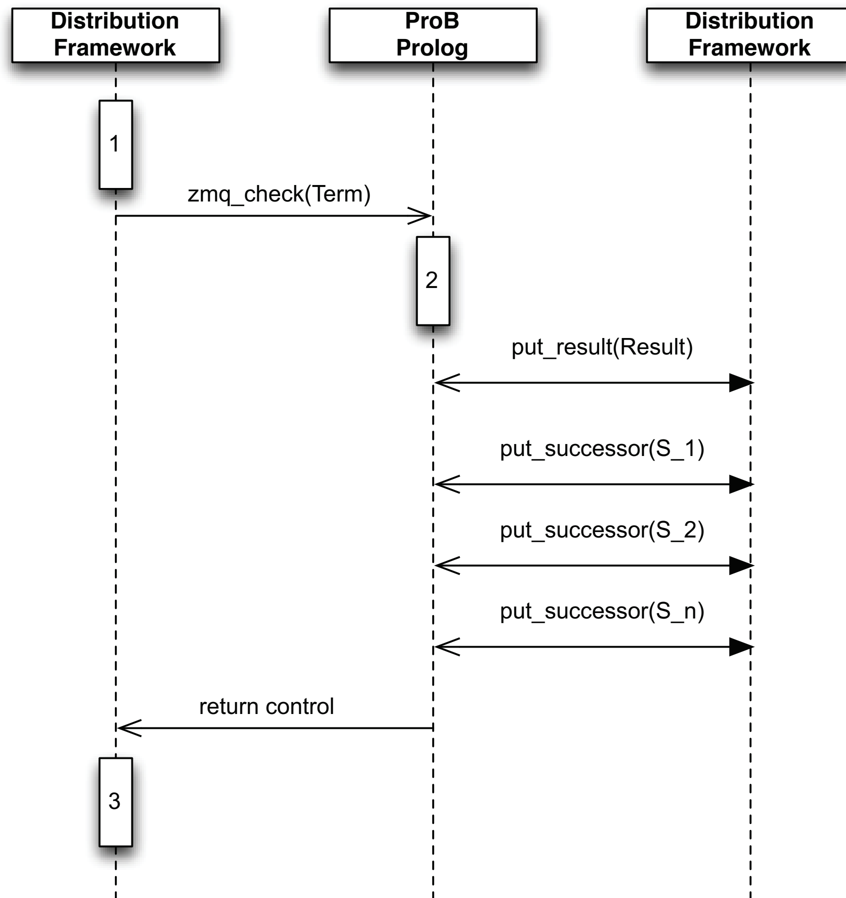


Figure 4.14: Sequence Diagram of work item processing

The reactor loop of a worker consists of the following queues:

1. **Receive work items.** Packages received from this queue contain work items that the worker should process. The worker extracts all items and enqueues them into its working queue. Currently, the master only initiates a work item transfer if the queue of the worker is empty.

2. **Receive share request.** The master notifies the worker to share a part of its queue with another worker. Upon receiving a notification from this queue, the worker will split its working queue and send half of it. This queue forms the basis of the work stealing approach together with the previously introduced receive work items queue and the balancer algorithm in the master.

3. **Receive command.** This queue contains commands from the master, e.g., a terminate command that is issued if the master notices that a user provided maximum number of states have been reached.

4. **Process state.** This queue contains a single token. If the worker receives the token it will dequeue a task from the working queue and check it. After completion, the worker will send itself a new token. The purpose of this queue is to properly interleave the checking process with the other tasks the worker performs. Most of the time, the worker will only read from this queue.

Fault tolerance

In the previous sections, we gave a slightly incomplete view on the communications between the components. We left out the durability extension implemented by Körner in [74] under the author's supervision. If we want to check large models, i.e., models where the checker would have to run for weeks, it is important to be able to recover from errors. In particular, we want to be able to resume model checking if, for instance, one of the nodes crashed. In [74] an extension to the parB prototype was developed that uses the node's harddisk to save all necessary information to recover the model checking process after the crash. This extension ensures that we can kill any process at any time and still be able to resume model checking. For instance, dequeueing a work item does not change the representation on disk until every important result has been written to some harddisk. This enables recovering the item, if the process is killed while the item was being processed.

However, the durability extension requires some additional queues for the acknowledgement of received data and heartbeating. For instance, not only the hashcode managers but also the worker subscribe to the master's hash code publication queue. The worker uses the broadcast of a hash code it has sent as the signal that the master has correctly received the data.

4.4.7 Counterexample Generation

When the normal version of ProB finds a state during model checking that violates the invariant, it presents the counterexample to the user in the form of a trace, not as a single state. In parB this is only possible with some additional work. We always discard the original state, so if we want to present a trace to the user, we have to reconstruct

it. We have only access to pairs of SHA-1 hash codes of states (v, v') where v' is the successor of v for some event. Trace reconstruction works as follows:

- We start from the state that violates the invariant and we look for its predecessor SHA-1 code. We repeat this until we reach the root state.
- Now we have a sequence of SHA-1 codes leading from the root state to the target state. We can use this sequence to construct a counterexample trace.
- We start a regular search in the root state. We compute the successor states and select the state with the matching hash code. We repeat this until we reach the target state.
- Finally, we have a trace of states that form the counterexample.

4.4.8 Symmetry Reduction

An important feature of the ProB model checker is symmetry reduction. There are three modes: nauty, permutation flooding and symmetry marker. The nauty reduction [82] uses the nauty tool to create a canonical representation of a state. Because this reduction produces the exact same state for states that are symmetrical, the distribution framework works out of the box.

The framework also supports symmetry markers [83]. The idea behind symmetry markers is to use a hash function where two symmetrical states produce the same hash value. This reduction technique is often very efficient, but it can produce collisions, i.e., it may return the same hash value for two states that are not symmetrical. This means in particular that we can miss states that violate the invariant. The distribution framework has the option to change the term that is used to produce our SHA-1 hash code. So instead of using the work item, we can tell the framework to use the symmetry marker. Figure 4.15 shows the situation with and without reduction. Without symmetry markers, the work item consists of the term that represents the task and the SHA-1 hash code of that term. If we switch on symmetry markers, we keep the task, but we change the SHA-1 code to the hash code of the corresponding symmetry marker. If another term with the same symmetry marker is discovered, it will be discarded, because the hash code is already present.

The third symmetry reduction is permutation flooding. While the first two reductions actually reduced the size of the state space, permutation flooding [84] reduces the work of checking symmetrical states by marking them as checked. If a state is checked, the

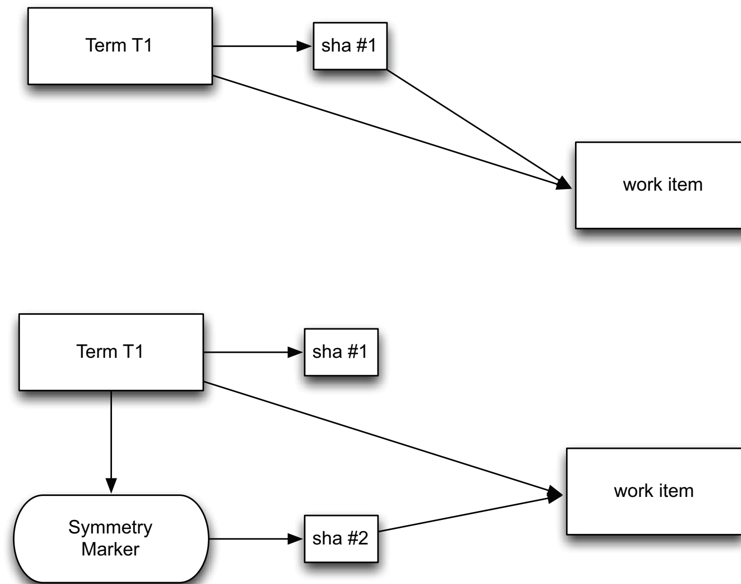


Figure 4.15: Work items without and with symmetry markers

flooding algorithm creates all symmetrical states and marks them as checked. The states are produced, but they are never checked. This can yield in a significantly improved performance. The distribution framework does not yet support permutation flooding, but we think it can be implemented without many problems, e.g., by making it possible to add a list of terms to produce the SHA-1 hash codes instead of a single term. Then we would add all hash codes to the set of known states as enqueued states. This would prevent enqueuing future occurrences of one of the symmetrical states. Another option is to add an additional parameter to the process predicate that contains a list of SHA-1 hash codes. The framework would add these SHA-1 values to the set of known states as checked. We are leaning towards the second option, but haven't implemented one of them yet.

4.4.9 Proof Support

We currently do not have an implementation for proof supported distributed model checking, but the tool is prepared to support it. As shown in Figure 4.12, we can add additional data to each hash code. For proof support, we would reserve one bit for each event, i.e., if event e at position i led to the state represented by the hash code, we would set the i-th bit to 1. If a worker discovers the same hash code again using

another event f at position j, the tool would set this bit. If a work item is processed, we can read the additional information and use the bits that are set to 1 in order to reduce the invariant. From a design point of view, we would call a combine function if we hit the same hash code passing in both values of the additional data. For model checking this combine function would simply be the boolean or operator, but for other purposes the function could be different.

4.4.10 Control UI

Starting a distributed model checking job using ProB directly is a bit cumbersome because it requires starting multiple processes on each node. As a first simple solution for a single multicore computer we wrote a shell script to start a number of workers and a master. We also developed a small web application that allows us to run the benchmark experiments more comfortably. We can produce an experiment descriptor containing parameters for the benchmark experiment such as the number of workers, the maximal number of states we want to check, the number of repetitions or ProB specific settings. We put a number of experiment descriptors together with the corresponding B models into a zip file and upload this file via the web interface. The application runs all experiments and produces result tables containing the most important information, such as the runtime, number of states, or invariant violations. For the HILBERT cluster, we developed a small job script template that could be used to start the computation. The web application could not be used for this.

We are currently developing a more sophisticated web application that will allow us to coordinate computations on the Amazon Cloud, e.g., we want to be able to add or remove computers and get more detailed information about the load on each computer. For instance, if we exhaust the memory on a single computer, we could automatically add a second computer from the cloud and move some workers to the second computer.

4.4.11 Assertion checking and Data validation

Another feature that is supported by our tool is distributed assertion checking. Typically, this feature is used for data validation. For instance, [85] describes a case study where ProB was used to verify that the assumptions made in a formally developed component for an automatic railway system are satisfied by the actual deployment. ProB was able to drastically reduce the amount of manual work.

Assertion checking is in principle just a special case of model checking, but the requirements for the parallelization framework are different enough to justify special treatment. It is performed on a single and typically very complex state consisting of large relations. The processing time for a single assertion is in the order of multiple seconds or minutes.

From a parallelization point of view, assertion checking is very simple. We exchange less information than when performing regular model checking and all tasks are statically known at loading time.

Our tool uses a different setup procedure for assertion checking than for model checking. While model checking only loads the model in each worker, assertion checking also initializes the model, and the term also contains a mapping from numbers to assertion term. This means that in the work item we can easily refer to an assertion using a number. Therefore a work item is as simple as `check(3)`. Checking an assertion never produces a successor work item. Instead, the master has some special treatment. It produces all work items, i.e., if the model has k assertions, the master generates `check(1)` ... `check(k)` and puts them in a local queue. When the master considers load balancing, it will first send a packet from that queue if there is one. We use the same mechanism to start the model checking. The master generates a special *root* work item and places it in its own queue. The load balancer will send this packet to one of the workers. In the case of assertion checking we have multiple packets and the master will send them in a round-robin fashion.

The experiments show the expected scaling. However, in our experiments we mostly had a few dominating assertions. Because these assertions are treated as one work package, the minimal time of assertion checking is at least the maximal time a single assertion requires.

We applied the tool on a model for a Communication-Based Train Control (CBTC) System of a metro line in São Paulo, Brazil. The results are shown in Table 4.1. We can see that the speedup is not linear. If we plot the duration of a single assertion check against the order of checking as in Figure 4.16 we can see the reason. The most expensive work items — which take 2 to 6 seconds each — are located at the bottom in the machine file, i.e., that they run late in the checking process which leads to a non-optimal speedup. It takes 6789 ms to check the most expensive work item, this is the lower bound for the assertion checking. We did not bother rearranging the assertions, but we expect to get closer to the optimum if we sort the assertions by difficulty. However, it is not always obvious whether an assertion is difficult to check or not.

Workers	Runtime (s)
1	34.0
2	20.0
4	15.2
5	14.7

Table 4.1: Assertion Checking of a CBTC system

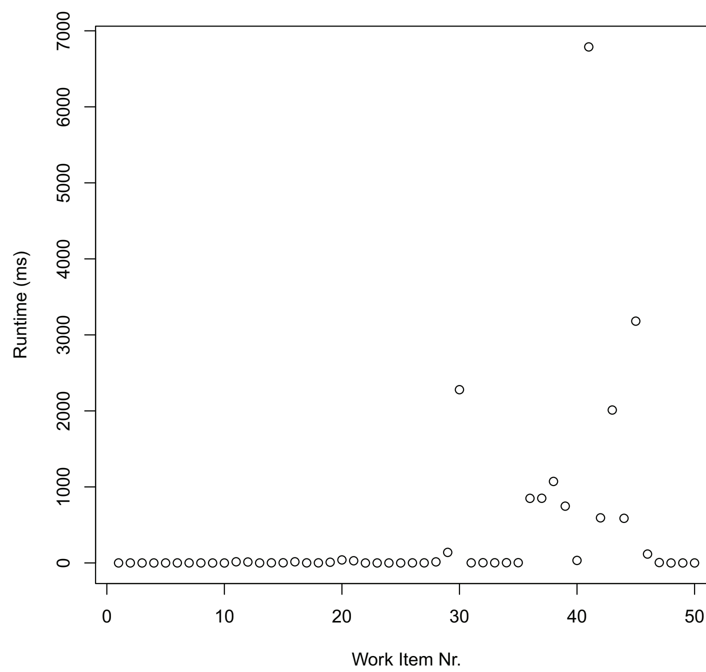


Figure 4.16: Runtimes of work items of the CBTC system

4.5 Empirical Evaluation

For our experiments we used three types of machines:

- A six core 3.33 GHz Mac Pro with 16 GB of RAM
- Multiple c3.8xlarge instances in the Amazon Cloud. A c3.8xlarge instance has 32 virtual CPUs and is equipped with 64 GB of RAM. The documentation states that "Each virtual CPU (vCPU) on C3 instances is a hardware hyper-thread from a 2.8 GHz Intel Xeon E5-2680v2 (Ivy Bridge) processor [...]" [86]. According to the Intel website, the Xeon E5-2680v2 is a 10 core processor with 20 threads.
- More recently we experimented with the high performance cluster at the university of Düsseldorf called HILBERT. Our measurements were performed on a Bull INCA cluster consisting of 112 nodes, each with 24 cores yielding a total of 2688 cores. The CPUs in use are 2.8 GHz Intel Xeon E5-2680 (Ivy Bridge) processors.

We used the Mac Pro to get an impression if and how well the B models scale. From the experiments, we chose those models that seemed to scale well and ran the benchmarks on the Amazon EC computer with a higher number of workers.

We also did few experiments using two or four c3.8xlarge instances which were connected via a 10 GBit Ethernet connection. From the tool's point of view there is no difference whether the workers are located on one computer or on multiple machines. Each worker gets the IP Address of the master, and then connects to that master, running all workers and the master on the same host is just a special case of distributed model checking.

We repeated each experiment three times on the Mac Pro. On the Amazon Cloud we only did a single run. However, in previous experiments we observed that the results did not differ by much.

For all experiments we used ProB 1.4.0-rc1. On the Mac Pro we used the revision tagged with `bm_macpro`, which was built on April 14th, 2014. On the Amazon Cloud we used a slightly newer revision which is tagged with `bm_ec2` and which was built on April 16th, 2014.

This thesis includes examples where our approach works quite well and models where our approach does not. We will analyze why the tool does not work for some of the models and give some guidelines on when not to use the parallel model checker. In particular, we will describe a very simple experimental approach to determine a good number of workers.

All experiments use breadth first search. This is not a restriction of the tool; one can freely chose among BFS, DFS or ProB's mixed mode. However, using BFS is a bit nicer for analysis. In the case of incomplete checking (e.g., if the state space is infinite), breadth first search eliminates one source of non-determinism, i.e., the choice of the successor state.

The results of our measurement are summarized in Table 4.2, 4.3, 4.4, and 4.5. Due to resource constraints, we did not run the model checker using a single worker on the Amazon cloud.

In summary, we can see that the experiments were very successful, and that our distributed model checking algorithm is clearly useful. We have achieved considerable speedups for all real-life benchmarks. For instance, we reduced the runtime of an interlocking model from around 119 minutes down to around 26 minutes by using 5 workers on the 6-core MacPro. On the Amazon cloud, we have further reduced the runtime to under 7 minutes by using an instance with 32 virtual CPUs.

Below we analyse the experimental results in more detail. In Section 4.5.2 we will analyse the (few) small benchmarks with limited scaling, and in Section 4.5.3 we will examine the other benchmarks with good scaling. Before that, we will study the impact of hyperthreading on the achievable scaling in Section 4.5.1 and the overhead of our framework in Section 4.5.1.

		Workers (#)			
Model	States (#)	1	2	4	5
Cruise control system	1361	4.8 s	2.8 s	2.4 s	2.5 s
Counter	100000[a]	18.9 s	19.2 s	20.8 s	21.8 s
Hanoi Towers	6563	26.1 s	13.8 s	9.1 s	8.9 s
Stuttgart 21	10000[a]	546.1 s	272.4 s	143.4 s	118.2 s
Interlocking	672175	7120.4 s	3599.5 s	1873.3 s	1563.8 s
USB Bus-4	16858	61.6 s	29.9 s	16.0 s	14.0 s
CAN Bus protocol	132599	137.7 s	71.7 s	44.5 s	38.1 s
RETHER	42254	83.5 s	40.8 s	22.0 s	19.3 s
Scheduler	24581	163.0 s	83.0 s	42.8 s	35.1 s
Mode Protocol	810948	4877.8 s	2436.4 s	1245.0 s	1023.5 s
Set Game	2000[a]	48.0 s	24.8 s	13.5 s	11.3 s

[a] State space not fully explored

Table 4.2: Runtime on Mac Pro

		Workers (#)			
Model	States (#)	4	8	16	31
Stuttgart 21	10000[a]	148.5 s	76.6 s	42.6 s	31.4 s
Interlocking	672175	-	1011.768 s	589.878 s	401.803 s
USB Bus-10	211042	185.5 s	104.5 s	60.6 s	39.7 s
RETHER	42.3	22.1 s	13.8 s	9.2 s	7.7 s
Scheduler	24581	44.1 s	24.4 s	15.6 s	10.9 s
Mode Protocol	810948	-	695.4 s	411.4 s	289.3 s
Set Game	10000	-	44.4 s	29.1 s	21.2 s
		Workers (#)			
Model	States (#)	8	16	32	63
Stuttgart 21	10000[a]	74.9 s	40.0 s	22.4 s	18.0 s
Interlocking	672175	988.0 s	542.5 s	326.3 s	204.6 s
Mode Protocol	810948	684.3 s	393.0 s	279.0 s	164.2 s

[a] State space not fully explored

Table 4.3: Runtime on Amazon EC2

		Workers (#)			
Model	States (#)	1	2	4	5
Cruise control system	1361	1.00	1.74	2.05	1.93
Counter	100000[a]	1.00	0.99	0.91	0.87
Hanoi Towers	6563	1.00	1.89	2.86	2.92
Stuttgart 21	10000[a]	1.00	2.00	3.81	4.62
Interlocking	672175	1.00	1.98	3.80	4.55
USB Bus-4	16858	1.00	2.06	3.84	4.41
CAN Bus protocol	132599	1.00	1.92	3.1	3.61
RETHER	42254	1.00	2.05	3.79	4.32
Scheduler	24581	1.00	1.96	3.81	4.64
Mode Protocol	810948	1.00	2.00	3.92	4.77
Set Game	2000[a]	1.00	1.93	3.55	4.24

[a] State space not fully explored

Table 4.4: Speedup factors on Mac Pro

		Workers (#)			
Model	States (#)	4	8	16	31
Stuttgart 21	10000[a]	4.00	7.75	13.95	18.92
Interlocking	672175	-	8.00	13.72	20.14
USB Bus-10	211042	4.00	7.10	12.24	18.68
RETHER	42254	4.00	6.40	9.60	11.53
Scheduler	24581	4.00	7.24	11.29	16.27
Mode Protocol	810948	-	8.00	13.52	19.24
Set Game	10000	-	8.00	12.19	16.79
		Workers (#)			
Model	States (#)	8	16	32	63
Stuttgart 21	10000[a]	8.00	14.99	26.71	33.28
Interlocking	672175	8.00	14.57	24.23	38.63
Mode Protocol	810948	8.00	14.29	19.62	33.35

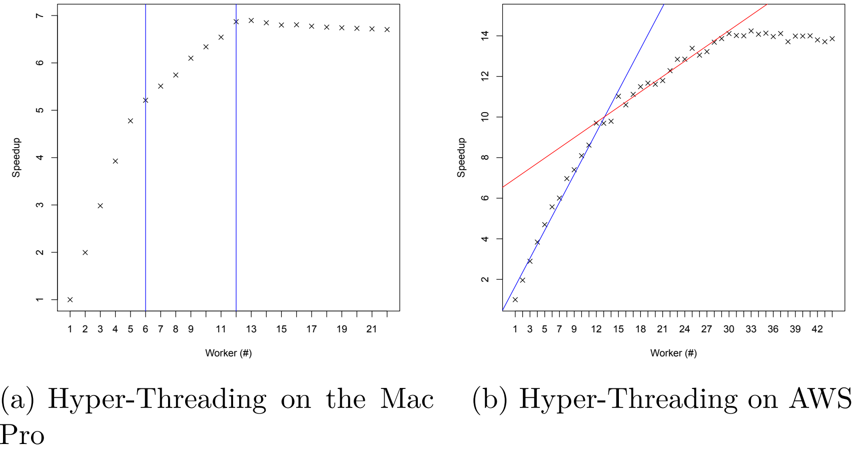
[a] State space not fully explored

Table 4.5: Speedup factors on Amazon EC2

4.5.1 Overhead

It turns out that ProB cannot scale perfectly if it does not run on a real core. Perfect scaling means that if we multiply the number of workers by k we get $1/k$ of the runtime. We can see this in Figure 4.17 where we plotted the speedup in dependency of the number of workers. We explored the full state space of a mode management protocol which we will describe in Section 4.5.3. The state space consists of 810948 states.

Our benchmark computer provides six real cores and twelve hyper-threads. If we fit a linear model on the segments we get about $speedup = 0.87 \times cores + 0.28$ for 1 to 6 cores with a correlation of 0.9938. We get $speedup = 0.95 \times cores + 0.09$ for 1 to 5 cores with a correlation of 0.9995. It does not scale up to exactly 6 because we also run the master process that uses a bit of the CPU. Between 6 and 12 workers it scales linearly, but the slope is much lower. Fitting a linear model yields $speedup = 0.27 \times cores + 3.59$ between 6 and 12 cores with a correlation of 0.9984. For more than 13 worker we actually lose performance if we add more workers. Another evidence in favor of this hypothesis is that if we look at the time that is spent within Prolog, we get 5.6 ms with a single worker and 10.1 ms with 12 workers (on the Mac Pro). The standard deviation is 1.0 ms in the case of a single core and 1.5 ms in the case of 12 cores.



(a) Hyper-Threading on the Mac Pro

(b) Hyper-Threading on AWS

Figure 4.17: Effects of the CPU / Hyper-Threads

If we use the same model on an Amazon EC2 instance, we get a similar curve. We did not use the full state space but 100000 states. The correlation is not as high as on the Mac Pro computer but we still see the same pattern. If we use a linear model, we get $speedup = 0.79 \times cores + 0.52$ for 1 to 10 cores with a correlation of 0.9955. For 11 to 32 cores we get $speedup = 0.27 \times cores + 6.40$ with a correlation of 0.9784.

From this experiment we can derive the rule that we must not run more workers than the computer has hyperthreads. We can expect good speedups only for real cores. Also, we should reserve one hyperthread for the master and hashcode manager.

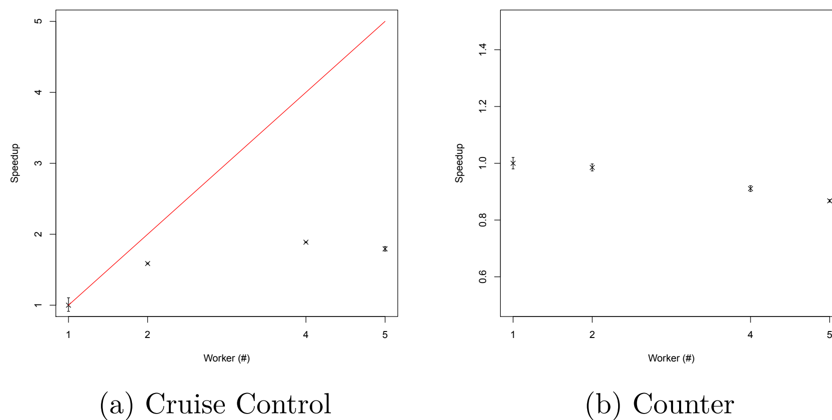
In order to measure the overhead introduced by our framework we did some experiments with models that are not supposed to scale well.

The model shown in Figure 4.18(a) does not scale well. We get a speedup of about 2 at most. However, the situation is not as bad as it seems if we look at Table 4.2. The total model checking of the cruise control system is only about 5 seconds for 1361 states. This is clearly not a very typical case where we would use parallel or distributed model checking. We would even accept a worse runtime for these kind of models. However, the experiment shows that the overhead introduced by our framework is reasonably low.

Figure 4.18(b) shows a model representing a simple counter. The model contains a single integer variable and one operation that increments the variable by one. As mentioned in Section 4.3, this is a worst case scenario for distributed model checking; exactly one worker has a single state to check and all other workers are idle. The loss of

performance is pure overhead of the distribution framework, Prolog is never called by the idle workers.

The experiment shows that we do lose a bit of performance, i.e., it is reasonable to put some effort into optimizing the number of workers. More importantly, if we use too many workers, we block resources for no good reason.



(a) Cruise Control (b) Counter

Figure 4.18: Impact of the parallelization framework

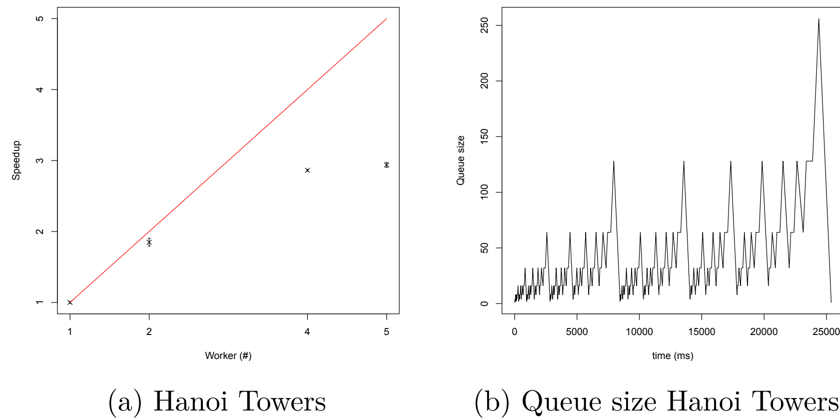
4.5.2 Benchmarks with limited scaling.

The speedup we get from parallel execution of multiple ProB processes is mainly determined by the degree of branching in the B model. Figure 4.19(a) shows the speedup for a model of the Hanoi towers[1]. Figure 4.19(b) shows the development of the queue size over time, exhibiting a repeating pattern. This pattern mirrors the recursive nature of the problem. From time to time, we reach a situation where the number of choices is very limited, e.g., if all disks are on one peg, there are only two possible moves. This means that at these points all the queues of all the workers are almost empty. This leads to the sub-linear scaling of the model.

Depth first search (DFS) performs slightly better, e.g., in the case of 4 workers the average model checking time is 7629 ms with a standard deviation of 107 ms. Breadth first search (BFS) takes 9150 ms with a standard deviation of 85 ms. Mixed mode takes 10384 ms. It performs worse than BFS and DFS. The standard deviation for mixed mode checking is 271 ms. The comparison between the search strategies was

[1] http://en.wikipedia.org/wiki/Tower_of_Hanoi

performed in a separate experiment. We did 30 repetitions for each strategy. The result in Table 4.2 matches the result of the second experiment.



(a) Hanoi Towers (b) Queue size Hanoi Towers

Figure 4.19: Limited scaling

4.5.3 Benchmarks with (almost) linear scaling

Stuttgart 21 Interlocking

The Stuttgart 21 Interlocking system (see Figure 4.20) was modeled by H. Wiegard. It is an interlocking system for the new railway station in Stuttgart. The formal model was developed with capacity simulation in mind, i.e., it should be able to answer questions like “How many trains can the station handle safely?”. We limited the experiment to 10,000 states. Checking a single state takes on average 54 ms, thus on a single core the overall model checking time is about 9 minutes. On the Mac Pro we get a speedup factor of 4.62 using 5 cores. On a single Amazon EC2 instance we get a factor of 18.9. Using two EC2 instances we get a factor of 33.3 using 63 workers. Finally, on four instances we get a factor of 68.8 using 127 workers. For this experiment we used 80,000 instead of 10,000 states. The runtime was 68 seconds, a single worker would have required about 1 hour and 18 minutes.

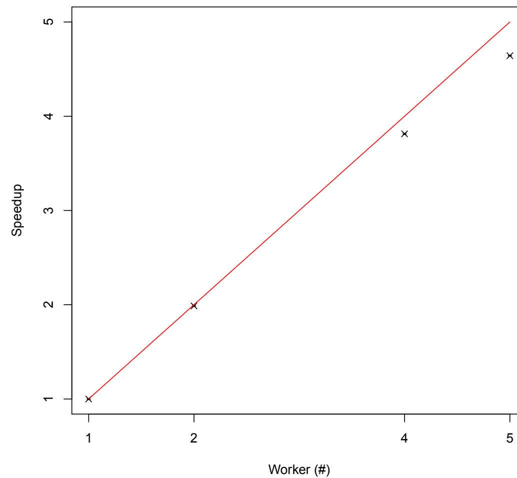


Figure 4.20: Stuttgart 21 Interlocking

Interlocking system

The other interlocking system (Figure 4.21) is a variation of the model from Abrial's book [11]. The model has a reduced state space, which was achieved by manually applying a partial order reduction. We used this reduced version on the Mac Pro and on the Amazon cloud. On a high performance cluster we did a few experiments using the unreduced model. The results of the high performance cluster are discussed separately in Section 4.5.4.

The speedup factor on the Mac Pro using 5 workers is 4.55. On a single Amazon EC2 we get a factor of 20.1 using 31 workers and using 63 workers on two instances we get a factor of 38.6.

Bus Systems

We also used two models of bus systems; a USB bus system and a Controller Area Network (CAN) Bus. CAN Busses are typically used in cars. They allow devices and controllers to communicate over a shared bus without the need of a special bus controller. Both models were developed by J. Colley. As we can see in Figure 4.22(a) and 4.22(b), both models scale reasonably well. The empirical data indicates that the CAN Bus model probably doesn't scale much further than 4 cores but at least we can

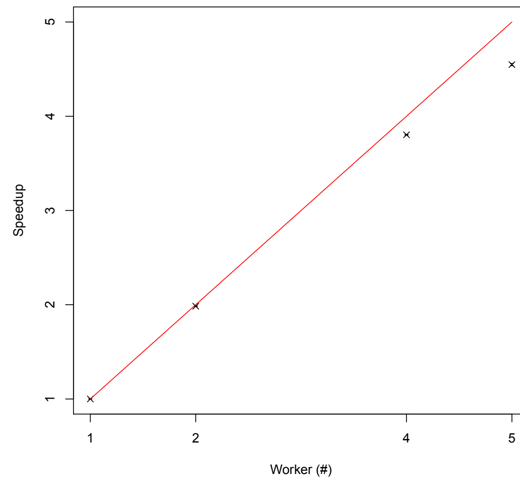
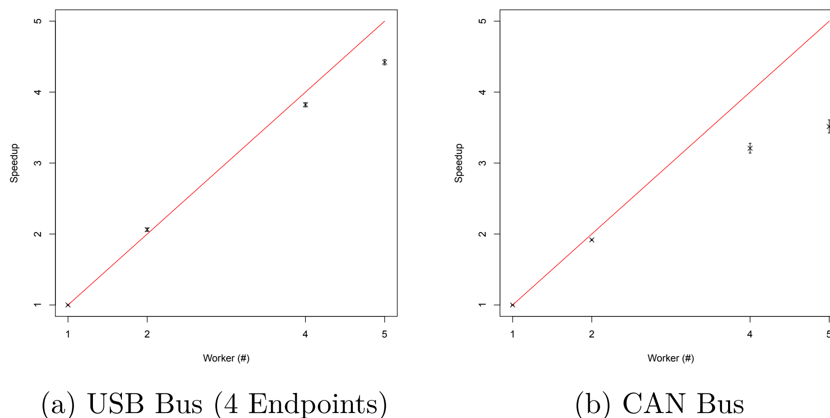


Figure 4.21: Interlocking from [11]

get a speedup factor of 3.6. The USB Bus model scales better. Using 5 cores we get a speedup factor of 4.41. If we compare the runtime for 1 and 2 workers for the USB Bus, we note something peculiar. The speedup factor is greater than 2 which is counter-intuitive. We think that neither ProB nor our tool is responsible for this behavior. It may be the CPU Turbo-Boost or CPU caching effects because the effect vanishes if we produce slightly more CPU load. Note that the CPU is only checking using two workers and one master, so it has at least 3 spare cores. On the Amazon EC2 instance we get a speedup factor of 18.7 for 31 workers.



(a) USB Bus (4 Endpoints) (b) CAN Bus

Figure 4.22: Bus Systems

Network Protocol

We used a protocol of a real time Ethernet protocol (RETHER) [87]. The B model by Büngener is a translation of a model written for the DiVinE model checker [88]. On the Mac Pro we get a speedup factor of 4.32 using 5 workers. On an Amazon instance the RETHER model scales very well up to about 8 cores yielding a speedup of a factor 6. Using 31 worker, we get a overall speedup factor of about 11.5.

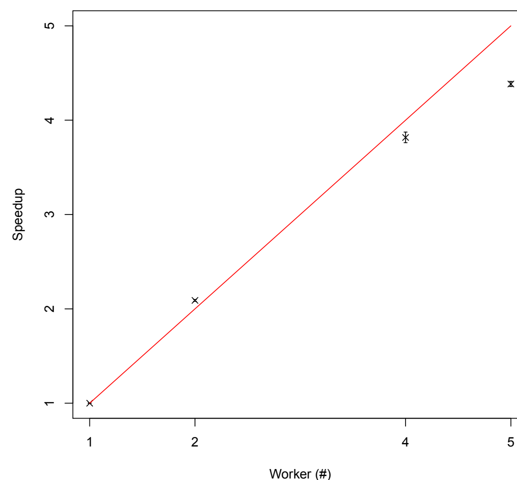


Figure 4.23: Real time Ethernet Protocol

Scheduler

We used a model of a kernel scheduler by J.-P.Bodeveix et al. [89]. On the Mac Pro we get an improvement by a factor of 4.64 using 5 cores. On the EC2 instance the scheduler also scales very well up to about 8 cores, yielding a speedup factor of about 7.24. Using 31 workers, we get a total speedup factor of about 16.25.

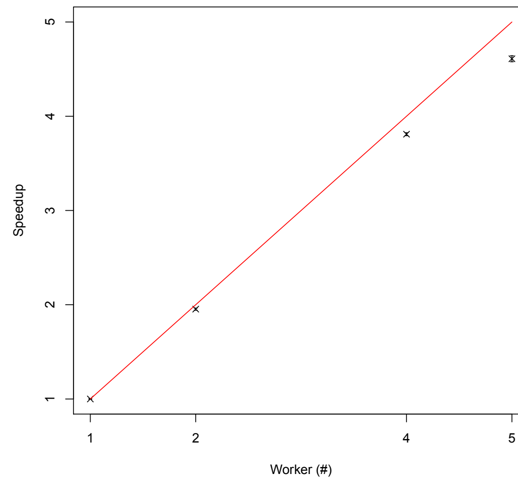


Figure 4.24: Process Scheduler

Mode Management Protocol

The model was developed by Space Systems Finland as part of a Distributed System for Attitude and Orbit Control for a Single Spacecraft (DSAOCSS) System [90, 91, 92]. The model was a part of the case study within the EU Project DEPLOY. On the Mac Pro we get a speedup factor of 4.77 using 5 workers. On a single EC2 instance we get a factor of 19.2 using 31 workers and on 2 instances we get a factor of 33.3 using 63 workers. This means we reduce the model checking time from about 1 hour 30 minutes to less than 3 minutes.

SET Game

We used a model of a game called Set[2]. The model has the interesting feature that its state space is actually a tree, i.e., for any state all the successor states are new. Also, checking the invariant becomes much more expensive the larger the depth of the state in the computation tree is. On the Mac Pro we get a speedup factor of 4.24 using 5 workers. On an Amazon EC2 instance we get a factor of 16.7.

[2] `http://en.wikipedia.org/wiki/Set_(game)`

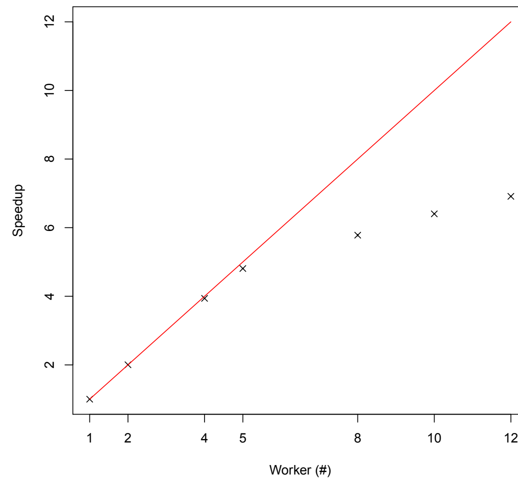


Figure 4.25: Mode protocol for a DSAOCSS system

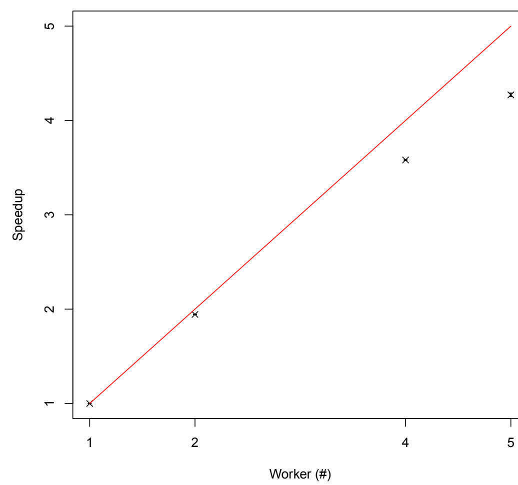
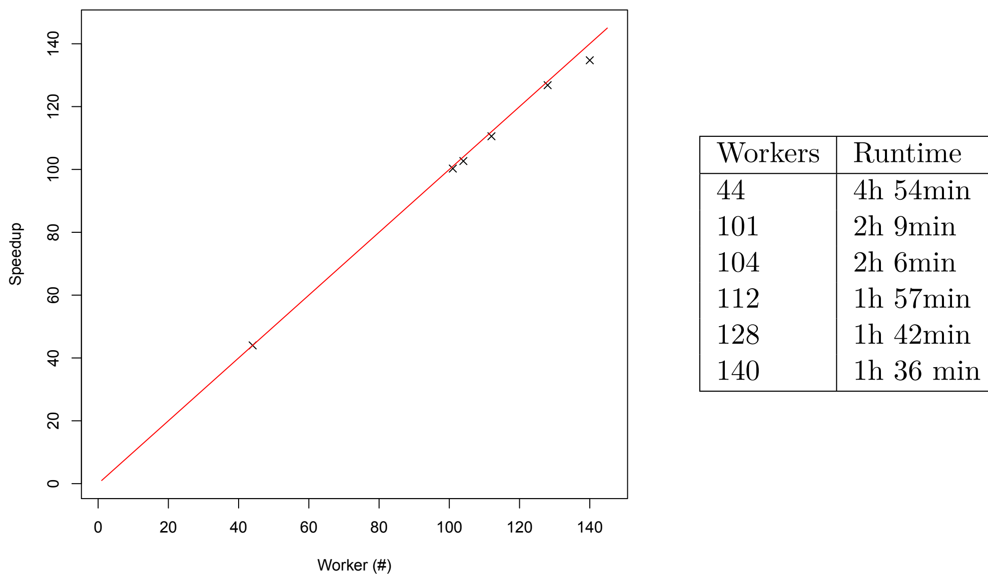


Figure 4.26: Set Game

4.5.4 High performance cluster

On the high performance cluster at the university of Düsseldorf (HILBERT) we performed an experiment with a classical B version of the interlocking system described in chapter 17 of [11]. The model's state space contains 61648077 states including all artificial states generated by ProB, e.g., the root state. As described in Section 4.1, we were not able to run the model using the regular ProB version in a reasonable time. On the Mac Pro we could get the model checking time down to 30 hours using 11 workers, on Amazon we got it down to 5 hours. In principle we could get it further down on the Amazon cloud, but decided to use the HILBERT cluster instead. Figure 4.27 shows the results; the model scales very well. We could reduce model checking time to 96 minutes and it does not yet seem to degrade much.

However, the high performance cluster exposed a problem with our reactor loop. Because each read from the queues has a timeout of 1 millisecond, the master's queue gets filled up with requests it cannot answer fast enough. This leads to timeouts on the workers and in turn a shutdown of the whole computation. We will address this issue in the near future, so that we will be able to use more workers. We hope that we can reduce the model checking time from *practically-impossible* down to *can-be-done-over-lunch*.



Workers	Runtime
44	4h 54min
101	2h 9min
104	2h 6min
112	1h 57min
128	1h 42min
140	1h 36 min

Figure 4.27: Interlocking system on a high performance cluster

4.6 Future Work

There are several points that could be addressed in the future:

- **Finding the right number of workers.** As mentioned in Section 4.3, we can use a scanning approach to find an appropriate number of workers. This process could be automated into a tool that automatically runs the experiments. However, this may not be reasonable on a cluster like HILBERT, where the jobs are under the control of an external scheduler. It may take quite some time between two jobs if the cluster has a high load.

- **Use Proof Information.** As mentioned in Section 4.4.9, during distributed model checking, we currently do not use information about discharged proof information as described in Section 2.1. As a first step we could at use the information about the transition the model checker took when encountering a state. This can be done locally by each worker, and can in some cases lead to a significant reduction in model checking time. If this first step is implemented, we could extend it with the combine function as described in Section 4.4.9.

- **Support for Message Passing Interface.** On a high performance cluster it is probably more efficient to use the Message Passing Interface (MPI) instead of ØMQ sockets. It may be worth to disentangle the communication part from the logic. The logic would call functions from an communications API. We could then exchange the implementation of the API depending on the environment. However, this refactoring is probably tricky.

- **Implement a tool for operations.** Currently running the model checker is manual work. In the case of a local computer we run the parB script or use our prototypical web application. On the cluster, we use a mixture of shell scripting and job scripts for the scheduling system. It would be very handy to have a single tool that can coordinate model checking jobs on different resources. The tool would run on the user's computer and connect to a central server that coordinates the resources. If the local resources are not sufficient the user could decide to turn the task into a job that runs on the cluster or the cloud. The server could even allocate resources on demand from the cloud. In the case of the Amazon cloud it could even use spot instances to reduce the costs. A spot instance has no fixed price. Instead, users can bid for unused instances. A drawback is that they can be shut-down by Amazon at any time if they are needed for a regular

customer. However the durability extension by Körner [74] allows resuming the model checking job even if it gets killed.

4.7 Related Work

Extending a model checker to do distributed or parallel model checking is very common. This means that there is a lot of related work already done. However, it is very different to directly compare model checking in ProB with other model checkers. On one hand, the high abstraction level of B leads to a very different characteristic when it comes to metrics like states per second. On the other hand, ProB only checks the invariant (and deadlock freedom) while most other model checker can check LTL properties.

Invariant checking is a very simple special case of LTL model checking, where the only kind of formula we consider is of the form $G\phi$ where ϕ is some atomic proposition and G is the temporal operator "globally". In our case the atomic proposition is the invariant.

Restricting the possible formulas to this special case makes a huge difference when it comes to the model checking algorithm. ProB can simply explore the state space and check each state individually. In contrast, a model checker that checks full LTL has to consider paths as well. One way to find counterexamples is to transform the state space and the negation of the LTL property into Büchi automatons and check if their intersection is empty, i.e., the product automaton must not accept any word. Typically this is done by finding strongly connected components (SCCs) that contain an accepting state. A counterexample is a path leading to a strongly connected component and a loop inside the component. This means an LTL counterexample has the shape of a lasso.

This clearly has an impact on how the model checker and also its distributed or parallel version works. As a result, we cannot claim to compete in the same league as DiVinE [88], TLC [93] or Spin [65, 94, 95]. These tools check real temporal properties, but on the other hand, their formalisms are at a very low level of abstraction.

Note that ProB can also check full LTL, but this is not part of the distributed version of the tool. If we want to extend ProB's LTL model checker to work in a parallel or distributed way, we would probably use similar techniques as the other model checkers.

TLC

The closest match for ProB is probably the model checker TLC for TLA$^+$, a rather high-level language. TLC can run in parallel and also in distributed mode. This work was presented by Markus Kuppe at the FM'12 TLA workshop and the TLA workshop co-located with ABZ 2014. Kuppe's latest work also contains a tool for controlling operations. It can automatically allocate resources on the Amazon cloud and will definitely be an influence for our own tool. In [96] and [97] Hansen has developed a translation from B to TLA$^+$ and from TLA$^+$ to B allowing us to use ProB on TLA$^+$ models TLC on some B models. Applied to low-level models TLC outperforms ProB, but many high-level B models cannot be model checked using TLC because it lacks the constraint solver.

Another downside of TLC is that the hash function it uses to create fingerprints of the states is limited to 64 bit. This makes hash collisions much more likely than the hash function used by ProB. However, TLC also checks real temporal properties.

DiVinE

The DiVinE model checker uses the DVE input language and it can also run on LLVM bitcode.

The LLVM support is particularly interesting because it can be used to check C/C++ programs using an LLVM back end of the compiler. No real IO is allowed in the C code because the model checker requires a fully controlled environment. However, some IO can be simulated, in particular the POSIX thread API. This "enables verification of unmodified multithreaded programs. In particular, DiVinE explores all possible thread interleavings systematically at the level of individual bitcode instructions. This allows DiVinE, for example, to virtually prove an absence of deadlock or assertion violation in a given multithreaded piece of code, which is impossible with standard testing techniques." [98]

DiVinE uses a so called "One way catch them young" algorithm [99] to detect cycles and, until version 3.0, a static partitioning approach with a per-thread hash table. In version 3.0 there is an experimental approach for a single shared hash table. In [98] the authors state that while "algorithms using traditional static partitioning and per-thread hash tables provide reasonable scalability, a single shared hash-table and dynamic work partitioning can give substantially better results".

Another interesting new feature of DiVinE is the Common Explicit-State Model Interface (CESMI). "The CESMI specification defines a simple interface between the model-checking core and a loadable module representing the model. Generation of model states is driven by the needs of the model checking engine" [98]. This is very close to the way ProB's own LTL model checker works. CESMI could be a way integrate ProB into DiVinE, allowing the application of DiVinE to a wider range of formalisms.

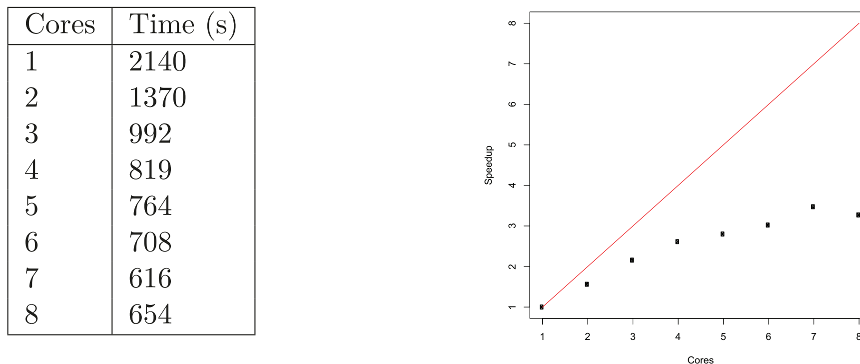
Spin

Spin is probably the best known model checker. It uses Promela as its input language. The development of Spin started around 1980. In 2002 it was awarded with the ACM Software System Award. It was successfully applied to many industrial models, for instance, for mission critical software at NASA.

In contrast to other tools, Spin does not perform the model checking. Instead it translates the specification into C code that also contains the verification code; the C code is then compiled to an executable. Running the executable performs the model checking for the specification. This is a bit cumbersome but it is also very fast.

Spin can also run on multi-core processors using an extension introduced in [100]. Holzman and Bosnaki conclude that they "provided evidence to show that the effect of both compiler optimization techniques and search optimization techniques such as partial order reduction diminish the benefits of multi-core processing. For applications of interest though, i.e., large applications with embedded C code and relatively costly transition functions, or large embedded data structures, the benefits especially for the verification of safety properties can be significant" and "Finding a liveness verification algorithm that retains the low complexity of the nested depth-first search method used in SPIN, yet can scale with increasing numbers of CPU cores, is as yet an open problem" [100].

Spin is incredibly fast on a single core. This makes it very hard to implement a parallel version that scales well. For instance, the following data was taken from the Spin website (`http://spinroot.com/spin/multicore/Table5_Fig10/Data_Gurdag/summary`). It describes the result of running Spin on up to 8 cores on a model. The model itself was not available, so we did not reproduce the results. The results and the speedup curve are shown in Figure 4.28. We can see that the scaling is worse than in ProB. However, this is mainly because it is much easier to scale up a model checker that runs with a rather low speed on states that are very abstract than to scale up a model checker that checks low level states at a very high rate.



Cores	Time (s)
1	2140
2	1370
3	992
4	819
5	764
6	708
7	616
8	654

Figure 4.28: Gurdag Model with multi-core Spin

Murϕ

There are at least two model checkers for the Murϕ language that support parallel or distributed model checking, Eddy_Murphi [101] and PReach. Eddy_Murphi assigns a specific worker to each state. To reduce the communication, states for other hosts are collected in a queue and bundled in a single message. Eddy_Murphi uses two threads, one thread performs the actual computation and the second thread is handling the communication. The threads communicate via shared memory. Communication between hosts is done using the Message Passing Interface (MPI).

A similar approach could also be used for ProB. Although Sicstus Prolog is single threaded, the C extension is not limited to a single thread. In principle we could split the worker into one thread that communicates and one thread that calls Prolog.

PReach [102] is a model checker that sits on top of a tweaked version of the Murϕ tool. It was written in Erlang and is impressively small. "We use the original Murphi front-end to parse the model description, a layer written in Erlang to handle the communication aspects of the algorithm, and also use Murphi as a back-end for state expansion and to store the hash table. This allowed a clean and simple implementation, with the core parallel algorithms written in under 1000 lines of code." [102]

Like Eddy_Murphi, PReach assigns each state to a specific worker. The worker's queues are stored on disk rather than kept in memory. This improves the memory footprint drastically. Our own implementation does something similar as described by Körner [74]. PReach has been applied to very large industrial use cases. The authors of PReach claim

that they “have used PREACH to model check an industrial cache coherence protocol with approximately 30 billion states”.

LTSmin

LTSmin [103, 104] is a language independent model checker. There are front ends for mu-CRL, mcrl2, DVE, Promela, UPPAAL’s timed automata. There is ongoing research on integration of ProB and LTSmin using the Partitioned Next-State Interface (PINS).

Load balancing is done using something similar to work stealing. If a worker becomes idle it asks another random worker for some work items until it finds another worker that is willing to share some work.

Chapter 5

Conclusion

The goal of this thesis was to significantly improve the performance of the ProB model checker using static analysis and parallel execution. We also wanted to extend the applicability of ProB for large formal models. Both goals have been achieved. We are now able to check formal models that are orders of magnitude larger.

5.1 Contributions

5.1.1 Proof supported and directed model checking

The status of a proof obligation carries valuable information for a model checker. If a part of the invariant has been proven to be preserved, we do not have to check it during model checking. Also we can further reduce the invariants using aggregation of incoming transitions during model checking. We have explained how the number of invariants left for a set of states can be exploited to direct the model checker. Finally, we have formalized the proof supported model checking in Event-B and we have proven its equivalence to regular model checking.

The following excerpt of the experimental results show that the introduction of proof information into the model checker significantly improved the performance of ProB. In some cases the speedup factor was 1.5 to 2. In no case we significantly lost performance.

	w/o proof information [ms]	using proof information [ms]	Speedup Factor
Mondex m2	1747 ± 21	1767 ± 38	0.99
Earley Parser m2	309810 ± 938	292093 ± 1076	1.06
Scheduler	9387 ± 124	8167 ± 45	1.15
SAP	50783 ± 232	34927 ± 114	1.45
Siemens	51560 ± 254	24127 ± 93	2.14
CXCC	18470 ± 151	6700 ± 36	2.76

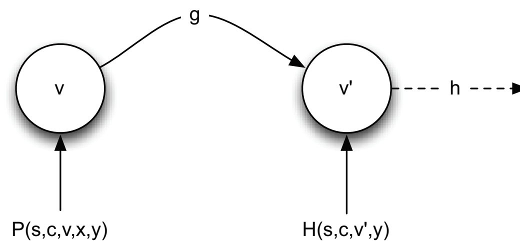
Using proof information it is now the default setting in ProB.

5.1.2 Flow Analysis

We define a theoretical framework that allows us to analyze a formal model to answer questions about the flow of execution in the model, i.e., we can infer the algorithmic structure of an Event-B model. This structure is typically hidden in the refinement chain, yet it bears valuable information for people who try to understand the model.

We introduced a notion of event independence that can be used to reduce the effort of evaluating guards and in a strict form it can be used to reduce the effort of computing complete states.

We also developed the notion of an enabled graph that can also be used to reduce the costs of guard evaluation. Informally, we allow the model checker to look one step into the future.



If P is true in state v and g happens, then in the next state H will be true and therefore event h will be possible.

Finally we introduced a flow graph that can be used to help covering events when generating test cases. It can be used for code generation, for directing the model checker

and also to prove deadlock freedom. However, computing the flow graph can become infeasible.

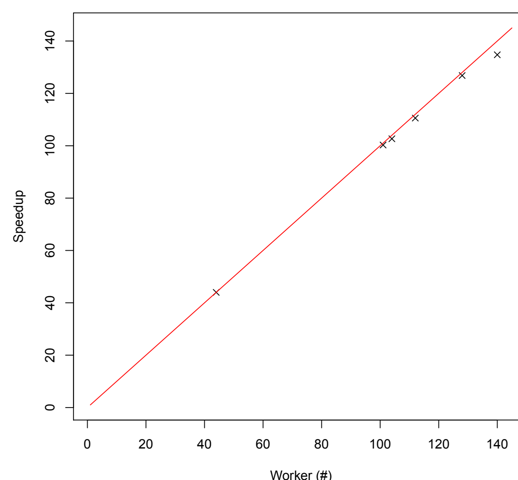
5.1.3 Distributed model checking

The distributed model checker is fundamentally changing ProB's applicability. We can now check very large models that were practically impossible to check previously. We are very close to perform model checking on specifications with a few billion states. Previously ProB was limited to some 10 million states.

Our implementation uses multiple processes coordinated using a master process. They share information about the set of known work items using an eventual consistent approach. Work is distributed using work stealing.

The framework is flexible, it is by no means restricted to model checking. Anything that can be implemented as a Prolog predicate which takes a work item as the input and produces a result and new work items can be run in parallel using our framework.

The tool can run on single multicore computers as well as on clusters, e.g., the Amazon cloud. It can also run on high performance clusters. We evaluated the framework on each of these types of systems and we can conclude that the scaling is very good provided the model is suited for parallel checking, i.e., the model's state space contains reasonable branching. Abrial's interlocking system [11] has a suited branching factor and results in an almost linear scaling even for more than 100 worker processes on a high performance cluster as shown in the following picture:



Appendix

Appendix A

Formal Model of Proof Supported Model Checking

These are the contexts and machines that we used to prove the equivalence of regular model checking and proof supported model checking.

C1 Context

CONTEXT c1_modelelements

SETS

INVARIANTS

EVENTS

STATES

CONSTANTS

truth

preserve

violate

AXIOMS

`type1` : $truth \subseteq \mathrm{STATES} \times \mathrm{INVARIANTS}$

`type2` : $partition(STATES, preserve, violate)$

`link1` : $preserve = \{s | \{s\} \times \mathrm{INVARIANTS} \subseteq truth\}$

thm3 : $\forall t \cdot (\forall i \cdot t \mapsto i \in truth) \Rightarrow t \in preserve$

thm4 : $preserve = \{s | \forall i \cdot s \mapsto i \in truth\}$

thm5 : $violate = \{s | \exists i \cdot s \mapsto i \notin truth\}$

$thm7$: $\forall t \cdot t \in preserve \Rightarrow (\forall i \cdot t \mapsto i \in truth)$

$thm7b$: $\forall t \cdot t \notin preserve \Rightarrow (\exists i \cdot t \mapsto i \notin truth)$

$thm7c$: $\forall t \cdot t \in violate \Rightarrow (\exists i \cdot t \mapsto i \notin truth)$

$thm8$: $\mathrm{STATES} \setminus violate = preserve$

$thm9$: $\mathrm{STATES} \setminus preserve = violate$

END

C2 Context

CONTEXT c2_statespace
EXTENDS c1_modelelements
CONSTANTS

$transitions$

$root$

AXIOMS

`type1` : $transitions \in \mathrm{STATES} \leftrightarrow \mathrm{STATES}$

`type2` : $root \in \mathrm{STATES}$

`axm1` : $\mathrm{STATES} \subseteq cls(transitions)[\{root\}] \cup \{root\}$

`axm2` : $root \in preserve$

END

C3 Context

CONTEXT c3_modelchecker
EXTENDS c2_statespace
SETS

MC_STATE

CONSTANTS

running

terminated_ok

terminated_ce

AXIOMS

`type` : $partition(\mathrm{MC_STATE}, \{running\}, \{terminated_ce\}, \{terminated_ok\})$

thm2 : $\forall s \cdot s \notin preserve \Leftrightarrow (\exists i \cdot i \in \mathrm{INVARIANTS} \land (s \mapsto i) \notin truth)$

END

C4 Context

CONTEXT c4_mc_states
EXTENDS c3_modelchecker
SETS

APC_MC1

APC_MC2

CONSTANTS

select_state

compute_successors

check_invariant

start_successors

step_successors

AXIOMS

`type1` : $partition(\text{APC_MC1}, \{select_state\}, \{compute_successors\}, \{check_invariant\})$

`type2` : $partition(\text{APC_MC2}, \{start_successors\}, \{step_successors\})$

END

C5 Context

CONTEXT c5_labels
EXTENDS c4_mc_states
CONSTANTS

$labels$

$initialization$

AXIOMS

`type1` : $labels \in transitions \rightarrow \mathrm{EVENTS}$

`type2` : $initialization \in \mathrm{EVENTS}$

END

C6 Context

CONTEXT c6_discharge_info
EXTENDS c5_labels
CONSTANTS

discharged

single_specialized_invariant

AXIOMS

`type1` : $discharged \in \mathrm{EVENTS} \rightarrow \mathbb{P}(\mathrm{INVARIANTS})$

`type2` : $single_specialized_invariant \in \mathrm{EVENTS} \rightarrow \mathbb{P}(\mathrm{INVARIANTS})$

`meaning` : $\forall s, t, e, i \cdot (s \mapsto t) \mapsto e \in labels \land i \in discharged(e) \land$
$s \in preserve \Rightarrow t \mapsto i \in truth$

`amx1` : $single_specialized_invariant = \{e, is \cdot e \in \mathrm{EVENTS} \land$
$is = \mathrm{INVARIANTS} \setminus discharged(e) | e \mapsto is\}$

`axm2` : $\forall e \cdot e \in \mathrm{EVENTS} \Rightarrow$
$partition(\mathrm{INVARIANTS}, discharged(e), single_specialized_invariant(e))$

`axm3` : $labels[\{root\} \lhd transitions] = \{initialization\}$

thm1 : $\forall s, t, e \cdot (s \mapsto t) \mapsto e \in labels \land s \in preserve \Rightarrow$
$(\forall i \cdot i \in discharged(e) \Rightarrow t \mapsto i \in truth)$

END

M0 Machine

MACHINE m0
SEES c3_modelchecker
VARIABLES

$result$

$counterexample$

INVARIANTS

`type1` : $result \in \text{MC_STATE}$

`type2` : $counterexample \subseteq violate$

EVENTS
Initialisation

begin

`setresult` : $result := running$

`setce` : $counterexample := \varnothing$

end

terminateBroken $\widehat{=}$

any

s

where

`grd1` : $s \in violate$

then

`act1` : $counterexample := \{s\}$

`act2` : $result := terminated_ce$

end

terminateOK $\widehat{=}$

when

`grd1` : $violate = \varnothing$

then

`act1` : $counterexample := \varnothing$

`act2` : $result := terminated_ok$

end

END

M1 Machine

MACHINE m1_abstract_mc
REFINES m0
SEES c3_modelchecker
VARIABLES

ok

$broken$

$unknown$

$result$

$counterexample$

INVARIANTS

`type1` : $ok \subseteq \mathrm{STATES}$

`type2` : $broken \subseteq \mathrm{STATES}$

`type3` : $unknown \subseteq \mathrm{STATES}$

`type4` : $result \in \mathrm{MC_STATE}$

`type5` : $counterexample \subseteq \mathrm{STATES}$

`correct1` : $ok \subseteq preserve$

`correct2` : $broken \subseteq violate$

`link` : $partition(STATES, ok, broken, unknown)$

thmlink1 : $partition(STATES, ok, violate, unknown \setminus violate)$

thmlink2 : $partition(STATES, preserve, broken, unknown \setminus preserve)$

lemma2 : $unknown = \varnothing \Rightarrow ok \cup broken = preserve \cup violate$

`correct4` : $result \neq running \Rightarrow (ok = STATES \vee counterexample \neq \varnothing)$

EVENTS
Initialisation

begin

`act1` : $ok, broken, unknown := \varnothing, \varnothing, STATES$

`act2` : $counterexample, result := \varnothing, running$

end

checkOK $\widehat{=}$

any

s

where

`grd1` : $s \in unknown$

`grd2` : $s \in preserve$

then

`act1` : $ok, unknown := ok \cup \{s\}, unknown \setminus \{s\}$

end

checkBroken $\widehat{=}$

any

s

where

`grd1` : $s \in unknown$

`grd2` : $s \notin preserve$

then

`act1` : $broken, unknown := broken \cup \{s\}, unknown \setminus \{s\}$

end

terminateBroken $\widehat{=}$

refines *terminateBroken*

any

s

where

`grd1` : $s \in broken$

then

`act1` : $counterexample := \{s\}$

`act2` : $result := terminated_ce$

end

terminateOK $\widehat{=}$

refines *terminateOK*

when

`grd1` : $broken = \varnothing$

`grd2` : $unknown = \varnothing$

then

`act1` : $counterexample := \varnothing$

`act2` : $result := terminated_ok$

end

END

M2 Machine

MACHINE m2_mc_cycle
REFINES m1_abstract_mc
SEES c4_mc_states
VARIABLES

$mcpc1$

$mcpc2$

$succs$

$current$

$queue$

ok

$broken$

$unknown$

$result$

$counterexample$

INVARIANTS

`type0` : $succs \subseteq \mathrm{STATES}$

`type1` : $mcpc1 \in \mathrm{APC_MC1}$

`type1b` : $mcpc2 \in \mathrm{APC_MC2}$

`type3` : $queue \subseteq \mathrm{STATES}$

`notinqueue` : $current \notin queue$

`pc` : $succs \neq \varnothing \Rightarrow mcpc1 = compute_successors$

EVENTS
Initialisation

extended

begin

`act1` : $ok, broken, unknown := \varnothing, \varnothing, STATES$

`act2` : $counterexample, result := \varnothing, running$

`initroot` : $current := root$

`initqueue` : $queue := \varnothing$

`pc1` : $mcpc1 := check_invariant$

`pc2` : $mcpc2 := start_successors$

`succs` : $succs := \varnothing$

end

select_state $\widehat{=}$

any
s
where
`grd1` : $s \in queue$
`grdpc` : $mcpc1 = select_state$
then
`act1` : $current := s$
`act2` : $queue := queue \setminus \{s\}$
`setpc` : $mcpc1 := check_invariant$
end

checkOK $\widehat{=}$
refines *checkOK*

when
`grd1` : $current \in preserve$
`grd2` : $current \in unknown$
`grd3` : $mcpc1 = check_invariant$
with
`s` : $current = s$
then
`act1` : $mcpc1 := compute_successors$
`act2` : $ok, unknown := ok \cup \{current\}, unknown \setminus \{current\}$
end

checkBroken $\widehat{=}$
refines *checkBroken*

when
`grd1` : $current \notin preserve$
`grd2` : $current \in unknown$
`grd3` : $mcpc1 = check_invariant$
with
`s` : $current = s$
then
`act1` : $mcpc1 := compute_successors$
`act2` : $broken, unknown := broken \cup \{current\}, unknown \setminus \{current\}$
end

successors_start $\widehat{=}$

any
$next$
where
`pc` : $mcpc1 = compute_successors$
`pc2` : $mcpc2 = start_successors$
`successors` : $next = (transitions[\{current\}] \cap unknown)$
then
`act` : $succs := next$

actpc : $mcpc2 := step_successors$

end

successors_step $\widehat{=}$

any

s

where

pc : $mcpc2 = step_successors$

grd1 : $s \in succs$

grd2 : $s \notin queue$

grd3 : $s \neq current$

then

act1 : $succs := succs \setminus \{s\}$

act2 : $queue := queue \cup \{s\}$

end

successors_skip $\widehat{=}$

any

s

where

pc : $mcpc2 = step_successors$

grd1 : $s \in succs$

grd2 : $s \in queue \vee s = current$

then

act1 : $succs := succs \setminus \{s\}$

end

successors_stop $\widehat{=}$

when

grd0 : $mcpc2 = step_successors$

grd1 : $succs = \varnothing$

then

pc2 : $mcpc2 := start_successors$

pc1 : $mcpc1 := select_state$

end

terminateBroken $\widehat{=}$

extends *terminateBroken*

any

s

where

grd1 : $s \in broken$

then

act1 : $counterexample := \{s\}$

act2 : $result := terminated_ce$

end

terminateOK $\widehat{=}$
extends *terminateOK*

when

`grd1` : $broken = \varnothing$

`grd2` : $unknown = \varnothing$

then

`act1` : $counterexample := \varnothing$

`act2` : $result := terminated_ok$

end

END

M3 Machine

MACHINE m3_INVARIANTS_cycle
REFINES m2_mc_cycle
SEES c4_mc_states
VARIABLES

invs

checked

unchecked

$mcpc1$

$mcpc2$

succs

current

queue

ok

broken

unknown

result

counterexample

INVARIANTS

`type1` : $invs \in \text{STATES} \leftrightarrow \text{INVARIANTS}$

`type2` : $checked \in \text{STATES} \leftrightarrow \text{INVARIANTS}$

`link1` : $\forall s \cdot s \in dom(invs) \Rightarrow ((\forall i \cdot (s \mapsto i) \in truth) \Rightarrow s \in preserve)$

`link0` : $partition(invs, checked, unchecked)$

link_p : $unchecked = invs \setminus checked$

`link_c` : $checked \subseteq truth$

`all_or_nothing` : $\forall s \cdot s \in dom(invs) \Rightarrow \{s\} \times \text{INVARIANTS} \subseteq invs$

link2 : $\forall s \cdot s \in dom(invs) \Rightarrow (s \notin preserve \Rightarrow \exists i \cdot (s \mapsto i) \in unchecked)$

EVENTS
Initialisation

extended

begin

`act1` : $ok, broken, unknown := \varnothing, \varnothing, STATES$

`act2` : $counterexample, result := \varnothing, running$

`initroot` : $current := root$

`initqueue` : $queue := \varnothing$

`pc1` : $mcpc1 := check_invariant$

`pc2` : $mcpc2 := start_successors$

`succs` : $succs := \varnothing$

`act3` : $invs := \{root\} \times \mathrm{INVARIANTS}$

`act4` : $unchecked := \{root\} \times \mathrm{INVARIANTS}$

`act5` : $checked := \varnothing$

end

select_state $\widehat{=}$

extends *select_state*

any

s

where

`grd1` : $s \in queue$

`grdpc` : $mcpc1 = select_state$

then

`act1` : $current := s$

`act2` : $queue := queue \setminus \{s\}$

`setpc` : $mcpc1 := check_invariant$

end

check_true_inv $\widehat{=}$

any

i

where

`grd2` : $i \in unchecked[\{current\}]$

`grd3` : $current \mapsto i \in truth$

`grd4` : $mcpc1 = check_invariant$

then

`remove1` : $checked := checked \cup (\{current \mapsto i\})$

`remove2` : $unchecked := unchecked \setminus (\{current \mapsto i\})$

end

checkOK $\widehat{=}$

refines *checkOK*

when

`grd2` : $checked[\{current\}] = \mathrm{INVARIANTS}$

`grd3` : $mcpc1 = check_invariant$

`grd4` : $current \in unknown$

then

`act1` : $mcpc1 := compute_successors$

`act2` : $ok, unknown := ok \cup \{current\}, unknown \setminus \{current\}$

end

checkBroken $\widehat{=}$
refines *checkBroken*

any
i
where
`grd1` : $i \in unchecked[\{current\}]$
`grd3` : $current \mapsto i \notin truth$
`grd4` : $current \in unknown$
`grdpc` : $mcpc1 = check_invariant$
then
`act1` : $mcpc1 := compute_successors$
`act2` : $broken, unknown := broken \cup \{current\}, unknown \setminus \{current\}$
end

successors_start $\widehat{=}$
extends *successors_start*

any
$next$
where
`pc` : $mcpc1 = compute_successors$
`pc2` : $mcpc2 = start_successors$
`successors` : $next = (transitions[\{current\}] \cap unknown)$
then
`act` : $succs := next$
`actpc` : $mcpc2 := step_successors$
end

successors_step $\widehat{=}$
refines *successors_step*

any
s
is
where
`pc` : $mcpc2 = step_successors$
`grd1` : $s \in succs$
`grd2` : $s \notin queue$
`grd3` : $s \neq current$
`grd4` : $\forall i \cdot i \in is \Rightarrow s \mapsto i \in truth$
`grd5` : $s \notin dom(invs)$
then
`act1` : $succs := succs \setminus \{s\}$
`act2` : $queue := queue \cup \{s\}$

act3 : $checked := checked \cup (\{s\} \times is)$
act4 : $unchecked := unchecked \cup (\{s\} \times (\mathrm{INVARIANTS} \setminus is))$
act5 : $invs := invs \cup (\{s\} \times \mathrm{INVARIANTS})$

end

successors_skip $\widehat{=}$
refines *successors_skip*

any

s

is

where

pc : $mcpc2 = step_successors$
grd1 : $s \in succs$
grd2 : $s \in queue \lor s = current$
grd3 : $\forall i \cdot i \in is \Rightarrow s \mapsto i \in truth$
grd4 : $(\{s\} \times \mathrm{INVARIANTS}) \subseteq invs$

then

act1 : $succs := succs \setminus \{s\}$
act2 : $unchecked := unchecked \setminus (\{s\} \times is)$
act3 : $checked := checked \cup (\{s\} \times is)$

end

successors_stop $\widehat{=}$
extends *successors_stop*

when

grd0 : $mcpc2 = step_successors$
grd1 : $succs = \varnothing$

then

pc2 : $mcpc2 := start_successors$
pc1 : $mcpc1 := select_state$

end

terminateBroken $\widehat{=}$
extends *terminateBroken*

any

s

where

grd1 : $s \in broken$

then

act1 : $counterexample := \{s\}$
act2 : $result := terminated_ce$

end

terminateOK $\widehat{=}$

extends *terminateOK*

when

`grd1` : $broken = \varnothing$

`grd2` : $unknown = \varnothing$

then

`act1` : $counterexample := \varnothing$

`act2` : $result := terminated_ok$

end

END

M4 Machine (Regular)

MACHINE m4_regular_mc
REFINES m3_INVARIANTS_cycle
SEES c4_mc_states
VARIABLES

$invs$

$checked$

$unchecked$

$mcpc1$

$mcpc2$

$succs$

$current$

$queue$

ok

$broken$

$unknown$

$result$

$counterexample$

EVENTS
Initialisation

extended

begin

`act1` : $ok, broken, unknown := \varnothing, \varnothing, STATES$

`act2` : $counterexample, result := \varnothing, running$

`initroot` : $current := root$

`initqueue` : $queue := \varnothing$

`pc1` : $mcpc1 := check_invariant$

`pc2` : $mcpc2 := start_successors$

`succs` : $succs := \varnothing$

`act3` : $invs := \{root\} \times \mathrm{INVARIANTS}$

`act4` : $unchecked := \{root\} \times \mathrm{INVARIANTS}$

`act5` : $checked := \varnothing$

end

select_state $\widehat{=}$
extends *select_state*

any

s

where

`grd1` : $s \in queue$

`grdpc` : $mcpc1 = select_state$

then

`act1` : $current := s$

`act2` : $queue := queue \setminus \{s\}$

`setpc` : $mcpc1 := check_invariant$

end

check_true_inv $\widehat{=}$

extends *check_true_inv*

any

i

where

`grd2` : $i \in unchecked[\{current\}]$

`grd3` : $current \mapsto i \in truth$

`grd4` : $mcpc1 = check_invariant$

then

`remove1` : $checked := checked \cup (\{current \mapsto i\})$

`remove2` : $unchecked := unchecked \setminus (\{current \mapsto i\})$

end

checkOK $\widehat{=}$

extends *checkOK*

when

`grd2` : $checked[\{current\}] = \mathrm{INVARIANTS}$

`grd3` : $mcpc1 = check_invariant$

`grd4` : $current \in unknown$

then

`act1` : $mcpc1 := compute_successors$

`act2` : $ok, unknown := ok \cup \{current\}, unknown \setminus \{current\}$

end

checkBroken $\widehat{=}$

extends *checkBroken*

any

i

where

`grd1` : $i \in unchecked[\{current\}]$

`grd3` : $current \mapsto i \notin truth$

`grd4` : $current \in unknown$

`grdpc` : $mcpc1 = check_invariant$

then

act1 : $mcpc1 := compute_successors$

act2 : $broken, unknown := broken \cup \{current\}, unknown \setminus \{current\}$

end

successors_start $\widehat{=}$

extends *successors_start*

any

$next$

where

pc : $mcpc1 = compute_successors$

pc2 : $mcpc2 = start_successors$

successors : $next = (transitions[\{current\}] \cap unknown)$

then

act : $succs := next$

actpc : $mcpc2 := step_successors$

end

successors_skip $\widehat{=}$

extends *successors_skip*

any

s

is

where

pc : $mcpc2 = step_successors$

grd1 : $s \in succs$

grd2 : $s \in queue \vee s = current$

grd3 : $\forall i \cdot i \in is \Rightarrow s \mapsto i \in truth$

grd4 : $(\{s\} \times \mathrm{INVARIANTS}) \subseteq invs$

grdis : $is = \varnothing$

then

act1 : $succs := succs \setminus \{s\}$

act2 : $unchecked := unchecked \setminus (\{s\} \times is)$

act3 : $checked := checked \cup (\{s\} \times is)$

end

successors_step $\widehat{=}$

extends *successors_step*

any

s

is

where

pc : $mcpc2 = step_successors$

grd1 : $s \in succs$

grd2 : $s \notin queue$

grd3 : $s \neq current$

grd4 : $\forall i \cdot i \in is \Rightarrow s \mapsto i \in truth$

grd5 : $s \notin dom(invs)$

grdis : $is = \varnothing$

then

act1 : $succs := succs \setminus \{s\}$

act2 : $queue := queue \cup \{s\}$

act3 : $checked := checked \cup (\{s\} \times is)$

act4 : $unchecked := unchecked \cup (\{s\} \times (\mathrm{INVARIANTS} \setminus is))$

act5 : $invs := invs \cup (\{s\} \times \mathrm{INVARIANTS})$

end

successors_stop $\widehat{=}$

extends *successors_stop*

when

grd0 : $mcpc2 = step_successors$

grd1 : $succs = \varnothing$

then

pc2 : $mcpc2 := start_successors$

pc1 : $mcpc1 := select_state$

end

terminateBroken $\widehat{=}$

extends *terminateBroken*

any

s

where

grd1 : $s \in broken$

then

act1 : $counterexample := \{s\}$

act2 : $result := terminated_ce$

end

terminateOK $\widehat{=}$

extends *terminateOK*

when

grd1 : $broken = \varnothing$

grd2 : $unknown = \varnothing$

then

act1 : $counterexample := \varnothing$

act2 : $result := terminated_ok$

end

END

M4 Machine (Proof Support)

MACHINE m4_use_proof
REFINES m3_INVARIANTS_cycle
SEES c6_discharge_info
VARIABLES

$invs$

$checked$

$unchecked$

$mcpc1$

$mcpc2$

$succs$

$current$

$queue$

ok

$broken$

$unknown$

$result$

$counterexample$

EVENTS
Initialisation

extended

begin

`act1` : $ok, broken, unknown := \varnothing, \varnothing, STATES$

`act2` : $counterexample, result := \varnothing, running$

`initroot` : $current := root$

`initqueue` : $queue := \varnothing$

`pc1` : $mcpc1 := check_invariant$

`pc2` : $mcpc2 := start_successors$

`succs` : $succs := \varnothing$

`act3` : $invs := \{root\} \times \mathrm{INVARIANTS}$

`act4` : $unchecked := \{root\} \times \mathrm{INVARIANTS}$

`act5` : $checked := \varnothing$

end

select_state $\widehat{=}$
extends *select_state*

any

s

where

`grd1` : $s \in queue$

`grdpc` : $mcpc1 = select_state$

then

`act1` : $current := s$

`act2` : $queue := queue \setminus \{s\}$

`setpc` : $mcpc1 := check_invariant$

end

check_true_inv $\widehat{=}$

extends *check_true_inv*

any

i

where

`grd2` : $i \in unchecked[\{current\}]$

`grd3` : $current \mapsto i \in truth$

`grd4` : $mcpc1 = check_invariant$

then

`remove1` : $checked := checked \cup (\{current \mapsto i\})$

`remove2` : $unchecked := unchecked \setminus (\{current \mapsto i\})$

end

checkOK $\widehat{=}$

extends *checkOK*

when

`grd2` : $checked[\{current\}] = \mathrm{INVARIANTS}$

`grd3` : $mcpc1 = check_invariant$

`grd4` : $current \in unknown$

then

`act1` : $mcpc1 := compute_successors$

`act2` : $ok, unknown := ok \cup \{current\}, unknown \setminus \{current\}$

end

checkBroken $\widehat{=}$

extends *checkBroken*

any

i

where

`grd1` : $i \in unchecked[\{current\}]$

`grd3` : $current \mapsto i \notin truth$

`grd4` : $current \in unknown$

`grdpc` : $mcpc1 = check_invariant$

then

`act1` : $mcpc1 := compute_successors$

`act2` : $broken, unknown := broken \cup \{current\}, unknown \setminus \{current\}$

end

successors_start $\widehat{=}$
extends *successors_start*

any

$next$

where

`pc` : $mcpc1 = compute_successors$

`pc2` : $mcpc2 = start_successors$

`successors` : $next = (transitions[\{current\}] \cap unknown)$

then

`act` : $succs := next$

`actpc` : $mcpc2 := step_successors$

end

successors_skip $\widehat{=}$
refines *successors_skip*

any

s

e

where

`pc2` : $mcpc2 = step_successors$

`grd1` : $s \in succs$

`grd2` : $s \in queue \vee s = current$

`grd4` : $(\{s\} \times \text{INVARIANTS}) \subseteq invs$

`grd5` : $(current \mapsto s) \mapsto e \in labels$

`grd6` : $current \in ok$

with

`is` : $discharged(e) = is$

then

`act1` : $succs := succs \setminus \{s\}$

`act2` : $unchecked := unchecked \setminus (\{s\} \times discharged(e))$

`act3` : $checked := checked \cup (\{s\} \times discharged(e))$

end

successors_step $\widehat{=}$
refines *successors_step*

any

s

e

where

`pc` : $mcpc2 = step_successors$

grd1 : $s \in succs$
grd2 : $s \notin queue$
grd3 : $s \neq current$
grd5 : $s \notin dom(invs)$
grd6 : $current \in ok$
grd7 : $(current \mapsto s) \mapsto e \in labels$

with

is : $discharged(e) = is$

then

act1 : $succs := succs \setminus \{s\}$
act2 : $queue := queue \cup \{s\}$
act3 : $checked := checked \cup (\{s\} \times discharged(e))$
act4 : $unchecked := unchecked \cup (\{s\} \times single_specialized_invariant(e))$
act5 : $invs := invs \cup (\{s\} \times \mathrm{INVARIANTS})$

end

successors_stop $\widehat{=}$
extends *successors_stop*

when

grd0 : $mcpc2 = step_successors$
grd1 : $succs = \varnothing$

then

pc2 : $mcpc2 := start_successors$
pc1 : $mcpc1 := select_state$

end

terminateBroken $\widehat{=}$
extends *terminateBroken*

any

s

where

grd1 : $s \in broken$

then

act1 : $counterexample := \{s\}$
act2 : $result := terminated_ce$

end

terminateOK $\widehat{=}$
extends *terminateOK*

when

grd1 : $broken = \varnothing$
grd2 : $unknown = \varnothing$

then

act1 : $counterexample := \varnothing$

`act2` : $result := terminated_ok$

end

END

Appendix B

Publications

As mentioned before, this thesis is based on two papers that have been published previously and one paper that is currently being prepared for submission. The content of all the papers is, of course, the result of fruitful discussions with other researchers. However, I am the main author of all three papers that form the foundation of this thesis.

The first paper "Proof Assisted Model Checking for B" [36] was co-authored by Michael Leuschel. He did most of the related work section, some of the figures, and improved the writing. Of course, Michael also verified my result and was an invaluable help when it came to the first prototype implementation. He also continued the further development of the proof support from the prototype into its present form. The paper was published at the International Conference of Formal Engineering Methods (ICFEM) in 2009.

The second paper "Automatic Flow Analysis for Event-B" [52] was also co-authored by Michael Leuschel. He wrote the first version of the simplifier and helped a lot with the writing of this article. He and Dobrikov also worked later on an approach using ProB's contraint solver to compute the enable graph instead of a prover. The paper was published at the International Conference on Fundamental Approaches to Software Engineering (FASE) in 2011.

The third paper "Scalable Distributed Model Checking for High-Level Formal Models" is currently in preparation for submission. This paper was co-authored by Philipp Körner and Michael Leuschel. Philipp Körner implemented most of the code under my supervision. Michael Leuschel helped me to shape the paper into the present form. In particular he helped to improve the paper's presentation a lot. He also wrote the related work section. A draft version of the paper is included in the appendix. We also submitted an extended abstract [64] to the Rodin User and Developer Workshop 2013.

Scalable Distributed Model Checking for High-Level Formal Models

Jens Bendisposto, Philipp Körner and Michael Leuschel

Institut für Informatik, Universität Düsseldorf**
Universitätsstr. 1, D-40225 Düsseldorf
{bendisposto,leuschel}@cs.uni-duesseldorf.de

Abstract Model checking is an important validation technology for formal models, where it is often challenging to keep validation time within reasonable bounds. Parallel and distributed model checking promises substantial improvements in that respect. However, actually delivering upon that promise is very challenging and there are not that many successful and easy to use parallel or distributed model checkers.
In this paper we present a scalable approach to parallel and also distributed model checking of high-level specifications in B, Event-B, Z and TLA$^+$. The work builds upon several worker instances of ProB connected by the ØMQ library and controlled by one master process. A queue management algorithm ensures even work distribution. We present the architecture of our system, along with extensive empirical evidence gathered on multi-core systems run locally and in the cloud. For several case studies we achieve almost perfect scaling. This is not only due to our architecture: we also benefit here from working on a very high-level formalism, where treating each individual state induces a high overhead.

1 Background

ProB [19] is a model checker and animator mainly aimed at the B/Event-B language, but which can also deal with Z and TLA$^+$ specifications. Some of the tasks ProB performs would clearly benefit form parallel execution, in particular model checking. Model checking here refers to the process of checking that an invariant holds for every reachable state, and that no deadlock occurs. For a typical case study, the processing time per state is in the order of 1 to 100 ms. For exceptionally simple models, such as a counter, the processing time for a single state is in the order of 100 μs, i.e., ProB can calculate at most 10,000 states per second. This is a rather low number compared with other explicit model checkers such as Spin, but in [18] we made a case for using ProB and high level languages such as B (sometimes a single high-level states can represent thousands or millions of low-level states). This also means that there is considerable potential for more sophisticated parallel and distributed model checking algorithms. We exploit this fact in this paper, and develop a scalable distributed model checking system for ProB, which ensures load balancing using a technique called "work stealing". As the experimental results will show, this approach does indeed achieve very good (and sometimes near optimal) scaling characteristics.

** This research is being carried out as part of the DFG funded research project GEPAVAS.

2 Implementation

The core of ProB was developed in SICStus Prolog. Unfortunately SICStus Prolog does not support parallel execution, so we could not implement parallel model checking within a single process; we need to run multiple Prolog process instances of ProB in order to achieve parallel or distributed model checking.

2.1 First Attempt

Initially, we experimented with the Linda library provided by SICStus. The library implements a tuple space [9], i.e., data is stored in a central repository and can be retrieved by worker processes. In the case of ProB, the state space as well as the states that haven't been explored yet are stored in the repository.

A worker retrieves a state, checks the invariant and sends all successor states to the repository. The big disadvantage of the library is that we always have to move states around; this is an expensive operation. The approach was beneficial for only very few models, because of a rather big performance overhead caused by the framework.

2.2 Architecture

The architecture we propose in this paper consists of two components: A single master that coordinates the computation and several instances of a worker that perform the actual model checking.

Both components are implemented in Prolog and C, to allow a very tight integration into ProB. Data is transferred very efficiently between Prolog and C using an undocumented feature of SICStus Prolog, namely fastread and fastwrite. To avoid duplicate checking of states, we store SHA-1 hash values of states that have been processed or are queued for processing. If we encounter two states with the same SHA-1 hash code, we assume that the two states are actually the same state. This can lead to unsoundness of the model checking if two different states produce the same hash value. However, the probability of hash collisions is very low. SHA-1 is a hash function that is used in cryptography[1]. It produces 160 Bit hash values, therefore the approximate collision probability for a billion elements is less than 2^{-100}. Currently the speed of computers and the performance of ProB makes checking more than a billion states practically impossible.

The communication between components is implemented using a library called ØMQ (pronounced Zero MQ) [10]. ØMQ is oriented around message queues that can be used to implement typical communication patterns, such as direct messaging or publish-subscribe.

We use the so called reactor pattern, a loop that queries a set of message queues in a round robin fashion and calls a specific function if a queue contains a value. The reactor is part of the ØMQ library. Without the reactor loop we probably would have used threads that concurrently modify the state of the component. For instance, one thread would receive information about the currently known state

[1] SHA-1 is specified in RFC 3174 `http://tools.ietf.org/html/rfc3174`

space, while another thread would check a state. Both threads would write to the same state, which is hard to implement correctly. The reactor pattern on the other hand allows us to decompose the concurrent algorithm into simple functions. Each of these functions has exclusive access to the state of the component because only one function is run at a time.

Figure 1 shows an overview of the architecture. Master and worker can exchange messages via the ZMQ library. Messages are stored in a messaging queue and asynchronously received and processed. The figure only shows a single message queue, but the design actually uses multiple queues. The reason is that we require different pattern for different message types. For example, information about the currently

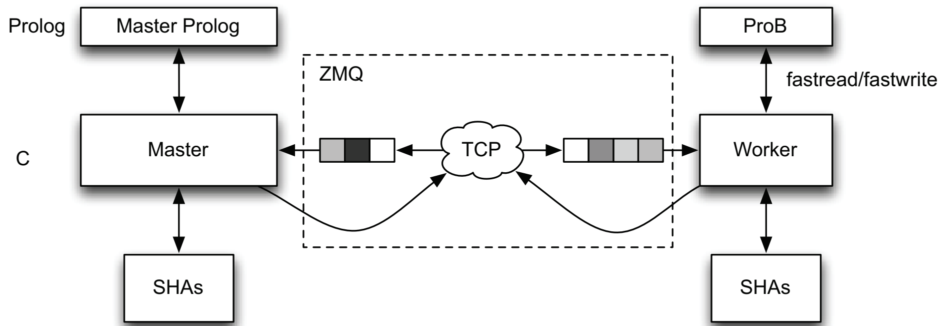


Figure1. Architecture of worker and master components

The master component has the responsibility of coordinating the work load, managing and broadcasting the information about known states of the model, collecting data about the progress, and terminating the workers upon completion of the task. Each time a worker processes a state, it sends the result (i.e., whether the invariant holds and whether the state contains a deadlock) together with some metadata to the master. The metadata includes the SHA-1 hashes of each successor state and some statistics.

The master maintains a data structure that contains all known hashes that have either been processed or are stored in some worker's queue. We will discuss the details of the data structure in section 2.3. For now we can think of it as a set containing tuples that consist of a SHA-1 hash and a boolean flag indicating whether the state has been processed. The master broadcasts changes to all workers, which will integrate them into their own local copy of that data structure.

The master can detect if a state has been checked independently by two workers. This can happen because the data structures are updated asynchronously, i.e., one worker may use an outdated data structure. This double checking of states cannot be completely avoided, but in most of our experiments the number is reasonably low.

The master also gets statistical information. The most important bit is the size of the worker's queues. If the master detects that worker W_1 has an empty queue while worker W_2 has a certain amount of elements in its queue, the master will initiate work stealing. It sends a message to W_2 who will transmit half of the elements in its queue to the master. The master forwards the package to W_1. We use a parameter *min_queuesize* to control the minimal amount of elements in the queue. The master will only ask a worker to send elements from its queue, if the queue size is greater than $2 \times min_queuesize$. This does not guarantee that the sending worker has at least *min_queuesize* after sending but it will prevent unnecessary transmissions. If we were close to the end of a model checking job, the system would start to "jitter" if we did not use a threshold.

One could argue that it might be more efficient to directly send the work packages from one worker to the other without the master. This is true, but it also comes at a price. We either have to establish a new connection between two workers, which takes time, or we have to establish a quadratic number of connections in advance. We could also setup a linear number of connections in advance if we restrict who can share work with whom, but in this case, it could impact the fair distribution of load. However, this is a design parameter that we want to explore in the future.

The master's reactor loop consists of four queues:

1. **Receive join request.** If a worker connects to the master, it sends a join request. The master answers request with a unique worker ID and the Prolog term that represents the B model. We provide an initial number of workers that need to connect before the master initiates the model checking process. We can dynamically add more workers, each worker that is connected after the process was started gets the information that it is late when it joins. We need the information to ensure that the new worker gets all hashes that have been processed.
2. **Receive hashes request.** If a worker joins late, it will ask the master to send the hash codes that have already been broadcast. The master answers this request with all hash codes that are known.
3. **Receive statistics.** After checking a state, the worker sends a message containing statistical information such as the time spent to check the state but more importantly the current queue size of the worker. Based on the queue size the master may initiate a work sharing request.
4. **Receive results.** Also after checking a state, the worker will send a message containing the result of the check (i.e., if the invariant was violated or a deadlock was found) and the hash codes of the checked state and all successors. The master extracts the hash codes and forwards them to all workers. For performance reasons, the master automatically combines information from multiple workers.

The worker components perform the actual computations. They take a state that has not yet been checked from their work queue, check if the machine's invariant is true for that state, and compute successor states. Like the master, a worker contains a set of states that are globally known. If the worker dequeues a state from

its work queue that has already been checked, the state is dropped. A worker also drops all successor states that are in the global set, because they already must be in the queue of some worker. This can lead to a loss of potentially useful information, namely that there is a transition from the currently checked state to that particular state. In future we consider keeping and using this information, e.g., to exploit proof information as shown in previous work [7]. We will discuss this issue in section 4.

The reactor loop of a worker consists of five queues:

1. **Receive hashes.** The master sends hash codes of known states to this queue. If a worker receives an item from this queue, it extracts all contained hash codes and adds them to its own set of hashes.
2. **Receive tasks.** Packages received from this queue contain states that the worker should check. The worker extracts all working tasks and enqueues them into its working queue. Currently this only happens if the queue of the worker is empty.
3. **Receive share request.** The master notifies the worker to share the half of its queue with another worker. Upon receiving a notification from this queue, the worker will split its working queue and send half of it.
4. **Receive command.** This queue contains commands from the master, e.g., a terminate command that is issued if the master notices that a user provided maximum number of states have been reached.
5. **Process state.** This queue contains a single token. If the worker receives the token it will dequeue a task from the working queue and check it. After completion, the worker will send itself a new token. The purpose of this queue is to properly interleave the checking process with the other tasks the worker performs.

2.3 Storing states

To store the hashes we use a Trie (also referred to as a prefix tree). Our implementation was inspired by Phil Bagwell's Hash Array Mapped Tries [3]. We use a trie that has a branching factor of 32, therefore each 5 bit chunk of the 160 bit input is represented by one level in the trie. We extend the trie structure on demand. As an example, take Figure 2. On the bottom we have chunks of linear memory in which we store the hash codes sequentially. This is very important because if workers join late we want to be able to efficiently send them the hash codes without traversing the trie. On the top of the left side we have a node in our trie structure. In the figure we use a branching factor of 4 instead of 32. The small squares are bits that allow us to tell pointers to linear memory from pointers to trie nodes. In the figure we see how the trie is changed if we add a hash that is supposed to be stored in a slot that is already occupied. We create a new internal node, we move the pointer to the old content into the new node using the next 5 Bits of the hash code as the index (here 1). We add the new content, point from the original trie node to the new node and flip the bit to represent a pointer to an internal node.

We use SHA-1 as the hashing function for a state, i.e., each state produces a 160 Bit hash code, thus the maximal depth of the trie is $log_{32}(2^{160}) = 32$. But because

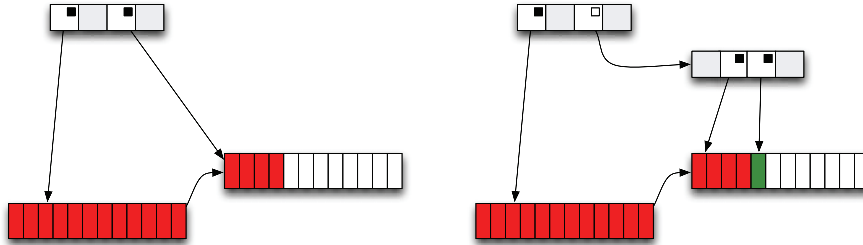


Figure2. Adding a new layer to the hashtrie

SHA-1 is actually a cryptographic hash function, the hashes can be expected to be practically randomly distributed, i.e, we can expect that the trie is almost balanced. Because we use a high branching factor the trie is supposed to be shallow. We measured the depth of the tree by putting the SHA-1 values of the numbers from 0 to 9999999 into the trie. We found that for these ten million elements the depth was 8.

The complexity of looking up or adding hash codes is $O(log\ n)$. However we can expect that the maximal number of operations for any (currently) reasonable input size is about 10.

The memory footprint is a trade-off between a number of requirements. We want to keep the footprint as small as possible. However, for a single state we need 161 bits. 160 bits are for the hash code and one bit is to indicate if a state has been checked. In fact we could even save more memory by dropping the prefix, because it is in principle already encoded in the path throughout the trie. However, this complicates the implementation and the performance if a worker is late. A late worker requires all hash codes that have been broadcasted so far. If we save memory by dropping the prefixes, we need to recreate all hashes and traverse the trie. If the hashes are complete and stored sequentially we can use `memcpy` to put the hashes into a ØMQ message.

We also decided to use 21 byte instead of 20 byte and a single bit for each hash code and thus “waste” 7 bits. There are clearly other solutions but we decided to use 21 bytes for the sake of a simpler implementation.

2.4 Control UI

Starting a distributed model checking using ProB directly is a bit cumbersome. As a first simple solution for a single multicore computer we wrote a shell script to start a number of workers and a master. We also developed a small web application that allows us to run the benchmark experiments more comfortably. We can produce an experiment descriptor containing parameters for the benchmark experiment such as the number of workers, the maximal number of states we want to check, the number of repetitions or ProB specific settings. We put a number of experiment descriptors together with the corresponding B models into a zip file and upload this file via the web interface. The application runs all experiments and produces result

tables containing the most important information, such as the runtime, number of states, or invariant violations. We are currently developing a more sophisticated web application that will allow us to coordinate computations on the Amazon Cloud, e.g., we want to be able to add or remove computers and get more detailed information about the load on each computer. For instance, if we exhaust the memory on a single computer we could automatically add a second computer from the cloud and move some workers to the second computer.

3 Empirical Evaluation

For our experiments we used two types of machines:

- A six core 3.33 GHz Mac Pro with 16 GB of RAM
- Multiple c3.8xlarge instances in the Amazon Cloud. A c3.8xlarge instance has 32 virtual CPUs and is equipped with 64 GB of RAM. The documentation states that "Each virtual CPU (vCPU) on C3 instances is a hardware hyper-thread from a 2.8 GHz Intel Xeon E5-2680v2 (Ivy Bridge) processor [...]" [2]. According to the Intel website, the Xeon E5-2680v2 is a 10 core processor with 20 threads.

We used the Mac Pro to get an impression if and how good the B models scale. From the experiments we chose those models that seem to scale well and benchmarked them on the Amazon EC computer with a higher number of workers.

We also did few experiments using two or four c3.8xlarge instances which were connected via a 10 GBit Ethernet connection. From the tool's point of view there is no difference whether the workers are located on one computer or on multiple machines. Each worker gets the IP Address of a master and then connects to that master, running all workers and the master on the same host is just a special case of distributed model checking.

The sample size for experiments on the Mac Pro was 3, for the experiments on the Amazon Cloud we only did a single run. However, in previous experiments we observed that the results do not differ by much.

For all experiments we used ProB 1.4.0-rc1. On the Mac Pro we used revision `d1829ce0fb6e968f35ffa39c60712e2d5b674a9a`, which was built on April 14th, 2014. On the Amazon Cloud we used a slightly newer revision, namely `735295934f31114e1e6a7362d458a88e744282a9` which was built on April 16th, 2014.

This paper includes examples where our approach works quite well and models where our approach does not. We will analyze why the tool does not work for some of the models, and give some guidelines on when not to use the parallel model checker. In particular, we will describe a very simple experimental approach to determine a good number of workers.

All experiments use breadth first search. This is not a restriction of the tool, one can freely chose BFS, DFS or ProB's mixed mode. However, using BFS is a bit nicer for analysis. In the case of incomplete checking (e.g., if the state space is infinite) breadth first search eliminates one source of non-determinism, i.e., the choice of the successor state.

The results of our measurement are summarized in Tables 1 and 2. Due to resource constraints, we did not run the model checker using a single worker on the

Amazon cloud. The results for one worker are approximated using either the 4 or the 8 worker result. This means we assume that below 4 or 8 the model scales linear with a slope of 1. This can only lead to an under-approximation of the runtime for one worker and therefore to an under-approximation of the speedup factor, i.e., our estimation of the speedup factor is safe.

Summary In summary we can see that the experiments were very successful, and that our distributed model checking algorithm is clearly worthwhile. We achieve considerable speedups for all real-life benchmarks. E.g., we reduce the runtime of the interlocking model from around 119 minutes down to around 26 minutes by using 5 workers on the 6-core MacPro. On the Amazon cloud, we can further reduce the runtime to under 7 minutes, by using an instance with 32 virtual CPUs. Below we analyse the experimental results in more detail. In Subection 3.3 we analyse the (few) small benchmarks with limited scaling, while in Subsection 3.4 we examine the other benchmarks with good scaling. Before that, we study the impact of Hyperthreading on the achievable scaling in Subsection 3.1 and the overhead of our framework in Subsection 3.2.

		Workers (#)			
Model	States (#)	1	2	4	5
Cruise control system	1361	1.00 (4870 ms)	1.74	2.05	1.93
Counter	100000[a]	1.00 (18919 ms)	0.99	0.91	0.87
Hanoi Towers	6563	1.00 (26091 ms)	1.89	2.86	2.92
Stuttgart 21	10000[a]	1.00 (546093 ms)	2.00	3.81	4.62
Interlocking	672175	1.00 (7120411 ms)	1.98	3.80	4.55
USB Bus-4	16858	1.00 (61596 ms)	2.06	3.84	4.41
CAN Bus protocol	132599	1.00 (137717 ms)	1.92	3.1	3.61
RETHER	42254	1.00 (83508 ms)	2.05	3.79	4.32
Scheduler	24581	1.00 (162982 ms)	1.96	3.81	4.64
Mode Protocol	810948	1.00 (4877777 ms)	2.00	3.92	4.77
Set Game	2000[a]	1.00 (47964 ms)	1.93	3.55	4.24

[a] State space not fully explored

Table1. Benchmarks on Mac Pro

3.1 Hardware influences: Hyper-Threading

It turns out that ProB cannot scale perfectly if it does not run on a real core. Perfect scaling means that if you multiply the number of workers by k you get $1/k$ of the runtime. We can see this in Figure 3. We explored the full state space of a mode management protocol which we will describe in section 3.4. The state space consists of 810948 states.

Our benchmark computer provides six real cores and twelve hyper-threads. If we fit a linear model on the segments we get about $speedup = 0.87 \times cores + 0.28$ for 1 to 6 cores with a correlation of 0.9938. We get $speedup = 0.95 \times cores + 0.09$

		Workers (#)				
Model	States (#)	1	4	8	16	31
Stuttgart 21	10000[a]	1.00 (593808 ms)[b]	4.00	7.75	13.95	18.92
Interlocking	672175	1.00 (8094144 ms)[c]	-	8.00	13.72	20.14
USB Bus-10	211042	1.00 (741984 ms)[b]	4.00	7.10	12.24	18.68
RETHER	42254	1.00 (88240 ms) [b]	4.00	6.40	9.60	11.53
Scheduler	24581	1.00 (176288 ms)[b]	4.00	7.24	11.29	16.27
Mode Protocol	810948	1.00 (5562928 ms)[c]	-	8.00	13.52	19.24
Set Game	10000	1.00 (355312 ms)[c]	-	8.00	12.19	16.79

		Workers (#)				
Model	States (#)	1	8	16	32	63
Stuttgart 21	10000[a]	1.00 (599328 ms)[c]	8.00	14.99	26.71	33.28
Interlocking	672175	1.00 (7904000 ms)[c]	8.00	14.57	24.23	38.63
Mode Protocol	810948	1.00 (5474200 ms)[c]	8.00	14.29	19.62	33.35

[a] State space not fully explored
[b] Approximation using 4 worker experiment
[c] Approximation using 8 worker experiment

Table2. Benchmarks on Amazon EC2

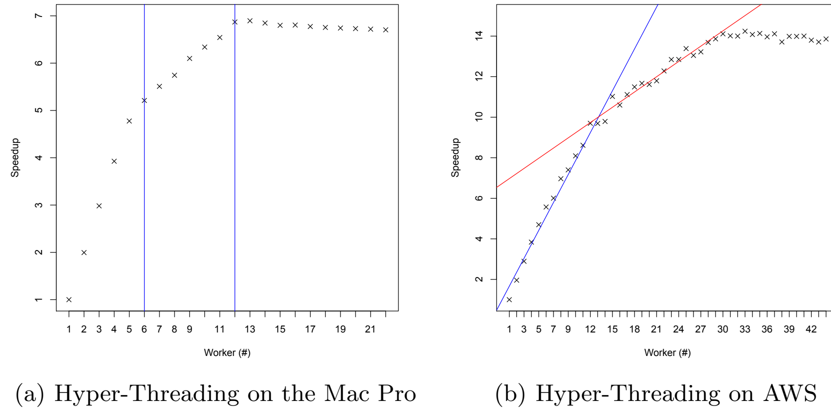
for 1 to 5 cores with a correlation of 0.9995. It does not scale up to exactly 6 because we also run the master process that uses a bit of the CPU. Between 6 and 12 workers it scales linearly but the slope is much lower. Fitting a linear model yields $speedup = 0.27 \times cores + 3.59$ between 6 and 12 cores with a correlation of 0.9984. For more than 13 worker we actually lose performance if we add more workers. Another evidence in favor of the hypothesis is that if we look at the time that is spent within Prolog, we get 5.6 ms with a single worker and 10.1 ms with 12 workers (on the Mac Pro). The standard deviation is 1.0 ms in the case of a single core and 1.5 ms in the case of 12 cores.

If we use the same model on an Amazon instance, we get a similar curve. We did not used the full state space but 100000 states. The correlation is not as high as on the Mac Pro computer but we still see the same pattern. If we fit a linear model, we get $speedup = 0.79 \times cores + 0.52$ for 1 to 10 cores with a correlation of 0.9955. For 11 to 32 we get $speedup = 0.27 \times cores + 6.40$ with a correlation of 0.9784.

3.2 Overhead of the framework.

In order to measure the overhead introduced by the parallelization framework we did some experiments with models that are not supposed to scale well.

The model shown in Figure 4(a) does not scale well. We get a speedup of about 2 at most. However, the situation is not as bad as it seems if we look at Table 1. The total model checking of the cruise control system is only about 5 seconds for 1361 states. This is clearly not a very typical case where we would use parallel or distributed model checking. We would even accept that the runtime is worse for these kind of models. The experiment shows that the overhead introduced by our framework is reasonably low.

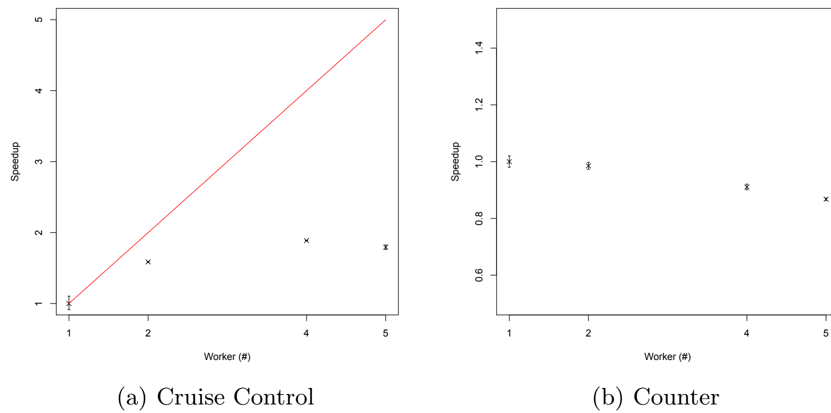


(a) Hyper-Threading on the Mac Pro (b) Hyper-Threading on AWS

Figure3. Effects of the CPU / Hyper-Threads

Figure 4(b) shows a quite artificial model, representing a simple counter with a single operation for incrementation. From the perspective of parallel model checking this is the worst case because the working queue cannot contain more than a single task. The experiment shows that we do lose performance, i.e., finding a good number of workers is important.

We will discuss a method how we can estimate a good number in section 4. We hope that we can reduce the performance loss in the future because in theory a work-stealing approach can deal well with workers that are idle.

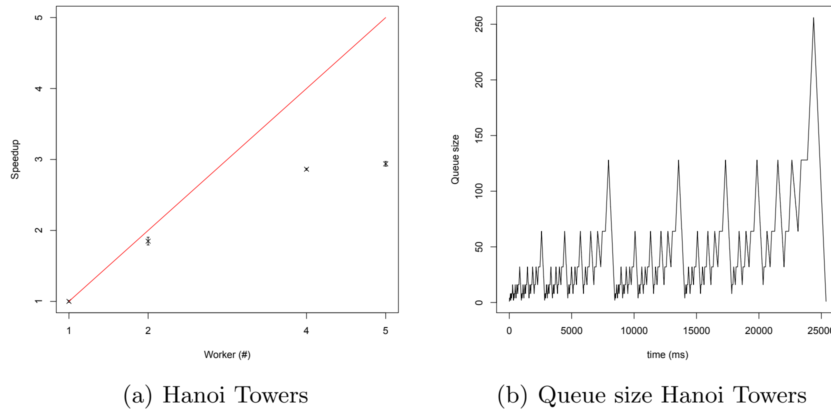


(a) Cruise Control (b) Counter

Figure4. Impact of the parallelization framework

3.3 Benchmarks with limited scaling.

The speedup we get from parallel execution of multiple ProB processes is mainly determined by the degree of branching in the B model. Figure 5(a) shows the speedup for a model of the Hanoi towers[2]. Figure 5(b) shows the development of the queue size over time, exhibiting a repeating pattern. This pattern mirrors the recursive nature of the problem. From time to time, we reach a situation where the number of choices is very limited, e.g., if all disks are on one peg, there are only two possible moves. This means that at these points all the queues of all the workers are almost empty. This leads to the sub-linear scaling of the model.



(a) Hanoi Towers (b) Queue size Hanoi Towers

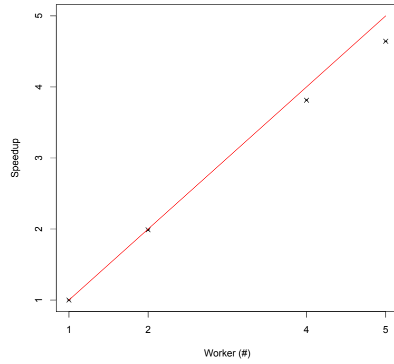
Figure5. Limited scaling

3.4 Benchmarks with (almost) linear scaling

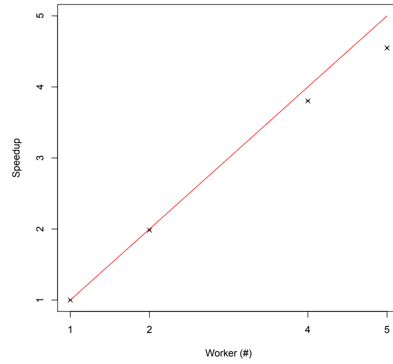
We mainly used case studies that are close to industrial use cases, such as

- **Stuttgart 21.** The Stuttgart 21 Interlocking system (see Figure 6(a) was modeled by H. Wiegard. It is an interlocking system for the railway station in Stuttgart, which is currently being built. It was developed with capacity simulation in mind, i.e., it should be able to answer questions like “How many trains can the station handle safely?”. We limited the experiment to 10000 states. Checking a single state takes about 54 ms, thus on a single core the overall model checking time is 9 minutes. On the Mac Pro we get a speedup factor of 4.62 using 5 cores. On a single Amazon EC2 instance we get a factor of 18.9. Using two EC2 instances we get a factor of 33.3 using 63 workers. Finally on 4 instances we get a factor of 68.8 using 127 workers. For the experiment on 127 cores we used 80000 instead of 10000 states and we checked them in 68 seconds, a single worker would have required about 1 hour and 18 minutes.

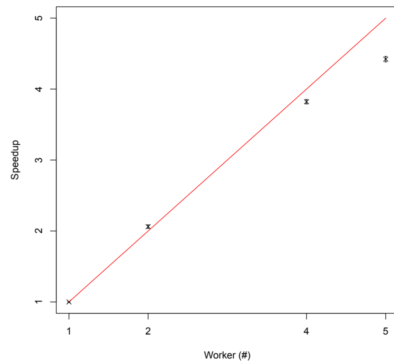
[2] `http://en.wikipedia.org/wiki/Tower_of_Hanoi`



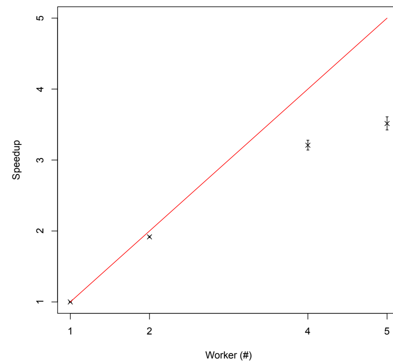
(a) Stuttgart 21 Interlocking



(b) Interlocking System



(c) USB Bus (4 Endpoints)



(d) CAN Bus

Figure6. (Almost) Linear scaling (Mac Pro)

- **Interlocking system.** The other interlocking system (Figure 6(b)) is a variation of the model from Abrial's book [1]. The model has a reduced state space, which was achieved by manually applying a partial order reduction. The reduction was not applied for the parallel experiment, we have chosen this variation because the absolute model checking time of the original model (about 8 days) was too high for this experiment. The speedup factor on the Mac Pro is 4.55 using 5 workers. On a single Amazon EC2 we get a factor of 20.1 using 31 workers and using 63 workers on two instances we get a factor of 38.6.
- **Bus Systems.** We also used two models of bus system, A USB bus system and a Controller Area Network (CAN) Bus. CAN Busses are typically used in cars. They allow devices and controllers to communicate over a shared bus without the need of a special bus controller. Both models were developed by J. Colley. As we can see in Figures 6(c) and 6(d), both models scale reasonably well. The

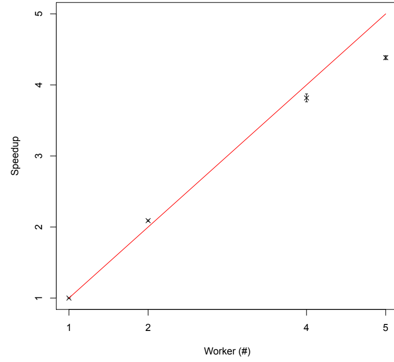
CAN Bus model probably doesn't scale much further than 4 cores, but at least we can get a speedup factor of 3.6. The USB Bus model scales better, using 5 cores we get a speedup factor of 4.41. If we compare the runtime for 1 and 2 workers for the USB Bus we note something peculiar. The speedup factor is greater than 2 which is counter-intuitive. We think that neither ProB nor our tool is responsible for this behavior. It may be the CPU Turbo-Boost or CPU caching effects because the effect vanishes if we produce slightly more CPU load. Note that the CPU is only checking using two workers and one master, so it has at least 3 spare cores. On the Amazon EC2 instance we get a speedup factor of 18.7 for 31 workers.

- **Network Protocol.** We used a protocol of a real time Ethernet protocol (RETHER) [21]. The B model by M. Büngener is a translation of a model written for the DiVinE model checker [4]. On the Mac Pro we get a speedup factor of 4.32 using 5 workers. On an Amazon instance the RETHER model scales very well up to about 8 cores yielding a speedup of a factor 6. Using 31 worker, we get a overall speedup factor of about 11,5.
- **Scheduler.** We used a model of a kernel scheduler by J.-P.Bodeveix et al. [8]. On the Mac Pro we get an improvement by a factor of 4.64 using 5 cores. On the EC2 instance the scheduler also scales very well up to about 8 cores, yielding a speedup factor of about 7.24. Using 31 workers, we get a total speedup factor of about 16.25.
- **Mode Management Protocol.** The model was developed by Space Systems Finland as part of a Distributed System for Attitude and Orbit Control for a Single Spacecraft (DSAOCSS) System [14],[16],[15]. The model was a part of the case study within the EU Project DEPLOY. On the Mac Pro we get a speedup factor of 4.77 using 5 workers. On a single EC2 instance we get a factor of 19.2 using 31 workers and on 2 instances we get a factor of 33.3 using 63 workers. This means we reduce the model checking time from about 1 hour 30 minutes to less than 3 minutes.
- **SET Game.** We used a model of a game called Set[3]. The model has the interesting feature that its state space is actually a tree, i.e., for any state all the successor states are new. Also, checking the invariant becomes much more expensive the larger the depth of the state in the computation tree is. On the Mac Pro we get a speedup factor of 4.24 using 5 workers. On an Amazon EC2 instance we improve by a factor of 16.7.

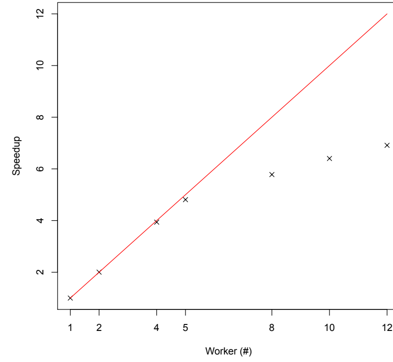
4 Future Work

Finding the right number of workers As we have seen in Sections 3.2 and 3.1 having too many workers can have a negative effect on the performance, it also blocks valuable resources. However, it is very simple to find a reasonable amount of workers using a scan technique. We start the model checker with a single worker for a fixed amount of time and measure how many states are checked. We keep doubling the number of workers and repeating the procedure until we do not get a good

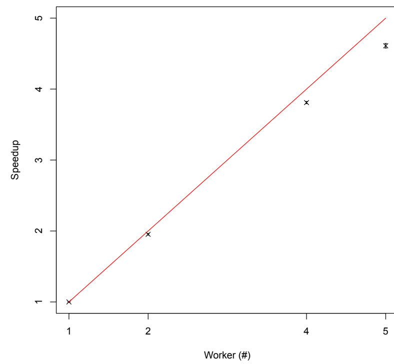
[3] `http://en.wikipedia.org/wiki/Set_(game)`



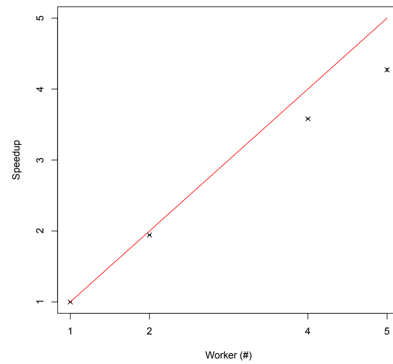
(a) Real time Ethernet Protocol



(b) Mode Protocol



(c) Process Scheduler



(d) Set Game

Figure7. (Almost) Linear scaling (Mac Pro)

improvement anymore. The improvement could be judged by a simple heuristic. For instance, if we double from k workers to $2 \times k$ workers we require that we check at least $0.25 \times k$ more states. The number 0.25 stems from the experiments described in Section 3.1. Once we have an interval, we can narrow the search down to a reasonable number of workers, e.g. using a binary search.

There are several other points that could be addressed in the future:

– **Use Proof Information.** During distributed model checking, we currently do not use information about discharged proof information as described in previous work [7]. As a first step we could at use the information about the transition the model checker took when encountering a state. This can be done locally by each worker, and can in some cases lead to a significant reduction in model checking time. However, it would be interesting to try and track all incoming

transitions, allowing us to check [7] only the invariants that have no proof for all incoming events.
Also, we do support invariant, assertion and deadlock checking, but not general LTL model checking. The latter still needs to be run by a single instance of ProB.
– **Reduce Memory footprint.** We are currently working on an implementation that reduces the footprint of the hash trie by sharing it among all workers that are located on the same computer. This will obviously reduce the global memory footprint considerably.

5 Related Work and Conclusion

There is a substantial body of research towards achieving parallel and distributed model checking, ranging from early work for Murϕ [20], to work on parallel symbolic model checking [6], and developments for the Spin [11] model checker [13,12]. Many works focus on safety properties (just like we do). A notable exception is for example [5] and more recently the DiVinE model checker [4]. The work [17] looks at parallelization in the context of directed model checking using A*.

As far as high-level formalisms is concerned, we want to mention the model checker TLC [22] for the high-level language TLA$^+$, which can be run in parallel. There also exists research on a distributed version of TLC, presented by Kuppe at the FM'12 TLA workshop.[4]

We have introduced a distributed model checker built on top of ProB that works very well for industrial case studies provided that the model's state space contains reasonable branching. We use work stealing to automatically distribute work among various instances of ProB. We have empirically demonstrated that the performance of the implementation is good enough to allow scaling even across multiple computers.

References

1. J.-R. Abrial. *Modeling in Event-B: System and Software Engineering.* Cambridge University Press, 2010.
2. Amazon Web Services, Inc. Amazon EC2 Instances. `http://aws.amazon.com/ec2/instance-types/`. Accessed: 2014-04-01.
3. P. Bagwell. Ideal hash trees. *Es Grands Champs*, 1195, 2001.
4. J. Barnat, L. Brim, and P. Ročkai. Scalable shared memory ltl model checking. *International Journal on Software Tools for Technology Transfer*, 12(2):139–153, 2010.
5. J. Barnat, L. Brim, and J. Stríbrná. Distributed LTL model-checking in spin. In M. B. Dwyer, editor, *SPIN*, LNCS 2057, pages 200–216. Springer-Verlag, 2001.
6. S. Ben-David, O. Grumberg, T. Heyman, and A. Schuster. Scalable distributed on-the-fly symbolic model checking. *International Journal on Software Tools for Technology Transfer (STTT)*, 4(4):496–504, 2003.
7. J. Bendisposto and M. Leuschel. Proof assisted model checking for B. In K. Breitman and A. Cavalcanti, editors, *Proceedings of ICFEM 2009*, volume 5885 of *Lecture Notes in Computer Science*, pages 504–520. Springer, 2009.

[4] See `http://tla2012.loria.fr/program.html`, "Current State of Distributed TLC".

8. J.-P. Bodeveix, M. Filali, J. Lawall, and G. Muller. Formal methods meet domain specific languages. In *Proceedings of the 5th international conference on Integrated Formal Methods*, IFM'05, pages 187–206, Berlin, Heidelberg, 2005. Springer-Verlag.
9. D. Gelernter. Generative communication in linda. *ACM Trans. Program. Lang. Syst.*, 7(1):80–112, Jan. 1985.
10. P. Hintjens. *ZeroMQ: Messaging for Many Applications*. O'Reilly, 2013.
11. G. J. Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, 2004.
12. G. J. Holzmann. A stack-slicing algorithm for multi-core model checking. *Electronic Notes in Theoretical Computer Science*, 198:3–16, 2008.
13. G. J. Holzmann and D. Bosnacki. The design of a multi-core extension of the Spin model checker. *IEEE Trans. Software Eng.*, 33(10):659–674, 2007.
14. D. Ilic, L. Laibinis, T. Latvala, E. Troubitsyna, and K. Varpaaniemi. Deployment in the Space Sector. In *Industrial Deployment of System Engineering Methods*, pages 45–62. Springer, 2013.
15. D. Ilic, T. Latvala, L. Nummila, T. Räsänen, P. Väisänen, K. Varpaaniemi, L. Laibinis, Y. Prokhorova, E. Troubitsyna, A. Iliasov, A. Romanovsky, M. Butler, A. S. Fathabadi, A. Rezazadeh, J.-C. Deprez, R. D. Landtsheer, and C. Ponsard. DEPLOY Deliverable D39 D3.2 – Report on Enhanced Deployment in the Space Sector. Technical report, 2011. Available at `http://www.deploy-project.eu/html/deliverables.html`.
16. D. Ilic, T. Latvala, P. Väisänen, K. Varpaaniemi, L. Laibinis, and E. Troubitsyna. DEPLOY Deliverable D20 D3.1 – Report on Pilot Deployment in the Space Sector. Technical report, 2010. Available at `http://www.deploy-project.eu/html/deliverables.html`.
17. S. Jabbar and S. Edelkamp. Parallel external directed model checking with linear i/o. In E. A. Emerson and K. S. Namjoshi, editors, *VMCAI*, LNCS 3855, pages 237–251. Springer-Verlag, 2006.
18. M. Leuschel. The high road to formal validation:. In E. Börger, M. J. Butler, J. P. Bowen, and P. Boca, editors, *ABZ*, volume 5238 of *Lecture Notes in Computer Science*, pages 4–23. Springer, 2008.
19. M. Leuschel and M. J. Butler. ProB: an automated analysis toolset for the B method. *STTT*, 10(2):185–203, 2008.
20. U. Stern and D. L. Dill. Parallelizing the murϕ verifier. *Formal Methods in System Design*, 18(2):117–129, 2001.
21. C. Venkatramani and T. Chiueh. Design, implementation, and evaluation of a software-based real-time ethernet protocol. *SIGCOMM Comput. Commun. Rev.*, 25(4):27–37, Oct. 1995.
22. Y. Yu, P. Manolios, and L. Lamport. Model checking TLA$^+$ specifications. In L. Pierre and T. Kropf, editors, *Proceedings CHARME'99*, LNCS 1703, pages 54–66. Springer-Verlag, 1999.

6 Appendix: Absolute Runtimes

		Workers (#)			
Model	States (#)	1	2	4	5
Cruise control system	1361	4870 ms	2797 ms	2375 ms	2523 ms
Counter	100000[a]	18919 ms	19203 ms	20774 ms	21805 ms
Hanoi Towers	6563	26091 ms	13769 ms	9129 ms	8942 ms
Stuttgart 21	10000[a]	546093 ms	272401 ms	143364 ms	118223 ms
Interlocking	672175	7120411 ms	3599504 ms	1873272 ms	1563760 ms
USB Bus-4	16858	61596 ms	29855 ms	16021 ms	13955 ms
CAN Bus protocol	132599	137717 ms	71681 ms	44488 ms	38123 ms
RETHER	42254	83508 ms	40754 ms	22017 ms	19330 ms
Scheduler	24581	162982 ms	83037 ms	42827 ms	35114 ms
Mode Protocol	810948	4877777 ms	2436374 ms	1245027 ms	1023481 ms
Set Game	2000[a]	47964 ms	24800 ms	13496 ms	11312 ms

[a] State space not fully explored

		Workers (#)				
Model	States (#)	1	4	8	16	31
Stuttgart 21	10000[a]	593808 ms [b]	148452 ms	76595 ms	42582 ms	31377 ms
Interlocking	672175	8094144 ms[c]	-	1011768 ms	589878 ms	401803 ms
USB Bus-10	211042	741984 ms[b]	185496 ms	104515 ms	60614 ms	39719 ms
RETHER	42254	88240 ms[b]	22060 ms	13796 ms	9189 ms	7653 ms
Scheduler	24581	176288 ms[b]	44072 ms	24352 ms	15611 ms	10851 ms
Mode Protocol	810948	5562928 ms[c]	-	695366 ms	411416 ms	289202 ms
Set Game	10000	355312 ms[c]	-	44414 ms	29137 ms	21168 ms

		Workers (#)				
Model	States (#)	1	8	16	32	63
Stuttgart 21	10000[a]	599328 ms[c]	74916 ms	39973 ms	22437 ms	18009 ms
Interlocking	672175	7904000 ms[c]	988000 ms	542479 ms	326262 ms	204586 ms
Mode Protocol	810948	5474200 ms[c]	684275 ms	392991 ms	279032 ms	164165 ms

[a] State space not fully explored
[b] Approximation using 4 worker experiment
[c] Approximation using 8 worker experiment

Bibliography

[1] Nancy G Leveson and Clark S Turner. An investigation of the Therac-25 accidents. *Computer*, 26(7):18–41, 1993.

[2] Marc Andreessen. Why Software Is Eating The World'. *Wall Street Journal*, 20, 2011.

[3] Chris Newcombe. Why Amazon Chose TLA+. In *Abstract State Machines, Alloy, B, TLA, VDM, and Z*, pages 25–39. Springer, 2014.

[4] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. Use of Formal Methods at Amazon Web Services. 2013.

[5] Thomas Ball, Ella Bounimova, Byron Cook, Vladimir Levin, Jakob Lichtenberg, Con McGarvey, Bohus Ondrusek, Sriram K. Rajamani, and Abdullah Ustuner. Thorough Static Analysis of Device Drivers. *SIGOPS Oper. Syst. Rev.*, 40(4):73–85, April 2006.

[6] Sebastian Wieczorek, Vitaly Kozyura, Andreas Roth, Michael Leuschel, Jens Bendisposto, Daniel Plagge, and Ina Schieferdecker. Applying Model Checking to Generate Model-based Integration Tests from Choreography Models. In *Proceedings TESTCOM/FATES 2009*, volume 5826 of *Lecture Notes in Computer Science*, pages 179–194. Springer-Verlag, 2009.

[7] J.-R. Abrial. *The B-Book: Assigning Programs to Meanings.* Cambridge University Press, New York, NY, USA, 1996.

[8] Steve Schneider. *The B-method: An introduction.* Palgrave Oxford, 2001.

[9] ClearSy. *Atelier B, User and Reference Manuals.* Aix-en-Provence, France, 2009. Available at `http://www.atelierb.eu/`.

[10] ClearSy, Aix-en-Provence, France. *B4Free: Tool and Manuals*, 2006. Available at `http://www.b4free.com`.

[11] Jean-Raymond Abrial. *Modeling in Event-B: System and Software Engineering.* Cambridge University Press, New York, NY, USA, 1st edition, 2010.

[12] Alexander Romanovsky. Rigorous Open Development Environment for Complex Systems - RODIN. *ERCIM News*, 65:40–41, 2006.

[13] Renato Silva, Carine Pascal, Thai Son Hoang, and Michael Butler. Decomposition Tool for Event-B. *Software: Practice and Experience*, 41(2):199–208, 2011.

[14] Michael Butler and Issam Maamria. Practical Theory Extension in Event-B. In *Theories of Programming and Formal Methods*, pages 67–81. Springer, 2013.

[15] Thomas Bouton, Diego Caminha B. de Oliveira, David Déharbe, and Pascal Fontaine. veriT: An Open, Trustable and Efficient SMT-Solver. In Renate A. Schmidt, editor, *Proc. Conference on Automated Deduction (CADE)*, Lecture Notes in Computer Science, pages 151–156. Springer-Verlag, 2009.

[16] Clark Barrett and Cesare Tinelli. CVC3. In Werner Damm and Holger Hermanns, editors, *Proceedings of the* 19th *International Conference on Computer Aided Verification (CAV '07)*, volume 4590 of *Lecture Notes in Computer Science*, pages 298–302. Springer-Verlag, July 2007. Berlin, Germany.

[17] David Déharbe, Pascal Fontaine, Yoann Guyot, and Laurent Voisin. SMT Solvers for Rodin. In *Proceedings of the Third International Conference on Abstract State Machines, Alloy, B, VDM, and Z*, ABZ'12, pages 194–207, Berlin, Heidelberg, 2012. Springer-Verlag.

[18] Sebastian Krings, Jens Bendisposto, and Michael Leuschel. Turning Failure into Proof: Evaluating the ProB Disprover. In *Proceedings of the 1st International Workshop about Sets and Tools*, 2014.

[19] Matthias Schmalz. Term Rewriting in Logics of Partial Functions. In Shengchao Qin and Zongyan Qiu, editors, *Formal Methods and Software Engineering*, volume 6991 of *Lecture Notes in Computer Science*, pages 633–650. Springer Berlin Heidelberg, 2011.

[20] Michael Jastram. The ProR Approach: Traceability of Requirements and System Descriptions. 2012.

[21] Colin Snook and Michael Butler. UML-B: Formal modeling and design aided by UML. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 15(1):92–122, 2006.

[22] Colin Snook, Abdolbaghi Rezazadeh, Kriangsak Damchoom, Michael Leuschel, Jens Bendisposto, Olivier Ligot, Apostolos Niaouris, Thierry Lecomte, and Ian Oliver. RODIN Deliverable D31 - Public Versions of Plug-in Tools. Technical report, 2007. Available at `http://rodin.cs.ncl.ac.uk/deliverables.htm`.

[23] Jens Bendisposto, Fabian Fritz, Michael Jastram, Michael Leuschel, and Ingo Weigelt. Developing Camille, a text editor for Rodin. *Software: Practice and Experience*, 41(2):189–198, 2011.

[24] Christophe Métayer. *AnimB 0.1.1*, 2010. Available at `http://wiki.event-b.org/index.php/AnimB`.

[25] Thierry Servat. BRAMA: A New Graphic Animation Tool for B models. In Jacques Julliand and Olga Kouchnarenko, editors, *Proceedings B'2007*, LNCS 4355, pages 274–276. Springer-Verlag, 2007.

[26] Nicolas Beauger, Jens Bendisposto, Michael Butler, Andreas Fürst, Alexei Iliasov, Michael Jastram, Issam Maamria, Daniel Plagge, Renato Silva, and Laurent Voisin. DEPLOY Deliverable D23 D9.2 Model Construction Tools and Analysis Tools II. Technical report, 2010. Available at `http://rodin.cs.ncl.ac.uk/deliverables.htm`.

[27] Michael Leuschel and Michael Butler. ProB: A Model Checker for B. In Keijiro Araki, Stefania Gnesi, and Dino Mandrioli, editors, *FME 2003: Formal Methods*, LNCS 2805, pages 855–874. Springer-Verlag, 2003.

[28] Michael Leuschel and Michael Butler. ProB: An Automated Analysis Toolset for the B Method. *Journal Software Tools for Technology Transfer*, pages –, 2008.

[29] Daniel Plagge and Michael Leuschel. Seven at one stroke: LTL model checking for high-level specifications in B, Z, CSP, and more. *Software Tools for Technology Transfer (STTT)*, 12(1):9–21, Feb 2010.

[30] Stefan Hallerstede and Michael Leuschel. Constraint-Based Deadlock Checking of High-Level Specifications. *Theory and Practice of Logic Programming*, 11(4–5):767–782, 2011.

[31] F. Ambert, F. Bouquet, S. Chemin, S. Guenaud, B. Legeard, F. Peureux, M. Utting, and N. Vacelet. BZ-Testing-Tools: A Tool-Set for Test Generation from Z and B using Constraint Logic Programming. In *Proceedings of FATES'02, Formal Approaches to Testing of Software*, pages 105–120, August 2002. Technical Report, INRIA.

[32] Kevin Lano and Howard Haughton. *Specification in B: An introduction using the B toolkit*. World Scientific, 1996.

[33] Idir Ait-Sadoune and Yamine Ait-Ameur. Animating Event B Models by Formal Data Models. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation*, volume 17 of *Communications in Computer and Information Science*, pages 37–55. Springer Berlin Heidelberg, 2008.

[34] Douglas Schenck and Peter Robert Wilson. *Information modeling: the EXPRESS way*, volume 126. Oxford University Press New York, 1994.

[35] Paulo J Matos, Bernd Fischer, and João Marques-Silva. A Lazy Unbounded Model Checker for Event-B. In *Formal Methods and Software Engineering*, pages 485–503. Springer, 2009.

[36] Jens Bendisposto and Michael Leuschel. Proof Assisted Model Checking for B. In Karin Breitman and Ana Cavalcanti, editors, *Proceedings of ICFEM 2009*, volume 5885 of *Lecture Notes in Computer Science*, pages 504–520. Springer, 2009.

[37] M Butler and D Yadav. An incremental development of the Mondex system in Event-B. *Formal Aspects of Computing*, 20(1):61–77, 2008.

[38] B. Legeard, F. Peureux, and Mark Utting. Automated Boundary Testing from Z and B. In L.-H. Eriksson and P. Lindsay, editors, *Proceedings FME'02*, LNCS 2391, pages 21–40. Springer-Verlag, 2002.

[39] Björn Scheuermann. *Reading Between the Packets - Implicit Feedback in Wireless Multihop Networks*. VDM Verlag Dr. Müller, Saarbrücken, Germany, 2008.

[40] Jens Bendisposto, Michael Jastram, Michael Leuschel, Christian Lochert, Björn Scheuermann, and Ingo Weigelt. Validating Wireless Congestion Control and Realiability Protocols using ProB and Rodin. *FMWS 2008: Workshop on Formal Methods for Wireless Systems*, August 2008.

[41] Jens Bendisposto, Michael Leuschel, Olivier Ligot, and Mireille Samia. La validation de modèles Event-B avec le plug-in ProB pour RODIN. *TSI*, pages 1065–1084, 2008.

[42] Dominique Cansell, Stefan Hallerstede, and Ian Oliver. UML-B Specification and Hardware Implementation of a Hamming Coder/Decoder. In *UML-B Specification for Proven Embedded Systems Design*, pages 261–277. Springer, 2004.

[43] Olaf Müller and Tobias Nipkow. Combining Model Checking and Deduction for I/O-Automata. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 1–16. Springer-Verlag, 1995.

[44] Gerard Tel. *Introduction to Distributed Algorithms*. Cambridge university press, 2000.

[45] K. A. Bartlett, R. A. Scantlebury, and P. T. Wilkinson. A Note on Reliable Full-duplex Transmission over Half-duplex Links. *Commun. ACM*, 12(5):260–261, May 1969.

[46] Natarajan Shankar. Combining Theorem Proving and Model Checking through Symbolic Analysis. In *CONCUR*, pages 1–16, 2000.

[47] Amir Pnueli and Elad Shahar. A Platform for Combining Deductive with Algorithmic Verification. *Lecture Notes in Computer Science*, pages 184–195, 1996.

[48] E Clarke, O Grumberg, H Hiraishi, and S Jha. Verification of the Futurebus+ Cache Coherence Protocol. *Formal Methods in System Design*, Jan 1995.

[49] Konstantine Arkoudas, Sarfraz Khurshid, Darko Marinov, and Martin C. Rinard. Integrating Model Checking and Theorem Proving for Relational Reasoning. In *RelMiCS*, pages 21–33, 2003.

[50] Elsa Gunter and Doron Peled. Model Checking, Testing and Verification working together. *Formal Aspects of Computing*, 17(2):201–221, 2005.

[51] Peter Dybjer, Qiao Haiyan, and Makoto Takeyama. Verifying Haskell Programs by Combining Testing, Model Checking and Interactive Theorem Proving. *Information and software technology*, 46(15):1011–1025, 2004.

[52] Jens Bendisposto and Michael Leuschel. Automatic Flow Analysis for Event-B. In Dimitra Giannakopoulou and Fernando Orejas, editors, *Proceedings of Fundamental Approaches to Software Engineering (FASE) 2011*, volume 6603 of *Lecture Notes in Computer Science*, pages 50–64. Springer, 2011.

[53] Stefan Hallerstede. Structured Event-B Models and Proofs. In *ABZ 2010*, Lecture Notes in Computer Science. Springer-Verlag, 2010.

[54] Michael Leuschel and Edward Turner. Visualizing Larger States Spaces in ProB. In Helen Treharne, Steve King, Martin Henson, and Steve Schneider, editors, *Proceedings ZB'2005*, LNCS 3455, pages 6–23. Springer-Verlag, April 2005.

[55] Didier Bert, Marie-Laure Potet, and Nicolas Stouls. GeneSyst: A Tool to Reason About Behavioral Aspects of B Event Specifications. Application to Security Properties. In *ZB 2005*, pages 299–318, 2005.

[56] Renaud Marlet and Cédric Mesnil. Demoney: A demonstrative electronic purse–Card specification. *Trusted Logic*, 2002.

[57] Helen Treharne and Steve Schneider. How to Drive a B Machine. In *ZB'2000*, pages 188–208, 2000.

[58] M.J. Butler. csp2B: A Practical Approach to Combining CSP and B. *Formal Aspects of Computing*, 12:182–198, 2000.

[59] Michael Butler and Michael Leuschel. Combining CSP and B for Specification and Property Verification. In *Proceedings of Formal Methods 2005*, LNCS 3582, pages 221–236, Newcastle upon Tyne, 2005. Springer-Verlag.

[60] Michael Butler. Decomposition Structures for Event-B. In Michael Leuschel and Heike Wehrheim, editors, *Integrated Formal Methods, 7th International Conference, IFM 2009, Düsseldorf, Germany, February 16-19, 2009. Proceedings*, volume 5423 of *Lecture Notes in Computer Science*, pages 20–38. Springer, 2009.

[61] Dominique Cansell, Dominique Mery, and Stephan Merz. Diagram Refinements for the Design of Reactive Systems. *Journal of Universal Computer Science*, 7(2):159–174, feb 2001.

[62] Ivaylo Dobrikov and Michael Leuschel. Optimising the ProB Model Checker for B using Partial Order Reduction. In Dimitra Giannakopoulou and Gwen Salaün, editors, *SEFM 2014*, LNCS 8702, pages 220–234, Grenoble, 2014.

[63] Ivaylo Dobrikov and Michael Leuschel. Optimising the ProB Model Checker for B using Partial Order Reduction. Technical Report STUPS/2014/xx, Institut für Informatik, Heinrich-Heine-University Düsseldorf, 2014.

[64] Jens Bendisposto, Philipp Körner, and Michael Leuschel. Parallel Model Checking of B Specifications. In *Proceedings of the 4th Rodin User and Developer Workshop*, TUCS Lecture Notes. TUCS, 2013.

[65] Gerard J. Holzmann. *The Spin Model Checker: Primer and Reference Manual.* Addison-Wesley, 2004.

[66] Michael Leuschel. The High Road to Formal Validation: Model Checking High-Level versus Low-Level Specifications. In *ABZ 2008*, volume 5238 of *Lecture Notes in Computer Science*, pages 4–23. Springer-Verlag, September 2008.

[67] Unlocking the Mysteries of a Formal Model of an Interlocking System. In Michael Butler and Stefan Hallerstede, editors, *Proceedings of the 5th Rodin User and Developer Workshop, 2014*. University of Southampton, June 2014.

[68] Brian Hayes. The easiest hard problem. *American Scientist*, 90(2):113–117, 2002.

[69] Sami Evangelista and Lars Michael Kristensen. Dynamic State Space Partitioning for External Memory Model Checking. In *Formal Methods for Industrial Critical Systems*, pages 70–85. Springer, 2009.

[70] David Gelernter. Generative Communication in Linda. *ACM Trans. Program. Lang. Syst.*, 7(1):80–112, January 1985.

[71] Matthias Radig. Development and Evaluation of a Framework for Parallelization of ProB. Bachelor thesis, 2011.

[72] Jens Bendisposto, Michael Leuschel, and Markus Borgermans. GEPAVAS Gerichtete und parallele Validierung von abstrakten Spezifikationen - Projektreport. Technical report, University of Düsseldorf, 2010.

[73] Werner Vogels. Eventually Consistent. *Queue*, 6(6):14–19, October 2008.

[74] Philipp Körner. Improving Distributed Model Checking in ProB. Bachelor thesis, 2014.

[75] Mahmoud Sayrafiezadeh. The Birthday Problem revisited. *Mathematics Magazine*, pages 220–223, 1994.

[76] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, Clifford Stein, et al. *Introduction to Algorithms*, volume 2. MIT press Cambridge, 2001.

[77] Thomas Ottmann, Peter Widmayer, Thomas Ottmann, Thomas Ottmann, Peter Widmayer, Informaticien Economiste, and Peter Widmayer. *Algorithmen und Datenstrukturen.* Spektrum, Akad. Verlag, 2002.

[78] Donald E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching.* Addison-Wesley, 1973.

[79] Phil Bagwell. Ideal hash trees. *Es Grands Champs*, 1195, 2001.

[80] Rich Hickey. The Clojure Programming Language. In *Proceedings of the 2008 symposium on Dynamic languages*, DLS '08, pages 1:1–1:1, New York, NY, USA, 2008. ACM.

[81] Pieter Hintjens. *ZeroMQ: Messaging for Many Applications.* " O'Reilly Media, Inc.", 2013.

[82] Corinna Spermann and Michael Leuschel. ProB gets Nauty: Effective Symmetry Reduction for B and Z Models. In *Proceedings TASE 2008*, pages 15–22. IEEE, June 2008.

[83] Michael Leuschel and Thierry Massart. Efficient Approximate Verification of B via Symmetry Markers. *Annals of Mathematics and Artificial Intelligence*, 59(1):81–106, 2010.

[84] Michael Leuschel, Michael Butler, Corinna Spermann, and Edd Turner. Symmetry Reduction for B by Permutation Flooding. In *Proceedings B'2007*, volume 4355 of *Lecture Notes in Computer Science*, pages 79–93. Springer-Verlag, 2007.

[85] Michael Leuschel, Jérôme Falampin, Fabian Fritz, and Daniel Plagge. Automated Property Verification for Large Scale B Models with ProB. *Formal Aspects of Computing*, 23(6):683–709, 2011.

[86] Amazon Web Services, Inc. Amazon EC2 Instances. `http://aws.amazon.com/ec2/instance-types/`. Accessed: 2014-04-01.

[87] Chitra Venkatramani and Tzicker Chiueh. Design, Implementation, and Evaluation of a Software-based Real-time Ethernet Protocol. *SIGCOMM Comput. Commun. Rev.*, 25(4):27–37, October 1995.

[88] J. Barnat, L. Brim, and P. Ročkai. Scalable Shared Memory LTL Model Checking. *International Journal on Software Tools for Technology Transfer*, 12(2):139–153, 2010.

[89] Jean-Paul Bodeveix, Mamoun Filali, Julia Lawall, and Gilles Muller. Formal Methods Meet Domain Specific Languages. In *Proceedings of the 5th international conference on Integrated Formal Methods*, IFM'05, pages 187–206, Berlin, Heidelberg, 2005. Springer-Verlag.

[90] Dubravka Ilic, Linas Laibinis, Timo Latvala, Elena Troubitsyna, and Kimmo Varpaaniemi. Deployment in the Space Sector. In *Industrial Deployment of System Engineering Methods*, pages 45–62. Springer, 2013.

[91] Dubravka Ilic, Timo Latvala, Pauli Väisänen, Kimmo Varpaaniemi, Linas Laibinis, and Elena Troubitsyna. DEPLOY Deliverable D20 D3.1 - Report on Pilot Deployment in the Space Sector. Technical report, 2010. Available at `http://www.deploy-project.eu/html/deliverables.html`.

[92] Dubravka Ilic, Timo Latvala, Laura Nummila, Tuomas Räsänen, Pauli Väisänen, Kimmo Varpaaniemi, Linas Laibinis, Yuliya Prokhorova, Elena Troubitsyna, Alexei Iliasov, Alexander Romanovsky, Michael Butler, Asieh Salehi Fathabadi, Abdolbaghi Rezazadeh, Jean-Christophe Deprez, Renaud De Landtsheer, and Christophe Ponsard. DEPLOY Deliverable D39 D3.2 - Report on Enhanced Deployment in the Space Sector. Technical report, 2011. Available at `http://www.deploy-project.eu/html/deliverables.html`.

[93] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. Model Checking TLA+ Specifications. In *Proceedings CHARME'99*, pages 54–66, 1999.

[94] Gerard J. Holzmann and Dragan Bosnacki. The Design of a Multi-Core Extension of the Spin Model Checker. *IEEE Trans. Software Eng.*, 33(10):659–674, 2007.

[95] Gerard J. Holzmann. A Stack-Slicing Algorithm for Multi-Core Model Checking. *Electronic Notes in Theoretical Computer Science*, 198:3–16, 2008.

[96] Dominik Hansen and Michael Leuschel. Translating B to TLA + for Validation with TLC. In *Proceedings ABZ'14*, LNCS 8477, pages 40–55, 2014.

[97] Dominik Hansen and Michael Leuschel. Translating TLA+ to B for Validation with ProB. In *Proceedings iFM'2012*, LNCS 7321, pages 24–38. Springer, 2012.

[98] J. Barnat, L. Brim, V. Havel, J. Havlíček, J. Kriho, M. Lenčo, P. Ročkai, V. 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.

[99] Kathi Fisler, Ranan Fraer, Gila Kamhi, Moshe Y. Vardi, and Zijiang Yang. Is There a Best Symbolic Cycle-Detection Algorithm? In *In Proc. Tools and Algorithms for Construction and Analysis of Systems, volume 2031 of LNCS*, pages 420–434. Springer, 2001.

[100] Gerard J. Holzmann and Dragan Bosnacki. The Design of a Multicore Extension of the SPIN Model Checker. *IEEE Transactions on Software Engineering*, 33(10):659–674, 2007.

[101] Igor Melatti, Robert Palmer, Geoffrey Sawaya, Yu Yang, Robert M Kirby, and Ganesh Gopalakrishnan. Parallel and Distributed Model Checking in Eddy. *International Journal on Software Tools for Technology Transfer*, 11(1):13–25, 2009.

[102] 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 Methods in Verification, 2010 Ninth International Workshop on, and High Performance Computational Systems Biology, Second International Workshop on*, pages 28–36. IEEE, 2010.

[103] Stefan Blom, Jaco C. van de Pol, and Michael Weber. LTSmin: Distributed and Symbolic Reachability. In Tayssir Touili, Byron Cook, and Paul Jackson, editors, *Computer Aided Verification*, volume 6174 of *LNCS*, pages 354–359. Springer, 2010.

[104] Alfons W. Laarman, Jaco C. van de Pol, and Michael Weber. Multi-Core LTSmin: Marrying Modularity and Scalability. In M. Bobaru, K. Havelund, G. J. Holzmann, and R. Joshi, editors, *NFM 2011*, volume 6617 of *LNCS*, pages 506–511. Springer, 2011.

List of Figures

1.1 Example erroneous Event-B machine 9
1.2 Explicit Consistency and Deadlock Checking 16
1.3 Exploration of a part of the state space 17

2.1 Example: Proof supported checking 21
2.2 State space of the model in Figure 2.1 21
2.3 Organization of the formal model 23
2.4 State space of the m0 machine 25
2.5 Introduce sequential processing 25
2.6 Algorithm of m2 26
2.7 Proof Supported Model Checking 34

3.1 Lifting $x :\in \mathbb{N}$ 46
3.2 Lifting $x, y : |x' = y + x \wedge y' = y - x$ 46
3.3 Lifting $f(x) := f(x) + 1 \parallel x = x + 1$ 47
3.4 Example transformation into the deterministic inlined form 48
3.5 Independent events where $(g, h) \notin \eta$ 51
3.6 Example demonstrating the difference between Definition 3.6 and 3.8. . 54
3.7 Example demonstrating the difference between TLF and DIF in Definition 3.8. 56
3.8 Application of effect independence: State patching. 57
3.9 Explanation of the enable predicate 58
3.10 Counter event 59
3.11 Incrementing values in an array 60
3.12 Graph Representations of Dependence for a Simple Model 61
3.13 No enabling predicate in presence of non-deterministic assigment 62
3.14 Transformed event allows definition of enabling predicate 62
3.15 Simplification of P_E 63
3.16 Algorithm for simplifying the weakest precondition in the context of the invariant and guard of g 64
3.17 Example Model for automatic flow analysis 67
3.18 Enable graph for example from Figure 3.17 69
3.19 Enable graph for example from [53] 70
3.20 Example for combining information 71
3.21 Algorithm for constructing a Flow Graph 73
3.22 Algorithm for expanding the Enable Graph (i.e., computing successor configurations) 73

3.23 Simple Flow Graph Construction 74
3.24 Deadlock finding using flow analysis 76
3.25 Enable graph for deadlock model 77
3.26 Flow graph for deadlock model 77
3.27 Flow graph for deadlock model (after fix) 77
3.28 Structural model from [53] 78
3.29 Flow Graph 79
3.30 Top level specification and first refinement. Taken from [61]. 81

4.1 Visualization of the model checking speedup for an interlocking system . 85
4.2 Dynamic state space partitioning example from [69] 87
4.3 Running the distributed model checker on three nodes 91
4.4 Prolog interface of the distribution framework 92
4.5 Droping work items 93
4.6 Two workers with a different view on the set of known states 94
4.7 Single Node 94
4.8 Storing words in a trie 96
4.9 Example of a Hash Trie 97
4.10 Adding a new layer to the hashtrie 98
4.11 Inserting a hashcode into the trie 98
4.12 Memory layout of an hashtrie entry 99
4.13 Mapping of shared memory segments into different addresses in the virtual address space 99
4.14 Sequence Diagram of work item processing 104
4.15 Work items without and with symmetry markers 107
4.16 Runtimes of work items of the CBTC system 110
4.17 Effects of the CPU / Hyper-Threads 115
4.18 Impact of the parallelization framework 116
4.19 Limited scaling 117
4.20 Stuttgart 21 Interlocking 118
4.21 Interlocking from [11] 119
4.22 Bus Systems 119
4.23 Real time Ethernet Protocol 120
4.24 Process Scheduler 121
4.25 Mode protocol for a DSAOCSS system 122
4.26 Set Game 122
4.27 Interlocking system on a high performance cluster 123
4.28 Gurdag Model with multi-core Spin 128

List of Tables

1.1 Exploring the state space in Figure 1.3 16

2.1 Experimental results (single refinement level check) 40
2.2 Experimental results (multiple refinement level check) 40
2.3 Experimental results (more recent ProB) 40
2.4 Number of invariants evaluated (single refinement level check). 41

3.1 Read/write sets for the model in Figure 3.11 60
3.2 Enable Predicates for the model in Figure 3.11 60
3.3 Read/Write sets 68
3.4 Enable predicates 68

4.1 Assertion Checking of a CBTC system 110
4.2 Runtime on Mac Pro 112
4.3 Runtime on Amazon EC2 113
4.4 Speedup factors on Mac Pro 113
4.5 Speedup factors on Amazon EC2 114