HEINRICH HEINE
UNIVERSITÄT DÜSSELDORF

# Supporting Validation and Verification of State-Based Formal Models

Inaugural-Dissertation

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

vorgelegt von

**Daniel Plagge**

aus Damme

Düsseldorf, September 2015

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. Marc Denecker
Katholieke Universiteit Leuven

Tag der mündlichen Prüfung: 19. November 2015

# Abstract

The four presented articles in this work address distinct aspects of supporting the development, validation and verification of state-based systems with formal models. State-based models describe a system by defining what constitutes a state and when and how a transition to a new state can be performed.

Software tools play a central role in supporting system development. *P*roB is such a tool that visualises and analyses the behaviour of a formal specified system via "animation" and that performs automated model checking to discover errors in the specification. Animation and model checking contribute to the development of critical software systems and the validation of systems under development.

Each of the presented researches had lead to an implementation of an extension to the development and validation tool *P*roB that has been written originally for the B method.

The first presented article explains how *P*roB has been extended to support the widely-used specification language Z. It shows how errors were found just by animating specifications written in Z.

Another extension to *P*roB covers also a specification language, Event-B, a successor of the B method. A main characteristic of Event-B is its refinement mechanism and the main challenge was to support refinement and allow a user to identify problems in refined models.

With formulas in temporal logic it is possible to express requirements with dependencies between a sequence of states. The extension presented in the third part verifies whether a model fulfils the given properties and therefore it allows to validate if the model meets the requirements.

The last presented article applies a SAT solver via an external library to specifications loaded into *P*roB. SAT solving is used as a complementary technique in *P*roB to improve its capability to animate specifications. A specific aspect of this translation is that the target formalism is less expressive than the formalism supported by *P*roB and not all encountered parts of a problem statement can be translated effectively.

For every presented article it is shown how it contributes to validation and verification of systems and which of *P*roB's components are affected by the implementation of the corresponding extension.

# Zusammenfassung

Zustandsbasierte Modelle beschreiben ein System, indem definiert wird, was einen Zustand des Systems ausmacht und auf welche Weise und unter welchen Umständen Zustandswechsel vollzogen werden können. Die hier vorgestellte Arbeit befasst sich mit der Unterstützung von Entwicklung, Validierung und Verifikation von Systemen, die mittels zustandsbasierten formalen Modellen entworfen werden.

Eine zentrale Rolle bei der Unterstützung der Systementwicklung bilden Softwarewerkzeuge. Bei *P*roB handelt es sich um solch ein Werkzeug, welches das Verhalten formaler Spezifikationen für einen Benutzer begreifbar macht („Animation“) und automatisierte Modellprüfungen darauf ausführt. Dies trägt bei der Entwicklung kritischer Software mit zur Validierung und Verifizierung des zu entwickelnden Systems bei. Alle Arbeiten resultierten in konkrete Erweiterungen des Entwicklungs- und Validierungswerkzeuges *P*roB, das ursprünglich für die B-Methode geschrieben wurde.

Zunächst wird erläutert wie *P*roB erweitert wurde, um die Spezifikationssprache Z zu unterstützen. Es wird gezeigt, wie allein durch Animation von Z Spezifikationen sicherheitskritischer Anwendungen in diesen Fehler gefunden werden konnten.

Auch die nächste Arbeit erweitert *P*roB um eine Spezifikationssprache. Ein wesentliches Merkmal von Event-B ist die Möglichkeit zur Modellverfeinerung („Refinement“), auf deren Unterstützung ein besonderes Augenmerk gelegt wurde. Entwickler sollen in die Lage versetzt werden, schnell Probleme im Refinement zu entdecken und zu beheben.

Mittels temporal-logischer Formeln lassen sich Anforderungen an das System formulieren, die sich über eine Abfolge von Zuständen und Zustandsänderungen erstrecken. Die vorgestellte Erweiterung für *P*roB kann prüfen, ob ein System diese Anforderungen erfüllt.

Der letzte Teil befasst sich damit, wie Modelle, die von *P*roB analysiert werden, so übersetzt werden könne, dass mit Hilfe von SAT Solvern Lösungen gesucht werden können. SAT Solving steht damit als komplementäre Technologie zur Verfügung, um um die Möglichkeiten zur Analyse komplexer Modelle zur verbessern.

Zu allen Arbeiten wird kurz vorgestellt, wie sie zur Validierung und Verifikation von Systemen beitragen und welche Komponenten von *P*roB die Implementierung der entsprechenden Erweiterung betrifft.

# Acknowledgements

First of all I want to express my heartfelt thanks to Michael Leuschel. At the University of Düsseldorf, Michael created a fantastic environment to work and research together. I always enjoyed being part of the team in Düsseldorf and the work, research and discussions in the friendly atmosphere. I thank Michael very much for his support, especially during those times when it was occasionally a challenge to combine work and family life. And last but not least he proved that he has an incredible patience by waiting for me finishing this thesis.

I also want to thank my dear colleagues in Düsseldorf. They of course made the atmosphere there the way it was. In this context I have to mention explicitly Jens Bendisposto. I had the luck to stay in the same office with him for a while and could observe his commitment and how he helped whereever it was possible.

# Contents

# 1 Introduction

The articles collected in this thesis have all in common that they explore different techniques to analyse state-based models with the tool *P*roB. They all describe new functionality implemented as additions to *P*roB.

In Section 1.1 of this introduction, we want to gain a basic understanding of what state-based models are and what we want to achieve with the newly developed aspects of the tool. We have a brief look at the concepts of verification and validation to ensure the quality of a software system and how a tool like *P*roB can support verification and validation of such state-based systems.

Because *P*roB plays such a central part in this thesis, Section 1.2 gives an overview about its structure and main features.

The next four sections (1.3–1.6) give brief introductions to the topics of the articles that are included in this thesis and covers some questions:

- What new capabilities are gained for *P*roB by the extension?
- How are verification and validation affected?
- Where has it been integrated into *P*roB and how does it affect the tool's structure?
- How does the extension affects the interaction with the user?
- How has the feature been developed after the release of the article and what are potential future works?

Also some bureaucratic aspects are presented:

- All articles have two or more authors. What is the contribution of this thesis' author to the specific article? Every participating author had contributed to each article as whole and most ideas and concepts have been elaborated together.
- Where was the article first published and what is the journal's impact factor if there exists any?

## 1.1 State-based Formal Methods and Animation

Requirements and specifications expressed in a natural language usually contain ambiguities which can be a source to many errors during the development of a software system. Formal methods approach this problem by specifying properties of the system with mathematical logic.

The effect of applying formal methods are twofold. Firstly, writing specifications and expressing properties of the system in a rigorous way that avoids ambiguities remove a major source of errors during the development. Secondly, the availability of a specification with clearly defined semantics allow to reason about the specification's properties and apply verification techniques like mathematical proofs or model checking (see Section 1.1.4). When a formal method is applied to the complete development phase of the software, it can even been proved that an implementation meets its specification.

### 1.1.1 Validation and Verification (V&V)

ISO 8402-1994 [fS94] defines validation as (emphasis by us):

> Confirmation by examination and provisions of objective evidence that the particular *requirements for a specific use are fulfilled*.

The same standard defines verification as

> Confirmation by examination and provisions of objective evidence that the *specified requirements have been fulfilled*.

Often, validation and verification are described as "Are we building the right thing?" (validation) and "Are we building it right?" (verification).

We now give a small introduction to state-based formal methods and how are they used to support validation and verification.

### 1.1.2 Formal Methods and V&V

Formal specifications provide a rigorous description of a system due to their mathematical nature. On the other hand they are usually provide a high level of abstraction in comparison to code written in a programming language. The use of formal methods has several impacts to verification and validation:

- The most prominent use of a formal specification is to show that a software implementation meets is specification. This verification can be done by formal proof or in some cases also by model-checking.

- Using a formal method affects already the way we think about the original problem statement and helps to find flaws in the requirements — in other words it helps us to validate the requirements. The following quote comes from a report about the experiences in industrial use of formal methods [Bar11]:

  > [...] that benefits are first realised as the very act of expression within a formal notation causes the author to explore the problem domain with a logical mindset — thereby detecting and investigating incompleteness in the requirements early in the life cycle.

- Formal methods allow to express properties that a system should have. A specification can then be checked if it has those properties. An example for this are temporal properties encoded in LTL that can be checked by a model-checker. Another example would be to specify series of actions that the model should be able to perform. These "test cases" provide a kind of validation that needed functionality is present in the developed system.

- Animation (see Section 1.1.4) of a specification allows the user to gain a better understanding on how the specified system works and to see if it fits the requirements.

The benefit of a formal specification can already be so large that it is used even in situations where the development of the software system continues in an informal way and without additional verification techniques like formal proof [Bar11]. In a case study [DVR96] for the NASA the authors applied formal methods to the requirements analysis and uncovered several flaws that have not been noticed in the existing informal process.

### 1.1.3 State-based Formal Methods

State-based formalisms focus on an explicit description of what constitutes the state of a system and how a state can be altered under what conditions. Usually this is done by defining a set of variables and their possible range of values.

Transitions between states are defined by operations or events. Usually in software systems the term operation is used whereas in system modelling the term event is preferred because also non-software events like "train A moves from track segment X to segment Y" are part of the specification. A definition of an event usually contains a condition when it can happen (called "guard") or in which situation the operation can be called (called "precondition") and by an effect that describes how the variables of the system will be changed. The behaviour of such systems is implicitly given by observing how the system can evolve starting from initial states by successively applying operations resp. events to the state.

In this work, we will use the terms operation and event synonymously.

Later we will see that the B method and Event-B also provide a mechanism called refinement which makes it possible to start with a very abstract and concise specification of a system and gradually add more complex properties.

#### Formal methods besides state-based methods

There are other forms of formal methods that do not focus on the system's state. We use the data structure of a stack as small example to illustrate the different approaches to specify its properties.

In **state-based** approaches like B, we define the possible states by variables and its possible values. In this case, we define a variable *size* for the number of values currently stored in the stack and a variable *stack* that contains a mapping from the position (0 at the bottom, $size - 1$ at the top) to the concrete value (an integer):

$$\begin{aligned}&size \in \mathbb{Z}\\ &stack \in 0\,..\,size - 1 \longrightarrow \mathbb{Z}\end{aligned}$$

The operation *push* increases the stack and puts the new element $x$ on top:

```
$push(x) = \mathrm{PRE}\ x \in \mathbb{Z}$
  BEGIN
    $size := size + 1 \parallel$
    $stack := stack \cup \{size \mapsto x\}$
  END
```

The stack must be non-empty when the operation *pop* can be called. It decreases the stack and returns the top element $x$ (the expression $1\,..\,size - 2 \lhd stack$ contains all elements of the stack between position 1 and $size - 2$):

```
$x \longleftarrow pop = \mathrm{PRE}\ size > 0$
  BEGIN
    $size := size - 1 \parallel$
    $stack := 1\,..\,size - 2 \lhd stack \parallel$
    $x := stack(size - 1)$
  END
```

**Process algebras** (e.g. CSP [Hoa85] and CCS [Mil82]) specify in which order events can happen in a system and how concurrently running processes can interact via messages.

In CSP we can define a process $P$ which can either do nothing (*skip*) or an element $x$ can be pushed to the stack which can be retrieved later via *pop*. Between these events, the process can happen recursively and arbitrarily often.

$$P = skip \quad \Box \quad push!x \rightarrow P \rightarrow pop?x \rightarrow P$$

The operator $\Box$ denotes external choice, i.e. the process can behave either like its left or its right argument but the choice which one is taken is made by the environment.

Actually, *P*roB supports CSP and the LTL model-checker (see Section 1.5 or the article in Chapter 4) is also applicable to CSP specifications. But the main focus of this work are state-based formalisms.

**Algebraic methods** concentrate on the properties of mathematical objects. For example, it could be stated that first adding and then removing an element from a data structure would leave it effectively unchanged. So the following holds for a stack $s$ and two functions *push* and *pop* that return a modified stack as result:

$$pop(\,push(s,x)\,) = s$$

### 1.1.4 Animation and Model-Checking

**Animation**

Animation of a formal model denotes the search for concrete valuations of variables that match specified properties. The term animation is at least used in the B, Event-B and Z community. An alternative term is "simulation" as it is used in the ASM community [FGG07].

Animation also means that events or operations can be evaluated which again can be used to explore the reachable states of a model in a state-based formalism. It shows the user concrete examples of how the system can evolve with time. Thus animation plays a important role for the validation of a specification by giving the user a deeper understanding on how the specified system will behave and if it fits the informal requirements. Even without checking the model against further formal properties, animation can help to uncover flaws in the specification. In our article about animating Z specifications (Section 1.3) we describe how we animated a (modified) industrial specification of a safety critical system and discovered errors just by inspecting the animation.

**Model-Checking**

With model checking [CGP99] we can verify that a model fulfils certain properties. Usually this is done by trying to find a counter-example to the desired property which makes it also easier to communicate to the user why the check failed. In contrast to formal proofs the verification is performed fully automatically, thus model checking is often referred to as a "push-button" technology.

*Explicit-state* model checking systematically computes all reachable states of a model and checks if the explored state space satisfies the desired properties. Animation can be used to generate the possible states. *P*roB supports basically two types of explicit-state model checking. Firstly, it can be used to check each state against a given property (e.g. if an invariant holds or if the state is a deadlock state where no more events are enabled) or secondly the LTL model checker (see Section 1.5) verifies if every possible executable sequence of states and events satisfies a given formula in temporal logic.

*Symbolic* model checking on the other hands avoids the enumeration of reachable states. Instead it tries to find a solution to a logical formula that describes a possible counter-example. *P*roB also supports symbolic model checking for several purposes. E.g. under the name "constraint-based invariant check" it can be checked if the occurrence of an event $E$ in a valid state always leads to a valid state. The logical formula to solve can be stated as follows: Let the variable $v$ represent a state of the system where the invariant $I$ holds. The event $E$'s before-after-predicate $BA(E)$ connects $v$ with a successor state $v'$. In a counter-example the invariant $I$ does not hold for $v'$ anymore. Formally, we look for a solution for:

$$\exists\, v, v' \mid I \wedge BA(E) \wedge \neg\, I'$$

> Animation can be used to search for solutions of this formula.

As we have seen, animation can serve — and does so in *P*roB — as a technological fundament for both explicit and symbolic model checking.

Model checking has the advantage over proof that it can be done automatically without user interaction. But the downside of model checking is that the success of the method depends on the complexity of the model. Even simple models can lead to a huge or even infinite number of possible states that are impossible to check and where the absence of errors cannot be guaranteed anymore. Also for model checking often additional assumptions are made to limit the complexity of the search. For example, the size of a deferred sets is fixed to a small integer for a check whereas the specification states no further limitations on its cardinality.

But even then model checking can be a valuable tool for verification to the engineer. In case a proof cannot be discharged automatically, a model checker can serve for "debugging" the specification. In situations where a prover is not able to discharge a proof obligation it is not clear whether there is no proof (i.e. the proof obligation is not valid) or the prover is to weak to find it. In general this problem (deciding whether a predicate can be proven given a set of hypotheses) is known as the "Entscheidungsproblem" and it had been shown by Church and Turing [Chu36, Tur36] that no general solution exists. For the engineer this means concretely that he has to decide whether to spend more effort into trying to prove the goal interactively or to understand how the model must be changed to work correctly. A model checker can now try to find a counter-example to the proof obligation. If such a counter-example can be presented, it avoids unnecessary effort and helps the engineer to comprehend the problem in model. Of course, the model-checker can neither give a definite answer whether a predicate holds or not in general. That would contradict the undecidability of the "Entscheidungsproblem".

As we have seen, model-checking is primarily used as a verification technique. But it also has an impact to validation because it enables us to check if the a specification has certain additional properties (e.g. a temporal formula given in LTL holds). With this ability we can formalise requirements and check if the system we are specifying implements these. Thus we are asking again the question "are we building the right thing?" and have an answer (specific to a certain feature the model should have) by the model-checker.

### 1.1.5 The B Method and Event-B

Since *P*roB was originally developed for the B method we give a brief overview about the core features of this formalism and its successor Event-B. For a more detailed explanation, see [Sch01, Abr96] for classical B resp. [FM, Abr10] for Event-B.

To express formal statements both formalisms use predicate logic with arithmetic, set theory and relations.

### The B Method

The B method is designed to support specification and implementation of software components. The central artefact of a specification is a so-called abstract machine. An abstract machine can have the following components:

*Constants and properties* The values of constants do not change during the operation of a machine. A predicate is used to define which properties these constants have. For example, for a railway interlocking system we can define a constant *topo* that represents the track topology and in the properties we express that *topo* is a relation between track segments that represents an undirected graph.

*Variables and invariant* With variables we define what constitutes a state of a system. A predicate, called the invariant, describes which states are assumed to be always valid during the execution of the system. In the interlocking example from above, a variable *pos* could be used to represent the current position of trains. The invariant states that it is a function from a train to a track segment and that two trains are not allowed to be located on the same segment.

*Operations* Operations describe which precondition must be satisfied when they are called and how the machine's variables will be changed. B provides an extensive set of "generalised substitutions" to enable a user to specify the behaviour of the operations. These can be simple assignments (like $x := E$ for $x$'s new value will be $E$), more complex structures like if-then-else constructs, up to "any"-statements where local variables can be introduced whose values are specified by a predicate.

The machine's initialisation is a special case of an operation.

The consistency of a machine (i.e. the invariant holds in any reachable state) is proven by induction: First it must be shown that the initial states fulfil the invariant. Then for each operation it must be proven that the new state also fulfils the invariant under the assumption that the original state already fulfilled it and that the precondition holds.

Thus the machine's invariant is used to encode two kinds of properties:

- Safety properties (something that must never be false) that originate directly from the system's requirements.
- Properties that are added to the invariant because they are needed for a proof of invariant preservation for an operation.

A central concept of the B method is the stepwise refinement of an abstract machine where subsequently more details are added to the specification. The aim of the refinement steps is to develop an implementation starting from an abstract and easy-to-read specification. It must be proven for each operation in a refinement that its behaviour corresponds to the behaviour of the refined operation. Finally, the result is an implementation that is proven to behave like the abstract machine specifies.

### Event-B

Whereas the focus of the B methods lies on the specification, development and implementation of software components, Event-B's purpose is system development. In general, Event-B can be regarded as a simplification of classical B:

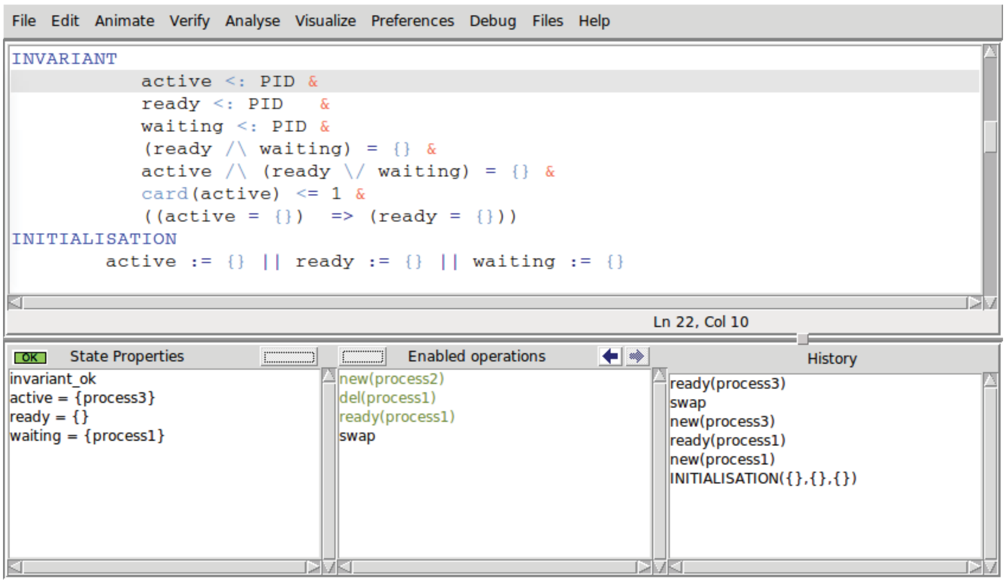
- The mathematical toolkit is very similar to classical B but some features like sequences have been removed. Theories [BM13] introduce a generic way to define new data types and operators.
- Classical B has many forms to combine and re-use machines (constructs like "includes", "uses" or "sees"). Event-B has two types of artifacts: Machines with variables, invariants and events to encode the dynamic behaviour of a system where refinement is the only supported relation between two machines. Contexts contain constants and axioms and describe the static properties of a system. A context can extend an arbitrary number of other contexts.
- An event just contains parameters, guards and assignments. The assignments have basically the form $var := expression$. Classical B in contrast supports many general substitutions like `ANY`, `PRE` or `IF–THEN–ELSE`. They can be arbitrarily nested.
- The proof obligations for refinement have been simplified. During refinement Event-B allows to strengthen a guard with the result that events might not be able to happen in situations where the abstract specification allows the event to happen. This simplifies the design of comlex systems but can be a limitation if software components are developed where an implementation must be able to act upon every possible specified input.
- Code generation to executable languages like C or ADA is (at the time of writing) not an integral part of Event-B. Several methods to generate code from refined Event-B models have been proposed (e.g. [MS11, EBM+12, FHB+14]).

### 1.1.6 *P*roB

*P*roB [LB08] was originally developed to animate and model check specifications that are written with the B method.

Figure 1.1 shows *P*roB animating a scheduler specified in B. The specification's source code is shown in the upper half of the picture, in the lower half you can see three panels. The left panel shows properties of a concrete state that has been reached by executing successively some operations. In this case, it shows that the invariant is not violated and a list of the variables' values. The right panel the operations that have been executed to reach the current state with the last one on top and the first at the bottom. The panel in the middle shows the operations that are currently available. When a user clicks on one of the operations, the new state is shown.

For the support of Event-B another user interface based on the Eclipse platform was developed that allowed *P*roB to be integrated into the Rodin development platform.


File Edit Animate Verify Analyse Visualize Preferences Debug Files Help
Ln 22, Col 10
OK
State Properties
Enabled operations
History

```
INVARIANT
           active <: PID &
           ready <: PID   &
           waiting <: PID &
           (ready /\ waiting) = {} &
           active /\ (ready \/ waiting) = {} &
           card(active) <= 1 &
           ((active = {})  => (ready = {}))
INITIALISATION
        active := {} || ready := {} || waiting := {}
```

```
invariant_ok
active = {process3}
ready = {}
waiting = {process1}
```

```
new(process2)
del(process1)
ready(process1)
swap
```

```
ready(process3)
swap
new(process3)
ready(process1)
new(process1)
INITIALISATION({},{},{})
```

Figure 1.1: Screenshot of *ProB*'s Tcl/TK user interface, version 1.5.0

Arguably, that interface is more user friendly but does not support all features that are provided by *ProB*. At the time of writing "*ProB* 2" is under active development which should replace the Tcl/TK user interface by an Eclipse and HTML-based interface in future [BCD$^+$13].

### Supported formalisms

B remained not the only formalism supported by *ProB*, amongst others it supports the state-based methods:

- Event-B (see Chapter 1.4)
- The Z notation (see Chapter 1.3)
- TLA$^+$ [HL12, Lam02]

With the process algebra CSP (Communicating Sequential Processes [Hoa85]) *ProB* supports even a formal method that is not state-based [LF08]. Also a combination of B and CSP, called CSP||B, is supported. The CSP part of such specifications defines in which order B operations can be called.

### Basic features of *ProB*

*ProB* provides a number of features that allow the validation and verification of formal models. We give a brief list of important features:

*User friendliness* One of the basic principles of *P*roB is that the barrier for a user to animate a specification should be kept as low as possible. Usually opening a specification file is already enough to start its animation. Only for some more complex models a user needs to alter preferences that influence how the animation or the model check behaves.

*Animation* *P*roB tries to find possible values for variables and constants that match the given properties without further interaction with the user. It presents the computed solutions in the user interface and shows which operations are available in the current state.

Animation gives the user insights into the specification and allows him to check if the model behaves as expected.

*Model checking* The state space can be explored automatically via animation and each encountered state is then checked for errors like invariant violations. With the implementation of the LTL extension (see Chapter 1.5) an alternative approach to model checking can be used. Both modes of model checking are forms of explicit model checking because each encountered state of the model is stored and handled explicitly.

To reduce the possible number of states that have to be checked, *P*roB implements various forms of "symmetry reduction" [SL08, LBST07, LM07].

*Graphical visualisation of a state* BMotionStudio is an extension to *P*roB that allows to create a graphical representation of the animated system. This improves the comprehensibility of a system's behaviour and can make the system's state comprehensible even for domain experts who may not be able to read and understand the formalism used to specify the system.

*Visualisation of the state-space* The state space can be visualised as a graph where each state is represented by a node and operations that lead from one state to another are represented by an edge.

*Evaluation of expressions* A user has the ability to enter arbitrary expressions or predicates that can be evaluated in every state of the model. This can be done in a console view, similar to command line interfaces of operating systems or via a tree that represents the formula (see Fig. 1.2) and that can be used to inspect sub-formulas of expressions.

*Constraint based checks* The model checking that we presented above is based on exploring the state space. Another approach to check properties is to set up a predicate that describes the counter-example to such a property and try to find a solution to the predicate with *P*roB's animation facilities.

Examples for constraint based checks are

- invariant preservation: For a system with variables $v$ and invariant $I$ and an event $e$ with before-after-predicate $BA(e)$ find a solution for:

  $$\exists\, v, v' \cdot I \land BA(e) \land \neg\, I'$$

- deadlock freedom: For a system with variables $v$, invariant $I$ and events $e_1$, ..., $e_n$ with their guards $G_1, \ldots, G_n$, find a solution for:

  $$\exists\, v \cdot I \land \neg\, (G_1 \lor \ldots \lor G_n)$$

- test case generation: To generate test cases based on a given model, the goal is to find sequences of events that cover certain events or states that match given constraints. One approach as described in [WKR+09] is to search for matching sequences in the generated state space.

  Another approach is to set up a predicate that describes a specific sequence of events to check if there exists a valuation for the constants and variables which fulfils the predicate and additional constraints. This approach is a kind of constraint based check and had been successfully applied in models where the coverage of some situations depends on the choice of constants or initial variables.

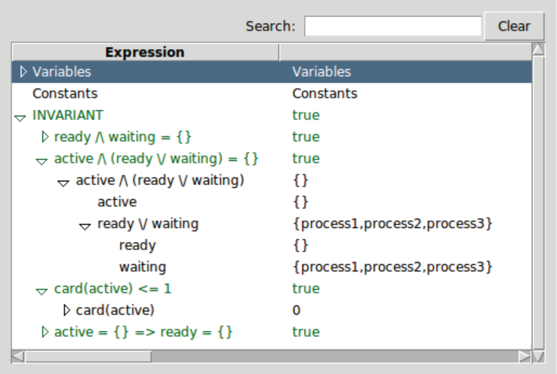
At the time of writing, various other examples for constrained based checks and evaluations are added, e.g. a check if an invariant is always fulfilled if other invariants are fulfilled. In that case the superfluous invariant could be marked as a theorem by the user to save effort in proving of the model's correctness.

*Trace checking* During an animation, a user has the ability to save the current history – the sequence of events from an initial state to the current state – to a file. Later the file can be loaded again by *P*roB to check whether it is possible to replay the sequence of events in a model. Used this way, the trace is a kind of test case for the developed model. This feature provides a mean to validate the model by checking that it shows a certain behaviour.

*Disprover* The development platform Rodin generates proof obligations for a model that must be proven automatically or interactively by an user to show the model's correctness. A proof obligation basically consists of a set of hypotheses $H$ and a goal $G$ and the formula $H \Rightarrow G$ must be proven. *P*roB can be used as a "disprover", it tries to find a counter-example for $H \land \neg\, G$. This can save a lot of time and effort for an engineer who is developing a model of a system because it spares him the work of proving an incorrect property.

This feature can be regarded as a special case of the constraint based checks. Together with other constraint based checks and a search for invariant violations these are core features to support an engineer in his verification effort.

In some cases we can be certain that *P*roB did an exhaustive check when trying to find a counter-example. In those special cases the failure to find one can be regarded as a proof for the validity of the proof obligation [KBL14].



| Expression | |
|---|---|
| ▷ Variables | Variables |
| Constants | Constants |
| ▽ INVARIANT | true |
| ▷ ready /\ waiting = {} | true |
| ▽ active /\ (ready \/ waiting) = {} | true |
| ▽ active /\ (ready \/ waiting) | {} |
| active | {} |
| ▽ ready \/ waiting | {process1,process2,process3} |
| ready | {} |
| waiting | {process1,process2,process3} |
| ▽ card(active) <= 1 | true |
| ▷ card(active) | 0 |
| ▷ active = {} => ready = {} | true |

Figure 1.2: Evaluation as a tree in *P*roB's Tcl/TK interface (version 1.5.0)

## 1.2 The *P*roB Architecture

Since all four articles in the chapters below describe work that had directly resulted in extensions to *P*roB, we give a brief overview about how *P*roB is structured. Later we will describe which parts are affected by each extension.

Besides the user interfaces, most parts of *P*roB are written in the programming language Prolog [Ste94]. Especially the constraint solving core of *P*roB makes use of modern Prolog features like co-routining and constraint solving.

### 1.2.1 A Walkthrough through *P*roB

To see how *P*roB works, we have a look at what happens when a user wants to load a B specification. Figure 1.3 shows a simplified outline of *P*roB's architecture. The blue rectangles represent components that are presented in the following example. These components are the central part of *P*roB's animation and model checking capabilities.

The user interface at the top of the figure invokes and controls the different functionalities below. At the time of writing their are at least four different user interfaces:

- A graphical interface written in Tcl/TK that provides most functions of *P*roB.
- A plug-in to the Rodin platform, which is another graphical interface that has an arguably more modern feel to it then the Tcl/TK interface, but it does not provide access to *P*roB's complete functionality and is restricted to Event-B specifications.
- A web-based interface (*P*roB 2) that is also integrated in Eclipse is currently under development and should replace the two interfaces above.
- *P*roB can also be invoked from the command line to perform tasks without any further user interaction and need for a graphical user interface.

Many modules of *P*roB are not directly accessible to the user, they provide various internal functions that are used by other parts of the system. They are represented by the

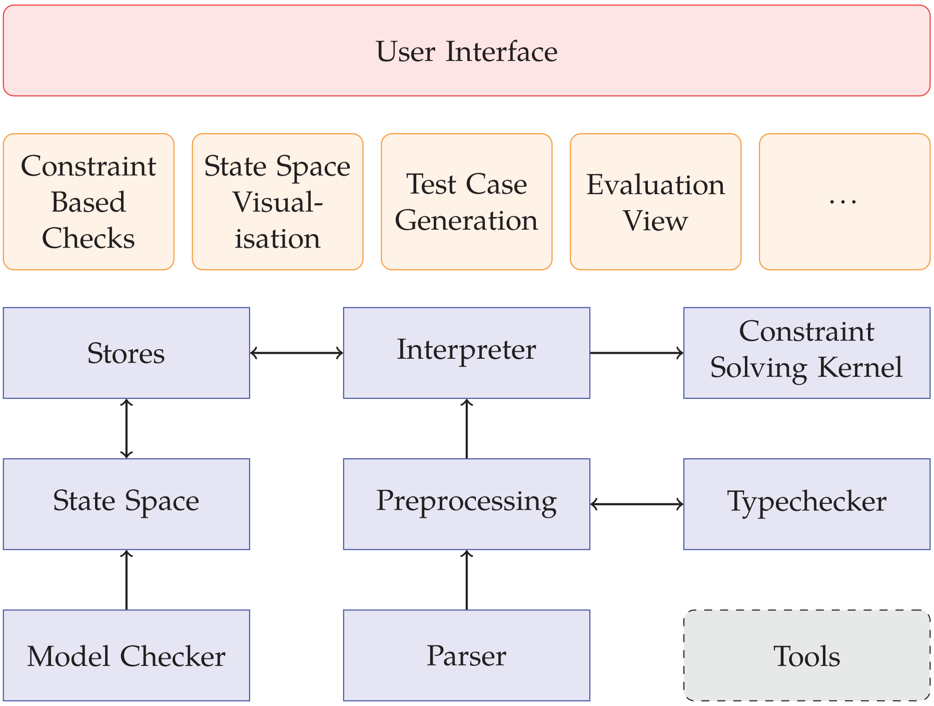


Figure 1.3: Overview over *ProB*'s structure

grey box labelled "Tools" in the figure. Examples are manipulations to abstract syntax trees, logging, etc.

We have described various aspects of *ProB* in Section 1.1.6 above. Many of those features are an integral part of *ProB* and are represented by the orange boxes in the figure.

### Loading the specification

After a user has selected the file containing the model the parser is invoked, a preprocessing module extracts the different part of the specification (like variables, invariant, operations, etc.), determines visibility rules for identifiers and applies the typechecker to the syntax tree. The result is a representation of the loaded machine with abstract syntax trees for the various expressions and predicates in the specification.

### Interpreting the specification

Let's assume the specification defines two constants $x, y$ and properties $y = 2 \cdot x \wedge x \in \{2, 3\}$ that should be fulfilled by them. *ProB* sets up a "store" with entries for $x$ and $y$ that initially contain their undefined values.

Then the interpreter is invoked on the predicate – the interpreter looks up values in the store whenever a constant is referenced. The constrained solving core is called for each of the encountered expressions or properties. For the left part ($y = 2 \cdot x$) of our small example, the interpreter would look up the values of $x$ and $y$ in the store which would still not be bounded to concrete values. Then it would call a Prolog predicate in the core that computes the value of $2 \cdot x$ and then another call would be used to state that the result of the computation is equal to $y$'s value. Since $x$ is not yet bound to any value, $2 \cdot x$ neither has a concrete value at this stage, but constraints are bound to the Prolog variables that represent them. After the interpreter has evaluated both the left and right side of the predicate, no concrete values for $x$ and $y$ have been found but the constraint solver is able to infer that $x$'s value is between $2$ and $3$.

Because there are still variables in the store that have no concrete value, enumeration of possible values is initiated. Since the bounds for $x$ are known, *P*roB sets first $x$ to $2$, then to $3$. Due to the constraints that were set up by the interpreter, the value for $y$ will be directly computed. *P*roB will find two solutions for the stores with $x = 2, y = 4$ and $x = 3, y = 6$.

**State space**

The two possible states that have been found are stored in the state space as two possible configurations for the constants. In an animation, the user can choose between both solutions as if the set up of the constants were operations of the specification.

When a user selects one of the solutions, the selected store is retrieved from the state space and a new store with the model's variables is created. The interpreter then evaluates the initialisation of the machine to find values for the variables. Again, after a possible enumeration phase, found solutions are stored in the state space.

After initialisation, this process is then repeated for each newly encountered state and each event. Beside the store that holds the values of variables and constants, also the transitions between those states are stored which are usually the operations and the values for their arguments.

**Model checking**

A user might then choose to model check the specification. In that case the model-checker drives automatically the process described above. It selects a state in the state space that has not been explored yet. Then it uses the interpreter to find transitions to new or already encountered states. Each newly encountered state is also checked for a number of properties. Usually the invariant is checked by invoking the interpreter. A test for deadlocks would check if there are any outgoing transitions.

The model checking terminates when an error is encountered, all reachable states have been explored or the user interrupts the check.

## 1.3 Animating Z

Z's dead, baby. Z's dead.

*(Butch in "Pulp Fiction" [Tar94])*

The article (Section 2) describes an extension to *P*roB (called *P*roZ) that allows to animate specifications written with the Z notation. The extension basically translates a Z specification to *P*roB's internal representation of a B machine. Since Z has some constructs like free types and some operators that cannot be effectively translated to B, some additions had also be made to *P*roB's interpreter and constraint solver. Due to Z's lack of structure it is assumed that the specification follows some conventions.

### 1.3.1 The Z Notation

The Z notation is a predecessor of the B method. Both formalisms share a common mathematical tool kit based on axiomatic set theory and first-order predicate logic.

As its name implies the B method is a software development methodology where an implementation of a formal specification is derived by refinement steps. The implementation represents executable code and can be transformed to C or ADA source code. The B method provides several constructs like variables, operations or invariants to identify various aspects of the system.

Z on the other hand provides a more general notation for specifications, but it flexibility is due to the fact that the surrounding natural language text must explain what the purpose of a formal statement is. Z's main mechanism to structure a specification is a schema. Schemas consist of a name, variable declarations and a predicate. They can be combined with the "schema calculus".

Schemas can be used to specify many things: E.g. they describe possible states of a system or just parts of it, operations or events can be specified, or they can be used to describe data structures.

Whereas the B-method provides concrete steps via the generation of proof obligations to proof the consistency of a model, a user of Z must take care himself of the verification of the desired properties. Similarly, there are approaches for refinement in Z [WD96] but it has no specific support for it and a user has to describe himself the purpose of the relevant schemas.

#### Specifying State-based Systems with Z

The usual approach to model a state-based system in Z is to specify a schema that describes the variables of the system and their possible values. Operations or events are then specified by schemas that connect two states with a "before-after-predicate".

We give a small example for the specification of a lift. We describe two components of the system with schemas, the cabin that is located at a specific floor (between the constants *bottom* and *top*) and the doors that are either open or closed:

$$\begin{array}{|l}\hline \textit{Cabin} \\ \textit{floor} : \mathbb{Z} \\ \hline \textit{bottom} \leq \textit{floor} \leq \textit{top} \\ \hline \end{array} \qquad \begin{array}{|l}\hline \textit{Door} \\ \textit{door} : \textit{DSTATUS} \\ \hline \end{array}$$

We assume that *DSTATUS* is a type with exactly the two values *OPEN* and *CLOSED*:

$$\textit{DSTATUS} ::= \textit{OPEN} \mid \textit{CLOSED}$$

The whole system can be described as the combination of both parts:

$$\textit{System} \mathrel{\widehat{=}} \textit{Cabin} \land \textit{Door}$$

We specify an operation by a schema *Up* to move the lift one level up. We include the schema *System* and a variant *System′* where all variables are primed (like *floor′*). The latter refer to the new state. We define the operation by specifying the before-after-predicate:

$$\begin{array}{|l}\hline \textit{Up} \\ \textit{System} \\ \textit{System}' \\ \hline \textit{floor} < \textit{top} \\ \textit{floor}' = \textit{floor} + 1 \\ \textit{door} = \textit{CLOSED} \\ \textit{door} = \textit{door}' \\ \hline \end{array}$$

You may have noticed the last predicate $\textit{door} = \textit{door}'$. It is necessary to specify that variables stay unmodified. This is difference to B or Event-B where variables are modified by explicit substitutions resp. assignments. If the last predicate would have been omitted, the door might be open in the new state.

Another noteworthy difference to B is the role of the first predicate $\textit{floor} < \textit{top}$. In a B specification the predicate $\textit{floor} \leq \textit{top}$ from the schema *Cabin* above would usually be part of the invariant. Thus for each operation that modifies *floor* a proof obligation would be generated to show that $\textit{floor} \leq \textit{top}$ still holds after executing the operation. To proof this, a guard like $\textit{floor} < \textit{top}$ would be necessary to discharge the proof obligation.

The situation in Z is somewhat different: By including *System′*, we have $\textit{floor}' \leq \textit{top}$ as a part of the schema *Up*. With $\textit{floor}' = \textit{floor} + 1$, we can conclude $\textit{floor} + 1 \leq \textit{top}$ which is equivalent to $\textit{floor} < \textit{top}$. Thus the first predicate is actually superfluous and the specified behaviour of the operation would be the same if we remove it from the schema.

**Checking Z specifications**

As we have seen above, the invariants of a Z specification cannot be violated by an operation because the operation's behaviour implies the predicates of the state. The main focus of the work was to enable animation of Z models to provide a way to validate the specification.

But automated checks on the model are also possible. We introduced the option to specify a schema called *Invariant*. Analogously to the B model checking *P*roB checks in every reached state whether the predicate of the invariant holds. Also checks for dead locking states and checks if the system fulfils an LTL formula can be performed.

### 1.3.2 Significance for Verification and Validation

The Z notation has been used to specify aspects of safety critical systems. In the presented article, our case studies were modified real-world examples. By animation we were able to discover flaws in the specification that had not yet been spotted by the company's quality control. This demonstrates that animation alone can actually be a valuable tool to validate formal specifications.

Other means to validate specification like trace checking (test cases for models) and LTL model checking are also applicable to Z models.

### 1.3.3 Affected Parts of *P*roB

We briefly explain which parts of *P*roB are affected by the implementation of the new functionality (see also Figure 1.3 on page 13).

We use the existing parser and type-checker "Fuzz" [Spi00] to read Z specifications. *P*roB reads the output of Fuzz and translates is to a B machine specification. Thus, the boxes "Parser" and "Preprocessing & Typechecking" in Figure 1.3 are completely replaced.

Only minor additions have been made to the interpreter to support some constructs that cannot be easily expressed with existing constructs in B. These are expressions and predicates to handle freetype values, the $\mu$-Operator, **let** predicates and expressions (B has only a LET substitution), and an **if-then-else** expression.

We had added the handling of freetype values to *P*roB's constraint solving kernel. A few functions regarding the handling of sequences and bags effectively have also been added.

### 1.3.4 Relationship to the Other Articles

*Animation of Refinements* In a later addition to the work presented in Chapter 1.4, we implemented support for Event-B theories (see 1.4.4). The freetype support that had been introduced for the *P*roZ extension has been re-used to cope for recursively defined data types in the theories.

*LTL model checking* Since the *P*roZ extension leaves the state-space internals of *P*roB unaffected, LTL model checking can be applied to Z specifications as well. The only restriction at the time of writing is that the LTL parser uses the B parser for atomic propositions such that a user has to specify those properties with B syntax.

*Employing SAT solvers* The translation to Kodkod can also directly be used because the *P*roZ extensions translates a Z specification to *P*roB's internal representation for B. The translation to Kodkod works on this abstract syntax tree.

### 1.3.5 Future Developments

#### Improved User Experience

Currently the tools relays on the fact that the specification follows certain conventions. Giving the user more control over what *P*roB uses as a state definition and what the operations are would be more convenient, because lesser modifications have to be done to a model to make it usable with *P*roB.

Another useful feature would be do let the user specify schemas and *P*roB searches for a concrete valuation of the schema's variables. This would give the user the ability to investigate details of a specification.

#### Compatibility with CZT

Like Rodin for Event-B specifications, CZT [MU05] is an integrated development platform for the Z notation based on the Eclipse platform. Currently *P*roB does not support Z models developed with CZT. It would increase the applicability and usability of *P*roB to implement a CZT plugin to animate Z specifications analogous to the Rodin plugin to animate Event-B specification.

### 1.3.6 Information about the Publication

The article "Validating Z Specifications using the *P*roB Animator and Model Checker" [PL07] was originally published in a conference proceedings of the "Lecture Notes in Computer Science" series, Springer-Verlag.

The article was presented at the conference "iFM 2007 integrated formal methods" which took place in Oxford, UK on July 2–5, 2007. Before acceptance, all papers of the conference went through a peer review process.

For the "Lecture Notes in Computer Science" series is no impact factor available.

The paper's authors are Daniel Plagge and Michael Leuschel.

D. Plagge's contribution to the article are

- the content of the sections 2.2–2.5 and
- the implementation of the presented tool.

M. Leuschel's contribution to the article are

- expert knowledge in *P*roB,
- the handling of symbolic closures in 2.5.1,
- the discussion section 2.6 beside the limitations (section 2.6) of the tool and
- work on all parts of the article to make it more readable and understandable.

## 1.4 Animation of Refinements

To understand and reason about complex systems it is necessary to have an abstract view on a system, its behaviour and its properties. Refinements in B and Event-B offer the ability to start the design of a system with such an abstract view and add details incrementally. For each refinement step it must be proven that the more concrete model meets the behaviour specified by the abstract model.

In the B method this approach is used to start with a specification of a software component and via refinement details are added and abstract concepts are replaced by concrete data structures. Finally a model is reached that can directly be translated into a programming language like C or ADA.

*P*roB has currently only limited support for refinement in B. It can animate refinements but it removes all abstract elements that are not directly part of the current level of abstraction. This makes it harder to understand the relation between abstract and concrete model and under certain conditions, *P*roB reports problems that are just introduced by ignoring the abstract parts or on the contrary does not detect certain kind of errors.

Event-B's refinement mechanism has been simplified in comparison to classical B. For a refined event, several proof obligations are generated. Together, they proof a correct refinement and each of them is relatively easy to understand.

Another aspect that has changed from B to Event-B is the field of application. Classical B is focused on software components where the initial specification already describes the complete set of preconditions under which the system has to work. In this scenario refinements must always be specified for every possible situation that the initial abstract model permits. Event-B, on the other hand, targets the specification of whole systems. It allows that an engineer restricts the behaviour of a system during refinement. E.g. a specification of a railway interlocking system might define how trains can move on the tracks with respect to the rail topology. In a later refinement one might add safety restrictions like two trains are not allowed to share the same track segments.

The article presented in Chapter 3 first gives an overview about the various aspects to show the consistence of an Event-B model. Then it explains how we have extended *P*roB such that multiple levels of refinement can be animated. We show how violations of proof obligations can be detected and presented to the user by the animator.

This addition gives the engineer better understanding of how a system behaves and particularly when applied to refinements it improves *P*roB's ability to support finding flaws in the specification compared to the previous animation of just one level of refinement in isolation.

### 1.4.1 Significance for Verification and Validation

Refinement is a central concept both in B and Event-B. But the concept of refinement has been simplified and is easier to use in Event-B than in classical B. As a result, specifications in Event-B make usually more use of refinement. With many levels of refinement, the animation of a single decoupled refinement level is not very easy to understand. The support to animate all levels together eliminates this deficit and increases the ease of use of refinement as a technique for developing and understanding models significantly.

On the other hand, refinement is a source for several proof obligations. As we have explained before, animation and constrained based techniques can lower the verification effort if a proof obligation cannot be discharged automatically.

### 1.4.2 Affected Parts of *P*roB

The Rodin platform provides already modules to parse and typecheck Event-B specifications. The result of Rodin's parser is used instead of *P*roB's B parser but a translation is applied that returns a syntax tree of the specification that is compatible with *P*roB's existing parser. Thus *P*roB's existing type checker is used to transform the output of Rodin's parser into *P*roB's internal format. The implementation of the preprocessing is replaced by a module specific to Event-B. In particular, instead re-using the existing representation for B operations the module generates a representation for events that incorporates Event-B specific behaviour for all levels of refinements.

The B interpreter was accordingly extended by a rule to handle events. This is the location where the algorithm described in the article is implemented.

Most other parts of *P*roB had been not affected by the extension because the internal representation is very similar to the representation of classical B machines.

### 1.4.3 Relationship to the Other Articles

*Animating Z* Similar to *P*roZ, the work about the multilevel refinement animation for Event-B makes *P*roB applicable for more methods of specification.

*LTL model checking* LTL model checking can also be applied to Event-B models with several refinement levels. The LTL parser had been parametrised to use the Event-B parser for atomic propositions and propositions on transitions. Thus, the LTL feature is well integrated with Event-B.

The visualisation of LTL counter-examples [Tol11] that has been added later has been implemented in the *P*roB Rodin plugin and is so far specific to Event-B models.

In the next paragraph (Section 1.4.4) we propose some additions to the LTL extension that are specific to refinements. At the time of writing no support in that respect has been implemented.

*Employing SAT solvers* The Kodkod translation can be equally applied to Event-B models as to classical B models. Additionally, the disprover plugin that is specific to Event-B specifications can make of Kodkod.

### 1.4.4 Further Developments

**Refinement Animation for Classical B**

At the time of writing, *P*roB still does not support animation of more than one level of refinement at once for classical B. Classical B's refinement concept is more complex than Event-B's because the situation in which an event might occur can always further constrained in a refinement step. In contrast, an operation must always behave as expected when it is called as long its precondition is fulfilled.

The algorithm shown in the article is bottom-up in the sense that it starts with the most concrete refinement level of an event and then continues with each more abstract level and tries to find a matching valuation .

A possible approach for classical B is to start with the most abstract specification of the operation and find possible values for its arguments that matches the precondition. After that we can evaluate the most concrete refinement level analogously to the existing method for Event-B with the additional constraint that there must always be at least be one solution if the precondition holds.

An additional obstacle is the lack of witnesses in classical B. Witnesses constitute the link from found values in a concrete event to its abstract specification. This has to be evaluated in more detail.

**Constraint-based Checks for Refinement**

*P*roB's constraint based core has been constantly improved over the last years. That made constraint based checks more and more effective. Constraint based checks for conditions that must hold in refinement (like guard strengthening) could be added to *P*roB.

**User Interface**

When *P*roB encounters an error in the specification of an event like a violation of guard strengthening, it presents the user a description of how the error can be found. The description contains what values have been chosen for parameters and how guards, witnesses and assignments are evaluated when following the chain of refinements from the most concrete event up to the abstract event where the error occurs.

One problem with these description is that they are usually hard to understand by a user. An approach to ease the understanding of a found error is the use of an interactive "worksheet" where textual descriptions and Event-B expressions and predicates are inserted that can be analysed by the user.

Such worksheets provide a much broader range of application than just the presentation of errors in the refinement chain. They would also provide a powerful tool to help a user in analysing, understanding and communicating various aspects of a formal specification. They subsume the functionality of a command-line console like it is already present in the Tcl/TK interface of *P*roB with the difference that several forms of presenting data could be mixed in one document. As an example, a relation could be shown as a graph which will be updated if the other parts of the worksheet are changed.

A prototype of such a "worksheet" for *P*roB has been implemented by René Goebbels as part of his bachelor thesis [Goe13].

#### Theory Support

A recent extension to Event-B are theories [BM13]. They allow the definition of new data types and operators using various approaches like direct, recursive or axiomatic definitions. *P*roB does now also – with some restrictions – support the use of theories in a model [PL14].

### 1.4.5 Information about the Publication

The article "Validation of Formal Models by Refinement Animation" [HLP13] was originally published in the journal "Science of Computer Programming", Elsevier. Before being published, articles in "Science of Computer Programming" run through a peer review process. The impact factor of "Science of Computer Programming" in 2011 due to the publisher Elsevier's data is 0.622 [Els11]. The SCImage Journal Rank (SJR) for 2011 is 0.792 [SCI15b].

The article is based on a previously published paper "Refinement-Animation for Event-B - Towards a Method of Validation" [HLP10] which was presented at the conference "Abstract State Machines, Alloy, B and Z" which took place in Orford, Canada, on February 22–25, 2010. Before acceptance, all papers of the conference went through a peer review process.

D. Plagge's contribution to the article are

- the design and description of the algorithm and its implementation in Section 3.3 and
- how specific problems that can occur in Event-B models are detected and displayed to the user as explained in Section 3.4.

S. Hallerstede and M. Leuschel's contributions to the article are

- the introduction and the in-depth description of Event-B's proof obligations in Sections 3.1 and 3.2,

- the application to case studies and the evaluation in Sections 3.4.2 and 3.5 and
- the related work and conclusion in Section 3.6 and 3.7.

## 1.5 Temporal Model-Checking

Requirements often involve propositions about the expected temporal behaviour of a system. An example for such a proposition is "if an event $E$ happens, later in the future an event $F$ must happen, too". A characteristic of a temporal requirement is that not only a state in isolation is considered unlike e.g. an invariant does.

Linear temporal logic (LTL) allows to express propositions about a possible infinite long sequence of alternating states and events. A system does satisfy a specification given as an LTL formula if every possible sequence of events and states that can happen starting from the initialisation of the system fulfils the formula.

The article in Chapter 4 describes how we have adapted an existing algorithm to make it suitable for models where deadlocks can occur and where we can express propositions on the events between two states.

A deadlock is a state of the system where no further events can occur. The handling of deadlocks is an important feature because for the formalism we consider there is no guarantee that a model does not have such states.

Propositions on events are a very useful feature because requirements contain often references to events that happen in a system, not just on the states itself. In general, it is not possible to formalise properties that describe the occurrence of an event in a sequence as a predicate on a single state.

We have also developed a concrete syntax that basically allows a user to enter LTL formulas for any formalism supported by *P*roB and we implemented support for LTL as an extension to *P*roB. Section 1.5.2 gives an overview about some of its details.

### Example

Again, we take a lift as an example for a specified system. Figure 1.4 shows the state space for a lift that connects two floors. It is located either in the upper or lower floor, its doors are closed (a cross in the cabin) or open and we store outstanding call request for each floor (the two circles on the left). The data types of the system allow $2^4$ possible states but 4 states where the lift is open and the current floor has an open call request are not reachable. So we have 12 reachable states remaining. The initial state (marked with a "start") is where the lift is in the lower floor, the door is open and there are no outstanding call requests.

Now, we want to check if the model fulfils the following requirement:

> Whenever the call button for the first floor is pressed, the lift will eventually be on the first floor with the door open.

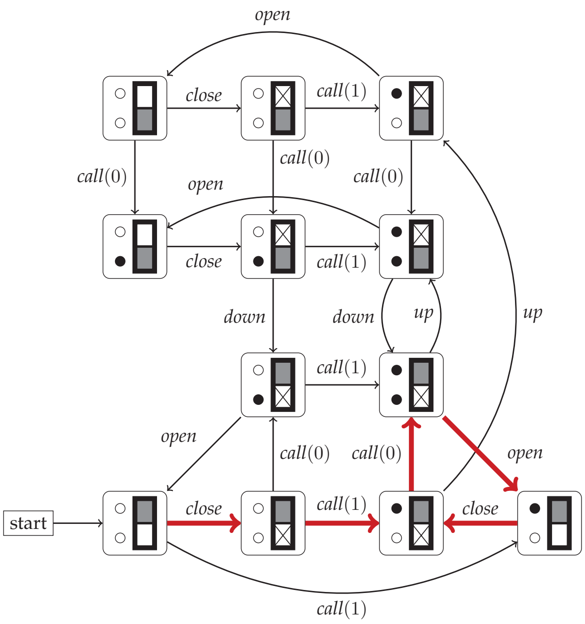


Figure 1.4: The state space of a simple elevator, each icon encodes outstanding call requests (the circles), the position of the lift and whether the door is open or closed

In LTL, "whenever" can be expressed with the *globally* ($G$) operator and "eventually" with the *finally* ($F$) operator.

$$G(\,[call(1)] \Rightarrow F\,\{floor = 1 \wedge door = OPEN\,\}\,)$$

The model checker can actually find a counter-example (which is marked with the bold red arrows in Figure 1.4):

1. Begin with the open lift on the lower floor, no call request are outstanding.
2. The door closes (*close*).
3. Somebody presses the button on the upper floor ($call(1)$), a call request is stored.
4. Somebody presses the button on the lower floor ($call(0)$), a call request is stored.
5. The door opens and the call request for the lower floor is discharged (*open*).
6. The door closes (*close*).
7. The state of the system is now the same as after the first $call(1)$ event, the counter-example continues with step 3 and goes into an infinite loop.

So what our lift actually does in the counter-example is to open and close the doors infinitely often because somebody presses the call button before it moves to another floor.

### 1.5.1 Significance for Verification and Validation

If a requirement like in the example above can be formalised with LTL formulas, we have gained the ability to check if a specification meet these requirement. Thus we have an additional aspect of validating the model.

Formalisms like B, Event-B and Z just provide very limited support for temporal properties. Invariants can be used to express properties that must hold in every state of a system. Liveness properties, i.e. the condition that something eventually happens, cannot be expressed easily. One exception is the concept of convergent events in Event-B, which can be used to show termination, i.e that some events are only finitely often enabled.

### 1.5.2 Affected Parts of *P*roB

The LTL model checker is an alternative implementation of the model checker as shown in Figure 1.3 on page 13. Whereas *P*roB's default model checker uses a mixed breadth-first and depth-first search [LB08], the LTL model checker uses a variant of a depth-first search.

The LTL code makes use of the state-space module to call functions that trigger the computation of enabled events in a state. It also uses the B interpreter to evaluate atomic propositions in a state or propositions on transitions.

There is an independent parser for the LTL formulas. The parser can be parametrized such that another parser will be called for atomic propositions (the part of a formula that is located between curly brackets) and propositions on transitions (between square brackets).

### 1.5.3 Further Developments

**Visualisation of Counter-examples**

In case a system does not fulfil an LTL formula, the model checker provides a counter-example to the user. A counter-example consists of a sequence of alternating states and events. Without a special presentation, the sequence is loaded in *P*roB's history as if the user has just initiated an animation by choosing the events himself. In case of the found counter-example for the lift, we would get the sequence

$$\mathit{close} \rightarrow \mathit{call}(1) \rightarrow \mathit{call}(0) \rightarrow \mathit{open} \rightarrow \mathit{close}$$

in the animator's history without any further hint why the formula is false. For more complex systems it can be very difficult for a user to see why the counter-example does actually not fulfil the given formula.

The counter-example has a "lasso form", where the first part is just $\mathit{close} \rightarrow \mathit{call}(1)$ and the looping part is $\mathit{call}(0) \rightarrow \mathit{open} \rightarrow \mathit{close}$.

Figure 1.5 shows a possible visualisation of the counter-example. The upper rectangle shows why the globally operator $G$ evaluates to false. The rectangle contains all states (the small boxes <a>,...,<e>) that could possibly have an impact to the result of the evaluation, in this case these are all states. The colour of each box indicates whether the argument of the operator (here the implication $[call(1)] \Rightarrow F\,\{floor = 1 \wedge door = OPEN\,\}$) evaluates to true (green) or false (red).

States that have a direct impact on the result of the operator are highlighted, e.g. the fact that the implication evaluates to false after the first *close* event (box <b>) makes *globally* evaluates to false. On the other hand, the first green box <a> is not highlighted because even if it would have been red the outcome of *globally* would have been the same.

The bold arrow that points to <b> indicates that the second rectangle explains why it evaluates to false. The user of the tool can click on the boxes to decide which boxes should be explained.

The implication is limited to just one state, unlike $G$ or $F$ has it no reference to future states. It evaluates to false because the antecedent (box <g>) is true and because the $call(1)$ event is the next event and the consequence (box <l>) is false.

Again, the outcome of box <l> is explained by another rectangle in the bottom row. It is false because the atomic proposition $floor = 1 \wedge door = OPEN$ does not hold in any of the future states (boxes <q>,. . . ,<t>).

Another issue in the visualisation are situations where the result of an operator in a certain state is unknown because the model-checker has truncated the counter-example when a prefix of the sequence is sufficient to show that the formula is false in the initial state.

Andriy Tolstoy implemented in his master's thesis [Tol11] an extension to *P*roB's Eclipse-based user interface that visualises the evaluation of an LTL formula on a given sequence.

### Fairness

When requirements are expressed as LTL formulas, often so-called fairness conditions arise. Again, we illustrate a fairness condition by the small lift example. The counter-example that has been presented above may not be very convincing for real-life applications: In the loop, the lift could move the first level (the event *up*) infinitely often but always somebody presses the button before that and the door opens again.

To prevent the model-checker to find such counter-examples, we can add the assumption that whenever the event *up* is enabled infinitely often, it must finally happen. This can be done directly in LTL:

$$(GFe(up) \Rightarrow GF[up]) \Rightarrow G(\,[call(1)] \Rightarrow F\,\{floor = 1 \wedge door = OPEN\,\}\,)$$

Such a constraint is called "strong fairness" [LP85]. A weak fairness condition would state that *up* must finally happen if it is *continuously* enabled:

$$(FGe(up) \Rightarrow GF[up]) \Rightarrow G(\,[call(1)] \Rightarrow F\,\{floor = 1 \wedge door = OPEN\,\}\,)$$

The computational complexity to check the formula doubles with each temporal operator in the formula, thus the complexity of the fair formula is $2^4 = 16$ times larger

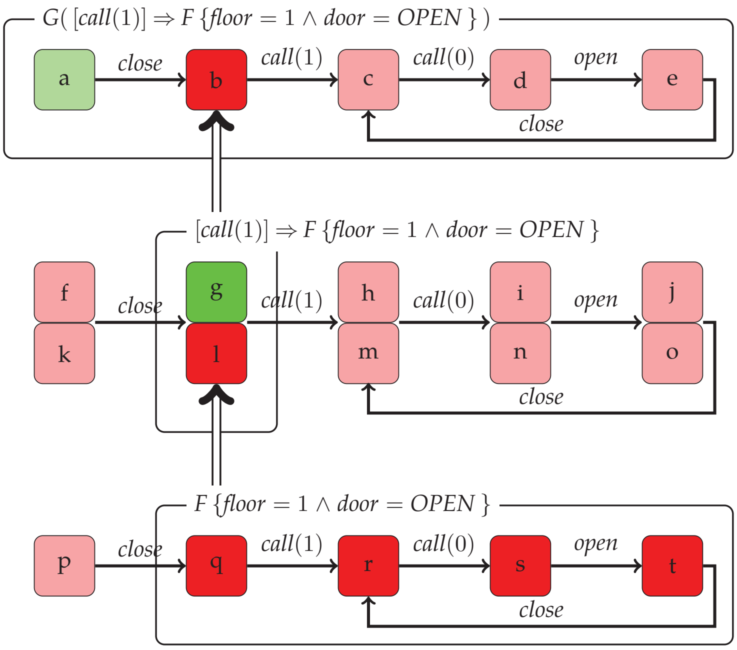

G( [call(1)] ⇒ F {floor = 1 ∧ door = OPEN } )
a
close
b
call(1)
c
call(0)
d
open
e
close
[call(1)] ⇒ F {floor = 1 ∧ door = OPEN }
f
k
close
g
l
call(1)
h
m
call(0)
i
n
open
j
o
close
F {floor = 1 ∧ door = OPEN }
p
close
q
call(1)
r
call(0)
s
open
t
close

Figure 1.5: Schema of a counter-example visualisation, the letters in the boxes are only used to refer to them more easily in the explanation

than the for the original formula. We (Ivaylo Miroslavov Dobrikov with support by Daniel Plagge) have implemented another approach based on the algorithm in [LP85] to check fairness conditions. With explicit support for fairness, the formula above could be expressed as:

$$SF(move) \Rightarrow G(\, [call \,|\, level = 1] \Rightarrow F\{lift\_level = 1\}\,)$$

The operator *SF* denotes strong fairness and can only be used on the left side of an implication. Accordingly there is an operator *WF* for weak fairness.

With direct support for fairness the model-checker searches for counter-examples of the original formula and then checks if the counter-examples is actually fair. If not, it continues the search. This approach has only a small computational overhead to the check of the original formula.

### Fairness: Future Work

In fairness conditions as described above, we look at enabled events and events that actually happen. The question arises when events are regarded equal. Consider that we want to specify a fairness condition regarding the *call* event in the lift system. If *call*(1) is enabled infinitely often in a counter-example, would the event *call*(0) be sufficient to fulfil the fairness condition (in that case the specifier is not interested in the event's parameter) or must it be *call*(1)?

A possible solution for such scenarios is to introduce the generation of a normalised form for each event based on a specification by the user. For example, if the floor parameter is significant, a normalisation of *call*$(1)$ would keep it unmodified. If the parameter can be ignored, the normal form would be *call*. The algorithm to check the fairness property of counter-examples would then work as before but uses the normalised form to compare events.

At the time of writing the implementation had not been finished yet and it was still under discussion how the specification can be integrated in the LTL syntax in a user friendly way.

### 1.5.4 Information about the Publication

A first version of the article "Seven at one stroke: LTL model checking for High-level Specifications in B, Z, CSP, and more" has been published in the proceedings [LP07] of the conference ISoLA (International Symposium On Leveraging Applications of Formal Methods, Verification and Validation). An extended version has been published later in the journal Software Tools for Technology Transfer [PL10]. To the author's knowledge, no impact factor for the journal has been published. The SCImage Journal Rank (SJR) for 2011 is 0.841 [SCI15a].

D. Plagge's contributions to the article are

- The extension of the existing formalisms for LTL$^{[e]}$ in Section 4.3,
- the development and implementation of the algorithm that supports deadlocks and propositions on transitions in Section 4.4 and
- the support for LTL operators that regard the past in Section 4.5

M. Leuschel's contribution are

- the introduction, motivation and conclusion in the Sections 4.1 and 4.8.3,
- the examples and experiments in Section 4.6 and
- the application of symmetry reduction in Section 4.7.

## 1.6 Employing SAT-Solvers

Another formal notation that has common ground with Z is Alloy [Jac02, Jac12]. A design decision in the development of Alloy was to have the expressive power that provides Z with it's mathematical toolkit of sets and relations to model software on one hand and to restrict that expressiveness in a way that it can be easier supported by tools.

As a result, Alloy makes it possible to analysis models fully automatically for given maximum cardinality of carrier sets. The major restriction that was made regarding the expressiveness is that Alloy provides only relations and sets of basic types. It does not support higher-order types like set of sets.

To find solutions to the specified problems, Alloy translates the model to a Boolean satisfiability problem (SAT) and uses existing SAT solvers to search for valid variable assignments. The development of SAT solvers have been highly successful and they have been applied to various domains like analysis of hardware design. Edmund Clark described the rapid development in his Turing Award paper [CES09]:

> Nevertheless, the increase in power of modern SAT solvers over the past 15 years on problems that occur in practice has been phenomenal. It has become the key enabling technology in applications of Model Checking to both computer hardware and software.

*P*roB uses a completely different technology — constraint solving with Prolog — to search for solutions. The basic question that lead to the article presented in Chapter 5 is if and how can we make use of this development. Since Alloy provides already data structures and logic that are very similar to Z and B's logic, it is very convenient to translate problems specified in B to Alloy and re-use the translation from predicates and relations to Boolean formulas. Alloy's core functionality that we are interested in (i.e. specifying carrier sets, variables and a predicate to solve) has been moved in a separate programming library, called Kodkod [TJ07].

The main problem in such a translation is the limited expressiveness of Alloy resp. Kodkod. In a B specification, we encounter data types and expressions that cannot be directly translated. Here we have chosen a very pragmatic approach: A given specification will be analysed if it contains parts that can be translated. If there are such translatable parts, the B interpreter calls an external process that interfaces to the Kodkod library. *P*roB sends already known values to the external process and it receives possible values to the remaining variables that fulfil the translated specification.

The result is that Kodkod and with it SAT solvers will be applied if the structure of a given problem allows it. If not, *P*roB uses it existing constraint solving techniques. Thus the application of Kodkod does not result in limitations of language features that a user can apply in his specification. But in case it is applicable, it can bring a performance gain to the analysis which is very specific to the specified problems.

Our approach is not the first one that tries to make use of Alloy for specifications in Z or similar formal notations [MGL10, MMS08]. The main advantage of our approach is that it is embedded in an existing tool that provides a "fall back" technique that can always be applied if the specification contains not supported features. This can make the difference between a tool that is only usable for a few scenarios or a tool that is a standard part of the formal development process.

The paper provides more details on how the translation is done, how the interpreter interacts with the Kodkod library and how some specific problems (like numbers) are handled.

### 1.6.1 Significance for Verification and Validation

Unlike the other articles presented in this work, the translation to Kodkod is barely visible to the user of *P*roB. The only direct interaction with the new functionality is a preference where one can choose if the Kodkod translation should be turned on and off.

Nevertheless, the translation improves *P*roB's abilities to support verification and validation by being applicable to specifications where it failed before. Also it improves *P*roB's performance in some cases and thereby it improves the user experience with the tool.

### 1.6.2 Affected Parts of *P*roB

The translation to Kodkod is applied to a part of the specification's abstract syntax tree. If the translation succeeds, the part will be replaced by a AST node that describes the Kodkod representation of the problem. This part happens either in the preprocessing phase after the type checking or whenever other features like constrained based checks or the disprover construct an AST to describe the properties of a supposed counter-example.

The B interpreter is extended with the handling of such a predicate. If the interpreter encounters a Kodkod translation it looks up known values in the store and sends the translation and the values to the external Kodkod component. The answer of the external component consists of possible values for previously unknown variables. They are then written back to the store.

### 1.6.3 Relationship to the Other Articles

*Animating Z* The article's work can also directly applied to Z specifications because Z is translated to the same internal representation in *P*roB than that is used for B specifications.

*Animation of Refinements* Analogously, this translation can be applied to Event-B specifications. But the algorithm presented for the validation of multiple refinement levels is not directly affected. Future work might be to develop constraint based variants of the algorithm such that a predicate is set up that describes the various errors (e.g. a violation of the guard strengthening proof obligation). Such predicates can then benefit from an application to the Kodkod translation.

*LTL model checking* The LTL extension and the translation to Kodkod approach independent aspects of the validation and verification of formal models.

### 1.6.4 Future Developments

As pointed out in the article, the translation was only applied to properties (resp. axioms in Event-B) when trying to find matching values for the constants. At the time of writing the translation had also be applied to other constraint based search like dead-lock checking, assertion checking or the disprover.

### 1.6.5 Information about the Publication

The article "Validating B, Z and TLA$^+$ using *P*roB and Kodkod" [PL12] was originally published in a conference proceedings of the "Lecture Notes in Computer Science" series, Springer-Verlag.

The work was presented at the conference "FM 2012: 18th International Symposium on Formal Methods" which took place in Paris, France on August 27–31, 2012. Before acceptance, all papers of the conference went through a peer review process. The "Formal Methods" series of symposia is a distinguished conference in the field of formal methods.

For the "Lecture Notes in Computer Science" series is no impact factor available.

The paper's authors are Daniel Plagge and Michael Leuschel.

D. Plagge's contribution to the article are

- the comparison of B, Z and TLA$^+$ on one side with Kodkod on the other,
- the architecture and details of the translation and
- the execution of the experiments.

M. Leuschel's contribution to the article are

- the introduction,
- placing the experiments' outcome in relation to other techniques,
- exploring related work and
- overall advice and help in both the described research and the article itself.

# 2 Validating Z Specifications using the ProB Animator and Model Checker

Daniel Plagge, Michael Leuschel

**Abstract.** We present the architecture and implementation of the *P*roZ tool to validate high-level Z specifications. The tool was integrated into *P*roB, by providing a translation of Z into B and by extending the kernel of *P*roB to accommodate some new syntax and data types. We describe the challenge of going from the tool friendly formalism B to the more specification-oriented formalism Z, and show how many Z specifications can be systematically translated into B. We describe the extensions, such as record types and free types, that had to be added to the kernel to support a large subset of Z. As a side-effect, we provide a way to animate and model check records in *P*roB. By incorporating *P*roZ into *P*roB, we have inherited many of the recent extensions developed for B, such as the integration with CSP or the animation of recursive functions. Finally, we present a successful industrial application, which makes use of this fact, and where *P*roZ was able to discover several errors in Z specifications containing higher-order recursive functions.

## 2.1 Introduction

Both B [Abr96] and Z [ASM80, Spi92] are formal mathematical specification notations, using the same underlying set theory and predicate calculus. Both formalisms are used in industry in a range of critical domains.

The Z notation places the emphasis on human-readability of specifications. Z specifications are often documents where ambiguities in the description of the system are avoided by supporting the prose with formal statements in Z. LaTeX packages such as *f*uzz [Spi00] exists to support type setting and checking those documents. The formal part of a specification mainly consists of schemas which describe different aspects of a system using set theory and predicate logic. The schema calculus—a distinct feature of Z—enables system engineers to specify complex systems by combining those schemas.

B was derived from Z by Jean-Raymond Abrial (also the progenitor of Z) with the aim of enabling tool support. In the process, some aspects of Z were removed and replaced, while new features were added (notably the ASCII Abstract Machine Notation). We will discuss some of the differences later in depth. In a nutshell, B is more aimed at refinement and code generation, while Z is a more high-level formalism aimed for specification. This is, arguably, why B has industrial strength tools, such as Atelier-B [Ste09] and the B-toolkit [B-C02]. Recently the *P*roB model checker [LB03] and refinement checker [LB05] have been added to B's list of tools. Similar tools are lacking for Z, even though there are recent efforts to provide better tool support for Z [MU05].

In this paper we describe the challenge of developing a Z version of *P*roB, capable of animating and model checking realistic Z specifications. We believe an animator and model checker is a very important ingredient for formal methods; especially if we do not formally derive code from the specification (as is common in Z [Hal02]). This fact is also increasingly being realised by industrial users of formal methods.

At the heart of our approach lies a translation of Z specifications into B, with the aim of providing an integrated tool that is capable to validate both Z and B specifications, as well as inheriting from recent refinements developed for B (such as the integration with CSP [BL05]). One motivation for our work comes from an industrial example, which we also describe in the paper.

## 2.2 Specifications in Z

First we give a brief introduction to the Z notation. We want to describe the structure of Z specifications, especially how this differs from specifications in B as supported by *P*roB. The interested reader can find a tutorial introduction to Z inside the Z reference manual [Spi92]. A more comprehensive introduction with many examples is [Jac97].

### 2.2.1 A brief description of Z

Usually, a specification in Z consists of informal prose together with formal statements. In a real-life applications, the prose part is at least as important as the formal part, as a specification has to be read by humans as well as computers.

Usually, one describes state machines in Z, i.e., one defines possible states as well as operations that can change the state. The Z syntax can be split into two: a notation for discrete mathematics (set theory and predicate calculus) and a notation for describing and combining *schemas*, called the schema calculus.

For illustration, we use the simple database of birthdays (Fig. 2.1) from [Spi92]. The first line in the example is a declaration $[NAME, DATE]$ which simply introduces *NAME* and *DATE* as new basic types, without providing more information about their attributes (like generics in some programming languages). We can also see three boxes, each with a name on the upper border and a horizontal line dividing it into two parts. These boxes define the so-called schemas. Above the dividing line is the declaration part, where variables and their types are introduced, and below a list of predicates can be stated.

Without additional description, the purpose of each schema in the example is not directly apparent. We use the first schema *BirthdayBook* to define the state space of our system. *Init* defines a valid initial state and the schema *AddBirthday* is the description of an operation that inserts a new name and birthday into the database.

$$[\mathit{NAME}, \mathit{DATE}]$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{BirthdayBook}} \\
\hline
\mathit{known} : \mathbb{P}\,\mathit{NAME} \\
\mathit{birthday} : \mathit{NAME} \pfun \mathit{DATE} \\
\hline
\mathit{known} = \operatorname{dom} \mathit{birthday} \\
\hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{Init}} \\
\hline
\mathit{BirthdayBook} \\
\hline
\mathit{known} = \varnothing \\
\hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{AddBirthday}} \\
\hline
\Delta \mathit{BirthdayBook} \\
\mathit{name}? : \mathit{NAME} \\
\mathit{date}? : \mathit{DATE} \\
\hline
\mathit{name}? \notin \mathit{known} \\
\mathit{birthday}' = \mathit{birthday} \cup \\
\qquad \{\mathit{name}? \mapsto \mathit{date}?\} \\
\hline
\end{array}$$

Figure 2.1: The birthday book example

We describe the schemas in more detail. In *BirthdayBook* we have two variables: *known* is a set of names and *birthday* is a partial function that maps names to a date. The predicate states that *known* is the domain of the partial function, i.e., the set of names that have an entry in the function. A possible state of our system consists of values for these two variables which satisfy the predicate.

The declaration part of the *Init* schema contains a reference to the schema *BirthdayBook*. This imports all *BirthdayBook*'s variable declarations and predicates into *Init*. The predicate says that *known* is empty. Together with the predicate of *BirthdayBook* this implicitly states that the domain of *birthday* is empty, resulting in an empty function.

The schema defining the operation *AddBirthday* contains two variables with an appended question mark. By convention, variables with a trailing ? (resp. !) describe inputs (resp. outputs) of operations, thus *name*? and *date*? are inputs to the operation. The first line of the schema is $\Delta$*BirthdayBook*. This includes all declarations and predicates of *BirthdayBook*, as previously seen in *Init*. Additionally the variable declarations are included with a prime appended to their name, representing the state after the execution of the operation. The predicates are also included a second time where all occurring variables have a prime appended. To clarify this, we show the *expanded* schema:

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{AddBirthday}} \\
\hline
\mathit{known}, \mathit{known}' : \mathbb{P}\,\mathit{NAME} \\
\mathit{birthday}, \mathit{birthday}' : \mathit{NAME} \pfun \mathit{DATE} \\
\mathit{name}? : \mathit{NAME} \\
\mathit{date}? : \mathit{DATE} \\
\hline
\mathit{known} = \operatorname{dom} \mathit{birthday} \land \mathit{known}' = \operatorname{dom} \mathit{birthday}' \\
\mathit{name}? \notin \mathit{known} \\
\mathit{birthday}' = \mathit{birthday} \cup \{\mathit{name}? \mapsto \mathit{date}?\} \\
\hline
\end{array}$$

The schema thus defines the relation between the state before and after executing the operation *AddBirthday*. Accordingly the unprimed variables refer to the state before

```
MACHINE BirthdayBook
SETS NAME;DATE
VARIABLES known,birthday
INVARIANT
 known:POW(NAME) & birthday:NAME+->DATE & known=dom(birthday)
INITIALISATION known,birthday := {},{}
OPERATIONS
  AddBirthday(name,date) = PRE name:NAME & date:DATE & name/:known THEN
    birthday(name) := date || known := known \/ {name}
  END
END
```

Figure 2.2: The birthday book example in B

and the primed ones to the state after the execution. The effect of *AddBirthday* is that the function *birthday* has been extended with a new entry. But, together with the predicates from *BirthdayBook*, it is (again implicitly) stated that *name*? should be added to *known*.

Instead of the schema boxes there is also a shorter equivalent syntax. E.g., *Init* can also be defined with $Init \mathrel{\widehat{=}} [\, BirthdayBook \mid known = \varnothing \,]$. In addition to inclusion, as seen in the example, the schema calculus of Z provides more operators to combine schemas. E.g., the conjunction of two schemas $R \mathrel{\widehat{=}} S1 \wedge S2$ merges their declaration part in a way that the resulting schema $R$ has the variables of both schemas $S1$ and $S2$, and its predicate is the logical conjunction of both original predicates. The schema calculus is a very important aspect of the Z notation, because it makes Z suitable for describing large systems by handling distinct parts of it and combining them.

### 2.2.2 Some differences between Z and B

*P*roZ is an extension of *P*roB, a tool that animates specifications in B. To make use of its core functionality, we need to translate a Z specification into *P*roB's internal representation of a B machine. To illustrate the fundamental issues and problems, we describe some of the major differences between Z and B using our example.

Figure 2.2 shows the birthday book example as a B machine. Aside from the ASCII notation, one difference is the use of keywords to divide the specification into multiple sections. The `VARIABLES` section defines that `known` and `birthday` are the variables making up the state. There is an explicit initialisation and in the `OPERATIONS` section the operation `AddBirthday` is described. In a Z specification, on the other hand, the purpose of each schema must be explained in the surrounding prose.

If we look closer at the `INITIALISATION` section in the example, we see that both `known` and `birthday` are set to $\varnothing$. This is unlike the Z schema *Init* in Fig. 2.2, where only $known = \varnothing$ is stated and the value of *birthday* is implicitly defined. Also in the definition of the operation `AddBirthday` both variables are changed explicitly. Generally in B all changes to variables must be stated explicitly via generalised substitutions. All other variables are not changed, whereas in Z every variable can change, as long its values satisfy the predicates of the operation.

Another noteworthy difference is the declaration of an invariant in the B machine. An invariant in B is a constraint that must hold in every state. To prove that a machine is consistent it has to be proven that the initialisation is valid and that no operation leads to an invalid state if applied to a valid state. In Z the predicate of the state's schema is also called invariant, but unlike B the operations implicitly satisfy it by including the state's schema. Errors in a B specification can lead to a violation of the invariant. A similar error in Z leads to an operation not being enabled, which in turn can lead to deadlocks.

### 2.2.3 Translating Z to B

The notation of substitutions often results in specifications that are easier to animate than higher-level Z specifications. Hence, at the heart of *P*roZ is a systematic translation of Z schemas into B machines.

Figure 2.3 contains such a B translation of the birthday book Z specification, as computed by our tool (to make the specification more readable we use Z style identifiers, i.e., ending with $'$, ? or !, even though strictly speaking this is not valid B syntax). As can bee seen, we have identified that the variables *birthday* and *known* form part of the state, their types are declared in the invariant. The initialisation part is a translation of the expanded *Init* schema. One operation *AddBirthday* with two arguments *date*? and *name*? has been identified, a translation of the expanded *AddBirthday* schema can be found in the WHERE clause of its ANY statement. There are also several references to a constant *maxentries*. We added it and a constraint $\# known \leq maxentries$ to demonstrate the handling of axiomatic definitions (cf. Section 2.3.1).

The B machine from Figure 2.3 can be fed directly into *P*roB, for animation and model checking. However, Z has also two data types, free types and schema types, that have no counterpart in B. This means that some aspects of Z cannot be effectively translated into B machines, and require extensions of *P*roB. In the next section we present the overall architecture of our approach, as well as a formal explanation of how to derive a B model from a Z specification.

## 2.3 Architecture and the *P*roZ compiler

In the previous section we have examined the basic ingredients of Z specifications, and have highlighted why Z specifications are inherently more difficult to animate and model check than B specifications. In this and the next section we explain how we have overcome those issues; in particular:

- How to analyse the various schemas of Z specification, identifying the state of a Z specification, the state-changing operations and the basic user-defined data types (cf. Section 2.3.1).

- How to deal with the fact that Z specifications do not specify all changes to variables explicitly.

- How to deal with the new data types provided by Z.

```
MACHINE z_translation
SETS NAME;DATE
CONSTANTS maxentries
PROPERTIES
    (maxentries:INTEGER) & (maxentries>=5)
VARIABLES birthday, known
INVARIANT
    (birthday:POW(NAME*DATE)) & (known:POW(NAME))
INITIALISATION
  ANY birthday', known'
    WHERE
        (known':POW(NAME)) & (birthday':(NAME+->DATE))
      & (known'=dom(birthday')) & (card(known')<=maxentries)
      & (known'={})
    THEN
      birthday, known := birthday', known'
  END
OPERATIONS
  AddBirthday(date?, name?) =
    PRE (name?:NAME)
      & (date?:DATE)
      THEN
        ANY birthday', known'
          WHERE
              (known:POW(NAME)) & (birthday:(NAME+->DATE))
            & (known=dom(birthday)) & (card(known)<=maxentries)
            & (known':POW(NAME)) & (birthday':(NAME+->DATE))
            & (known'=dom(birthday')) & (card(known')<=maxentries)
            & (name?/:known) & (birthday'=(birthday\/{(name?,date?)}))
          THEN
            birthday, known := birthday', known'
        END
    END
END
```

Figure 2.3: The translated birthday book example

- How to deal with new operators and constructs.

**Overall Architecture** *P*roZ is an extension of *P*roB that supports Z specifications which can be parsed by the *f*uzz typechecker. Those specifications are given as a LaTeX file. When the user loads a specification into *P*roZ, the following steps are performed (see also Figure 2.4):

1. The specification is typechecked with *f*uzz. *f*uzz writes the formal content of the specification into a file which then is parsed by *P*roZ.

2. The different components of the specification (definition of constants, state, initialisation and operations) are identified.

3. All schemas are expanded and normalised, i.e., all schema inclusions are resolved and the type declarations of variables are strictly separated from constraints on

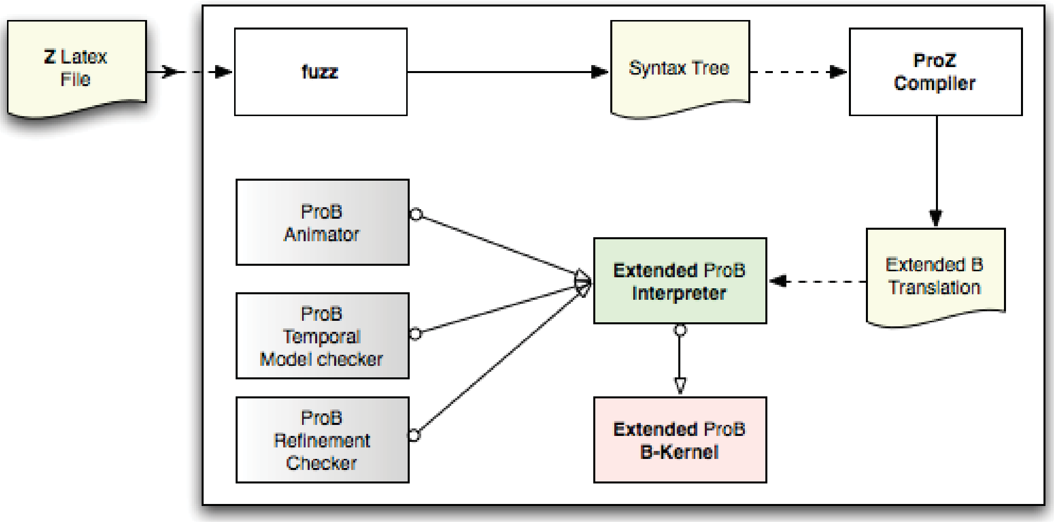


Figure 2.4: Overview of *ProZ* Architecture

their values.

4. *ProZ* then translates the specification to an internal representation of a B machine (with some small extensions, which are discussed later in the paper).

5. After the translation process *ProB* treats the specification the same way as other B machines are treated (with some extensions having been added to the *ProB* kernel).

Most of the expressions in Z have a direct counterpart in B, for those the translation in point 4 is just a conversion from one syntax into another. Some cases where there is more logic need in the translation process or where we extended the *ProB* interpreter are presented in Section 2.3.3. The support of two Z data types as discussed in the next section affects the translation process and requires extensions to the kernel as well.

### 2.3.1 Identifying components of the specification

As we have seen in the previous sections, the purpose of the schemas in a specification is not stated formally. But to interpret a given specification for animation and model checking, we must identify which schemas describe the state space, the initialisation and the operations. To be able to do so, we require that the specification satisfies some rules:

- There must be a schema called *Init* for initialisation.
- *Init* includes exactly one other schema. The included schema will be taken as the description of the state space.

- A schema with all variables of the state and their primed versions, that is not included by any other schema, will be used as an operation.

The rules can be applied to the birthday book example in Fig. 2.1: There is a schema *Init* which includes *BirthdayBook*. Thus *Init* is the initialisation of the state which consists of *BirthdayBook*'s variables *known* and *birthday*. Expanding *AddBirthday* shows that it has all variables of the state and also the primed versions *known*′ and *birthday*′. It is not included by any other schema. Thus *P*roZ would identify *AddBirthday* as an operation.

In the next two paragraphs we present two other components of a specification that are used by *P*roZ and explain how they relate to existing features of *P*roB.

**Invariant** As seen in the comparison in Sect. 2.2.2, the B invariant has no direct counterpart in Z. But it can be useful to search for states that violate a certain property by model checking. To make this feature available for Z specifications, *P*roZ looks for a schema named *Invariant*. If such an invariant is given, its predicate is checked for every visited state in an animation or in model checking. The predicate is then used analogously to the invariant in B.

**Axiomatic definitions** In our short introduction to Z we did not describe how global constants can be introduced in Z by *axiomatic definitions*. Like schemas, axiomatic definitions consist also of a declaration and a predicate part, but their declared variables can be used throughout the specification without a schema inclusion. E.g., we can define a constant *maxentries* which value is at least 5 with the axiomatic definition

$$\begin{array}{|l} maxentries : \mathbb{Z} \\ \hline maxentries \geq 5 \end{array}$$

We interpret axiomatic definitions analogously to how *P*roB interprets the sections CONSTANTS and PROPERTIES in a B machine: The very first step of an animation or model checking—before the initialisation of the state variables—consists in finding values for the constants which satisfy the predicates of the axiomatic definitions. After this step the predicates of the axiomatic definitions can be ignored. To illustrate how the axiomatic definitions are handled, we added the definition above to the birthday book example and appended the predicate $\# known \leq maxentries$ to the schema *BirthdayBook* before translating the specification to the result in Figure 2.3.

### 2.3.2 Translating initialisation and operations from Z to B

The *initialisation* schema *Init* consists of the declaration of all state variables and a predicate $I$. We annotate $T_v$ as the type of variable $v$.

$$\begin{array}{|l} \textit{Init} \\ \hline x_1 : T_{x_1};\ \ldots ; x_n : T_{x_n} \\ \hline I \\ \hline \end{array}$$

In B, the initialisation is a generalised substitution to all variables of the abstract machine. We can state "choose any values that satisfy $I$" with an ANY statement:

```
ANY $x'_1, \ldots, x'_n$
  WHERE $x'_1 \in T_{x_1} \wedge \ldots \wedge x'_n \in T_{x_n} \wedge$
    $I'$
  THEN $x_1, \ldots, x_n := x'_1, \ldots, x'_n$
END
```

Beside the predicate $I$, the WHERE clause of the ANY contains the type declaration of the variables. The types $T_v$ and the predicate $I$ are translated from Z to B syntax. Most of the types, predicates and expressions in Z have a direct counterpart in B and can be translated directly. In section 2.3.3 we show how we extended the B interpreter to support other constructs.

An *operation* schema $Op$ declares in addition to the state variables $x_1, \ldots, x_n$ their primed counterparts $x'_1, \ldots, x'_n$ and variables for input $i_1?, \ldots, i_k?$ and output $o_1!, \ldots, o_l!$. The predicate $P$ describes the effect of the operation.

$$\begin{array}{|l}
\hline
Op \\
x_1 : T_{x_1};\ \ldots;\ x_n : T_{x_n} \\
x'_1 : T_{x_1};\ \ldots;\ x'_n : T_{x_n} \\
i_1? : T_{i_1};\ \ldots;\ i_k? : T_{i_k} \\
o_1! : T_{o_1};\ \ldots;\ o_l! : T_{o_l} \\
\hline
P \\
\hline
\end{array}$$

ProZ translates such a schema to a B operation of the form

```
$o^*_1!, \ldots, o^*_l! \leftarrow Op(i_1?, \ldots, i_k?) =$
  PRE $i_1 \in T_{i_1} \wedge \ldots \wedge i_k \in T_{i_k}$ THEN
    ANY $x'_1, \ldots, x'_n, o_1!, \ldots, o_l!$
      WHERE $x'_1 \in T_{x_1} \wedge \ldots \wedge x'_n \in T_{x_n} \wedge o_1! \in T_{o_1} \wedge \ldots \wedge o_l! \in T_{o_l} \wedge$
        $P$
      THEN $x_1, \ldots, x_n, o^*_1!, \ldots, o^*_l! := x'_1, \ldots, x'_n, o_1!, \ldots, o_l!$
    END
  END
END
```

Like in the initialisation the central part of the operation is an ANY statement with the predicate $P$, but additionally we have to consider the possible result values. The surrounding PRE statement is just for declaring the types of the operation's arguments.

Often operations change only a subset of the state variables. ProZ checks if terms like $x = x'$ occur in the predicate $P$. If such a term is found, we know that $x$ does not change and so we can remove the substitution $x := x'$. Also we can replace all occurrences of $x'$ by $x$ in $P$. Then $x'$ is not used anymore in the statement and can be removed. If parts of the state or the complete state are not modified by an operation, the expression $\theta S = \theta S'$ is often used, where $S$ is a schema containing all variables that should not change. $\Xi S$ is an abbreviation for including $\Delta S$ and stating $\theta S = \theta S'$. Those expressions are transformed into $s_1 = s'_1 \wedge \ldots \wedge s_n = s'_m$ with $s_1, \ldots, s_m$ as the variables of S. This way the simplification of the ANY statement is also working with $\theta$-expressions.

### 2.3.3 New constructs and operators

Some constructs of Z's mathematical language do not have a direct counterpart in B, and below we show how we have treated those.

**Translation of Comprehension Sets** A comprehension set has the form $\{\, Decl \mid Pred \,\}$ and specifies a set with a declaration of variables and a predicate. E.g., the expression $\{\, i : \mathbb{N} \mid i \geq 5 \,\}$ is the set of all numbers greater or equal to 5. This kind of comprehension sets is also supported by B, but Z has an extended syntax of the form $\{\, Decl \mid Pred \bullet Expr \,\}$. E.g., the set $\{\, i : \mathbb{N} \mid i \geq 5 \bullet i * i \,\}$ is the set of all square numbers greater or equal to 25. We translate such comprehension sets as follows. Let $T$ be the type of *Expr*, then we express $\{\, Decl \mid Pred \bullet Expr \,\}$ by $\{\, v : T \mid (\exists Decl \mid v = Expr \wedge Pred) \,\}$ and translate this into B.

**Extensions to the B Interpreter** The following expressions are not easily translatable to B (or would entail a considerable efficiency penalty), and hence extensions were made to the *P*roB interpreter to support an extended B syntax:

- We added an **if**-expression to the standard B syntax. While B contains a substitution IF − THEN − ELSE, it can not be used as an expression that yields a value. The **if** expression of Z resembles to the ternary ?: operator known in C or Java.
- The **let** in Z can be used as an expression or as a predicate. Both can not be stated directly in B, which again only has the **LET** as a substitution.
- The operations $\upharpoonleft$ (extraction) and $\upharpoonright$ (filter) on sequences are defined with the function *squash*. We added the *squash* function to the interpreter.
- We added the definite description quantifier $\mu$.

## 2.4 New Types

To deal with the Z specifications we have seen so far, it was sufficient to translate Z to B, possibly with some some syntactic extensions. There are, however, two important features of Z which cannot be effectively dealt with in that way: Z's schema and free types. Supporting those features in an effective manner requires a fundamental addition to the core datatypes of the *P*roB kernel.

**Overview of the *P*roB-kernel** The *P*roB kernel is responsible for storing and finding values for the values of the variables in a specification. In order to avoid naive enumeration of possible values, the *P*roB kernel is written in Prolog works in multiple phases (controlled by Prolog's `when` co-routining mechanism). In the first phase, only deterministic propagations are performed (e.g., the predicate $x = 1$ will be evaluated but the predicates $x \in \mathbb{N}$ will suspend until they either become deterministic or until the second phase starts). In the second phase, a restricted class of non-deterministic enumerations will be performed. For example, the predicate $x \in \{a, b\}$ will suspend during the first phase but will lead to two solutions $x = a$ and $x = b$ during the second phase. In the final phase, *all* variables, parameters and constants that are still undetermined (or partially determined) are enumerated.

**New Data types** Adding a new basic data type to the kernel requires the extension of four Prolog predicates: `equal_object` to check two objects for equality, `not_equal_object` to check two objects for disequality, one predicate to type check an object and one predicate to enumerate all possible values of an object given its type. So far the kernel supported basic user-defined types (defined in B's SET clause), integers, pairs and sets (relations are represented as sets of pairs). Below, we present two new data types, schema types and free types, which are needed for Z.

### 2.4.1 Schema Types

In Z each schema defines a new data type, a *schema type* which resembles record types known from other languages. Basically, a record data value $rec(f)$ consists of a list $f = [n_1/v_1, \ldots, n_k/v_k]$ of field names $n_i$ along with values $v_i$ for each field. We require that all field names are sorted alphabetically. Two record values are identical iff they have the exact same field names and all field values are identical. In the kernel this gives rise to two new inference rules:

$$\frac{x_1 = y_1 \qquad \ldots \qquad x_k = y_k \qquad n_1 < n_2 < \ldots < n_k}{rec([n_1/x_1, \ldots, n_k/x_k]) = rec([n_1/y_1, \ldots, n_k/y_k])}$$

$$\frac{x_i \neq y \qquad 0 \leq i \leq k \qquad n_1 < n_2 < \ldots < n_k}{rec([n_1/x_1, \ldots, n_i/x_i, \ldots, n_k/x_k]) \neq rec([n_1/x_1, \ldots, n_i/y, \ldots, n_k/x_k])}$$

The type of a record contains the name of the fields and the types of each field. This gives rise to two new inference rules for type inference and enumeration, where we use the k-ary type constructor *Record* for records with $k$-fields:

$$\frac{x_1 : \tau_1 \qquad \ldots \qquad x_k : \tau_k \qquad n_1 < n_2 < \ldots < n_k}{rec([n_1/x_1, \ldots, n_k/x_k]) : Record(n_1/\tau_1, \ldots, n_k/\tau_k)}$$

$$\frac{x_1 \in enum(\tau_1) \qquad \ldots \qquad x_k \in enum(\tau_k) \qquad n_1 < n_2 < \ldots < n_k}{rec([n_1/x_1, \ldots, n_k/x_k]) \in enum(Record(n_1/\tau_1, \ldots, n_k/\tau_k))}$$

Classical B does not have a record type, but a record type extension and syntax has been introduced by the tool Atelier-B [Ste09].[1] In extending the kernel, ProB now also supports those records in B.

Note that in Z, possible instances of a schema type (the *bindings*) can be further constrained by the predicates of the schema. E.g. the schema

$$\mathit{ExampleRecord} \mathrel{\widehat=} [\, x, y : \mathbb{Z} \mid x < y \,]$$

can be used as a record with the constraint $x < y$. The kernel does not support this directly, instead an unconstrained record $[\, x, y : \mathbb{Z} \,]$ can be used. We show how the constraints can be preserved in the translation process by *normalisation*.

ProZ normalises all schemas of a specification, i.e. it strictly separates type information and additional predicates on the instances. E.g. the normalised form of the schema $[\, x : \{1, 2, 3\} \,]$ is $[x : \mathbb{Z} \mid x \in \{1, 2, 3\}]$. Given a normalised schema $A \mathrel{\widehat=} [\, \mathit{Decl} \mid \mathit{Pred} \,]$, we define $A^* \mathrel{\widehat=} [\, \mathit{Decl} \,]$ as the schema with just the type information and without any additional constraints. If $A$ is used as a type for a variable $v$, in the normalisation process it is split into the type $A^*$ and the additional constraint $v \in \{\, \mathit{Decl} \mid \mathit{Pred} \bullet \theta A\}$. Because the type $A^*$ does not have further constraints, it's supported by the kernel. The constraint had been made explicit by the normalisation and can be translated to B. The used $\theta$-operator creates an instance of type $A$. We can translate it directly to a record constructor.

### 2.4.2 Free Types

Another feature of the Z notation is the definition of *free types*. E.g.,

$$T ::= \mathit{empty} \mid \mathit{value}\langle\!\langle \{1, 2, 3\} \rangle\!\rangle$$

defines a new data type $T$ with a constant value *empty* and a constructor function *value* which maps values from $\{1, 2, 3\}$ to $T$. Contrary to schema types, free types can also be recursive, as in the following example, defining a binary tree with integers:

$$\mathit{BinTree} ::= \mathit{empty} \mid \mathit{leaf}\langle\!\langle \mathbb{Z} \rangle\!\rangle \mid \mathit{node}\langle\!\langle \mathit{BinTree} \times \mathbb{Z} \times \mathit{BinTree} \rangle\!\rangle$$

In Z, free types are only syntactic sugar and can also be expressed with axiomatic definitions and basic types. But for the purpose of animating the specification it is essential for efficiency to implement this type directly.

There is no counterpart for free types in B, so we extended the ProB core. The representation of data values and the inference rules for equality and typing are similar to record types; one just needs to also store the constructor used (e.g., in the case of $T$ above we need to know whether we are in the case *empty* or in the case *value*). Two free type data values are thus identical iff they have the same constructor and if the values for that constructor are identical.

[1] See also [EB02] for a theoretical foundation of records.

Free type definitions can be made recursive, so the implementation of enumeration must prevent the generation of infinitely many values. We solved this by introducing a maximum recursion depth when enumerating free types. The maximum is adjustable by the user. The introduction of a maximum recursion depth has the effect that the model checker might not find all possible solutions (similarly to integer variables whose enumeration is restricted to MININT..MAXINT).

The *P*roB interpreter is extended by a constructor *FreeConstructor* for creating instance values of free types. The arguments are the free type, the case (*empty* or *value* in the $T$ example) and the tuple containing the arguments to the constructor. Also there is the inverse of the constructor *FreeDestructor*, which takes a free type instance and returns the type, the case and the tuple of arguments. Finally, we have a predicate *FreeCase* that takes the identifier of a case and a free type instance as arguments and evaluates to true if the free type value has the given case.

The kernel itself does not support constraints on the values of a constructor. In the $T$ example above the type of the constructor *value* is $\mathbb{Z}$ but the domain constrained to $\{1, 2, 3\}$. Like with the schema types, the constraints have to be handled separately in the translation. This is done by normalisation as follows.

Given a free type $F$ of the form

$$F ::= c_1 \mid \ldots \mid c_n \mid d_1\langle\!\langle S_1\rangle\!\rangle \mid \ldots \mid d_m\langle\!\langle S_m\rangle\!\rangle$$

we define the type $F^*$ which has just the type information of $F$ without other constraints, where $T_i$ is the underlying type of $S_i$ (e.g. $S_i = \{1, 2, 3\} \Rightarrow T_i = \mathbb{Z}$):

$$F^* ::= c_1 \mid \ldots \mid c_n \mid d_1^*\langle\!\langle T_1\rangle\!\rangle \mid \ldots \mid d_m^*\langle\!\langle T_m\rangle\!\rangle$$

Then we convert $F$ and the constructors $d_i$, $1 \leq i \leq m$ to

$$\begin{array}{l} F == \{\, x : F^* \mid \\ \qquad x \in \operatorname{ran} d_1^* \Rightarrow d_1^{*\sim}(x) \in S_1 \wedge \ldots \wedge x \in \operatorname{ran} d_m^* \Rightarrow d_m^{*\sim}(x) \in S_m \,\} \\ d_i == (\lambda\, x : T_i \mid x \in S_i \bullet d_i^*(x)) \end{array}$$

The schema normalisation transforms a variable $v$ of type $F$ to a variable of type $F^*$ and adds the constraint $v \in F$.

The transformed example would be

$$\begin{array}{l} T^* ::= \mathit{empty} \mid \mathit{value}^*\langle\!\langle \mathbb{Z}\rangle\!\rangle \\ T == \{\, x : T^* \mid x \in \operatorname{ran} \mathit{value}^* \Rightarrow \mathit{value}^{*\sim}(x) \in \{1, 2, 3\} \,\} \\ \mathit{value} == (\lambda\, x : \mathbb{Z} \mid x \in \{1, 2, 3\} \bullet \mathit{value}^*(x)) \end{array}$$

Finally, expressions of the form $x \in \operatorname{ran} d_i^*$ are translated to the predicate $\mathit{FreeCase}(F, d_i^*, x)$ and constructor calls of the form $d_i^*(x)$ are translated to $\mathit{FreeConstructor}(F^*, d_i^*, x)$ (resp. the inverse $d_i^{*\sim}(y)$ is translated to the expression $\mathit{FreeDestructor}(F^*, d_i^*, y)$). The result can then be dealt with by the extended *P*roB interpreter and kernel.

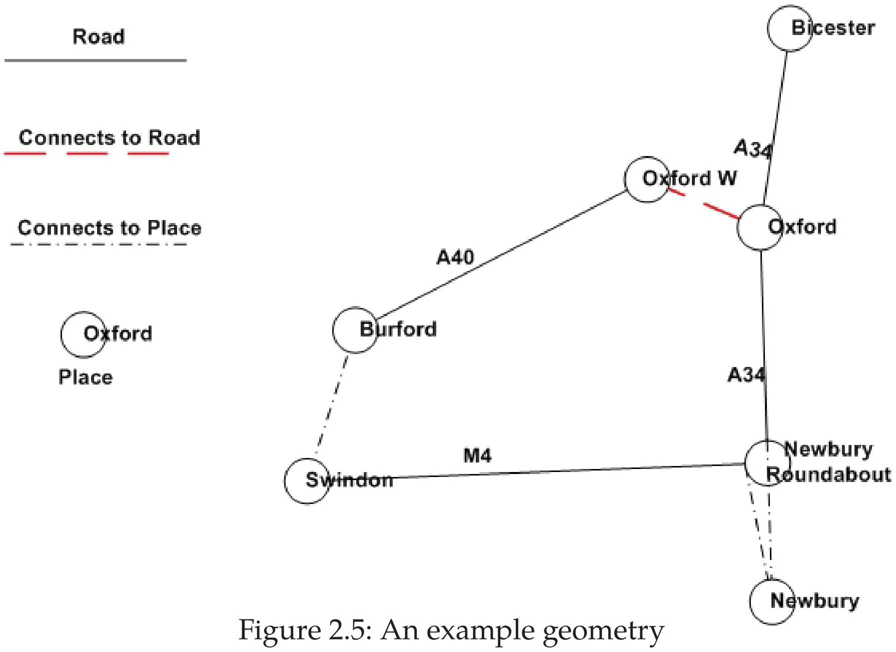


Figure 2.5: An example geometry

## 2.5 Case study

The case study was inspired by a real industrial example. The specifications are very high level and, using the guidelines from [Hal90], were not destined to be refined into code. These Z specifications thus[2] provide a particular challenge for our tool. Below we present two sub-components of the system, the challenges in animating and validating them, as well as an indication on the errors located by our tool.

### 2.5.1 Route calculation

The *route calculation* component is a key component of the overall system, containing several intricate algorithmic aspects. It is important to ascertain the correctness of the algorithms (e.g., before proceeding with an implementation).

This system component calculates routes through a given geometry. The geometry (mainly places and roads) is stored in the system state. The main part of the specification consists of the definition of a function that takes a route as input. The input route is a sequence that starts and ends with a place and between both is a list of places or roads. The result of the function is the *expansion* of the route, i.e. the sequence of all places that lie between the first and the last place. E.g., in the given geometry in figure 2.5 the expansion of $\langle Bicester, A34, M4, Swindon\rangle$ is $\langle Bicester, Oxford, NewburyRoundabout, Swindon\rangle$. For sake of simplicity we ignore below the connections which are not roads. The expansion

[2] Some of the features of B, such as generalised union, are rarely used in formal refinements as the existing B provers do not support them very well. It is our experience that formal B specifications that are refined to code are easier to animate than more liberal specifications.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{ExpandElems}} \\
\hline
\mathit{ExpandElemexpandElems} : \mathit{Expansion} \pfun \mathit{Expansion} \\
\hline
\mathit{expandElems} = \{\ \Delta \mathit{Expansion} \mid \\
\quad \theta \mathit{Expansion}' = \mathbf{if}\ \mathit{error} \neq \varnothing \lor \mathit{currentElem} \notin \operatorname{dom} \mathit{proposedRoute} \\
\qquad \mathbf{then}\ \theta \mathit{Expansion} \\
\qquad \mathbf{else}\ \mathit{expandElems}(\mathit{expandElem}(\theta \mathit{Expansion})) \bullet \\
\quad \theta \mathit{Expansion} \mapsto \theta \mathit{Expansion}'\ \} \\
\hline
\end{array}
$$

Figure 2.6: Example: The recursive definition of the function *expandElems*.

function is constructed by combining several other functions, which do not work directly on the input route. Instead a record is created that contains the original route, information about which part has already been processed, the result so far, and a set of errors. An error could be "no connection found", for example. A recursive function (Fig. 2.6) expands every single element in the route until the complete route is expanded or an error is found. In total the specification consists of 8 function definitions which are combined to calculate the result. Most of these functions are defined by comprehension sets.

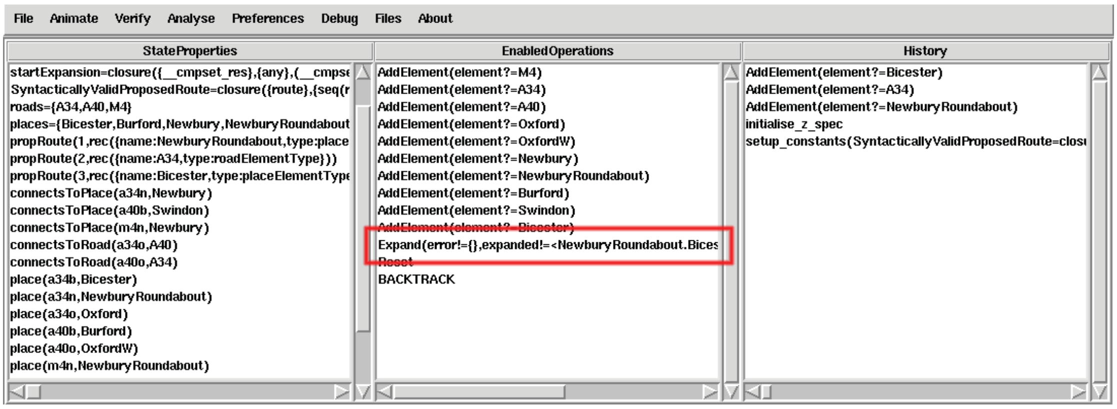
Due to the complexity of the defined functions, it was not feasible to enumerate them (i.e., to store all possible inputs and outputs). Fortunately, *P*roB [LCB09] has the ability to compile these kind of definitions into *symbolic* closures, which are evaluated and expanded on demand. For example, given a set comprehension $S = \{x \mid x \in \mathbb{N} \Rightarrow P\}$ and the condition $y \in S$ the Kernel will "only" check that $P$ holds for $x = y$ and *not* compute the entire set $S$. A similar situation arises for lambda abstractions. Take for example, $f = \lambda x.(x \in \mathbb{N} \bullet E)$. In that case, to evaluate $f(y)$ the Kernel "only" evaluates $E$ with $y$ substituted for $x$ and *not* the entire function $f$.[3] The kernel, even supports recursive function definitions, such as the one presented in Fig. 2.6.

Storing comprehensions sets and $\lambda$-expressions symbolically was an essential feature to allow the animation of the specification. By integrating *P*roZ into *P*roB we inherit this feature, which allows us to validate this specification.

To make an animation possible, the system was initialised with test data that describes a map with six cities. Running the animation the user can simply click on the *AddElement* operations to construct an input and sees immediately the result. Figure 2.7 shows a screenshot of the animator after entering the route "Newbury → A34 → Bicester". In the middle the list of enabled operations can be seen where the *Expand* operation contains the solution (which is truncated in the screenshot).

The animation of the specification quickly exhibited one error in the specification. For each road a sequence of places to which it connects is stored in the geometry. When a route contains a road, the entry and exit points are calculated and the section between both is appended to the result. But under certain circumstances the section was appended in the wrong direction so that the route "Newbury → A34 → Bicester" was

[3] Some expressions, however, will require the computation of the entire function (e.g., $dom(f) \subseteq SetA$). In those circumstances the kernel converts the symbolic form into explicit form.



StateProperties

```
startExpansion=closure({__cmpset_res},{any},(__cmpse
SyntacticallyValidProposedRoute=closure({route},{seq(r
roads={A34,A40,M4}
places={Bicester,Burford,Newbury,NewburyRoundabout
propRoute(1,rec({name:NewburyRoundabout,type:place
propRoute(2,rec({name:A34,type:roadElementType}))
propRoute(3,rec({name:Bicester,type:placeElementType
connectsToPlace(a34n,Newbury)
connectsToPlace(a40b,Swindon)
connectsToPlace(m4n,Newbury)
connectsToRoad(a34o,A40)
connectsToRoad(a40o,A34)
place(a34b,Bicester)
place(a34n,NewburyRoundabout)
place(a34o,Oxford)
place(a40b,Burford)
place(a40o,OxfordW)
place(m4n,NewburyRoundabout)
```

EnabledOperations

```
AddElement(element?=M4)
AddElement(element?=A34)
AddElement(element?=A40)
AddElement(element?=Oxford)
AddElement(element?=OxfordW)
AddElement(element?=Newbury)
AddElement(element?=NewburyRoundabout)
AddElement(element?=Burford)
AddElement(element?=Swindon)
AddElement(element?=Bicester)
Expand(error!={},expanded!=<NewburyRoundabout.Bices
Reset
BACKTRACK
```

History

```
AddElement(element?=Bicester)
AddElement(element?=A34)
AddElement(element?=NewburyRoundabout)
initialise_z_spec
setup_constants(SyntacticallyValidProposedRoute=closu
```

Figure 2.7: Animation of the route calculation

calculated to "Newbury → Bicester → Oxford → Newbury → Bicester" instead of the much simpler correct solution "Newbury → Oxford → Bicester".

Figure 2.8 shows the function containing the error. The result of the expression (in the third **let** expression)

$$(\textbf{if } \mathit{entry} > \mathit{exit} \textbf{ then } \mathit{exit} \mathrel{..} \mathit{entry} \textbf{ else } \mathit{entry} \mathrel{..} \mathit{exit}) \upharpoonright \mathit{roadPlaces}(r)$$

are all places on the road $r$ that are between *entry* and *exit*. If $\mathit{roadPlaces}(r)$ is the sequence $\langle a, b, c, d, e \rangle$, *entry* is 4 and *exit* is 2, than the result is $\langle b, c, d \rangle$. Although the case $\mathit{entry} > \mathit{exit}$ is covered explicitly in the specification, it has been forgotten to reverse the resulting sequence to $\langle d, c, b \rangle$.

For this application we did not yet use the model checking facilities of ProZ, because we have no further properties about the result of the algorithm (and hence no way to automatically check the correctness of the result). But the animator alone gives the user a powerful tool to get more insight in the behaviour of a specification, as the quick detection of errors showed.

### 2.5.2 Network protocol

A second important component of the overall system implements access control to a shared resource, employing a simple network protocol. A number of workstations are connected via a network and share are critical resource. Whenever a workstation wants to access the resource it has to send a request to the other workstations. The protocol should assure that only one workstation can be in the critical section at the same time.

The specification distinguishes between the state and behaviour of the workstations and the the state and behaviour of the underlying middleware.

The specification of the middleware is the description of an existing system. Its state space consists of a sent and received buffer for each workstation. Messages can be added to a sent buffer, transferred between workstations and removed from a received buffer to deliver it to the workstation.

$$
\begin{array}{|l}
\hline
\textit{ExpandRoad} \\
\textit{FindConnections} \\
\textit{expandRoad} : \textit{Expansion} \mathrel{+\!\!\!\!\!\rightarrow} \textit{Expansion} \\
\hline
\textit{expandRoad} = \{ r : \textit{ElementName}; \textit{ExpansionOp} \mid \\
\quad r \in \textit{RoadName} \land \\
\quad (\textit{proposedRoute}(\textit{currentElem})).\textit{type} = \textit{roadElementType} \land \\
\quad (\textit{proposedRoute}(\textit{currentElem})).\textit{name} = r \land \\
\quad (\mathbf{let}\ \textit{entries} == \textit{findConnections}(r, \textit{proposedRoute}(\textit{currentElem} - 1)); \\
\qquad\quad \textit{exits} == \textit{findConnections}(r, \textit{proposedRoute}(\textit{currentElem} + 1)) \bullet \\
\qquad ((\textit{entries} = \varnothing \lor \textit{exits} = \varnothing) \\
\qquad\quad \land\ \textit{error}' = \{\textit{noConnection}\} \land \textit{expandedRoute}' = \langle\rangle) \lor \\
\qquad (\mathbf{let}\ \textit{entry} == \textit{min}(\textit{entries}); \textit{exit} == \textit{min}(\textit{exits}) \bullet \\
\qquad\quad (\mathbf{let}\ \textit{placesToAdd} == (\mathbf{if}\ \textit{entry} > \textit{exit}\ \mathbf{then}\ \textit{exit} \mathinner{..} \textit{entry} \\
\qquad\qquad \mathbf{else}\ \textit{entry} \mathinner{..} \textit{exit}) \upharpoonright \textit{roadPlaces}(r) \mathbin{⨟} \textit{place} \bullet \\
\qquad\quad \textit{expandedRoute}' = \mathbf{if}\ \textit{last}\ \textit{expandedRoute} = \textit{head}\ \textit{placesToAdd} \\
\qquad\qquad \mathbf{then}\ \textit{expandedRoute} \frown \textit{tail}\ \textit{placesToAdd} \\
\qquad\quad \mathbf{else}\ \textit{expandedRoute} \frown \textit{placesToAdd} \land \\
\qquad\quad \textit{error}' = \varnothing))) \bullet \\
\quad \theta\textit{Expansion} \mapsto \theta\textit{Expansion}'\} \\
\hline
\end{array}
$$

Figure 2.8: The definition of the function *expandRoad* containing an error.

The specification of the workstation defines their current states (*idle*, *waiting*, *editing* or *failed*) and their operations. They can send requests to the other workstations, read their responses, read other requests and send responses.

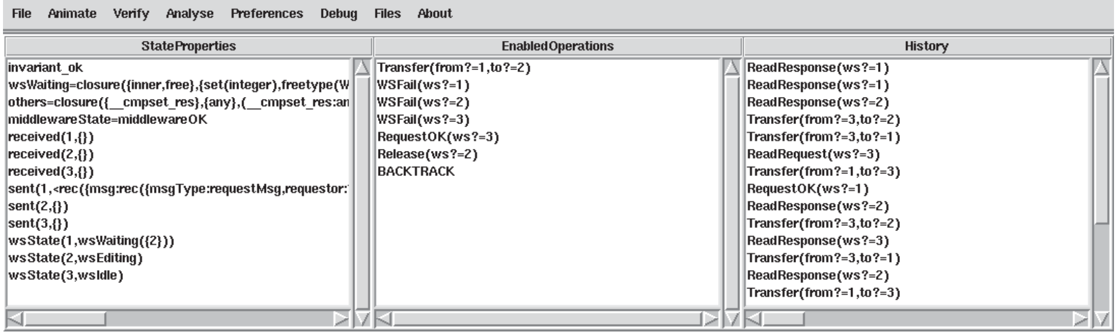
The components from both parts of the definitions are combined by using the schema calculus. Especially the pipe operator ($\gg$) was used to connect operations, where the result of one operation serves as the input for another operation. E.g. when a workstation sends a request, the operation describing the workstation behaviour outputs a message that is taken by a middleware operation as input:

$$\textit{RequestOK} \mathrel{\widehat{=}} \textit{RequestWorkstationOK} \gg \textit{AcceptMsgMiddleware}$$

A screenshot of the animator is shown in Fig. 2.9. On the left side the current state is displayed. It can be seen that workstation 1 is waiting for a response of workstation 2, workstation 2 is in editing mode and workstation 3 is idle. Also a message is still in the sent buffer of workstation 1.

Free types are used in the specification for distinguishing the different modes of a workstation. *wsIdle* and *wsEditing* are constants of the free type, whereas *wsWaiting* is a constructor, e.g. $\textit{wsWaiting}(\{1,3\})$ refers to the state "waiting for workstations 1 and 3".

First we used the model checker to find deadlocks in the protocol. It found a deadlock that was caused by an error in the specification. It was possible that a workstation could ignore a rejected request. The same error caused a situation where more than one workstation was in the critical section.



StateProperties

```
invariant_ok
wsWaiting=closure({inner,free},{set(integer),freetype(W
others=closure({__cmpset_res},{any},(__cmpset_res:an
middlewareState=middlewareOK
received(1,{})
received(2,{})
received(3,{})
sent(1,<rec({msg:rec({msgType:requestMsg,requestor:
sent(2,{})
sent(3,{})
wsState(1,wsWaiting({2}))
wsState(2,wsEditing)
wsState(3,wsIdle)
```

EnabledOperations

```
Transfer(from?=1,to?=2)
WSFail(ws?=1)
WSFail(ws?=2)
WSFail(ws?=3)
RequestOK(ws?=3)
Release(ws?=2)
BACKTRACK
```

History

```
ReadResponse(ws?=1)
ReadResponse(ws?=1)
ReadResponse(ws?=2)
Transfer(from?=3,to?=2)
Transfer(from?=3,to?=1)
ReadRequest(ws?=3)
Transfer(from?=1,to?=3)
RequestOK(ws?=1)
ReadResponse(ws?=2)
Transfer(from?=3,to?=2)
ReadResponse(ws?=3)
Transfer(from?=3,to?=1)
ReadResponse(ws?=2)
Transfer(from?=1,to?=3)
```

Figure 2.9: Animation of the network protocol

We added an *Invariant* schema to the specification to check automatically if more than one workstation is in the editing mode (*wsState* is a function defined in the schema *Workstations* that maps each workstation to its mode, and the operator $\rhd$ is used to restrict it to all entries which map to *wsEditing*):

$$Invariant \mathrel{\widehat=} [\, Workstations \mid \#(wsState \rhd \{wsEditing\}) \leq 1 \,]$$

The model checker was able to find states where the invariant was violated. Another error was found: Every response to a request was treated as if it was a grant, even rejections.

The model checker was not able to do an exhaustive search of the state space because the message buffers in the model are not limited.

## 2.6 Discussion, Related and Future Work

**Limitations** Z is a very large and extensive formal method, with many features and extension. While we provide a tool that can animate a considerable subset of Z, some of Z's features are obviously not yet supported:

- Bags (multisets) are not supported.
- Some expressions like disjoint and partition are not yet implemented.
- Generic definitions cannot be used in a specification yet. We plan to support them by determining with wich types a generic definition is used, and then creating for each such type a separate axiomatic definition.

**Related and Future Work** On the theoretical side, there are several works discussing the relationship and translations between Z and B [Dun04] or weakest precondition semantics [CW98].

On the practical side, several animators for Z exist, such as [WDK98], which presents an animator for Z implemented in Mercury, as well as the Possum animation tool [HST98].

Another animator for Z is ZANS [Jia95]. It has been developed in C++ and unlike *P*roB only supports deterministic operations (called explicit in [Jia95]). The more recent Jaza tool by Mark Utting [Utt00] looks very promising. There has also been a recent push [MU05] to provide more tool support for Z. However, to our knowledge, no existing Z animator can deal with the recursive higher-order functions present in our case study.

The most closely related work on the B side is [BLP02, ABC$^{+}$02, LPU02], which uses a special purpose constraint solver over sets (CLPS) to animate B and Z specifications using the so-called BZ-Testing-Tools. However, the focus of these tools is test-case generation and not verification, and the subset of B that is supported is comparatively smaller (e.g., no set comprehensions or lambda abstractions, constants and properties nor multiple machines are supported).

Another very popular tool for validating specifications and models is Alloy [Jac02], which makes use SAT solvers (rather than constraint solving). However, the specification language of Alloy is first-order and thus cannot be applied "out of the box" to our motivating industrial example.

**Conclusion** In this paper we presented *P*roZ, a tool for animating and model checking Z specifications. We pursued an approach to translate Z specifications to B, reusing the existing *P*roB toolset as much as possible. Some extensions to the *P*roB core were required (e.g., for free types and schema types), after which we have obtained an integrated tool that is now capable to animate and validate Z and B specifications. In principle our tool could now validate combined B/Z specifications,[4] and as a side effect we have added support for B specifications with records. By integrating *P*roZ with *P*roB our tool has also inherited from the recent developments and improvements originally devised for B, such as visualisation of large state spaces [LT05], integration with CSP [BL05], symmetry reduction [LBST07], and symbolic validation of recursive functions [LCB09].

Our tool was successfully applied to examples which were based on industrial specifications and also revealed several errors. Especially *P*roZ's ability to store comprehensions sets symbolically was essential to make the animations of those specifications possible.

**Acknowledgements** We would like to thank Anthony Hall for his very helpful contributions and feedback on the paper.

[4] It is not clear to us whether this has any practical benefit

# 3 Validation of Formal Models by Refinement Animation

Stefan Hallerstede, Michael Leuschel, Daniel Plagge

**Abstract.** We provide a detailed description of refinement in Event-B, both as a contribution in itself and as a foundation for the approach to simultaneous animation of multiple levels of refinement that we propose. We present an algorithm for simultaneous multi-level animation of refinement, and show how it can be used to detect a variety of errors that occur frequently when using refinement. The algorithm has been implemented in *P*roB and we applied it to several case studies, showing that multi-level animation is tractable also on larger models. We present empirical results and discuss how the algorithm can be combined with symmetry reduction.

**Keywords: Refinement – Model Checking – Constraint-Solving – Tools – Industrial Applications – Event-B**

## 3.1 Introduction and Motivation

We consider formal modelling of software systems an important phase in the development process. It permits us to reason about the system to be developed early on in the development using abstractions of the system. We believe that this reasoning is a key to improving the quality of the developed software. The approach to reasoning varies depending on what kind of problems one wants to uncover. It can reach from formal proof to just "trying out" a model. We expect to benefit from this by improving our understanding of a system (by analysing models of it) and by creating an adequate formal model that can serve as the reference for what is to be considered "correct" in later stages of the development process. As much as possible reasoning should be supported by a software tool. The work presented in this article is based on the Event-B modelling method [Abr10] and the complementing software tool Rodin [ABH06]. The core of the Rodin tool provides automatic generation of proof obligations that can be analysed to gain insight into a formal model. Often proof obligations give good indications of how to make an improvement in case of inconsistencies in a model. However, there are also many occasions where proof obligations do not point directly to a problem or where a model does not contain inconsistencies but is still "incorrect" (see, e.g., the Earley parser example discussed in [BLLS08]). In many such cases animation is an useful tool to gain further insight into a model. The Rodin plugins *P*roB [LB03, LB08], Brama[1] [Ser07], and AnimB [Ani] provide animation facilities for Event-B.

When dealing with complex models, refinement can be used to introduce the many details gradually, achieving a reduced complexity at each refinement level. Proof obligations, once discharged, demonstrate correctness of a refinement: the more detailed model behaves within the limits set by the more abstract model. However, it can be difficult to analyse a refinement relationship only by means of associated proof obligations. Animation can be used to complement proof as a method of validation providing a broader view on the properties of formal models. All three animation plugins mentioned above provide some means to animate refinements. In this article we investigate their relative capabilities and how to advance refinement animation in order to turn it into a tool for refinement validation. This serves as a blueprint for the evolution of *P*roB in terms of animation support.

Before starting the investigation we should be clear about the objectives of animation when used for validation. What is the purpose of animating a model across multiple levels of refinement? In Event-B several concepts play a role in refinement. Most prominently, these are invariants, guards, actions, and witnesses. If a refinement fails, any combination of those concepts may be involved. Animation should help to locate the cause of a problem in the model, pointing to specific invariants, guards, and so on, if possible. However, even if a refinement is formally correct, there can still be problems with the model. This concerns, in particular, properties that have not been formalised. Animation should make it easy to experiment with a model, visualising potential problems. We try to integrate this aspect of animation with the first one as far as possible. Otherwise consistent tool support for both would be difficult to realise.

In Section 3.2 we introduce the Event-B notation (using the syntax of the Event-B text editor “Camille” [BFL10]) and give a concise description of the fundamentals of Event-B refinement. As far as we are aware there is no single place where all the essential aspects are described and motivated in such detail. All of this is needed in order to present the basic refinement-animation algorithm in Section 3.3. The presentation of the algorithm is interspersed with methodical remarks on validation. In Section 3.4 we present some concrete examples on how to use *P*roB for refinement validation and some brief description of case studies to which it has been applied. Finally, Sections 3.6 and 3.7 contain a discussion of related work and a conclusion.

## 3.2 Modelling and Refinement in Event-B

Event-B can be used to model complex intricate systems. To understand the system and the model of the system we need to reason thoroughly about the model. Such reasoning is the principal purpose of Event-B. The basic concepts of Event-B are characterised by means of proof obligations; they are the core of the Event-B method. However, they are not an exclusive means of reasoning. Based on an operational interpretation of a model

[1] Brama requires an older version (0.9.2.x) of Rodin at the time of writing.

we can also animate the model to gain deeper understanding. In this section we parallel the presentation of Event-B proof obligations, in particular, refinement, with ideas of animation. This demonstrates well how animation complements proof. Because there is no single software tool for animation that provides all that is needed, we use the three tools *P*roB, Brama, and AnimB at the same time.

### 3.2.1 Contexts

Event-B models are described in terms of the two basic constructs: *contexts* and *machines*. Contexts specify static parts of a model, that is, *carrier sets* and *constants* that are constrained by *axioms*, introduced by the keywords sets, constants, and axioms, as shown in Fig. 3.1. Usually, axioms are quite simple formulas; contexts are intended to be used to parameterise machines. We mention contexts here mainly because of the role they play in animation. For any particular animation specific values for all constants have to be found. *P*roB does this automatically using constraint-solving techniques for finding proper values that satisfy all axioms. The constraint-solving also determines whether the axioms contain a contradiction. In order to use carrier sets, constants, and axioms of a context in a machine, the context needs to be referenced in the machine: the machine sees the context. For instance, in Fig. 3.3 machine *CoffeeM* sees context *CofCtxt*. Contexts are described in more detail in [Abr10, AH07].

```
context CofCtxt
constants full empty half level
sets FILL
axioms
  @Ffhe partition(FILL, {full}, {half}, {empty})
  @lvl level = (0 .. 2 × {empty}) ∪ (3 .. 7 × {half}) ∪ (8 .. 11 × {full})
end
```

Figure 3.1: Context of coffee dispenser model

### 3.2.2 Notation for Variables and Substitutions

We follow the style of [Abr10] of expressing variables and substitution in formulas. Let $v = v_1, \ldots, v_n$ be a sequence of $n$ distinct variables, $t = t_1, \ldots, t_n$ a sequence of $n$ formulas and $F$ a formula. Then $F[t/v]$ is obtained from $F$ by replacing simultaneously all free occurrences of each $v_i$ by $t_i$.

We let $F(v)$ denote a formula, whose free variables are among $v_1, \ldots, v_n$. Once the formula $F(v)$ has been introduced, we denote by $F(t)$ the formula $F[t/v]$ with $v$ replaced by $t$. By $v'$ we denote the variables $v'_1, \ldots, v'_n$, used for naming after-states of state transitions. For (disjoint) variable sequences $v$ and $w$ we denote their concatenation by $v, w$. These notions could also be used with Event-B constants. But we do not do this in this article where we focus on refinement relationships and consequently on variables.

### 3.2.3 Machines

*Machines* provide behavioural properties of Event-B models. Machines may contain *variables*, *invariants*, *events*, and *variants*. Variables $v$ define the state of a machine. They are constrained by invariants $I(v)$. Possible state changes are described by means of events. Each event is composed of a *guard* $G(t,v)$ and an *action* $S(t,v)$, where $t$ are *parameters* of the event. The guard states the necessary condition under which an event may occur, and the action describes how the state variables evolve when the event occurs. We denote an event $E(v)$ by one of the forms shown in Fig. 3.2. Form (b) is used if event $E(v)$ does not have parameters, and form (c) if in addition the guard equals *true*. A dedicated event of the third form is used for *INITIALISATION*. In the formal exposition below, we assume without loss of generality that the most general first form is used.

```
any  $t$  when
  $G(t,v)$
then
  $S(t,v)$
end
```

(a) Event (general form)

```
when
  $G(v)$
then
  $S(v)$
end
```

(b) Event without parameters

```
begin
  $S(v)$
end
```

(c) Event without parameters or guard

Figure 3.2: Syntax of events

The action of an event is composed of *assignments* of the form: $x := E(t,v)$ or $x :\in E(t,v)$ or $x :\mid Q(t,v,x')$, where $x$ is some or all of the variables $v$, $E(t,v)$ expressions, and $Q(t,v,x')$ a predicate. The second form assigns $x$ to an element of a set, and the third form assigns to $x$ a value satisfying a predicate. The first two can be formally defined in terms of the third form: $x := E(t,v) \;\widehat{=}\; x :\mid x' = E(t,v)$ and $x :\in E(t,v) \;\widehat{=}\; x :\mid x' \in E(t,v)$. The effect of an assignment is described by a before-after predicate:

$$\text{before-after predicate of ``}x \;:\mid\; Q(t,v,x')\text{''} \quad \widehat{=} \quad Q(t,v,x')$$

A before-after predicate describes the relationship between the state just before an assignment has occurred, $x$, and the state just after the assignment has occurred, $x'$. All assignments of an action $S(t,v)$ occur simultaneously which is expressed by conjoining their before-after predicates, yielding a predicate $A(t,v,x')$. Variables $y$ that do not appear on the left-hand side of an assignment of an action are not changed by the action. Formally, this is achieved by conjoining $A(t,v,x')$ with $y' = y$, yielding the predicate

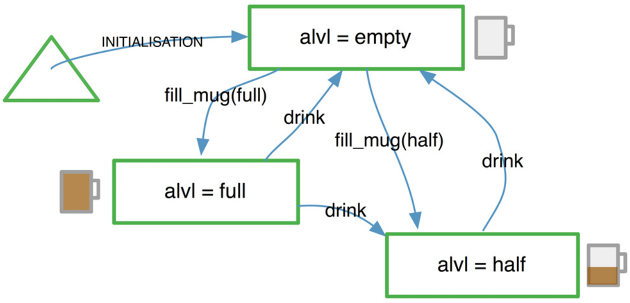
$$\mathbf{S}(t,v,v') \quad \widehat{=} \quad A(t,v,x') \;\wedge\; y' = y \quad .$$

#### Running Example

We use the coffee dispenser model in Fig. 3.3 (with the context shown in Fig. 3.1) and Fig. 3.19 for illustration of refinement-animation. In the abstract machine *CoffeeM* the dispenser can fill a mug half or fully; the state of the mug is represented by the variable *alvl* (abstract level). As a special service the dispenser can also drink the coffee.

```
machine $CoffeeM$ sees $CofCtxt$
variables $alvl$
invariants
  @inv1 $alvl \in FILL$
variant $(\{full \mapsto 2, half \mapsto 1, empty \mapsto 0\})(alvl)$
events
  event $INITIALISATION$
    begin
      @mf $alvl := empty$
    end
  event $fill\_mug$
    any $x$ when
      @g0 $alvl = empty$
      @g1 $x \neq alvl$
    then
      @a1 $alvl := x$
    end
  convergent event $drink$
    when
      @g1 $alvl \neq empty$
    then
      @a1 $alvl :\in \{empty, half\} \setminus \{alvl\}$
    end
end
```

(a) Coffee dispenser machine



(b) State space for *CoffeeM* (with additional mugshots)

Figure 3.3: Coffee dispenser model

In the first refined machine *CoffeeR1* a feature is introduced for inserting an arbitrary number of coins into the dispenser. A coin is consumed each time a mug is filled. In the second refined machine *CoffeeR2*, the number of coins maximally accepted is limited and the amount of coffee contained in a mug is represented numerically by a the variable *clvl* (concrete level).

### Animation in ProB

The before-after predicate can be used to compute the state space of a machine, a graph where each node represents a state of the machine and each arc the execution of an event. Fig. 3.3b contains the state space of the *CoffeeM* (Fig. 3.3a), as computed by the *P*roB tool. The triangle represent a special root node, where the variables and constants of a machine have not yet been set. An animator lets the user navigate the state space by choosing the events to be fired. A model checker will systematically explore the state space, looking for various errors in the machine.

### 3.2.4 Machine Consistency

Invariants are supposed to hold initially and whenever variable values are changed by an event. Obviously, this does not hold a priori and, thus, needs to be proved. The corresponding proof obligation for every event is called *invariant preservation*, formally,

$$I(v) \ \wedge \ G(t,v) \ \wedge \ \boldsymbol{S}(t,v,v') \Rightarrow I(v') \quad . \tag{3.1}$$

For the *INITIALISATION* of a machine there is a special form of this proof obligation, without invariant and guard in the hypothesis. By proving *action feasibility* for an event,

$$I(v) \ \wedge \ G(t,v) \ \Rightarrow \ (\exists\, v' \cdot \boldsymbol{S}(t,v,v')) \quad ,$$

as well, we achieve that $\boldsymbol{S}(t,v,v')$ provides an after state whenever $G(t,v)$ holds. This means that the guard indeed represents the enabling condition of the event. For the *INITIALISATION* action feasibility is just $(\exists\, v' \cdot \boldsymbol{S}(t,v,v'))$ because it does not have a guard and the invariant cannot be assumed to hold before initialisation of a machine.

#### Enabledness

We can prove that machine $M$ is deadlock-free, that is, always some event is enabled to occur. Let $E_1(v)$, $E_2(v)$, ..., $E_n(v)$ be the events of machine $M$ without *INITIALISATION*, with parameter lists $t_1$, $t_2$, ..., $t_n$, and guards $G_1(t_1,v)$, $G_2(t_2,v)$, ..., $G_n(t_n,v)$. Machine $M$ is deadlock-free if some choice of parameter values makes one of the guards of the events true:

$$I(v) \ \Rightarrow \ (\exists\, t_1 \cdot G_1(t_1,v)) \ \vee \ (\exists\, t_2 \cdot G_2(t_2,v)) \ \vee \ \ldots \ \vee \ (\exists\, t_n \cdot G_n(t_n,v))$$

The Rodin tool does not support enabledness proof obligations at the moment. But *P*roB supports analysis of liveness properties and animation can show a deadlock (where all events except for the initialisation are disabled).

### 3.2.5 Machine Refinement

A machine $N$ can refine at most one other machine $M$. We call $M$ the *abstract machine* and $N$ a *concrete machine*. The state of the abstract machine is related to the state of the concrete machine by a *gluing invariant* $J(v,w)$ associated with the concrete machine $N$, where $v$ are the variables of the abstract machine and $w$ the variables of the concrete machine. The restriction that there is at most one abstract machine to a concrete machine permits us to store the gluing invariant $J(v,w)$ in the concrete machine. The full invariant of the concrete machine is the conjunction of the (full) invariant of the abstract machine and the gluing invariant. The invariants are accumulated during refinements. In particular, they are immediately available as hypotheses in proofs.

We call events of an abstract machine *abstract events* and those of a concrete machine *concrete events*. Abstract events are refined by concrete events. Each abstract event must be refined by at least one concrete event. The most common case is an abstract event

```
event $E(v)$
  any
    $t$
  when
    $G(t, v)$
  then
    $S(t, v)$
  end
```

(a) Abstract event $E(v)$

```
event $F(w)$ refines $E(v)$
  any
    $u$
  when
    $H(u, w)$
  with
    $W(t, v', u, v, w, w')$
  then
    $T(u, w)$
  end
```

(b) Concrete event $F(w)$

Figure 3.4: Event refinement

$E(v)$ refined by one concrete event $F(w)$. Let abstract event $E(v)$ and concrete event $F(w)$ be as specified in Fig. 3.4. Informally, concrete event $F(w)$ refines abstract event $E(v)$ if, whenever the gluing invariant $J(v, w)$ is true,

1. the guard of $F(w)$ is stronger than the guard of $E(v)$, and

2. for every possible execution of $F(w)$ there is a corresponding execution of $E(v)$ which simulates $F(w)$ such that the gluing invariant remains true after execution of both events.

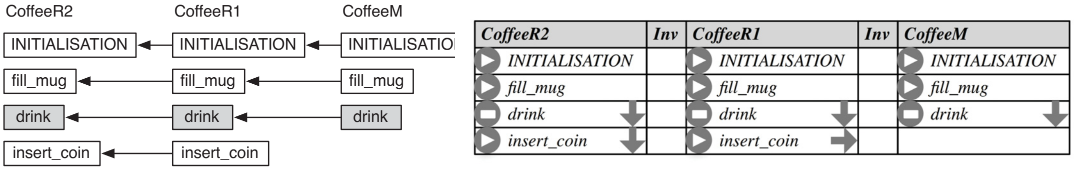
Superposition refinement [Bac89] is supported explicitly by way of *extending* events. For instance, in Fig. 3.19 some events carry the attribute `extends`. This means that all parameters, guards, and actions are copied literally from the abstract event. Note that the event $F(w)$ contains one more component $W(t, v', u, v, w, w')$ following the keyword `with`, called the *witnesses*. We return to its role in Section 3.2.7 below.

An event can be *split* in a refinement. This happens if more than one concrete event refines abstract event $E(v)$. A concrete event can also *merge* several abstract events into one concrete event. This is only permitted if the parameters of the merged abstract events agree on their types and their actions are identical. The latter restrictions have been introduced in order to keep the associated proof obligations simple. It is not possible for such a concrete event to extend the merged abstract events.

### Refinement Animation

To check whether the guard of a concrete event is stronger, we also need to animate the corresponding abstract machine. Fig. 3.5a shows a graphical visualisation (created with Brama) of an animation of the coffee dispenser model described earlier.[2] White boxes signal enabled events, grey boxes disabled events. As can be seen, all the refinement

[2] The drawing corresponds to the output generated by Brama. The original screenshot can be found in [HLP10].



| *CoffeeR2* | *Inv* | *CoffeeR1* | *Inv* | *CoffeeM* |
|---|---|---|---|---|
| ▶ *INITIALISATION* | | ▶ *INITIALISATION* | | ▶ *INITIALISATION* |
| ▶ *fill_mug* | | ▶ *fill_mug* | | ▶ *fill_mug* |
| ⊖ *drink* ↓ | | ⊖ *drink* ↓ | | ⊖ *drink* ↓ |
| ▶ *insert_coin* ↓ | | ▶ *insert_coin* → | | |

(a) in Brama

(b) Improved

Figure 3.5: Coffee dispenser refinement animation

levels are animated concurrently. Brama's representation also shows at a glance that whenever a refined event is enabled, then all of its ancestor events are also enabled.

The view underlying Fig. 3.5a is operational. It focuses solely on event execution. If we wanted to use it for analysing a formal model, we would need to add information. In particular, information about gluing invariants would be useful (Fig. 3.5b). For instance, it could be shown which event could violate the invariant. We have indicated more information that could be provided by the icons used in Fig. 3.5b: the icon “▶” indicates that an event is enabled; the icon “⊖” indicates that an event is not enabled; the icon “↓” indicates that a convergent event decreases the variant; the icon “→” indicates that an anticipated event does not increase the variant. Note, that this is not a purely cosmetic change: the animator must supply all necessary information.

### 3.2.6 Common Variables and Common Parameters

As far as animation and model checking are concerned, refinement introduces a new challenge: we no longer have just a single machine that needs to be animated as in Fig. 3.3b, but a series of machines, each with its own state.[3]

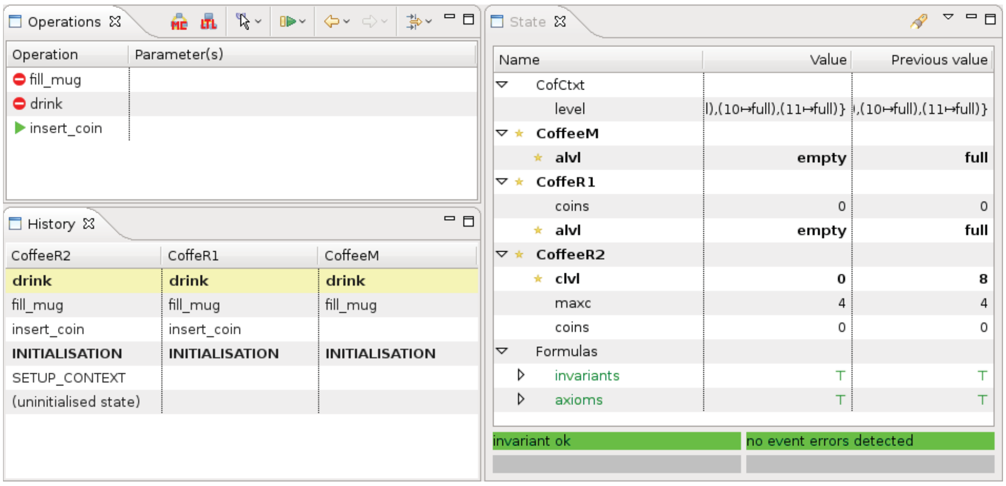
In order to check the gluing invariant, we need to access variables from various machines. This raises a new issue. In Section 3.2.5 we have simply assumed that all variables $v$ are refined by new variables $w$, and all parameters $t$ are refined by new parameters $u$. The variables $v$ and parameters $t$ “disappear” in the refinement. In practice, variables and parameters can be repeated in a refinement. Abstract machines and concrete machines can have variables in common, and abstract events and concrete events parameters. By convention, when repeating variables and parameters abstract and concrete counterparts are assumed to be equal.[4]

[3] Earlier versions of *P*roB avoided this problem by animating each refinement level separately, at the cost of not being able to check the gluing invariant and of less user feedback.

[4] Once a variable has disappeared in the course of several refinements it cannot reappear. The reason for this is that the equality cannot be established by means of a machine that does not contain the variable. Furthermore, invariants are accumulated in Event-B. So it is not possible to reintroduce a variable with a different meaning.

### Animation

For refinement animation of machines this means that we have basically two options: first, variables must be renamed in each machine and gluing invariants generated. This approach would allow us to visualise machines with deviating behaviour. A second approach would be to treat repeated variables internally as one, but we must ensure that the algorithm will still detect the errors and find a way to visualise them. After presenting the algorithm, we will explain in detail in Section 3.3.2 why we opt for the latter.



| Operation | Parameter(s) |
|---|---|
| fill_mug | |
| drink | |
| insert_coin | |

| CoffeeR2 | CoffeR1 | CoffeeM |
|---|---|---|
| **drink** | **drink** | **drink** |
| fill_mug | fill_mug | fill_mug |
| insert_coin | insert_coin | |
| **INITIALISATION** | **INITIALISATION** | **INITIALISATION** |
| SETUP_CONTEXT | | |
| (uninitialised state) | | |

| Name | Value | Previous value |
|---|---|---|
| CofCtxt | | |
| level | l),(10↦full),(11↦full)} | ,(10↦full),(11↦full)} |
| ★ **CoffeeM** | | |
| ★ **alvl** | **empty** | **full** |
| ★ **CoffeR1** | | |
| coins | 0 | 0 |
| ★ **alvl** | **empty** | **full** |
| ★ **CoffeeR2** | | |
| ★ **clvl** | **0** | **8** |
| maxc | 4 | 4 |
| coins | 0 | 0 |
| Formulas | | |
| invariants | T | T |
| axioms | T | T |

invariant ok

no event errors detected

Figure 3.6: ProB operations and state view after the trace *insert_coin*, *fill_mug*, *drink*

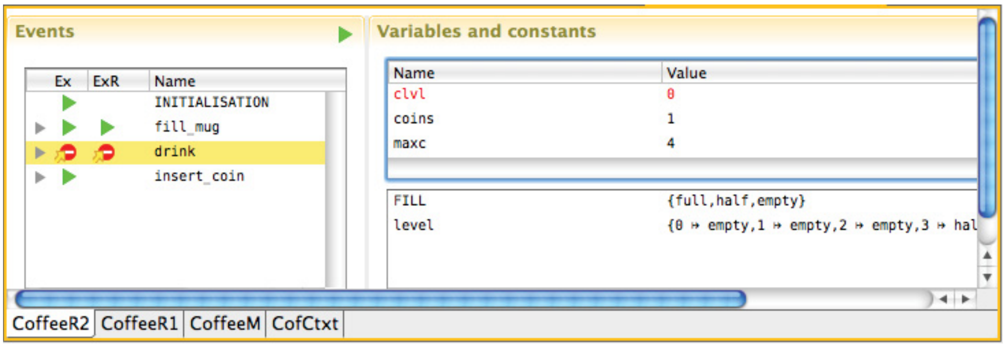
### Example

In the running example the machine *CoffeeM* of Fig. 3.3a and its refinement *CoffeeR1* of Fig. 3.19a have the variable *alvl* in common. Fig. 3.6 shows ProB animating the Coffee example. As can be seen in the newly developed hierarchical "State View", the variable *alvl* occurs twice, once in *CoffeeM* and once in *CoffeeR1*. We can also see that the variable *alvl* disappears in the refinement *CoffeeR2*. In Fig. 3.7 we show how the AnimB animator displays a state of multiple refinement-levels; each refinement level is given its own tab.

## 3.2.7 Refined Events and Witnesses

The predicate $W(t, v', u, v, w, w')$ denotes *witnesses*. Somewhat simplified, they link the abstract parameters $t$ and the abstract variables $v'$ to concrete parameters $u$ and variables and $w'$ (see also Fig. 3.9). Witnesses describe for each event separately how the refinement is achieved. Let $K(u, v, w, w') \;\widehat{=}\; I(v) \;\wedge\; J(v, w) \;\wedge\; H(u, w) \;\wedge\; \boldsymbol{T}(u, w, w')$.

We discuss splitting of events in more detail and comment below on the differences with merging events in a refinement. The proof obligations for concrete events are called



| Ex | ExR | Name |
|---|---|---|
| ▶ | | INITIALISATION |
| ▶ | ▶ | fill_mug |
| ⛔ | ⛔ | drink |
| ▶ | | insert_coin |

| Name | Value |
|---|---|
| clvl | 0 |
| coins | 1 |
| maxc | 4 |

| | |
|---|---|
| FILL | {full,half,empty} |
| level | {0 ↦ empty,1 ↦ empty,2 ↦ empty,3 ↦ hal |

Figure 3.7: AnimB view after *insert_coin,insert_coin, fill_mug, drink*

*guard strengthening*:

$$K(u,v,w,w') \;\wedge\; W(t,v',u,v,w,w') \;\Rightarrow\; G(t,v) \quad , \tag{3.2}$$

*action simulation*:

$$K(u,v,w,w') \;\wedge\; W(t,v',u,v,w,w') \;\Rightarrow\; \boldsymbol{S}(t,v,v') \quad , \tag{3.3}$$

and *invariant preservation*:

$$K(u,v,w,w') \;\wedge\; W(t,v',u,v,w,w') \;\Rightarrow\; J(v',w') \quad . \tag{3.4}$$

We have to prove *witness feasibility* in order to be able to add the witness predicate $W(t,v',u,v,w,w')$ to the premises in the proof obligations above:

$$K(u,v,w,w') \;\Rightarrow\; (\exists t,v' \cdot W(t,v',u,v,w,w')) \quad . \tag{3.5}$$

In general, witnesses would be required for all parameters $p$ of an event but when a parameter is repeated in a refined event, by convention, it is assumed to be equal to the corresponding abstract parameter. If a parameter is not repeated an explicit witness is required. (The Rodin tool creates the *default witness "true"* if none is specified in the latter case. This witness does not constrain the relationship between abstract and concrete parameters and variables. Hence, usually default witnesses are not sufficient to establish the refinement relationship.) For variables the rule when witnesses are needed is more complicated: whenever a variable $x$ that disappears occurs in a non-deterministic assignment in the abstract event, in the refined event a witness for the post-state variable $v'$ is required.

*Aside.* As described in [AH07], in order to verify that $F(w)$ refines $E(v)$ we have to prove

$$I(v) \;\wedge\; J(v,w) \;\wedge\; H(u,w) \;\wedge\; \boldsymbol{T}(u,w,w') \Rightarrow \exists t,v' \cdot G(t,v) \;\wedge\; \boldsymbol{S}(t,v,v') \;\wedge\; J(v',w') \quad . \tag{3.6}$$

In a proof of this statement we prefer to instantiate the existentially quantified parameters $t$ and variables $v'$ by expressions that can in some way be inferred from the premises. This idea is generalised to the witnesses used in Event-B. Witnesses are predicates that provide values to satisfy the existentially quantified conclusion of the statement.

Refinement according to (3.6) is established by proof obligations (3.2) to (3.5):

$$\begin{array}{ll}
 & (3.2)\ \wedge\ (3.3)\ \wedge\ (3.4) \\
\Rightarrow & \quad \{\ \text{predicate logic}\ \} \\
 & K(u,v,w,w')\ \wedge\ W(t,u,v,v',w,w')\ \Rightarrow\ G(t,v)\ \wedge\ \mathbf{S}(t,v,v')\ \wedge\ J(v',w') \\
\Rightarrow & \quad \{\ \text{instantiate } t,v' := t,v' \text{ in goal}\ \} \\
 & K(u,v,w,w')\ \wedge\ W(t,u,v,v',w,w')\ \Rightarrow\ \exists\, t,v' \cdot G(t,v)\ \wedge\ \mathbf{S}(t,v,v')\ \wedge\ J(v',w') \\
\Rightarrow & \quad \{\ t \text{ and } v' \text{ not free in } K(u,v,w,w')\ \} \\
 & K(u,v,w,w')\ \wedge\ (\exists\, t,v' \cdot W(t,u,v,v',w,w'))\ \Rightarrow\ \exists\, t,v' \cdot G(t,v)\ \wedge\ \mathbf{S}(t,v,v')\ \wedge\ J(v',w') \\
\Rightarrow & \quad \{\ \text{by (3.5)}\ \} \\
 & (3.6)
\end{array}$$

*Aside.* Two or more events can be merged in a refinement. In this case some restrictions apply for the abstract events being merged. Figure 3.8 shows a concrete event $F(w)$ that merges two abstract events $E_1(v)$ and $E_2(v)$. (It can be easily generalised to any finite number of merged events.) Merging of $E_1(v)$ and $E_2(v)$ is only defined if identically named parameters of $t_1$ and $t_2$ have identical types and the actions $S_1(t_1,v)$ and $S_2(t_2,v)$ are identical. Term $t_1 \cup t_2$ denotes the "union" of the parameter lists $t_1$ and $t_2$. Due to the restrictions we have imposed on the abstract events, only the guard strengthening proof obligation needs to be adapted to the merging case (with $K(u,v,w,w')$ as defined above):

```
event  $E_1(v)$
  any
    $t_1$
  when
    $G_1(t_1,v)$
  then
    $S_1(t_1,v)$
  end

event  $E_2(v)$
  any
    $t_2$
  when
    $G_2(t_2,v)$
  then
    $S_2(t_2,v)$
  end
```

(a) Abstract events $E_1(v)$ and $E_2(v)$

```
event  $F(w)$  refines  $E_1(v)$  $E_2(v)$
  any
    $u$
  when
    $H(u,w)$
  with
    $W(t_1 \cup t_2, v', u, v, w, w')$
  then
    $T(u,w)$
  end
```

(b) Concrete event $F(w)$ merging $E_1(v)$ and $E_2(v)$

Figure 3.8: Merging two events

$$K(u,v,w,w')\ \wedge\ W(t_1 \cup t_2, v', u, v, w, w')\ \Rightarrow\ G_1(t_1,v)\ \vee\ G_2(t_2,v) \quad .$$

Merging does not pose new challenges for animation, hence, we do not discuss it when presenting the animation algorithm. The brief presentation of event merging given here should suffice as evidence for this claim (and as justification for the omission of event merging in Section 3.3).

### Example

Event *fill_mug* in *CoffeeR2* contains the witness $x = level(clvl')$ for the abstract parameter $x$ of *fill_mug* in *CoffeeR1*. This means intuitively, that every execution of *fill_mug* in *CoffeeR2* corresponds to an execution of *fill_mug* in *CoffeeR1* with parameter $x$ set to $level(clvl')$. Event *fill_mug* in *CoffeeR1* must be enabled for $x = level(clvl')$ and the gluing invariant $alvl = level(clvl)$ must hold after executing the abstract and concrete event. Similarly, *drink* in *CoffeeR2* contains a witness $alvl' = level(clvl')$ for the abstract variable *alvl*. (Note, that it is just invariant @lvl in Fig. 3.19b with all variables primed.)

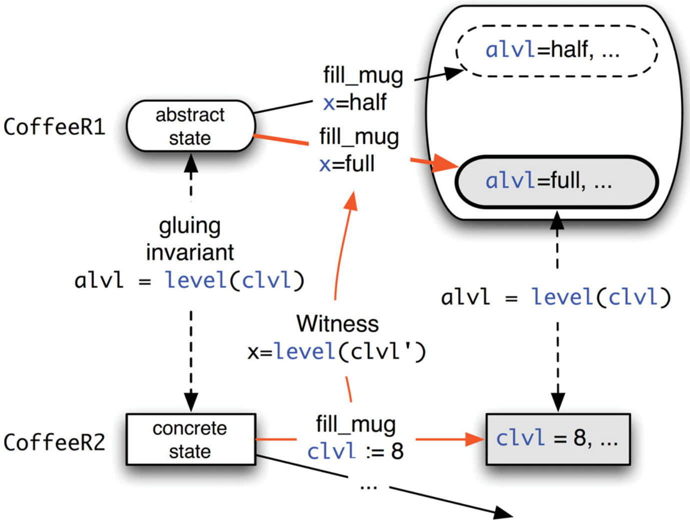


Figure 3.9: Witnesses and multi-level animation

### Animation in *ProB*

Witnesses are the key concept that makes refinement animation possible. Indeed, (trace) refinement animation and refinement checking in classical B (see Section 3.5) require for every concrete state to keep track of the *set* of all abstract states for which the gluing invariant holds. Only if this set becomes empty, have we found an error in the refinement. The size of state space necessary for simulation grows exponentially. In Event-B, by contrast, the witnesses pinpoint the states which have to satisfy the refinement relationship (see Fig. 3.9).

### 3.2.8 New Events, Convergence, and Anticipation

In the course of refinement, often *new events* $F(w)$ are introduced into a model. New events must be proved to refine the implicit abstract event "*skip*" that does nothing and is always enabled, that is, we have to prove *invariant preservation*:

$$K(u, v, w, w') \Rightarrow J(v, w') \quad .$$

Moreover, it may be proved that new events do not collectively diverge. This is done by means of *convergence* and *anticipation*. Convergence is verified by proving that a specified *variant* $Y(w)$ is bounded from below:

$$I(v) \wedge J(v, w) \wedge H(u, w) \Rightarrow Y(w) \geq 0$$

and is decreased by each new event

$$I(v) \wedge J(v, w) \wedge H(u, w) \wedge \boldsymbol{T}(u, w, w') \Rightarrow Y(w') < Y(w) \tag{3.7}$$

where we assume that the variant is an integer expression. (Instead of an integer expression also a finite set expression can be used.) We call events that satisfy these two proof obligations *convergent*.

An event is *anticipated* (to be convergent) if instead of (3.7) it satisfies the weaker condition (3.8):

$$I(v) \wedge J(v, w) \wedge H(u, w) \wedge \boldsymbol{T}(u, w, w') \Rightarrow Y(w') \leq Y(w) \quad . \tag{3.8}$$

Anticipated events can be used to prove convergence on a lexicographic order [ACM05] or just to delay convergence proofs [Hal09b]. Anticipated events can be refined by anticipated or convergent events, but must ultimately be refined by a convergent event. Convergent events can only be refined by convergent events, that is, they stay convergent. Fig. 3.10 shows a typical scenario of the use of convergent and anticipated events.[5] "Ordinary" events (events that are neither convergent nor anticipated) are shown as boxes □, anticipated events as ellipses ⬭, and convergent events as bold ellipses ◎. The new event $E_2$ in machine $M_2$ is convergent and stays convergent in machines $M_3$ to $M_5$. The new event $E_3$ in machine $M_2$ is anticipated and becomes convergent only in machine $M_5$. The new event $E_4$ in machine $M_3$ becomes convergent in machine $M_4$ "before" event $E_3$. The interest of anticipated events is mostly this possibility to carry out convergence proofs when it is most convenient, usually avoiding explicit specification of lexicographic variants.

Roughly speaking, if $X(v)$ is the variant of an anticipated event $E(v)$, and $Y(w)$ the variant of a convergent event $F(w)$ that refines $E(v)$, then $F(w)$ is convergent with respect to the lexicographic variant $(X(v), Y(w))$. For $X(v)$ we have verified that it is not increased by $E(v)$, that is, $X(v') \leq X(v)$. Because $F(w)$ refines $E(v)$ it also does not increase $X(v)$ (modulo the gluing invariant). In particular, if $F(w)$ leaves $X(v)$ unchanged, $X(v') = X(v)$, then $Y(w)$ is decreased, hence, $X(v') < X(v) \vee (X(v') = X(v) \wedge Y(w') < Y(w))$.

[5] See also Fig. 3.5b.

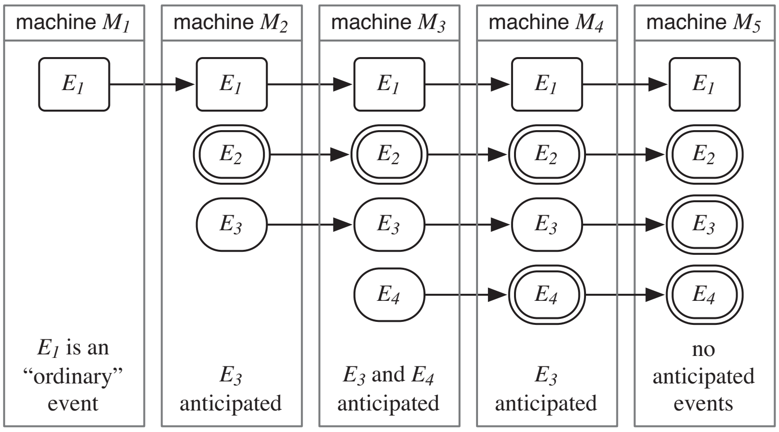


Figure 3.10: Convergent and anticipated events

**Example**

Event *insert_coin* in Fig 3.19a is anticipated in *CoffeeR1* and is then proven convergent in *CoffeeR2* (Fig 3.19b) by introducing an upper bound on the number of inserted coins.

### 3.2.9 Enabledness of Refined and New Events

The concrete machine $N$ is relatively deadlock-free with respect to the abstract machine $M$ if some concrete event is enabled whenever some abstract event is enabled. That is, the disjunction of the guards of all abstract events (without *INITIALISATION*) implies the disjunction of the guards of all concrete events (without *INITIALISATION*). This is equivalent to the claim that each abstract event (except for *INITIALISATION*) implies the disjunction of the guards of all concrete events (without *INITIALISATION*) . Whenever the abstract machine may continue by means of some event $E(v)$ with guard $G(t,v)$, the concrete machine may continue by means of at least one of the concrete events $F_1(w)$, $F_2(w), \ldots, F_n(w)$ with parameter lists $u_1, u_2, \ldots, u_n$, and guards $H_1(u_1,w), H_2(u_2,w), \ldots,$ $H_n(u_n,w)$,

$$I(v) \;\wedge\; J(v,w) \;\wedge\; G(t,v) \;\Rightarrow\; (\exists\, u_1 \cdot H_1(u_1,w)) \;\vee \ldots \vee\; (\exists\, u_n \cdot H_n(u_n,w)) \quad .$$

Neither the Rodin proof obligations generation nor *P*roB model-checking or animation support the notion of relative deadlock-freedom at the moment. *P*roB animation would show a deadlock, because first the concrete events are animated and then their abstractions if a solution is found.

## 3.3 Description of the Multi-Level Animation Algorithm

In this section we describe the validation and animation algorithm in detail. We point out in the presentation of the algorithm how it indicates problems with particular proof obligations. We also show how feedback to the user needs to be considered. Producing informative output from an animation with good performance is a challenge. For this reason the algorithm makes heavy use of *P*roB's existing functionality. In particular, *P*roB provides methods to find values for variables that satisfy predicates occurring in Event-B models.

Below we limit the discussion to animation; but the algorithm is identical for model checking: the model checker uses the same technique to determine the state space.

### 3.3.1 Preprocessing

The algorithm is applied to a particular refinement machine $M_i$ of a model. In a preprocessing step, all ancestor machines $M_0, \dots, M_{i-1}$ of $M_i$ are loaded and all contexts seen by $M_0, \dots, M_i$ are merged by collecting the declared constants and joining the axioms. The invariant is obtained by conjoining all invariants of $M_0, \dots, M_i$.

We transform each event of $M_i$ to an internal representation. The representation is outlined on the right hand side of Fig. 3.11. Usually the list of abstract events contains just one entry. If an event refines *skip* or belongs to the most abstract machine $M_0$, the list of abstract events is empty. If the event refines several events, it will contain all of those events.

E.g., if we animate *CoffeeR1*, the event *drink* has one abstract event, the event *drink* of *CoffeeM*, which itself has an empty list of abstract events. *insert_coin* would have no abstract events because it refines *skip*.

### 3.3.2 The Animation Algorithm

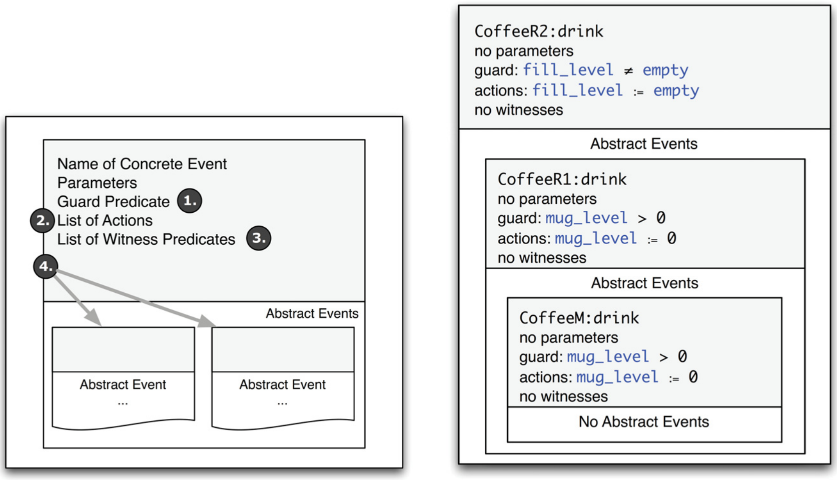


Figure 3.11: Illustration of the algorithm and one particular event structure

The animator executes events depending on the current state of a model. It maintains a state consisting of all constants of the seen contexts as well as all variables of the machines $M_0, \ldots, M_i$.

In a first step the animator tries to find values for the constants that satisfy all axioms. Subsequently, the animator executes in each step an event of the most concrete machine $M_i$; then it executes the corresponding abstract events from the concrete event to the most abstract event.

When all of this has been done, the animator is ready for the next step.

The algorithm to animate a particular event works as follows (item numbers correspond to those of Fig. 3.11, left hand side):[6]

1. Search for possible values for the parameters by evaluating its guard. If no values are found, the event is disabled.
2. Execute each action by evaluating the respective before-after predicate. If no solution is found, report an error. The possible reasons for a failing action are:
   a. The predicate $P$ of an action $v :\mid P$ is not satisfiable or the set $S$ of an action $v :\in S$ is empty. Both cases show violations of the event feasibility proof obligation.
   b. The new value $v'$ of a variable $v$ is constrained by a witness of a refined event (see step 3), but the abstract action cannot assign a corresponding value to $v'$. This indicates a violation of the action simulation proof obligation.
3. For each witness evaluate its predicate and try to find values for the witnessed variable. If no value is found for a witness, report an error, because a witness should have at least one solution (by the witness feasibility proof obligation).
4. a. If the list of abstract events is empty, a complete solution has been found for this event that leads to a new state. It consists of the values newly assigned by the actions plus the variables unchanged by the actions.
   b. If there are one or more abstract events, choose one non-deterministically and evaluate its guard like in step 1. If it evaluates to true, continue recursively with step 2, otherwise try the next event.
   If no guard evaluates to true, report an error, because the guard of the refinement is weaker than that of the abstract event (violation of the guard strengthening proof obligation).

All four steps can be non-deterministic and we generate all solutions (limited to a maximum number) with backtracking.

### Common Variables and Parameters

As mentioned above we have basically two options for dealing with common variables between a model and its refinement. One is to treat them as distinct variables $x_a$ and $x_c$ and to add the implicit gluing invariant $x_a = x_c$. If an abstract event $x := 1$ is now erroneously refined by $x := 2$, the animator can visualise the error by showing the post state $x_a = 1, x_c = 2$ and indicating that the invariant $x_a = x_c$ is violated.

But this approach has a fundamental problem as the following example shows. Let the action of the abstract event be $x :\in \{1, 2\}$ and the corresponding action of the refined

[6] With respect to animation *INITIALISATION* is not treated differently from any other events (see left of Fig. 3.11) except that it is enforced to occur once upon start of an animation.

event $x := 1$. Our algorithm would find two possible post states, $x_a = 1, x_c = 1$ and $x_a = 1, x_c = 2$. The first one is valid, the second one violates the implicit gluing invariant. Thus *P*roB would wrongly show an error for a correct refinement.

A solution to prevent the spurious error is to add implicit witnesses $x'_a = x'_c$ for all common variables. Then after assigning $x'_c = 1$, the evaluation of $x'_a = x'_c$ results in $x'_a = 1$. The abstract assignment $x :\in \{1, 2\}$ is then just a check $x'_a = 1 \in \{1, 2\}$. We still would find an error for an invalid concrete action (like $x := 3$), namely a violation of the action simulation proof obligation in step 2a. But adding the implicit witness has the effect that the algorithm never finds solutions with different abstract and concrete values. So there is no need to distinguish between them in the animation. The same applies to common parameters.

A slightly different solution would be to add implicit witnesses only for those variables which are assigned non-deterministically in the abstract event. This would allow us to generate a state which violates the gluing invariant that we could present the user in case of deterministic assignments. For non-deterministic assignments we still have to indicate a violation of the action simulation proof obligation in the event. It is not clear to us if the presentation of an invalid result state is easier to understand than the explanation of an error found in the event. So there is no clear benefit of this approach, but the downside would be a less consistent algorithm and a larger representation of a state by duplicating variables. Thus we have chosen the approach where we do not distinguish between common variables.

### Example

Fig. 3.12 shows a detailed example run for the event *fill_mug* in the refinement *CoffeeR2* to demonstrate the animation algorithm. The table displays what is happening for each refinement level and step of the algorithm. The last column contains an entry if new values for variables have been found by evaluating the predicates.

We have chosen an arbitrary state where *fill_mug* is enabled as a starting point: $alvl = empty$, $clvl = 0$, $coins = 2$, $maxc = 4$. The example shown leads to a new state $alvl = full$, $clvl = 9$, $coins = 1$, $maxc = 4$. The variable *maxc* has not been modified by any action, so its value is just copied to the new state.

Note that the evaluation of action @ffl (*CoffeeR2*, step 2) chooses the value 9 of $clvl'$ non-deterministically, in total there are 4 different possibilities, leading to 4 different states with $clvl \in 8\,..\,11$.

### Animation of Convergent and Anticipated Events

If we have successfully found a possible event leading from one state to another, we can easily check if the convergence criteria are satisfied. The principle is quite simple: for each convergent event of an animated model, we check if the variant $V$ is decreased and non-negative by the predicates $V > V'$ and $V \geq 0$ resp. $V \supset V'$ when the variant is a set. If the event refines another convergent event, we omit the test: in the lexicographic order constructed by refinement, events that have been shown to decrease a variant in an

| Refinement | Step | | found values |
|---|---|---|---|
| *CoffeeR2* | 1. | evaluate the guards to see whether the event | |
| | | is enabled: | |
| | | @gc2: $coins > 0 \equiv 2 > 0 \equiv \top$ | |
| | | @ml: $clvl \in level^{-1}[\{empty\}] \equiv 0 \in 0\,..\,2 \equiv \top$ | |
| *CoffeeR2* | 2. | action @delc2: $coins' = coins - 1 \equiv coins' = 2 - 1$ | $coins' = 1$ |
| | | action @ffl: $clvl' \in level^{-1}[\{full\}] \equiv clvl' \in 8\,..\,11$ | $clvl' = 9$ |
| *CoffeeR2* | 3. | witness for the abstract parameter $x$: | |
| | | $x = level(clvl') = level(9) = full$ | $x = full$ |
| *CoffeeR2* | 4. | check the guard of the abstract event: | |
| | | @gc: $coins > 0 \equiv \top$ | |
| *CoffeeR1* | 2. | action @delc: $coins' = coins - 1 \equiv 1 = 2 - 1 \equiv \top$ | |
| *CoffeeR1* | 3. | (no witness to check) | |
| *CoffeeR1* | 4. | check the guards of the abstract event: | |
| | | @g0: $alvl = empty \equiv empty = empty \equiv \top$ | |
| | | @g1: $x \neq alvl \equiv full \neq empty \equiv \top$ | |
| *CoffeeM* | 2. | action @a1: $alvl' = x$ | $alvl' = full$ |
| *CoffeeM* | 3. | (no witness to check) | |
| *CoffeeM* | 4. | no more abstract events, stop | |

Figure 3.12: Example run of the refinement algorithm for event *fill_mug*

abstraction of some concrete machine may increase the variant of that concrete machine. Similarly, we can check anticipated events (with $V \geq V'$ and $V \geq 0$ resp. $V \supseteq V'$), but we cannot omit the test if an event is a refinement of an anticipated event. Violations of specified variants are treated by *P*roB similarly to guard strengthening violations as shown in Fig. 3.13.

### Animating Only a Part of the Refinement Chain

Above we presumed that the user wants to animate a refinement $M_i$ and all its ancestors $M_0, \ldots, M_{i-1}$. But we also permit the user to limit the animation to the refinements between $M_i$ and an "upper" refinement $M_k$ with $0 \leq k \leq i$ instead of $M_0$. Then variables of not animated models and parts of predicates that contain references to those variables will be removed, resulting in weaker predicates.

Such a reduction of the animated refinement levels usually leads to smaller representation of states and allows the user to focus on the parts of the model he is currently interested in. However, the downside is that the limitation might hide some errors or introduce spurious errors:

- The gluing invariant can refer to abstract variables, weakening it means that some invariant violations are not detected anymore.
- Weakened witnesses can result in finding values for abstract parameters or vari-

ables that were not possible with the original predicates. Those values might cause other errors (e.g. violation of the simulation proof obligation).

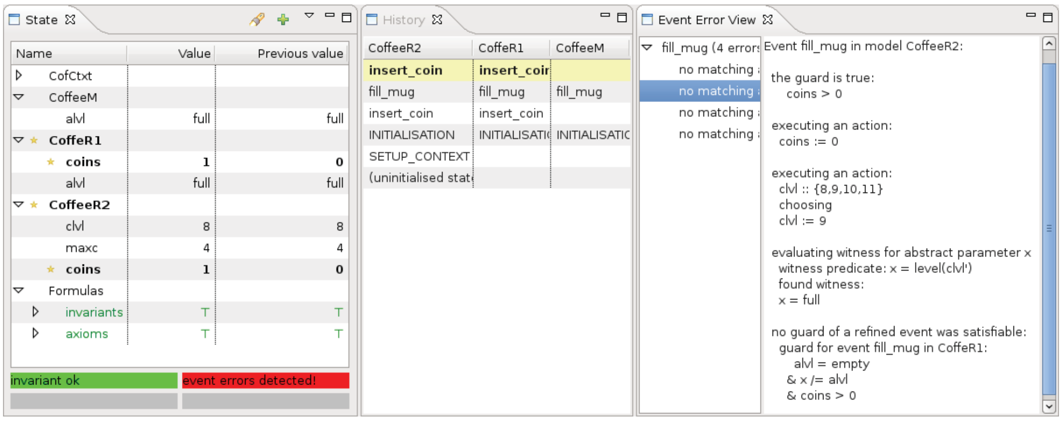
If we limit the animation to a single refinement ($k = i$) the animation behaves like that of *P*roB without multi-level support.

## 3.4 Refinement-Validation with *P*roB

Refinement animation can be used to validate models. We present some specific problems that can be analysed by animation and discuss a selection of case studies to which it has been applied.

### 3.4.1 Detection of Specific Problems

Below we show on various modified versions of the coffee model (Fig. 3.3a) and its refinements (Fig. 3.19), how the new multi-level animation algorithm allows *P*roB to detect a variety of refinement errors. Note that in contrast to AnimB and Brama, *P*roB is also a model checker that can systematically detect refinement errors.



| Name | Value | Previous value |
|---|---|---|
| CofCtxt | | |
| CoffeeM | | |
| alvl | full | full |
| CoffeR1 | | |
| coins | 1 | 0 |
| alvl | full | full |
| CoffeeR2 | | |
| clvl | 8 | 8 |
| maxc | 4 | 4 |
| coins | 1 | 0 |
| Formulas | | |
| invariants | T | T |
| axioms | T | T |

invariant ok | event errors detected!

| CoffeeR2 | CoffeR1 | CoffeeM |
|---|---|---|
| insert_coin | insert_coir | |
| fill_mug | fill_mug | fill_mug |
| insert_coin | insert_coin | |
| INITIALISATION | INITIALISATI | INITIALISATIC |
| SETUP_CONTEXT | | |
| (uninitialised stat | | |

```
fill_mug (4 errors
    no matching
    no matching
    no matching
    no matching
```

```
Event fill_mug in model CoffeeR2:

 the guard is true:
    coins > 0

 executing an action:
   coins := 0

 executing an action:
   clvl :: {8,9,10,11}
   choosing
   clvl := 9

 evaluating witness for abstract parameter x
   witness predicate: x = level(clvl')
   found witness:
   x = full

 no guard of a refined event was satisfiable:
   guard for event fill_mug in CoffeR1:
      alvl = empty
    & x /= alvl
    & coins > 0
```

Figure 3.13: Violation of guard strengthening (*P*roB)

#### Guard Weakening

If we remove the guard @ml from the event *fill_mug* in *CoffeeR2*, we violate the guard strengthening proof obligation. As can be seen in Fig. 3.13, *P*roB's model checker using our new algorithm detects this problem straightaway (case 4a of our algorithm), leading us to a state where *fill_mug* is enabled in *CoffeeR2* but not in the abstract machines.

#### Witness Disables Abstract Guard

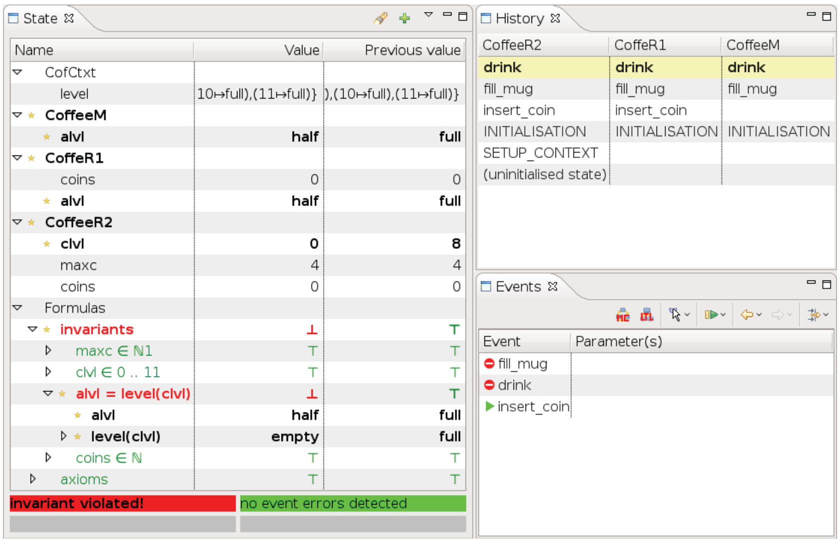
A similar error message appears if we keep the guards as they are, but inject an error in the witness. E.g, when using $x = empty$ as witness for *fill_mug*, *P*roB detects that there is a solution for the witness, but that the witness does not enable the abstract event.

### Witness Not Feasible

Next, let us use the witness $x = level(clvl') \wedge x = empty$ for event *fill_mug*. Here case (3) of our algorithm detects an error for *fill_mug* (after executing *insert_coin*), and ProB displays the error message: "No solution found for witness of the abstract parameter x in event CoffeeR2:fill_mug". The animator AnimB does not detect this error (but it did detect the previous two errors).

### Witness Violates Invariant

Finally, we try to specify the witness $alvl' \in \{empty, half\} - \{alvl\}$ for the event *drink*, which does not guarantee that the abstract event will satisfy the gluing invariant. As can be seen in Fig. 3.14, ProB finds an invariant violation error ($alvl = level(clvl)$ is false) directly after the *drink* event.



State

| Name | Value | Previous value |
|---|---|---|
| CofCtxt | | |
| level | 10↦full),(11↦full)} | ),(10↦full),(11↦full)} |
| CoffeeM | | |
| alvl | half | full |
| CoffeR1 | | |
| coins | 0 | 0 |
| alvl | half | full |
| CoffeeR2 | | |
| clvl | 0 | 8 |
| maxc | 4 | 4 |
| coins | 0 | 0 |
| Formulas | | |
| invariants | ⊥ | T |
| maxc ∈ ℕ1 | T | T |
| clvl ∈ 0 .. 11 | T | T |
| alvl = level(clvl) | ⊥ | T |
| alvl | half | full |
| level(clvl) | empty | full |
| coins ∈ ℕ | T | T |
| axioms | T | T |

invariant violated!

no event errors detected

History

| CoffeeR2 | CoffeR1 | CoffeeM |
|---|---|---|
| drink | drink | drink |
| fill_mug | fill_mug | fill_mug |
| insert_coin | insert_coin | |
| INITIALISATION | INITIALISATION | INITIALISATION |
| SETUP_CONTEXT | | |
| (uninitialised state) | | |

Events

| Event | Parameter(s) |
|---|---|
| fill_mug | |
| drink | |
| insert_coin | |

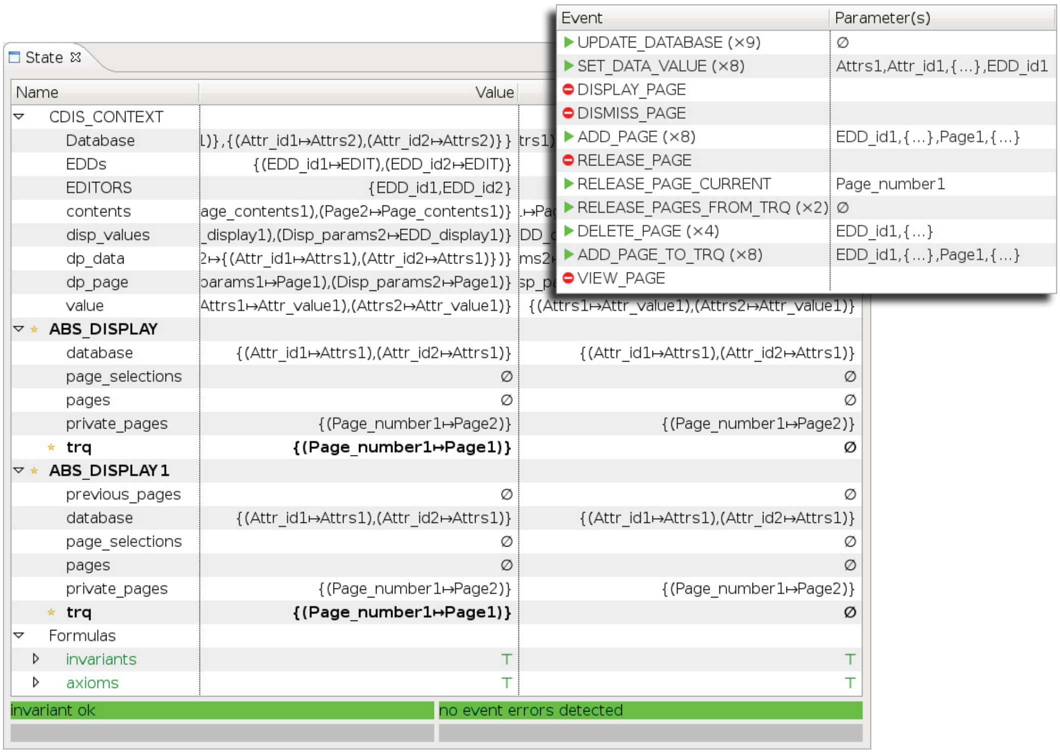
Figure 3.14: Violation of gluing invariant (ProB)

Note that, AnimB detects an error in the model, but only later when trying to execute the *fill_mug* event after the erroneous *drink* event.

In practice, validation by animation complements the proof-based methodology of core Event-B. Corresponding methodological benefits of using animation of Event-B models are discussed in more detail in [HL09].

## 3.4.2 Application to Case Studies

We have successfully applied the new multi-level animation of ProB on a two-level model of SAP service choreographies [WKR+09]. We have also tested the tool on the CDIS air traffic control case study carried out in the EU project Rodin. Figure 3.15 contains a screenshot of the first two levels; we have successfully animated all 7 levels of the full model concurrently.



| Event | Parameter(s) |
|---|---|
| ▶ UPDATE_DATABASE (×9) | ∅ |
| ▶ SET_DATA_VALUE (×8) | Attrs1,Attr_id1,{...},EDD_id1 |
| ⊖ DISPLAY_PAGE | |
| ⊖ DISMISS_PAGE | |
| ▶ ADD_PAGE (×8) | EDD_id1,{...},Page1,{...} |
| ⊖ RELEASE_PAGE | |
| ▶ RELEASE_PAGE_CURRENT | Page_number1 |
| ▶ RELEASE_PAGES_FROM_TRQ (×2) | ∅ |
| ▶ DELETE_PAGE (×4) | EDD_id1,{...} |
| ▶ ADD_PAGE_TO_TRQ (×8) | EDD_id1,{...},Page1,{...} |
| ⊖ VIEW_PAGE | |

State

| Name | Value | |
|---|---|---|
| CDIS_CONTEXT | | |
| Database | l)},{(Attr_id1↦Attrs2),(Attr_id2↦Attrs2)}} | trs1) |
| EDDs | {(EDD_id1↦EDIT),(EDD_id2↦EDIT)} | |
| EDITORS | {EDD_id1,EDD_id2} | |
| contents | age_contents1),(Page2↦Page_contents1)} | .↦Pa |
| disp_values | _display1),(Disp_params2↦EDD_display1)} | DD_ |
| dp_data | 2↦{(Attr_id1↦Attrs1),(Attr_id2↦Attrs1)})} | ms2 |
| dp_page | params1↦Page1),(Disp_params2↦Page1)} | sp_p |
| value | Attrs1↦Attr_value1),(Attrs2↦Attr_value1)} | {(Attrs1↦Attr_value1),(Attrs2↦Attr_value1)} |
| **ABS_DISPLAY** | | |
| database | {(Attr_id1↦Attrs1),(Attr_id2↦Attrs1)} | {(Attr_id1↦Attrs1),(Attr_id2↦Attrs1)} |
| page_selections | ∅ | ∅ |
| pages | ∅ | ∅ |
| private_pages | {(Page_number1↦Page2)} | {(Page_number1↦Page2)} |
| **trq** | **{(Page_number1↦Page1)}** | **∅** |
| **ABS_DISPLAY1** | | |
| previous_pages | ∅ | ∅ |
| database | {(Attr_id1↦Attrs1),(Attr_id2↦Attrs1)} | {(Attr_id1↦Attrs1),(Attr_id2↦Attrs1)} |
| page_selections | ∅ | ∅ |
| pages | ∅ | ∅ |
| private_pages | {(Page_number1↦Page2)} | {(Page_number1↦Page2)} |
| **trq** | **{(Page_number1↦Page1)}** | **∅** |
| Formulas | | |
| invariants | T | T |
| axioms | T | T |

invariant ok | no event errors detected

Figure 3.15: Animating the Rodin CDIS case study

Another case study was a complete development of the quicksort algorithm in Event-B, consisting of ten machines and two contexts. We have successfully animated and model-checked all the ten levels concurrently. Animation showed how the algorithm "works" at different abstraction levels. This is valuable for explaining an otherwise static model of an algorithm.

We have also successfully animated concurrently 14 levels of an elevator model solution by ETH Zürich. This has uncovered a potential problem in the model, namely that starting at a certain refinement level, the lift is no longer able to move (but the doors can be opened and closed and the buttons can be pressed; so there is no deadlock in the conventional sense).

## 3.5 Comparison with Trace Refinement and Empirical Evaluation

### 3.5.1 Comparing with Trace Refinement Checking

An important related work is the refinement checking algorithm in [LB05]. This algorithm checks trace refinement between the abstract model and the refined model, i.e., it checks that every trace of events of the refined model is also a valid trace of the abstract model.

For example, for our running *Coffee* example, all the possible traces of the abstract model can be deduced from the state space in Figure 3.3. If the refined model can perform the trace *INITIALISATION*, *drink*, then the algorithm from [LB05] would detect an error, as this trace cannot be performed by the abstract model *CoffeeM*.

The main differences between our new multi-level validation algorithm and [LB05] can be summarised as follows:

- As the algorithm from [LB05] works on the level of traces, it can be applied to other formalisms, such as CSP [Hoa85]. It can even be used to check refinement relations between CSP and classical B [Abr96] models.
- The tools for classical B and Event-B generate proof obligations for forward refinement, which is not complete. I.e., there are systems which are related by trace refinement but where no forward refinement can be established [HHS86]. The new algorithm in this paper adheres to the forward refinement methodology of Event-B, whereas [LB05] checks trace refinement.
- The algorithm of [LB05] checks the refinement relation between two models; our new algorithm checks an entire refinement chain in one go (but restricted by the state space of the most concrete model).
- The algorithm of [LB05] computes the state space of the refined model on-the-fly, but it has to compute the entire state space of the abstract model *before* starting the refinement checking (for the running *Coffee* example, this corresponds to Figure 3.3). The new algorithm in this paper performs the entire validation on-the-fly. In particular, only the relevant parts of the state space of the abstract model(s) are computed. We will illustrate this in the empirical results below.
- The algorithm of [LB05] performs the refinement checking depth-first; the new algorithm can use any of *P*roB's search methods (depth-first, breadth-first and the default mixed depth-first/breadth-first search). Especially for large state spaces, this increases the likelihood of detecting errors; see [Leu08].
- The algorithm of [LB05] computes the state space of the abstract model independently of the refined model and thus does not check the gluing invariant. Furthermore, in its current form, it cannot make use of Event-B's witnesses and hence has to keep track of *sets* of states in the abstract model. Our algorithm makes use of the witnesses, which often allows us to only relate a single abstract state to a refined state. It also allows us to relate parameters of the refined event to parameters of the abstract event. E.g., in our running example, *fill_mug* has no parameter in *CoffeeR2*, while it has the parameter $x$ in *CoffeeR1* and *CoffeeM*; the witness $x = level(clvl')$ in *CoffeeR2* tells us which parameter value to use in the abstract models. The algorithm of [LB05] was devised in a setting of CSP and classical B, where parameters of events or operations cannot be refined. E.g., in the running example it would

| Benchmark | Multi-level | Multi-level<br>with symmetry [LM07] | Trace-Refinement [LB05] |
|---|---|---|---|
| **Consistency Checking** | | | |
| *Scheduler0* | 1930 ms | 50 ms | 1930 ms |
| **Refinement Checking** | | | |
| *Scheduler1* | 8760 ms | 70 ms | 1930 ms + 7260 ms |
| *Scheduler1_err* | 360 ms | 30 ms | 1930 ms + *16090 ms |
| *Scheduler1_terr* | 10 ms | 20 ms | 1930 ms + 60 ms |

*: no error found, as gluing invariant not checked

Table 3.1: Comparing trace-refinement checking and multi-level validation

report a counter-example (*INITIALISATION*,*fill_mug*) for *CoffeeR2*, unless we tell it to ignore all parameters in traces.

To evaluate the practical performance difference between the two algorithms, we have adapted the scheduler example from [LB05] (and originally from [LPU02]) for Event-B. The abstract model *Scheduler0* schedules a certain number of processes (which we have fixed to 6 in the experiments), and keeps them in the sets *waiting*, *ready* and *active*. At most one process can be active at any one time. In the refinement *Scheduler1* there is a queue for the ready processes (the abstract model just keeps them in a set). We have also generated two incorrect refinements: in *Scheduler1_err* the event *swap_ready* is not shifting the indexes in the queue and in *Scheduler1_terr* there is a missing guard in the event *new*.

The experimental results are summarised in Table 3.1. The times are in milliseconds, and do not include the time to load the models. The experiments were run on a MacBook Pro with a 3.06 GHz Core2Duo processor, and *P*roB 1.3.2 compiled with SICStus Prolog 4.1.2. In essence, we can draw the following conclusions.

*Scheduler1*: there is no big performance difference here between the multilevel validation and the trace checking. However, the multilevel validation also checks the gluing invariant. Furthermore, symmetry reduction [LM07, SL08] can be applied; which in this case dramatically reduces the model checking time by a factor of 38.6. We explain why symmetry can be more easily exploited with our multi-level algorithm further below in subsection 3.5.3.

*Scheduler1_err*: Here we see that the trace refinement checking does not catch the error introduced in the model, as the error only manifests itself in the violation of the gluing invariant. In this case, this leads to a big difference in performance, as both the abstract and refined state space are fully explored by the trace refinement check.

*Scheduler1_terr*: Here, the trace refinement checking does catch the error, as this time it does result in a trace of the refined model which cannot be mimicked by the abstract model. Note, however, that there is a big performance difference, as the algorithm from [LB05] has to explore the full state space of the abstract model (even though only a small part of it was needed to find the counterexample).

### 3.5.2 Application to other Formalisms

In general we think the algorithm can be adapted to formalisms which support a concept of refinement similar to that of Event-B. In the following we have a closer look at classical B and ASM which have many concepts in common with Event-B.

#### Classical B

An obvious candidate for adapting the presented algorithm is classical B [Abr96]. Like Event-B, classical B has machines with variables and invariants on them. Operations can change the values of variables by *generalised substitutions*. Unlike events in Event-B operations can have preconditions.

Lets assume an abstract B machine with invariant $I$ and an operation with precondition $P_A$ and substitution $S_A$. Its refinement has a gluing invariant $J$ and an operation with precondition $P_R$ and substitution $S_R$. Let $[S]P$ denote the weakest precondition that ensures that an execution of $S$ results in a state satisfying $P$. A refinement must fulfil two conditions:

- Starting from a state where $I, J$ and $P_A$ hold, for each execution of $S_R$ there must exists a corresponding execution of $S_A$ such that $J$ holds in the resulting state. Formally this is expressed by $I \wedge J \wedge P_A \Rightarrow [S_R]\neg\,[S_A]\neg\,J$.
- The precondition of the refined operation must be weaker than the abstract operation's precondition, formally $I \wedge J \wedge P_A \Rightarrow P_R$.

Whereas the presented algorithm follows a strict bottom-up (from refinement to abstract machine) approach, animating classical B could probably require a combination of a top-down and bottom-up approach: First we evaluate $P_A$ on a state, then we check if $P_R$ also holds. Then we can execute $S_R$ to find the concrete values of a new state. Afterwards we check if there is a corresponding abstract substitution that fulfils $J$.

In future work we will further investigate this approach. There are still several open technical issues (e.g., how to give the user helpful feedback in case of errors) as well as fundamental issues (e.g., have we thought of all possible situations regarding preconditions?). We expect that witnesses or a similar concept will have to be introduced in classical B as well. Preconditions could be handled similarly to guards, except for the order in which preconditions have to be evaluated, as indicated above.

#### ASM

ASM provides a very generic framework for refinement [Bör03]. Unlike Event-B, it allows the refinement of an arbitrary number of abstract operations by an arbitrary number of concrete operations. Thus we do not see how we could apply this algorithm to that liberal form of refinement.

Of course, using a restricted form of ASM refinement could make our approach applicable with only small modifications. The only difference would be that ASM rules can be enabled simultaneously whereas Event-B events cannot.

### 3.5.3 Symmetry Reduction for Multi-Level Validation

The reason symmetry reduction cannot be (easily) applied when using the refinement checking algorithm from [LB05], is that the abstract and refined model are explored separately. Hence, symmetry reduction would have to be applied to these two steps separately as well. A big problem now is that the canonical form chosen for the abstract model may be incompatible with the one chosen for the refined model, leading to erroneous counterexamples and/or to real counterexamples not being found.

Let us look at the example in Figure 3.16. The abstract model *ProcSet* contains a variable $x$, which stores process identifiers taken from the carrier set *Proc*, declared in the context *PCtxt*. The event *new* can be used to add elements to $x$, and the event *del* to remove elements from $x$. In the refined model *ProcSeq* the process identifiers are stored in a sequence $q$. The model also stores the size of the sequence as a separate variable. Observe that the elements of *Proc* can be interchanged, leading to symmetries in the state space. Notably, all successor states of the initial state $x = \varnothing$ are symmetric in the abstract model. Similarly, all successor states of the initial state $q = \varnothing, n = 0$ are symmetric in the refined model. Let us assume that we have two process identifiers: $Proc = \{P1, P2\}$. The top of Figure 3.17 contains the initial state and the successor states for the abstract and refined model. The symmetry reduction algorithm from [SL08] will now chose one representative per equivalence class. Say the symmetry reduction chooses the successor of $new(P1)$ for the canonical representative for the abstract model, but for some reason (e.g., due to the other data structure used) chooses the successor of $new(P2)$ in the refined model. As we can see in the bottom of Fig. 3.17, in this case $new(P1), del(P2)$ or $new(P2), del(P2)$ are traces of the symmetry reduced refined model, which cannot be mimicked by the symmetry reduced abstract model.[7] Hence, the trace refinement checking algorithm would erroneously report that refinement fails. In multilevel animation, the canonical form is computed on the combined state of the abstract and refined model, and as such respects the relationship between them. This is illustrated in Fig. 3.18, where one representative is chosen for the combined abstract and refined state.

In Table 3.2 we show how effective symmetry reduction can be for multi-level validation. We have run exhaustive multi-level model checking for the refined *ProcSeq* model of Fig. 3.16 with and without symmetry reduction. We have used the symmetry reduction technique from [SL08].

[7] The permutation flooding technique [LBST07] could probably be made to work more easily, as it explicitly adds "permutation" actions to the state space.

```
context PCtxt
sets Proc
end
```

```
machine ProcSet
sees PCtxt
variables x
invariants

   @xproc x ⊆ Proc

events
   event INITIALISATION
      begin
         @i x := ∅
      end
   event new
      any p when
         @newp p ∉ x
      then
         @incx x := x ∪ {p}
      end
   event del
      any p when
         @pinx p ∈ x
      then
         @subx x := x \ {p}
      end
end
```

```
machine ProcSeq refines ProcSet
sees PCtxt
variables q n
invariants
   @nnat n ∈ ℕ
   @qinj q ∈ 1 .. n ↣ Proc
   @glue ran(q) = x
events
   event INITIALISATION
      begin
         @iq q, n := ∅, 0
      end
   event new refines new
      any p when
         @newp p ∉ ran(q)
      then
         @incq q, n := q <+ {n+1 ↦ p}, n+1
      end
   event del refines del
      any p when
         @pinq q[{n}] = {p}
      then
         @subq q, n := n ⩤ q, n−1
      end
end
```

Figure 3.16: A model symmetric in the set of processes *Proc*

### 3.5.4 More Experimental Results

The experiments below were again run on a MacBook Pro with a 3.06 GHz Core2Duo processor, and *P*roB 1.3.2 compiled with SICStus Prolog 4.1.2.

To evaluate the cost of treating multiple levels at once we have run our algorithm on an Event-B model for an elevator by ETH Zürich with an abstract machine and 13 refinements. We have measured the time to check 100 states for each of the 14 machines. The results are summarised in Table 3.3. Note that the state spaces for levels 0 to 4 have less than 100 nodes each, hence the full state space was explored in the given time. Starting at level 5, the state space has more than 100 states; we have stopped the algorithm at 100 nodes to be able to compare the time required to treat the same number of nodes. Also note that, we have always animated all levels above the given level (i.e., when validating the level 13 model, we have also validated levels 0–12). As we can see, there is no simple linear dependence between the validation time and the number of levels animated. However, up to level 8 the runtime seems to increase linearly with the number of levels animated. Then there is a big jump at level 9 (which introduces 6 new variables of type total function), after which the runtime seems to increase again linearly (albeit with a steeper slope).

We have conducted a similar experiment for the Quicksort model from [Hal09a], for

| Cardinality of Proc | States | Validation Time | States | Validation Time |
|---|---|---|---|---|
| | (without symmetry) | | (with symmetry) | |
| 1 | 3 | 0.001 s | 3 | 0.002 s |
| 2 | 6 | 0.004 s | 4 | 0.003 s |
| 3 | 17 | 0.012 s | 5 | 0.006 s |
| 4 | 66 | 0.049 s | 6 | 0.009 s |
| 5 | 327 | 0.278 s | 7 | 0.013 s |
| 6 | 1,958 | 1.618 s | 8 | 0.018 s |
| 7 | 13,701 | 11.325 s | 9 | 0.025 s |
| 8 | 100,001 | 90.930 s | 10 | 0.033 s |

Table 3.2: Experimental results for multi-level validation with symmetry reduction

| Level | Validation Time (100 states) | Cost for extra level | Time per level |
|---|---|---|---|
| 0 | 0 ms | | |
| 1 | 0 ms | | |
| 2 | 20 ms | | |
| 3 | 30 ms | | |
| 4 | 80 ms | | |
| 5 | 100 ms | | |
| 6 | 130 ms | 30.0% | 18.57 |
| 7 | 160 ms | 23.1% | 20.00 |
| 8 | 190 ms | 18.8% | 21.11 |
| 9 | 840 ms | 342.1% | 84.00 |
| 10 | 1110 ms | 32.1% | 100.91 |
| 11 | 1550 ms | 39.6% | 129.17 |
| 12 | 1790 ms | 15.5% | 137.69 |
| 13 | 2160 ms | 20.7% | 154.29 |

Table 3.3: Experimental results for multi-level validation of the ETH elevator (levels 0–4 have less than 100 states)

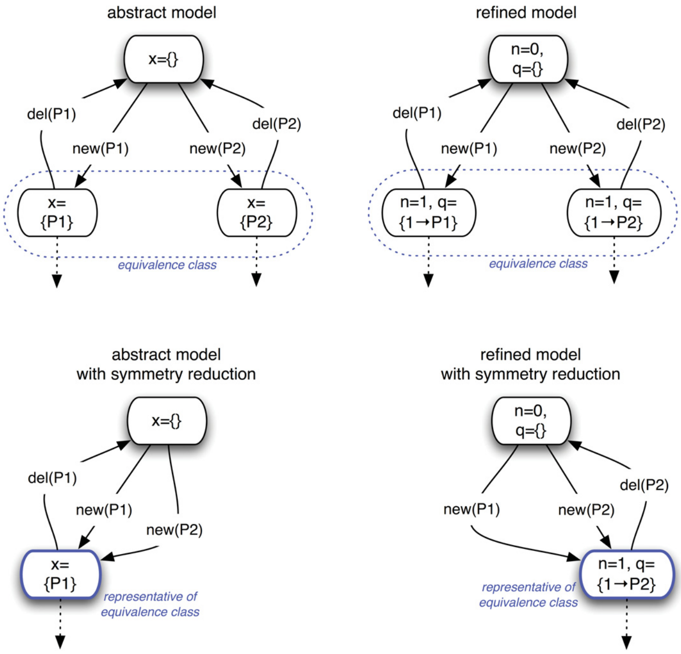


Figure 3.17: Illustration of symmetry reduction for trace refinement checking

a particular array to be sorted. The results are summarised in Table 3.4. Here, we can see that between level 0 and 1 the number of nodes and the validation time actually decreases: the refined model is more deterministic and has, hence, a reduced state space. The non-deterministic choice of a sorting permutation in the abstraction is implemented by a deterministic algorithm computing such a permutation. The validation time per node and level actually goes down until level 4, and then remains relatively stable from level 5 onwards.

We have also applied our algorithm on industrial models from SAP, Bosch and Siemens within the Deploy project. In all instances, the multi-level algorithm dealt very well with large number of refinement levels.

## 3.6 More Related Work

As already indicated above, the tools Brama and AnimB are also capable of performing multi-level animation of Event-B models, and have partially inspired this work. Unfortunately there is little scientific or technical documentation available for either of these tools. A few notable differences are

- Both Brama and AnimB require the user to specify explicitly values for constants; that is, we had to "calculate" the Cartesian products for the level constant in Fig. 3.1 by hand.

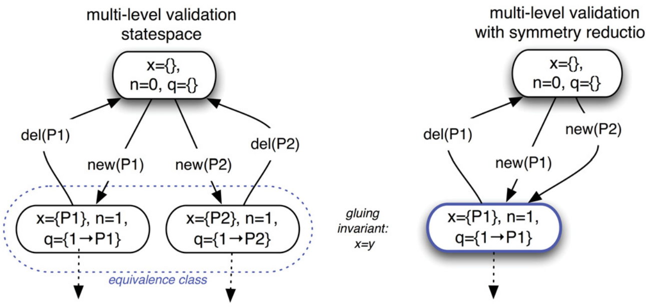


Figure 3.18: Illustration of symmetry reduction for multi-level validation

| Level | Validation Time (complete) | Nodes | Cost for extra level | Time per node and level |
|---|---|---|---|---|
| 0 | 300 ms | 34 | | 8.82 |
| 1 | 60 ms | 18 | -80.0% | 1.67 |
| 2 | 110 ms | 18 | 83.3% | 2.04 |
| 3 | 170 ms | 44 | 54.5% | 0.97 |
| 4 | 190 ms | 65 | 11.8% | 0.58 |
| 5 | 260 ms | 65 | 36.8% | 0.67 |
| 6 | 310 ms | 66 | 19.2% | 0.67 |
| 7 | 360 ms | 66 | 16.1% | 0.68 |
| 8 | 410 ms | 66 | 13.9% | 0.69 |
| 9 | 460 ms | 66 | 12.2% | 0.70 |

Table 3.4: Experimental results for multi-level validation of a Quicksort model

- *P*roB can be driven by a model checker to systematically search for errors, and to validate LTL formulas.
- *P*roB uses a constraint-solving approach to find solutions for predicates, while AnimB and Brama seem to rely on pure enumeration. As such, *P*roB can evaluate much more complicated guards and predicates than AnimB or Brama.

Another animator for Event-B is [ASA08]; but it does not yet seem to support refinement animation. The same is true for the animator in [ABC$^+$02] for classical B.

Animation was already considered valuable in the tools supporting the B Method, Atelier B [Ste09] and the B-Toolkit [B-C02]. Both contain support for animating formal models. However, the support is very rudimentary. It relies on the user to supply parameter values to carry out a step of the simulation engine. If the supplied values do not satisfy the guard, this is shown to the user but the simulation continues.

Abstract State Machines [BS03] have traditionally an operational conception of modelling. The tool CoreASM [FGG07] support animation of Abstract State Machines based on an executable core of the notation. Refinement [Bör03] is not supported yet. The refinement notion of Abstract State Machines is very general. Thus, it may be difficult to achieve efficient animation without introducing restrictions on permissible refinements. We also believe that the concept of witnesses could be a key for Abstract State Machine refinement animation, too.

Tools for VDM [Jon90] have also provided animation features. VDMTools [Sys10] provides animation but not of refinements. An early attempt on animation of VDM models has been made in [Led97]. However, the approach relies on the user to compare simulation traces of abstract and concrete models. We think automation of this is indispensable, because relying on the user for trace inspection the approach appears error-prone.

We have not considered animation of formalisms based on automata or Petri nets here because the challenges in their animation are quite different from those we face in Event-B, a state-based formalism with forward refinement at its core, where the model is described in terms of first-order predicate formulas.

## 3.7 Conclusion

We have presented a description of refinement in Event-B and have shown how a suitable animation and validation algorithm can be developed. The key ingredient that makes the algorithm tractable are the witnesses of Event-B. We have implemented the algorithm within *P*roB, and have shown how a variety of refinement errors can now be detected effectively. We have applied the technique to various case studies, and have animated up to 14 levels simultaneously. The algorithm presented in the article also achieves further performance improvements over previous work based on trace refinement, for instance, by applying symmetry reduction to multiple levels of refinement at once. It would be interesting to see how the new ideas apply to classical B, in particular. We leave this for future work.

We consider proof and animation complementary techniques of validation. Hence, animation does not need to provide all capabilities that formal proof provides. For instance, for finding candidates of invariants formal proof appears superior; for checking whether they are always satisfied, i.e., finding counter examples, animation appears superior. Moreover, animation can be used to reason about properties that have not (yet) been formalised.

We would like to incorporate some improvements into the animator. We would like to be able to better visualise relative deadlocks, i.e., states where all concrete events are disabled but some abstract event is enabled. For now, animation shows in case of a relative deadlock that all concrete events are disabled and does not analyse the abstract events further. Moreover, Event-B types and definitions would be needed that support animation better. For instance, transitive closures are defined as sets of all transitive closures of all relations of a certain type. For animation, it would be sufficient to compute only the closures of the relations actually appearing in a model. We would like to link the animator closer to the other components of the Rodin tool. We would also like to indicate certain errors found in Event-B models by *P*roB in associated proof obligations, for instance, so that the violated proof obligations can be marked as “not provable.”

**Acknowledgement** We are grateful to Thierry Lecomte and Christophe Métayer for their help with using Brama and AnimB and their help with technical questions concerning the two tools. We also want to thank the anonymous reviewers for their helpful comments. Part of this research has been EU funded FP7 project 214158: DEPLOY (Industrial deployment of advanced system engineering methods for high productivity and dependability).

## 3.8 Appendix: Refinements of the Coffee Model

```
machine CoffeeR1 refines CoffeeM
sees CofCtxt
variables alvl coins
invariants
  @ci $coins \in \mathbb{N}$
events
  event INITIALISATION
    extends INITIALISATION
    begin
      @ai $coins := 0$
    end
  event fill_mug
    extends fill_mug
    when
      @gc $coins > 0$
    then
      @delc $coins := coins - 1$
    end
  convergent event drink
    extends drink
  end
  anticipated event insert_coin
    begin
      @insc $coins := coins + 1$
    end
end

context ImpCtxt
constants maxc
axioms @amc $maxc \in \mathbb{N}_1$
end
```

(a) First refinement *CoffeeR1*

```
machine CoffeeR2 refines CoffeeR1
sees CofCtxt ImpCtxt
variables clvl coins
invariants
  @ifl $clvl \in 0\,..\,11$
  @lvl $alvl = level(clvl)$
variant $maxc - coins$
events
  event INITIALISATION
    begin
      @cci $coins := 0$
      @fli $clvl := 0$
    end
  event fill_mug refines fill_mug
    when
      @gc2 $coins > 0$
      @ml $clvl \in level^{-1}[\{empty\}]$
    with
      @x $x = level(clvl')$
    then
      @delc2 $coins := coins - 1$
      @ffl $clvl :\in level^{-1}[\{full\}]$
    end
  convergent event drink refines drink
    when
      @dgfl $clvl \notin level^{-1}[\{empty\}]$
    with
      @alvl' $alvl' = level(clvl')$
    then
      @dfl $clvl :\in level^{-1}[\{empty, half\} \setminus \{level(clvl)\}]$
    end
  convergent event insert_coin extends insert_coin
    when
      @gmc $coins < maxc$
    end
end
```

(b) Second refinement *CoffeeR2*

Figure 3.19: Refinements of coffee dispenser machine

# 4 Seven at one stroke: LTL model checking for High-level Specifications in B, Z, CSP, and more

Daniel Plagge, Michael Leuschel

**Abstract.** The size of formal models is steadily increasing and there is a demand from industrial users to be able to use expressive temporal query languages for validating and exploring high-level formal specifications. We present an extension of LTL, which is well adapted for validating B, Z and CSP specifications. We present a generic, flexible LTL model checker, implemented inside the *P*roB tool, that can be applied to a multitude of formalisms such as B, Z, CSP, B||CSP, as well as Object Petri nets, compensating CSP, and dSL. Our algorithm can deal with deadlock states, partially explored state spaces, past operators, and can be combined with existing symmetry reduction techniques of *P*roB. We establish correctness of our algorithm in general, as well as combined with symmetry reduction. Finally, we present various applications and empirical results of our tool, showing that it can be applied successfully in practice.

**Keywords: Validation and Verification – Notations and Languages – LTL – model checking – B-method – CSP – Z – Integrated Methods – symmetry reduction.**

## 4.1 Introduction and Motivation

The B-Method and Z are used in railway systems [DEF03], the automotive sector [Pou03], as well as avionics [Hal96]. The size of the formal models is steadily increasing and there is a big demand from industrial users to be able to animate and validate high-level specifications [ED07], in order to ensure that the correct system is built. *P*roB is an animator and model-checker primarily designed for B [LB08]. By now *P*roB supports other formalisms like Z specifications [PL07], CSP, B||CSP, Object Petri nets, compensating CSP and dSL (a programming language used for industrial controllers). Beside animation, it can also be used to detect invariant violations, deadlocks and check refinement. However, there is also an industrial demand for expressive temporal query and validation languages[1], in order to validate temporal properties of the system (not easily expressed

[1] Private communication from Kimmo Varpaaniemi, Space Systems Finland.

in B or Z), as well as to navigate in the state space, and ask questions about the future and past of the current state.

In this paper we present a methodology and implementation to satisfy this industrial need by

- using LTL as the core and—based on feedback from case studies—extending it to enable convenient property specification by the user,
- implementing the model checking algorithm and integrating it into the *P*roB tool set. Due to the flexible, high-level implementation our technology is not limited to B and Z, but can also be applied to CSP, combinations of B, CSP and Z, as well as to a few other domain specific formalisms.
- providing a practical evaluation of our language and tool, showing that we can express a large class of problems (covering many described in earlier literature) and also solve those problems in practice using our implementation.

The article is organized as follows: After discussing the approach in general, we give a formal definition of $\text{LTL}^{[e]}$ (Section 4.3) and describe the model-checking algorithm (Section 4.4), along with an extension for past-LTL (Section 4.5). In Section 4.6 we provide some examples and experiments. After showing that $\text{LTL}^{[e]}$ can be combined with symmetry reduction (Section 4.7), we conclude the article with look at related and future work.

## 4.2 LTL for Formal Models

LTL is a popular temporal logic for model checking [CGP99], and is now considered to be more expressive, intuitive and practically useful than CTL (see, e.g. [Var01]). Despite an apparent complexity problem (model checking LTL is exponential in the size of the formula), "efficient" algorithms exist for LTL model checking, notably by negating an LTL formula and translating it into a Büchi automata. The most prominent model checking tool that supports LTL is probably spin [Hol97]. But note that newer versions of SMV now also support LTL. Despite its popularity and usefulness, there are a number of formalisms which are still lacking an automatic LTL model checking tool.

- The B-Method [Abr96]
  There has also been considerable interest in trying to verify temporal properties for B specifications. In [BB02] proof obligations are defined for liveness properties in B. A way to reason about temporal properties of B systems is described in [BPS05] amongst others, e.g., checking properties about when operations are enabled. The work in [Gro06b] and the associated JAG tool [Gro06a] aim to prove LTL properties of B machines by translating Büchi automata into a B representation and generating suitable proof obligations. However, none of these provide a fully automatic model checker, as proof obligations still need to be discharged. The works [BDJK01] and [CJMB05] also study the use of LTL for B specifications,

in particular examining the link with refinement. The work in [Par00] describes a LTL model checker for B based on CLPS-B [BLP02].The system does not cover the full B language (e.g., no power set construction, no lambda abstractions nor set comprehensions are supported) and deals with standard LTL (albeit with fairness constraints).

Finally, the model checker *P*roB [LB08] is a fully automatic tool, but in its current form can only check safety properties, as well as perform refinement checks.

In summary, to our knowledge there is no automatic tool available to check LTL properties for full B (or at least a large subset thereof). The same can be said for the composition of B and CSP (see, e.g. [TS00] and [BL05]).

- CSP [Ros99]
  This formalism is supported by the refinement checker FDR [Oxf10]. Here, the idea is to model both the system and the property in the *same* formalism, e.g., as CSP processes, and perform refinement checks.

  The relationship between refinement checking and LTL model checking has been studied (e.g., [Ros05] and [DS04]) and we ourselves have even proposed a way to perform LTL model checking for CSP using FDR in [LMC01], by translating Büchi automata into CSP processes, language intersection into CSP synchronisation and the emptiness check into a refinement check. However, this approach is not that useful in practice (because the complexity is on the wrong side of the refinement check for FDR to be efficient, and because it requires several tools to be applied in sequence).

**Contributions:** In the rest of this paper we describe an extension of LTL, called LTL$^{[e]}$, which is well adapted for validating B, Z and CSP specifications, by allowing us to reason about enabled operations and the execution of operations. In addition, we present the implementation of a LTL$^{[e]}$ model checking algorithm inside *P*roB, which can

- deal with deadlock states and partially explored state spaces,
- be applied in conjunction with symmetry reduction,
- be directly applied to multiple formalisms, such as B, CSP, B || CSP, Z, Object Petri nets, StAC (CSP with compensations), and dSL.

We establish correctness of our algorithm in general, as well as combined with symmetry reduction. In addition we provide various applications and useful LTL$^{[e]}$ patterns, as well as empirical results. We also briefly present an extension to allow Past LTL operators [LS95].

**Discussion about the approach:** The interested reader may ask the question: "Why did we not translate our formal models into, e.g., Promela and use the SPIN LTL model checker?" Indeed, this approach is perfectly valid, and has proven to be successful for some lower-level languages (e.g., for Java in [HD01] or dSL in [WGMM05]). For very high-level languages, however, this approach becomes much more difficult. Indeed, translating B directly into Promela would be extremely challenging (it is already difficult enough to write a B interpreter in Prolog with constraint solving like we did for *P*roB), and it is furthermore very difficult to avoid additional state space explosion due to the smaller granularity of Promela (see, [Leu08]).[2] Another option would be to compute the state space with *P*roB, and then translate it to a Promela model. We have actually implemented such a translation, but it has so far not proven to be practically useful. First, the overhead of starting up an external tool can be considerable (typically 6 seconds were needed for SPIN to generate and compile the pan.c files). Also, translating the high-level properties into atomic Promela properties can be expensive, and it is not obvious how to exploit the symmetry present in the high-level model in the Promela model. Most importantly, the extensions of the LTL language, which are needed for most interesting practical applications discussed in Section 4.6, are not supported by SPIN. Still, we plan to reevaluate this approach in the future.

## 4.3 LTL$^{[e]}$

We want to use the LTL model checker for models specified in B, Z, etc. Those models can have deadlock states, but usually LTL formulas are defined over Kripke structures such that every state must have at least one successor state. To support models with deadlock states, we simply extend the definition of a Kripke structure to also allow states without successors.

Another approach to treat deadlock states is to add a 'dummy' loop transition to each deadlock state or even to add an additional state with a loop. But this introduces subtle differences, e.g. when a dummy transition is added, the formula $Xp$ is true in a deadlock state iff $p$ is true in the state itself. We think that $Xp$ should not be true at all because there is no next state. Our approach has the advantage that the underlying labeling transition system remains unaltered whereas the performance impact is very low.

From preliminary case studies it became clear that often it is interesting to know which kind of operation has been performed to get into a certain state. Especially in CSP models, we are often only interested in the operation performed, not in the state between operations. We add labels on transitions in the relation of the Kripke structure. LTL with support for labelled transitions can also be found in [CCO$^+$05], but the definition there is limited to infinite paths.

Propositions on transitions give us the possibility to express statements like 'the next operation on the path is *op1*$(x)$ with $x > 3$'. In the formalism below we denote such statements with $[t]$, where $t$ is the proposition on the transition.

**Definition 1.** *A* labelled Kripke structure *$M$ with possible deadlocks over atomic propositions $AP$ and transition propositions $TP$ is a tuple $M = (S, S_0, R, L)$ consisting of a set of states $S$, a*

[2] This process was actually attempted in the past — without success — within the EPSRC funded project ABCD at the University of Southampton.

*set of initial states $S_0 \subseteq S$, a ternary relation between states $R \subseteq S \times 2^{TP} \times S$, and a labeling function $L \in S \to 2^{AP}$.*

*For our purposes we do* not *restrict the relation to be total, so the structure may have deadlock states. The set of deadlock states is*

$$deadlocks = \{s \in S | \neg\, \exists\, t, s' : (s, t, s') \in R\}.$$

**Definition 2.** *A path $\pi$ in $M$ can be either infinite or finite ending in a deadlock state:*

- *A finite path of length $|\pi| = k$, $k \geq 1$ is a finite sequence $\pi = s_0 \xrightarrow{t_0} \ldots \xrightarrow{t_{k-2}} s_{k-1}$ with $s_0 \in S_0$, $s_{k-1} \in deadlocks$ and $\forall\, i : 0 \leq i < k-1 \Rightarrow (s_i, t_i, s_{i+1}) \in R$.*
- *Infinite paths have the form $\pi = s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_1} \ldots$, $s_0 \in S_0$, $\forall\, i \geq 0 : (s_i, t_i, s_{i+1}) \in R$. We denote $|\pi| = \omega$ for the infinite length of $\pi$.*

We denote $\pi^i$ as the suffix of $\pi$ without $\pi$'s first $i$ elements.

We extend the semantics of LTL formulas in two aspects: First we claim that a formula of the form $X\varphi$ is only true if the current state is not a deadlock. Second we allow to check if a transition proposition holds in the transition to the next state by using the $[t]$ construct. A state $s$ in $M$ satisfies a formula $\varphi$ (denoted $M, s \models \varphi$) if all paths starting in $s$ satisfy $\varphi$. Whether a path $\pi$ satisfies a formula $\varphi$ (denoted $M, \pi \models \varphi$, or shorter $\pi \models \varphi$ if $M$ is unambiguous) is defined by:

$$\begin{array}{lll}
\pi \models & true & \\
\pi \models & p & \Leftrightarrow \quad \pi = s_0 \ldots \text{ and } p \in L(s_0) \\
 & & \phantom{\Leftrightarrow} \quad \text{for atomic propositions } p \in AP \\
\pi \models & \neg\, \varphi & \Leftrightarrow \quad \pi \not\models \varphi \\
\pi \models & \varphi \vee \psi & \Leftrightarrow \quad \pi \models \varphi \text{ or } \pi \models \psi \\
\pi \models & X\varphi & \Leftrightarrow \quad |\pi| \geq 2 \text{ and } \pi^1 \models \varphi \\
\pi \models & \varphi\, U \psi & \Leftrightarrow \quad \exists\, k < |\pi| : \pi^k \models \psi \\
 & & \phantom{\Leftrightarrow} \quad \text{and } \forall\, i : 0 \leq i < k \Rightarrow \pi^i \models \varphi \\
\pi \models & [t] & \Leftrightarrow \quad |\pi| \geq 2 \text{ and} \\
 & & \phantom{\Leftrightarrow} \quad \pi = s_0 \xrightarrow{t_0} \pi^1 \text{ with } t \in t_0 \\
 & & \phantom{\Leftrightarrow} \quad \text{for transition labels } t \in TP
\end{array}$$

So far we have defined only a few basic LTL[e] operators. We introduce other operators like conjunction ($\wedge$), finally ($F$), globally ($G$), release ($R$) and weak until ($W$) in the usual way:

$$\begin{array}{rcl}
false & := & \neg\, true \\
\varphi \wedge \psi & := & \neg\, (\neg\, \varphi \vee \neg\, \psi) \\
F\varphi & := & true\, U\varphi \\
G\varphi & := & \neg\, F \neg\, \varphi = \neg\, (true\, U \neg\, \varphi) \\
\varphi\, R\psi & := & \neg\, (\neg\, \varphi\, U \neg\, \psi) \\
\varphi\, W\psi & := & G\varphi \vee \varphi\, U\psi = \neg\, (true\, U \neg\, \varphi) \vee \varphi\, U\psi
\end{array}$$

## 4.4 The Model Checking Algorithm

Below we adapt the LTL model checking algorithm from [LP85] and [CGP99]. One may ask why we did not use the "standard" LTL model checking algorithm based on Büchi automata. Our motivations were as follows:

- It can be easily extended to deal with "open" nodes, on which no information is available. This is especially useful for infinite state systems, where only part of the state space can be computed. Also, it is not clear to what extent Büchi automata can easily deal with the $[t]$ operator from $\mathrm{LTL}^{[e]}$.
- The state space for B and Z specifications is, due to the high-level nature of the operations, typically much smaller than for other more low-level formalisms such as Promela. This is especially true when we apply symmetry reduction (cf. Section 4.7). Hence, the bottleneck is generally not to be found inside the LTL model checking algorithm.
- The algorithm can also later be extended to $\mathrm{CTL}^*$ [CGP99].

We implemented the algorithm in C, using SICStus Prolog's C-Interface to integrate it into the *P*roB tool. The model checking module is not specific to the B formalism, in fact it uses callback mechanism to let the Prolog code evaluate the atomic propositions and outgoing transitions for every state.

### 4.4.1 Overview of the algorithm

To check if a model satisfies a given $\mathrm{LTL}^{[e]}$ formula, we use a modified version of the tableau algorithm given in [LP85] and [CGP99]. We adapted the algorithm in a way that deadlock states and propositions on transitions are supported.

To check if a state $s$ in the structure $M$ satisfies a given LTL formula, we try to find a counter-example by searching for a path starting in $s$ that satisfies the negated formula $\varphi$.

In the next paragraphs we explain how a graph can be constructed that contains some nodes (called atoms) for each state of the model. An atom represents a possible valuation of $\varphi$ and its subformulas that is consistent with the corresponding state. E.g. for a formula $\varphi = a \vee Xb$ with $a, b \in AP$, the valuations of $a$ and $b$ are defined by the state but there are two atoms, one where $Xb$ is true and one where $Xb$ is false. There is an edge between two atoms $A$ and $B$ if there is a transition between the corresponding states and if subformulas of the form $X\psi$ in $A$ have the same valuation as $\psi$ in $B$.

Then we search for a path of atoms that serves as a counter-example with the following properties: The path starts with an atom $(s_0, F_0)$ of the initial states ($s_0 \in S_0$) where $\varphi \in F_0$. And for each atom on the path where $\psi_1 U \psi_2$ is true, $\psi_1$ is true until a state is reached where $\psi_2$ is true. A counter-example may be infinitely long, then it consists of a finite path, followed by a cycle. To find also those cycles, we search for a strongly connected component (SCC) with certain properties.

We adapt the original algorithm's rules of how atoms can be constructed and when a transition from one atom to another exists. And in contrast to the original algorithm we consider deadlock states in the requirements of the SCC we search for.

After presenting the algorithm, we show how nodes that are not yet explored can be handled and present a proof for the correctness of our extensions to the algorithm.

For the interested reader we provide our algorithm in full detail below. The reader not interested in this can skip to Section 5.

### 4.4.2 The closure of a formula

The *closure* $Cl(\varphi)$ of a formula $\varphi$ is the smallest set of LTL$^{[e]}$ formulas satisfying the following rules:

$$\begin{array}{rcl}
\varphi \in Cl(\varphi) & & \\
\psi \in Cl(\varphi) & \Rightarrow & (\neg\, \psi) \in Cl(\varphi), \\
 & & \text{identifying } \neg\, \neg\, \varphi \text{ with } \varphi \\
\psi_1 \vee \psi_2 \in Cl(\varphi) & \Rightarrow & \psi_1 \in Cl(\varphi) \text{ and } \psi_2 \in Cl(\varphi) \\
X\psi \in Cl(\varphi) & \Rightarrow & \psi \in Cl(\varphi) \\
\neg\, X\psi \in Cl(\varphi) & \Rightarrow & X(\neg\, \psi) \in Cl(\varphi) \\
\psi_1\, U\psi_2 \in Cl(\varphi) & \Rightarrow & \psi_1 \in Cl(\varphi), \psi_2 \in Cl(\varphi), \\
 & & X(\psi_1\, U\psi_2) \in Cl(\varphi)
\end{array}$$

Informally, the closure $Cl(\varphi)$ contains all formulas that determine if $\varphi$ is true. This definition has been taken without modification from the original algorithm.

### 4.4.3 Atoms of a state

An *atom* is a pair $(s, F)$ with $s \in S$ and $F$ a consistent set of formulas $F \subseteq Cl(\varphi)$. $F$ is consistent if it satisfies the following rules:

- $p \in F$ iff $p \in L(s)$ for atomic propositions $p \in AP$
- $\psi \in F$ iff $(\neg\, \psi) \notin F$ for $\psi \in Cl(\varphi)$
- $\psi_1 \vee \psi_2 \in F$ iff $\psi_1 \in F$ or $\psi_2 \in F$ for $\psi_1 \vee \psi_2 \in Cl(\varphi)$
- $\psi_1 U \psi_2 \in F$ iff $\psi_2 \in F$ or $\psi_1, X(\psi_1 U \psi_2) \in F$ for $\psi_1\, U\psi_2 \in Cl(\varphi)$
- If $s \in$ *deadlocks* then $(\neg\, X\psi) \in F$ for $X\psi \in Cl(\varphi)$
- If $s \notin$ *deadlocks* then $X\psi \in F \Leftrightarrow (X\neg\, \psi) \notin F$ for $X\psi \in Cl(\varphi)$
- If $s \in$ *deadlocks* then $(\neg\, [t]) \in F$ for all transition propositions $t \in TP$.

Our changes to the original rules are as follows: We added the last rule for transition propositions and we introduced the distinction of the cases $s \in \mathit{deadlocks}$ and $s \notin \mathit{deadlocks}$ for the $X$ operator.

Whether a formula $\psi \in Cl(\varphi)$ is in $F$ or not for an atom $(s, F)$ thus depends on the current state's atomic propositions and whether formulas with next operators $X\psi$ and the transitions propositions $[t]$ are in $F$ or not. We denote the set of all possible atoms of a state $s$ with $A(s)$ and all atoms of the set of states $S$ with $A(S)$. The number of atoms of $s$ is limited by $|A(s)| \leq 2^{N_x} \cdot 2^{N_t}$, where $N_x$ is the number of next operators in $Cl(\varphi)$ and $N_t$ the minimum of the number of transition propositions in $Cl(\varphi)$ and the maximum vertex degree $\Sigma(G)$. Thus the maximum number of atoms grows exponentially with the number of next and until operators (because an until operator implies an additional next operator in the closure) and $N_t$.

### 4.4.4 Search for self-fulfilling SCCs

We construct a directed graph $G$ with the set of all atoms $A(S)$ as nodes. There is an edge from $(s_1, F_1)$ to $(s_2, F_2)$ labelled with $t$ iff

- there is a transition in $M$ from the state $s_1$ to $s_2$ with $(s_1, t, s_2) \in R$, and
- $X\psi \in F_1 \Leftrightarrow \psi \in F_2$ for all formulas $X\psi \in Cl(\varphi)$, and
- $[t'] \in F_1 \Leftrightarrow t' \in t$ for all $[t'] \in Cl(\varphi)$

The last rule is an addition to the original algorithm.

A strongly connected component (SCC) $C$ is a maximal subgraph of $G$ such that between all nodes in $C$ there exists a path in $C$. We search for an SCC $C$ that is reachable from an atom $(s_0, F_0)$ of an initial state $s_0 \in S_0$ with $\varphi \in F_0$ and that has the following properties:

- For every atom $(s, F)$ in the component $C$, and every formula $\psi_1\, U\psi_2 \in F$ there exists an atom $(s', F')$ in $C$ with $\psi_2 \in F'$. ('self-fulfilling')
- There exists an edge in $C$ ('nontrivial') or $C$ consists of exactly one deadlock state.

In contrast to the original algorithm we added the exception for deadlock states.

We use Tarjan's algorithm [Tar72] to identify the SCCs in the graph. If we find a nontrivial self-fulfilling SCC $C$, we can construct an $\alpha$-path (a path of atoms in $G$)

$$\pi_\alpha = \underbrace{(s_0, F_0) \xrightarrow{t_0} \ldots}_{\pi_1} \underbrace{(s_c, F_c) \xrightarrow{t_c} \ldots (s_c, F_c)}_{\pi_2}$$

with $s_0 \in S_0$, $\varphi \in F_0$ and $(s_c, F_c)$ in $C$. The first part $\pi_1$ is the $\alpha$-path from an initial atom to an atom in the found SCC $C$. The second part $\pi_2$ is a loop in $C$ that includes an atom $(s, F)$ with $\psi_2 \in F$ for each $\psi_1\, U\psi_2 \in Cl(\varphi)$.

The path $\pi = s_0 \xrightarrow{t_0} \ldots s_c \xrightarrow{t_c} \ldots s_c$ then acts as a counter-example.

If the found SCC consists of exactly one atom $(s_d, F_d)$ of a deadlock state $s_d$, the found $\alpha$-path has the form $\pi_\alpha = (s_0, F_0) \xrightarrow{t_0} \ldots (s_d, F_d)$, and the counter-example is $\pi = s_0 \xrightarrow{t_0} \ldots s_d$.

### 4.4.5 Handling of open nodes

An open node in the state space $S$ is a node, whose outgoing transitions are not calculated yet. The algorithm explained above can be easily modified to work with state spaces that contain open nodes. Whenever the outgoing transitions of a node are needed in Tarjan's algorithm, we check if the current node is an open node. If so, all transitions starting in the node will be calculated. This way the LTL$^{[e]}$ model checker can drive the exploration of the state space. Also, part of the state space can remain unexplored, while still ensuring the correctness of the result.

Another alternative is to leave a node unexplored if we want to explore only a limited number of states or restrict the search to a given subset of states. Then we only have to check if the state is a deadlock. If not, we just store the information that we encountered an such a node. The constructed atoms have no outgoing transitions and are trivial SCCs. So the node won't lead to a false counter-example. If we do not find a counter-example we give the user a warning that not the whole state space was considered.

### 4.4.6 Correctness of the algorithm

The algorithm is used to find a counter-example for a given LTL formula. This is done by searching for a path of atoms to a self-fulfilling SCC or an SCC that consists of a deadlock state. The proof consists of two steps: First we show the equivalence between the existence of a counter-example and the existence of an eventuality sequence, then we show the equivalence of the existence of an eventuality sequence and the existence of a path to an SCC.

The complete proof (without our additions to the LTL semantics and the algorithm) can be found in [CGP99], we extend it only in a way that it covers our additions.

**Definition 3.** *An* eventuality sequence $\pi_\alpha$ *is an infinite $\alpha$-path or finite $\alpha$-path ending in an atom of a deadlock state such that if $\psi_1 U \psi_2 \in F_A$ for an atom $A$ on $\pi_\alpha$, there exists an atom $B$ on $\pi$ after $A$ with $\psi_2 \in F_B$.*

**Lemma 1.** *There exists a path $\pi$ starting in $s_0$ with $\pi \models \varphi$ iff there exists an eventuality sequence starting at an atom $(s_0, F_0)$ such that $\varphi \in F$.*

*Proof.* We first show that the existence of an eventuality sequence implies the existence of $\pi$, then we show the reverse direction.

1. Let $\pi_\alpha = (s_0, F_0) \xrightarrow{t_0} (s_1, F_1) \xrightarrow{t_1} \ldots$ be an eventuality sequence starting in $(s_0, F_0) = (s, F)$ with $\varphi \in F$. The corresponding path is $\pi = s_0, s_1, \ldots$. We prove $\pi \models \varphi$ by showing that $\pi^i \models \psi \Leftrightarrow \psi \in F_i$ holds for every $\psi \in Cl(\varphi)$ and $0 \leq i < |\pi|$. The proof is done by induction on the subformulas of $\varphi$. We describe only the two cases that are affected by our changes to the LTL semantics.
   a) If $\psi = X\varphi_1$: If $s_i \in \mathit{deadlocks}$ then by definition $X\varphi_1 \notin F_i$ and $\pi_i \not\models X\varphi_1$.

      If $s_i \notin \mathit{deadlocks}$ then we have a transition from $(s_i, F_i)$ to $(s_{i+1}, F_{i+1})$, implying that $X\varphi_1 \in F_i \Leftrightarrow \varphi_1 \in F_{i+1}$. By induction we know $\varphi_1 \in F_{i+1} \Leftrightarrow \pi^{i+1} \models \varphi_1$ and by definition $\pi^{i+1} \models \varphi_1 \Leftrightarrow X\pi^i \models \varphi_1$ holds.

b) If $\psi = [t]$ with $t \in TP$: If $s_i \in deadlocks$ then it follows from the definition of an atom that $[t] \notin F_i$ and from the definition of $\models$ if follows $\pi_i \not\models [t]$.

If $s_i \notin deadlocks$ then there exists a transition $(s_i, F_i) \xrightarrow{t'} (s_{i+1}, F_{i+1})$. By definition $[t] \in F_i \Leftrightarrow t = t'$ and $\pi^i = s_i \xrightarrow{t} s_{i+1} \ldots \Leftrightarrow \pi^i \models [t]$.

2. Let $\pi = s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_1} \ldots$ be a path with $\pi \models \varphi$. Let $F_i = \{\psi \mid \psi \in Cl(\varphi) \wedge \pi^i \models \psi\}$. First we show that there is an $\alpha$-path $\pi_\alpha = (s_0, F_0) \xrightarrow{t_0} (s_1, F_1) \xrightarrow{t_1} \ldots$.

a) All $(s_i, F_i)$ are atoms. This can be seen by comparing the definition of $\models$ with the definition of consistency of an atom's formulas.

b) If $s_i \in deadlocks$: $s_i$ is the last element of $\pi$. Because there is no transition in $R$ starting in $s_i$, there is neither a transition starting in $(s_i, F_i)$.

If $s_i \notin deadlocks$: There is a state $s_{i+1}$ in $\pi$. We show that there is a transition from $(s_i, F_i)$ to $(s_{i+1}, F_{i+1})$: For every $X\psi \in Cl(\varphi)$ holds by definition that $\pi^i \models X\psi \Leftrightarrow \pi^{i+1} \models \psi$ and we have $X\psi \in F_i \Leftrightarrow \pi^i \models X\psi$ and $\psi \in F_{i+1} \Leftrightarrow \pi^{i+1} \models \psi$, it follows that $X\psi \in F_i \Leftrightarrow \psi \in F_{i+1}$.

Also for every $[t] \in Cl(\varphi)$, $t \in TP$ holds $\pi^i \models [t] \Leftrightarrow t \in t_i$, $\pi^i \models [t] \Leftrightarrow [t] \in F_i$ , it follows that $t \in F_i \Leftrightarrow t \in t_i$.

For every $\psi_1 U \psi_2 \in F_i$ there is a $F_j$, $i \leq j < |\pi|$, such that $\psi_2 \in F$, because $\psi_1 U \psi_2 \in F_i \Leftrightarrow \pi_i \models \psi_1 U \psi_2$ and by definition there is a $j$ with $i \leq j < |\pi|$, such that $\pi_j \models \psi_2$ and that implies $\psi_2 \in F_j$. So $\pi_\alpha$ is an eventuality sequence.

□

**Lemma 2.** *There exists an eventuality sequence starting at an atom $(s_0, F_0)$ iff there is a path in $G$ from $(s_0, F_0)$ to a self-fulfilling SCC or an SCC consisting of a deadlock state.*

*Proof.* We only consider the case where the path ends in a deadlock state. The other, infinite, case can be found in the original proof.

1. Let $\pi_\alpha = (s_0, F_0) \xrightarrow{t_0} \ldots \xrightarrow{t_{d-1}} (s_d, F_d)$ be a finite eventuality sequence ending in the deadlock state $s_d$. We know that $(s_d, F_d)$ has no outgoing edges, so the atom is the only node in the SCC.

2. Let there be a path $(s_0, F_0) \xrightarrow{t_0} \ldots \xrightarrow{t_{d-1}} (s_d, F_d)$ to an SCC consisting of just a deadlock state $(s_d, F_d)$. We have to show that $\pi_\alpha$ is an eventuality sequence by proving that for every occurrence of $\psi_1 U \psi_2 \in F_i$ $(0 \leq i \leq d)$ there is an atom $(s_j, F_j)$, $i \leq j \leq d$ with $\psi_2 \in F_j$. Then, by the definition of an atom, $\psi_2 \in F_i$ or $X(\psi_1 U \psi_2) \in F$ implies $\psi_1 U \psi_2 \in F_{i+1}$. Now we assume $\psi_2 \notin F_j$ for all $j$ with $i \leq j \leq d$. Then we know that $X(\psi_1 U \psi_2) \in F_j$, it follows that $X(\psi_1 U \psi_2) \in F_d$. But $s_d$ is a deadlock state, so $X(\psi_1 U \psi_2) \notin F_d$, we have a contradiction. So there is an atom $(s_j, F_j)$ with $\psi_2 \in F_j$.

□

The lemmas 1 and 2 together show that there exists a counter-example to a formula iff the presented algorithm finds a suitable SCC.

## 4.5 LTL$^{[e]}$ with past

The definition of LTL$^{[e]}$ in Section 4.3 only considers operators that enable us to express properties about current or future states of the system. LTL with past [LS95] makes it possible to reason about previous states and transitions by introducing operators such as

- Y, which stands for "yesterday" and is the dual of the LTL$^{[e]}$ next state operator X, and
- S, which stands for "since" and is the dual of the LTL$^{[e]}$ operator until.

### 4.5.1 Definition of Past-LTL$^{[e]}$

To support Past-LTL$^{[e]}$, we first have to make small modifications to the formal definition of the LTL$^{[e]}$ operators. In the original definition of $\models$ in Section 4.3, the current state of a sequence $\pi = s_0 \xrightarrow{t_0} s_1 \dots$ is always the first state $s_0$. This prevents us from using this notation to reason about the past of a sequence. E.g. $\pi \models X\varphi$ is defined by $\pi^1 \models \varphi$, so $\varphi$ cannot consider the state $s_0$ anymore. To allow this, an index $i$ is introduced that indicates which state of the sequence is the current one.

Whether $\pi = s_0 \xrightarrow{t_0} s_1 \dots$ satisfies a Past-LTL$^{[e]}$ formula $\varphi$ in the $i^{\text{th}}$ state of the sequence $\pi$ in the Kripke structure $M$ (denoted $M, (\pi, i) \models_p \varphi$ or shorter $(\pi, i) \models_p \varphi$), is defined by:

$$\begin{array}{lll}
(\pi,i) \models_p true & \Leftrightarrow & 0 \le i < |\pi| \\
(\pi,i) \models_p p & \Leftrightarrow & 0 \le i < |\pi| \text{ and } p \in L(s_i) \\
 & & \text{for atomic propositions } p \in AP \\
(\pi,i) \models_p \neg\,\varphi & \Leftrightarrow & (\pi,i) \not\models_p \varphi \\
(\pi,i) \models_p \varphi \vee \psi & \Leftrightarrow & (\pi,i) \models_p \varphi \text{ or } (\pi,i) \models_p \psi \\
(\pi,i) \models_p X\varphi & \Leftrightarrow & i+1 < |\pi| \text{ and } (\pi,i+1) \models_p \varphi \\
(\pi,i) \models_p \varphi U \psi & \Leftrightarrow & \exists k : i \le k < |\pi| \text{ with } (\pi,k) \models_p \psi \\
 & & \text{and } \forall j : i \le j < k \Rightarrow (\pi,j) \models_p \varphi \\
(\pi,i) \models_p [t] & \Leftrightarrow & i+1 < |\pi| \text{ and } t \in t_i \\
 & & \text{for transition labels } t \in TP \\
(\pi,i) \models_p Y\varphi & \Leftrightarrow & i > 0 \text{ and } (\pi,i-1) \models_p \varphi \\
(\pi,i) \models_p \varphi S \psi & \Leftrightarrow & \exists k : 0 \le k \le i \text{ with } (\pi,k) \models_p \psi \\
 & & \text{and } \forall j : k < j \le i \Rightarrow (\pi,j) \models_p \varphi
\end{array}$$

The definition of $(\pi, 0) \models_p \varphi$ is equivalent to $\pi \models \varphi$ for formulas that do not contain the past operators $Y$ or $S$.

We introduce operators like once ($O$, dual to finally), history ($H$, dual to globally) and trigger ($T$, dual to release) as usual:

$$\begin{aligned} O\varphi &:= \mathit{true}\, S\varphi \\ H\varphi &:= \neg\, O \neg\, \varphi = \neg\, (\mathit{true}\, S \neg\, \varphi) \\ \varphi\, T \psi &:= \neg\, (\neg\, \varphi\, S \neg\, \psi) \end{aligned}$$

We decided not to include a dual operator to $[t]$, because it can be easily expressed by $Y[t]$.

### 4.5.2 Closure and atoms of a Past-LTL$^{[e]}$ formula

We add two rules to the definition of a closure $Cl(\varphi)$ (cf. Section 4.4.2).

$$\begin{aligned} Y\psi \in Cl(\varphi) &\Rightarrow \psi \in Cl(\varphi) \\ \psi_1\, S\psi_2 \in Cl(\varphi) &\Rightarrow \psi_1 \in Cl(\varphi), \psi_2 \in Cl(\varphi), \\ & \quad\; Y(\psi_1\, S\psi_2) \in Cl(\varphi) \end{aligned}$$

To the consistency definition of an atom $(s, F)$, we add rules for $Y$ and $S$ (cf. Section 4.4.3):

- If $s \notin S_0$ then $Y\psi \in F$ iff $(Y\neg\, \psi) \notin F$ for $Y\psi \in Cl(\varphi)$
- If $s \in S_0$ then
  - $Y\psi \in F$ iff $(Y\neg\, \psi) \notin F$ for $Y\psi \in Cl(\varphi)$
  - or $Y\psi \notin F$ for $Y\psi \in Cl(\varphi)$
- $\psi_1 S\psi_2 \in F$ iff $\psi_2 \in F$ or $\psi_1, Y(\psi_1 S\psi_2) \in F$ for $\psi_1 S\psi_2 \in Cl(\varphi)$

The first rule is analogous to the non-deadlock case of the $X$ operator. It states that either $Y\psi$ or $Y\neg\, \psi$ is true, but not both. But in the first state $s_0$ of a sequence $\pi = s_0 \xrightarrow{t_0} s_1 \ldots$, we have the additional case that all formulas $Y\psi$ are false, even $Y\mathit{true}$. Because there might be an $s_i$ in $\pi$ with $s_i = s_0$, we can have both cases in an initial state $s \in S_0$. The third rule is analogous to the rule of the $U$ operator.

We need an additional condition when there is an edge in $G$ from an atom $(s_1, F_1)$ to an atom $(s_2, F_2)$ (cf. Section 4.4.4):

- $\psi \in F_1 \Leftrightarrow Y\psi \in F_2$ for all formulas $Y\psi \in Cl(\varphi)$

The search for an SCC in $G$ starts in the atoms $(s_0, F_0)$ with $s_0 \in S_0$ where $Y\psi \notin F_0$ for all formulas $Y\psi \in Cl(\varphi)$.

The proofs of Lemma 1 and Lemma 2 can be easily extended to Past-LTL$^{[e]}$. In the case distinctions of the proofs, the operator $Y$ can be handled analogously to the operator $X$ and the operator $S$ analogously to the operator $U$ where the path ends in a deadlock state.

## 4.6 Some Examples and Experiments

### 4.6.1 LTL$^{[e]}$: Supported syntax and usage patterns

We provide several types of atomic propositions in our implementation:

- One can check if a B predicate holds in the current state by writing the predicate between curly braces, e.g. `{card(set) > 0}`.
- And with `e(op)` it can be tested if an operation `op` is currently enabled.
- If the user animates the model, he can use an atomic proposition `current` to check if a state is the state that is shown in the animator. This allows the user to check a formula $f$ in that state for all paths that go through that state by using `current` $\Rightarrow f$.

And some types of transition propositions:

- With `[op]` it can be checked if *op* is the next executed operation in the path.
- We provide simple pattern matching for the arguments of an operation. E.g. with `[op(5,_)]` one can check if the next operation is *op* and if its first argument is 5. Any B expression (including constants and variables of the machine) or underscores as placeholders for any value can be used as an argument.
- In future, we also plan to support combined pre- and post-conditions like $x' > x$ as transition propositions (where $x'$ refers to the value of $x$ after the transition has been executed).

Note that while checking if an operation is enabled with `e(op)` is an atomic proposition, the check if the next transition is done via a certain operation with `[op]` depends on the actual computation path, not only on the current state.

Some useful patterns of LTL$^{[e]}$ formulas for B/Z specifications (and sometimes also CSP) are as follows:

- quasi-deadlock `G (e(O1) or ... or e(On))` where `O1`,..,`On` are the real state-changing operations (as opposed to query operations). To improve support for this pattern, we introduced another atomic proposition keyword `sink` that is true for states that have no outgoing transitions to other states.
- operation post-condition `G ( [Op] => X {Post} )`
  (`[Op]` tests if the next executed operation is `Op`)
- operation pre-condition `G ( e(Op) => {Pre} )`.
- While animating a model, one may ask if an operation `Op` is executed in the past of the current state, regardless of the computation path being taken to reach it: `current => YO[Op]`.

Later, in Section 4.6, we will see that LTL$^{[e]}$ is useful in practice to solve a variety of other problems, and can also be used to encode fairness constraints.

In the rest of this section we exhibit the flexibility and practical usefulness of our approach. Notably, we show how our tool can now be used to solve a variety of problems mentioned in the literature. We also show that the tool is practically useful on a variety of case studies.

All experiments were run on a Linux PC with an AMD Athlon 64 Dual Core Processor running at 2 GHz, and using *P*roB 1.2.8 built from SICStus Prolog 4.0.2. Our model checker can actually drive the construction of the state space on demand. However, to clearly separate the time required for the LTL checking and the state space construction, we have first fully explored the state space in the examples below.

### 4.6.2 B Examples: Volvo Vehicle Function, Robot, and Card Protocol

We have tried our tool on a case study performed at Volvo on a typical vehicle function (see [LB03]). The B machine has 15 variables, 550 lines of B specification, and 26 operations and was developed by Volvo as part of the European Commission IST Project PUSSEE (IST-2000-30103).

To explore the full state space (1360 states and 25696 transitions) *P*roB required 25.29 seconds. Some of the LTL$^{[e]}$ formulas we checked are as follows:

- `G (e(SetFunctionParameter)`
  `=> e(FunctionBecomesNotAllowed))`
  The formula is valid and the model checking time is 0.12 seconds.

- `G (e(FunctionBecomesAllowed)`
  `=> X e(SetFunctionParameter))`
  A counter-example was found after 0.14 seconds.

- `G ([FunctionBecomesAllowed]`
  `=> X e(SetFunctionParameter))`
  The formula is valid and the model checking time is 0.20 seconds.

- `G(e(FunctionOff)`
  `=> YO[SetFunctionParameter])`
  The formula is valid and the model checking time is 0.21 seconds.

We have also applied our tool to the (very) small robot specification from [Gro06a]. The original LTL formula `G(({Dt=TRUE} & X{Dt=FALSE}) => {De=FALSE})` from [Gro06a] can now be validated fully automatically (and instantaneously). It is interesting to observe that the intended temporal property can be more naturally encoded in our extension LTL$^{[e]}$ as follows: `G([Unload] => {De=FALSE})`.

We have applied our tool on the T=1 protocol[3] specification from [CJMB05]. Computing the state space took 0.02 seconds (for 15 nodes). We tested the formula $P_1$ =

[3] En27816-3, European Standard—identification cards—integrated circuit(s) card without contacts—electronic signal and transmission protocols, 1992.

`G({CardF2=bl} => F{CardF2=lb})` from [CJMB05]. This took less than 0.01 seconds (and 36 atoms were computed). However, our model checker provided a counter-example. This is not surprising, as [CJMB05] also takes fairness constraints into account. These fairness constraints are written in [CJMB05] as $FAIRNESS = \{Eject, Csends\,if\,(CardF2 = bl), Rsends\,if\,(ReaderF2 = bl)\}$. Fortunately, these fairness constraints can be expressed in our LTL[e] language as follows:

```
f = GF[Eject] & GF{CardF2=bl => GF[Csends])
     & (GF{ReaderF2=bl} => GF[Rsends])
```

Checking the formula $f \Rightarrow P_2$ was successful (no counter-example found); this took 12.65 seconds (98,304 atoms where computed). The time is an illustration that LTL model checking is exponential in the size of the formula; it may be worthwhile to investigate adapting our algorithm to incorporate fairness, rather than encoding fairness in the LTL[e] formula itself.

Finally, we believe that our LTL model checker can be used to check probabilistic Event B models ([HH07]), by enabling the encoding of the required fairness constraints.

### 4.6.3 CSP Examples: Peterson and Train Level-Crossing

First we tried a standard CSP example from the book web page of [Sch99][4], Peterson's Algorithm version 1. Computing the state space, consisting of 58 nodes and 115 transitions, took 0.44 seconds with *P*roB (which has recently been extended to handle full CSP-M). Some of the LTL[e] formulas checked are as follows:

- `G ([p1critical] => X(!e(p2critical)) )`
  The formula is valid; the model checking time is 0.01 seconds.

-
  ```
  G ([p1critical]
      => X((!e(p2critical)) W [p1leave] ))
  ```
  The formula is valid; the model checking time is 0.01 seconds.

-
  ```
  G ([p2critical]
      => X((!e(p1critical)) W [p2leave] ))
  ```
  The formula is valid; the model checking time is 0.02 seconds.

We have also tested version 2 of the same algorithm. Computing the state space with 215 nodes and 429 transitions took 1.28 seconds.The CSP model is more generic and elegant than the first version, which enables us to write a single LTL[e] formula basically covering the last two formulas from above.

-
  ```
  G ([critical]
      => X((!e(critical)) W [leave]))
  ```
  The formula is valid; the model checking time is now 0.06 sec.

[4] http://www.cs.rhul.ac.uk/books/concurrency/

Another example we tested is `crossing.csp`, also from [Sch99]. This model by Bill Roscoe describes a level crossing gate using discrete-time modelling in untimed CSP. Computing the state space, consisting of 5517 nodes and 12737 transitions, took 53.76 seconds. Some of the LTL$^{[e]}$ formulas we checked are as follows:

- `G F e(enter)` The formula is valid; the model checking time is 0.36 seconds.
- `G F [enter]` A counter-example (of length 554) was found after 0.17 seconds.

### 4.6.4 CSP || B Examples: Control Annotations and Philosophers

In [IST06] it is proposed to check compatibility of a CSP controller with a particular B machine by adding proof obligations. For this the NEXT annotation is introduced, from which the proof obligations are derived. It turns out that these annotations can also be checked (now automatically) by our LTL model checker. For example, for the traffic light controller from [IST06], the NEXT annotation for the `Stop_All` operation can be checked by the following LTL$^{[e]}$ formula: `G ([Stop_All] => X (e(Go_Moat) & e(Go_Square)))`. This check can be done instantaneously. We have checked all the NEXT assertions from [IST06] fully automatically and instantaneously.

We have also applied our LTL$^{[e]}$ model checker to a fully combined CSP and B model. The B model is the generic dining philosophers example from [LM07] instantiated for three philosophers and three forks, using symmetry reduction (cf. Section 4.7), and where the protocol is specified by a CSP Controller.

```
datatype BPhils = p1 | p2 | p3
channel think,eat,TakeLeftFork,
        TakeRightFork,DropFork : BPhils
MAIN = ||| p: BPhils @ PHIL(p)
PHIL(P) = think!P -> TakeLeftFork!P ->
          TakeRightFork!P -> eat!P ->
          DropFork!P -> DropFork!P -> PHIL(P)
```

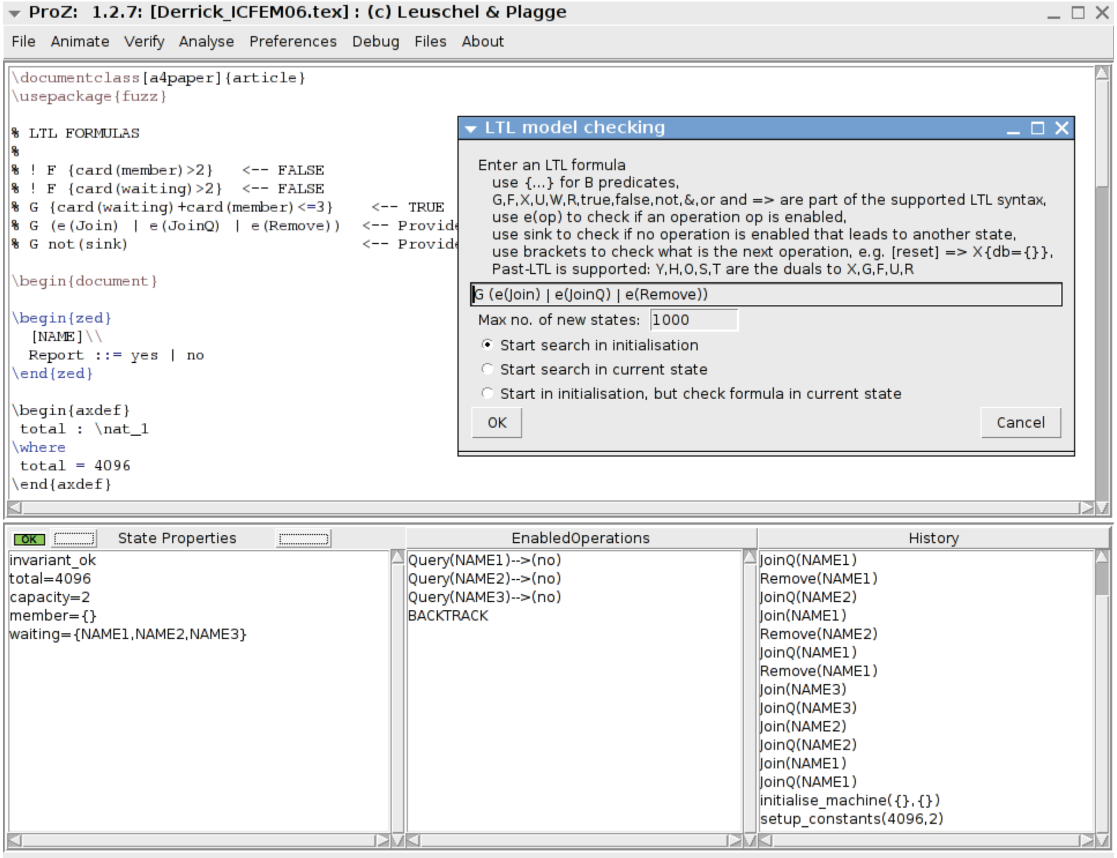
We have validated the following LTL$^{[e]}$ formula in 0.06 seconds:

-
  ```
  G (e(DropFork) U
      ([TakeLeftFork] or [TakeRightFork]))
  ```

### 4.6.5 Z Examples: SAL Example and Workstation Protocol

In [PL07], *P*roB was extended to deal with Z specifications. We examined the example from [DNS06], formalising the process of joining an organisation. We were able to check the three LTL formulas described there:

- `! F {card(member)>2}` (*P*roB provides a counter-example)
- `! F {card(waiting)>2}` (*P*roB provides a counter-example)


ProZ: 1.2.7: [Derrick_ICFEM06.tex] : (c) Leuschel & Plagge
File Animate Verify Analyse Preferences Debug Files About
LTL model checking
OK
Cancel
OK
State Properties
EnabledOperations
History

```
\documentclass[a4paper]{article}
\usepackage{fuzz}

% LTL FORMULAS
%
% ! F {card(member)>2}   <-- FALSE
% ! F {card(waiting)>2}  <-- FALSE
% G {card(waiting)+card(member)<=3}    <-- TRUE
% G (e(Join) | e(JoinQ) | e(Remove))  <-- Provid
% G not(sink)                         <-- Provid

\begin{document}

\begin{zed}
  [NAME]\\
 Report ::= yes | no
\end{zed}

\begin{axdef}
 total : \nat_1
\where
 total = 4096
\end{axdef}
```

Enter an LTL formula
use {...} for B predicates,
G,F,X,U,W,R,true,false,not,&,or and => are part of the supported LTL syntax,
use e(op) to check if an operation op is enabled,
use sink to check if no operation is enabled that leads to another state,
use brackets to check what is the next operation, e.g. [reset] => X{db={}},
Past-LTL is supported: Y,H,O,S,T are the duals to X,G,F,U,R

G (e(Join) | e(JoinQ) | e(Remove))

Max no. of new states: 1000

● Start search in initialisation
○ Start search in current state
○ Start in initialisation, but check formula in current state

```
invariant_ok
total=4096
capacity=2
member={}
waiting={NAME1,NAME2,NAME3}
```

```
Query(NAME1)-->(no)
Query(NAME2)-->(no)
Query(NAME3)-->(no)
BACKTRACK
```

```
JoinQ(NAME1)
Remove(NAME1)
JoinQ(NAME2)
Join(NAME1)
Remove(NAME2)
JoinQ(NAME1)
Remove(NAME1)
Join(NAME3)
JoinQ(NAME3)
Join(NAME2)
JoinQ(NAME2)
Join(NAME1)
JoinQ(NAME1)
initialise_machine({},{})
setup_constants(4096,2)
```

Figure 4.1: Counter-example found for the Z model

- `G {card(waiting)+card(member)<=3}` (the formula is true)

Model checking time is 0.08 sec to construct the state space plus less than 0.01 sec for each LTL check. This is faster than the times reported in [DNS06] (ranging from 3 seconds to 12 hours depending on the translation to SAL). In addition, we were able to uncover an error in the specification, namely that it is possible to reach a quasi-deadlock state where only probing operations are possible and no "real" operation can be performed, i.e., the following LTL$^{[e]}$ formula is false:

- `G (e(Join) or e(JoinQ) or e(Remove))`

Note that this error was not uncovered in [DNS06].

We have also tested our tool on the workstation protocol industrial case study from [PL07]. Computation of the state space for 2 workstations took 2.49 seconds, resulting in 68 states.

The formula `G([Transfer]=>X(e(ReadRequestOK) or e(ReadResponse)))` was checked in 0.04 seconds, using 421 atoms.

### 4.6.6 Other Formalisms: StAC, Object Petri nets, dSL

*P*roB has also the ability to load specifications via custom Prolog interpreters following the style of [LM00], describing the initial states, the properties and the transition relation by using the Prolog predicates `start/1`, `prop/2`, `trans/3`.

This directly opens up LTL model checking for three further formalisms, for which we have such interpreters: Compensating CSP (StAC) [BF00], Object Petri Nets, [FL04], and dSL [WGMM05].

We have applied our LTL model checker to a door control system specified in dSL. Computing the state space with 9968 nodes and 37357 transitions took 13.25 seconds. Checking a safety property `G e(action)` took 39.21 seconds.

## 4.7 LTL Model Checking with Symmetry Reduction

Combining full blown LTL model checking and symmetry reduction is not always easy. If one is not careful, the application of symmetry reduction can lead to unsoundness for more complicated LTL formulas. Quite often, only safety properties or some other subset of LTL is supported. E.g., murphi ([ID93]) only deals with safety properties. An exception is, e.g., SMC ([SGE00]), which does deal with safety and liveness properties (expressed as an automata).

It turns out that the presented LTL$^{[e]}$ language is the ideal companion to the existing symmetry reduction techniques developed for *P*roB [LBST06, LM07, TLSB07], i.e., we can apply the symmetry reduction techniques and need to impose no restrictions whatsoever on the LTL$^{[e]}$ formulas. This meant that, in preliminary experiments, we were actually able to model check some examples considerably faster, than using SPIN with partial order reduction on hand-translated Promela models.

Let us recall some of the results from [LBST06]. First, the notion of a permutation is introduced, which can permute elements of deferred sets. *DS* is the set of all deferred sets of the B machine under consideration.

**Definition 4.** *Let DS be a set of disjoint sets. A* permutation $f$ *over* $DS$ *is a total bijection from* $\cup_{S \in DS} S$ *to* $\cup_{S \in DS} S$ *such that* $\forall\, S \in DS$ *we have* $\{f(s) \mid s \in S\} = S$.

The following results show that the deferred sets induce a symmetry in the state space: if in a given state $s$ we permute the deferred set elements, the resulting state will be symmetrical to $s$.

**Theorem 1.** *For any expression* $E$*, predicate* $P$*, state* $[V := C]$ *and permutation function* $f$*:*

$$\begin{array}{rcl} f(E[V := C]) & = & E[V := f(C)] \\ P[V := C] & \Leftrightarrow & P[V := f(C)] \end{array}$$

**Corollary 1.** *Every state permutation* $f$ *for a B machine* $M$ *satisfies*

- $\forall s \in S : s \models I$ *iff* $f(s) \models I$

- $\forall s_1 \in S, \forall s_2 \in S$:
  $s_1 \rightarrow^M_{op.a.b} s_2 \;\Leftrightarrow\; f(s_1) \rightarrow^M_{op.f(a).f(b)} f(s_2)$.

From these theoretical results in [LBST06] we can deduce a new result for LTL$^{[e]}$:

**Proposition 1.** *Let $f$ be a permutation function, $s$ a state of B machine $M$ and $\phi$ a LTL$^{[e]}$ formula. Then $M, s \models \phi$ iff $M, f(s) \models \phi$.*

*Proof.* (Sketch)

- By Theorem 1, if a predicate `{ Pred }` is true in a state then it is true in all permutation states.

- By Corollary 1, if a sequence of operations is possible in $s$ then a permuted sequence is possible in the state $\pi(s)$. As the permutation does not affect the enabled operations nor the operation label itself (just the arguments): we can deduce that a LTL$^{[e]}$ formula is true in $s$ iff it is true in $\pi(s)$.

□

In other words, there exists a LTL$^{[e]}$ counter-example for a B machine iff there exists one with symmetry reduction. As Z specifications are translated internally into B machines ([PL07]), all of the above also applies when model checking Z specifications.

## 4.8 Related and Future Work, Discussion and Conclusion

### 4.8.1 More Related Work

There are variety of relatively generic CTL model checkers, such as [LM00, NL00, DP01]. Both [NL00] and [DP01] are based on constraint logic programming. [NL00] requires constructive negation, and as such only a prototype implementation seems to exist. [DP01] is tailored towards verification of infinite state systems.[5] The CTL model checker in [LM00] is generic, and can be applied to any specification language that can be encoded in Prolog. Unfortunately, the model checker relies on tabling, and as such it can only run on XSB Prolog [SSW94], which does not support co-routines and hence the system can *not* be applied to our interpreters for CSP and B (and hence also Z). The same can be said for the pure LTL model checker from [PR00] or the XMC system [CDD$^+$98, RRR$^+$97] for the modal mu-calculus and value-passing CCS.

[5] Unfortunately, the corresponding DMC prototype available at `http://www.disi.unige.it/person/DelzannoG/DMC/dmc.html` no longer runs on current SICStus Prolog versions and the code is no longer maintained.

### 4.8.2 Future Work

Adding fairness constraints to an LTL$^{[e]}$ formula leads to an exponential growth of the search graph. We plan to incorporate support for fairness directly into the algorithm by validating if a found SCC satisfies the fairness constraints or not.

We think that expressive transition propositions have many useful applications. We want to extend the model checker in a way that combined pre- and post-conditions can be checked, optionally with access to the parameters of the executed operations.

Currently, a counter-example is presented to the user by moving the animation into the final state of the path and having the complete path in the history of the animation (in Fig. 4.1, the box in the right bottom corner shows the history). This is unsatisfactory, as it is not always easy to see why the presented path is a counter-example. We plan to improve the presentation and investigate how the length of the counter-example can be minimised.

### 4.8.3 Conclusion

In summary, we have presented LTL$^{[e]}$ to conveniently express temporal properties of formal models. Indeed, LTL$^{[e]}$ can be used, e.g., to express pre- and post-conditions of operations, fairness constraints as of [CJMB05], the NEXT control annotations from [IST06], as well as a large class of interesting properties which cannot be directly expressed in pure LTL. We have shown an algorithm for LTL$^{[e]}$, proven it correct with and without symmetry reduction, and have integrated it into the *P*roB tool set. In the empirical section, we have shown that LTL$^{[e]}$ is expressive enough and that our tool is fast enough for a variety of practical applications.

# 5 Validating B,Z and TLA$^+$ using *P*roB and Kodkod

Daniel Plagge, Michael Leuschel

**Abstract.** We present the integration of the Kodkod high-level interface to SAT-solvers into the kernel of *P*roB. As such, predicates from B, Event-B, Z and TLA$^+$ can be solved using a mixture of SAT-solving and *P*roB's own constraint-solving capabilities developed using constraint logic programming: the first-order parts which can be dealt with by Kodkod and the remaining parts solved by the existing *P*roB kernel. We also present an empirical evaluation and analyze the respective merits of SAT-solving and classical constraint solving. We also compare to using SMT solvers via recently available translators for Event-B.

**Keywords: B-Method – Z – TLA – Tool Support – SAT – SMT – Constraints.**

## 5.1 Introduction and Motivation

TLA$^+$ [Lam02], B [Abr96] and Z are all state-based formal methods rooted in predicate logic, combined with arithmetic and set theory. The animator and model checker *P*roB [LB03] can be applied to all of these formalisms and is being used by several companies, mainly in the railway sector for safety critical control software [LFFP09, LFFP11]. At the heart of *P*roB is a kernel dealing with the basic data types of these formalisms, i.e., integers, sets, relations, functions and sequences. An important feature of *P*roB is its ability to solve constraints; indeed constraints can arise in many situations when manipulating a formal specification: the tool needs to find values of constants which satisfy the stipulated properties, the tool needs to find acceptable initial values of a model, the tool has to determine whether an event or operation can be applied (i.e., is there a solution for the parameters which make the guard true) or whether a quantified expression is true or not. Other tasks involve more explicit constraint solving, e.g., finding counterexamples to invariant preservation or deadlock freedom proof obligations [HL11]. While *P*roB is good at dealing with large data structures and also at solving certain kinds of complicated constraints [HL11], it can fare badly on certain other constraints, in particular relating to relational composition and transitive closure. (We will illustrate this later in the paper.)

Another state-based formalism is Alloy [Jac02] with its associated tool which uses the Kodkod [TJ07] library to translate its relational logic predicates into propositional formulas which can be fed into SAT solvers. Alloy can deal very well with complicated constraints, in particular those involving relational composition and transitive closure. Compared to B, Z and TLA$^+$, the Alloy language and the Kodkod library only allow first-order predicates, e.g., they do not allow relations over sets or sets of sets.

The goal of this work is to integrate Kodkod into *P*roB, providing an alternative way of solving B, Z and TLA$^+$ constraints. Note that we made sure that the animation and model checking engine as well as the user interface of *P*roB are agnostic as to how the underlying constraints are solved. Based on this integration we also conduct a thorough empirical evaluation of the performance of Kodkod compared to solving constraints with the existing constraint logic programming approach of *P*roB. As we will see later in the paper, this empirical evaluation provides some interesting insights. Our approach also ensures that the whole of B is covered, by delegating the untranslatable higher-order predicates to the existing *P*roB kernel.

## 5.2 B, Z, TLA$^+$ and Kodkod in comparison

*P*roB can support Z and TLA$^+$ by translating those formalisms to B, because these formalisms have a common mathematical foundation. In the case of TLA$^+$ a readable B machine is actually generated, whereas a Z specification is translated to *P*roB's internal representation because some Z constructs did not have a direct counterpart in B's syntax. In the next sections we refer only to B, but because all three notations share the same representation in *P*roB, all presented techniques can be applied likewise to the two other specification languages.

If we specify a problem in B, we basically have a number of variables, each of a certain type and a predicate. The challenge for *P*roB is then to find values for the variables that fulfil the predicate. For simplicity, we ignore B's other concepts like machines, refinement, etc.

Kodkod provides a similar view on a problem. We have to specify a number of relations (these correspond to our variables in B) and a formula (which corresponds to a predicate in B) and Kodkod tries to find solutions for the relations.

From this point of view, the main difference between B and Kodkod is the type system: Instead of having some basic types and operations like power set and Cartesian product to combine these, Kodkod has the concept of a universe consisting of atoms. To use Kodkod, we must define a list of atoms and for each relation we must specify a *bound* that determines a range of atoms that can be in the relation.

The bound mechanism can also be used to assign an exact value to a relation. This is later useful when we have already computed some values by *P*roB.

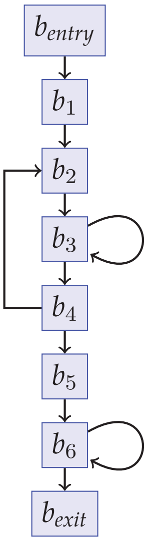


Figure 5.1: Control flow graph of a program

## 5.3 Architecture

### 5.3.1 Overview

We use a small example to illustrate the basic mechanism how Kodkod and SAT solving is integrated into ProB's process to find a model for a problem. Details about the individual components are presented below after this overview.

Our small problem is taken from the "dragon book" [ALSU07] and formalised in B. The aim is to find loops in a control flow graph of a program (see Figure 5.1).

We model the basic blocks as an enumerated set *Blocks* with the elements $b_1$, $b_2$, $b_3$, $b_4$, $b_5$, $b_6$, $b_{entry}$, $b_{exit}$. The successor relation is represented by a variable *succs*, the set of the nodes that constitute the loop by $L$ and the entry point of the loop by *lentry*. The problem is described by the B predicate:

$$\begin{aligned}
&succs = \{b_{entry} \mapsto b_1, b_1 \mapsto b_2, b_2 \mapsto b_3, b_3 \mapsto b_3,\\
&\qquad b_3 \mapsto b_4, b_4 \mapsto b_2, b_4 \mapsto b_5,\\
&\qquad b_5 \mapsto b_6, b_6 \mapsto b_6, b_6 \mapsto b_{exit}\}\\
\wedge\ & lentry \in L\\
\wedge\ & succs^{-1}[L \setminus \{lentry\}] \subseteq L\\
\wedge\ & \forall l.(l \in L \Rightarrow lentry \in (L \lhd succs \rhd L)^{+}[\{l\}])
\end{aligned}$$

In total, there are seven different solutions to this problem, for instance $L = \{b_2, b_3, b_4\}$ with $lentry = b_2$.

After parsing and type checking the predicate, we start a static analysis (the box "Analysis" in Fig. 5.2) to determine the integer intervals of all integer expressions. In our simple case, the analysis is not necessary. In Section 5.3.4 we describe how this analysis

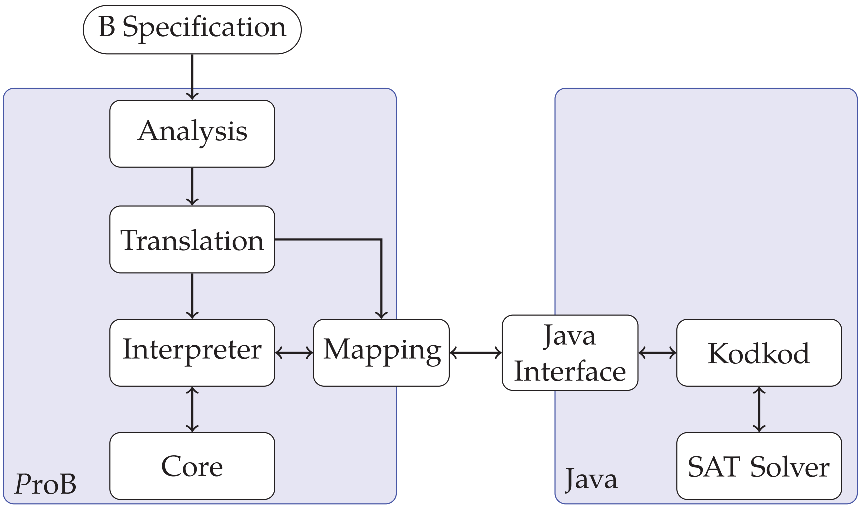


Figure 5.2: Overview of the architecture

works and under which circumstances it is needed.

In the next phase, we try to translate the formula from B to Kodkod ("Translation" in Fig. 5.2). First we have a look at the used variables and their types: *succs* is of type $\mathbb{P}(Blocks \times Blocks)$, $L$ of type $\mathbb{P}(Blocks)$ and *lentry* of type *Blocks*. *Blocks* is here the only basic type that is used. Thus we have to reserve 8 atoms in the Kodkod universe to represent this type; each atom in the universe corresponds directly to a block $b_i$. The variables can be represented by binary (*succs*) and unary ($L$ and *lentry*) relations, where we have to keep in mind that the relation for *lentry* must contain exactly one element. The B predicate can be completely translated to a Kodkod problem. In Section 5.3.2 we will describe the translation in more detail. It can be useful to keep a part of the formula untranslated: since the part $succs = \{\ldots\}$ is very easy to compute by *P*roB, we leave it untranslated. The translated formula has the form:

```
one lentry &&
lentry in L &&
((L-lentry) . ~succs) in L &&
all l: one Blocks | (l in L =>
  lentry in (l.^(((L->Blocks)&succs)&(Blocks->L))))
```

The translated description of the formula is then stored and a mapping between *P*roB's internal representation and Kodkod's representation of values is constructed ("Mapping" in Fig. 5.2). The message "new problem with following properties…" is sent to the Java process.

The Kodkod problem gets a unique identifier and the translated part of the B predicate is replaced by a reference to the problem, i.e., $succs = \{\ldots\} \wedge \texttt{kodkod}(ID)$, and then given to the B interpreter of *P*roB.

When the *P*roB interpreter starts to evaluate the predicate, it prioritises which parts should be computed first. It chooses $succs = \{\ldots\}$ because it can be computed deterministically by *P*roB's core and finds a value for *succs*. Then a message "We have these values for *succs*, try to find values for the other variables" is sent to the Java process.

The Java process has now a complete description of the problem. It consists of the universe (with 8 atoms) and relations for the variables and the type *Blocks* itself. The bounds define the value of *succs* and *Blocks* and ensure that all relations contain only atoms that match their corresponding types. This information is then given together with the formula to the solver of the Kodkod library ("Kodkod" in Fig. 5.2) that translates the problem into a SAT problem and passes this to the SAT solver.

The SAT solver finds solutions that are transformed by Kodkod to instances of the relations that fulfil the given formula The values of the previously unknown relations that represent *L* and *lentry* are sent back in an answer to the *P*roB process. The answer is then mapped to *P*roB's internal representation of values. The B interpreter can now continue with the found values. Now all predicates have been evaluated and the solutions can be presented to the user.

### 5.3.2 Translation

#### Representing values

It turns out that the available data types in Kodkod are the main limitation when trying to translate a problem described in B. Let's first have a look at the available data types in B and how they can be translated to Kodkod. We have the basic data types:

*Enumerated Sets* Enumerated sets can directly be translated to Kodkod. For each element of the set, we add an atom to the universe and create a unary relation that contains exactly that atom. The relation is needed in case the element is referred to in an expression. We create another unary relation for the whole set that contains exactly all atoms of the enumerated set.

*Deferred Sets* Deferred Sets in B can have any number of elements that are not further specified. For animation, *P*roB chooses a fixed finite cardinality for the set, either by an analysis of the axioms or by using user preferences. Then we can treat deferred sets just like a special case of enumerated sets.

*Booleans* The set of Booleans is a special case of an enumerated set with two elements TRUE and FALSE.

*Integer* Integers in B represent mathematical numbers, they can be arbitrary large. It is possible to represent integer values in Kodkod, but the support is very limited and special care has to been taken. We describe the handling of integers in Section 5.3.3 in detail.

Thus, we can map a B variable of a basic data type to a Kodkod relation. Since Kodkod treats every relation as a set, we must ensure explicitly that the relations for such variables contain exactly one element.

**Example.** Let's assume that we use two types in our specification, an enumerated set $E = \{a, b, c\}$ and BOOL. Treating the Booleans as enumerated set $\mathrm{BOOL} = \{\mathrm{TRUE}, \mathrm{FALSE}\}$, we have the following universe with five atoms:

| | $E$ | | | BOOL | |
|---|---|---|---|---|---|
| B value | $a$ | $b$ | $c$ | TRUE | FALSE |
| atom | 0 | 1 | 2 | 3 | 4 |

We can now represent a variable of type $E$ by a unary relation `r1` whose elements are bounded to be a subset of the atoms $0\,..\,2$. We also have to add the Kodkod formula `one r1`.

In B, two or more basic types can be combined with the Cartesian product. Variables of such a type can be represented by a relation.

**Example.** If we have a variable of type $(E \times E) \times \mathrm{BOOL}$, we can represent it by a ternary relation `r2` whose elements are bound to subsets of the atoms $0\,..\,2 \times 0\,..\,2 \times 3\,..\,4$. Like in the example above, we have to add the condition `one r2`.

We can construct the power set $\mathbb{P}(\alpha)$ for any type $\alpha$ in B. A variable of type $\mathbb{P}(\alpha)$ can be mapped to a Kodkod relation if $\alpha$ is itself not a set. A relation for $\mathbb{P}(\alpha)$ is defined exactly as a relation for $\alpha$ but without the additional restriction that it must contain exactly one element.

Finally, let's have a look at what we *cannot* translate. All "higher-order" data-types, i.e. sets of sets are not translatable. E.g a function $f \in A \mathrel{+\!\!\!\to} \mathbb{P}(B)$ cannot be handled.

It turned out that unary and binary relations are handled very well. With relations of a higher arity we encounter the problem that many operators in Kodkod are restricted to binary relations. Thus it is not as easy to translate many properties using these data types.

### Translating predicates and expressions

One of the central tasks in combining *P*roB and Kodkod is the translation of the B predicate that specifies the problem to a Kodkod formula. Many of B's most common operators can be directly translated to Kodkod, especially when basic set theory and relational algebra is used. It is not strictly necessary to cover all operators that B provides, because we always have the possibility to fall back to *P*roB's own constraint solving technique. Of course, we strive to cover as many operators as possible.

**Operators on Predicates.** The basic operators that act on predicates like conjunction, disjunction, etc. have a direct counterpart in Kodkod. This includes also universal and existential quantification.

**Arithmetic operators.** Addition, subtraction and multiplication of integers can also directly be translated, whereas division is not supported by Kodkod. Other supported integer expressions are constant numbers and the cardinality of a set. If we want a

| B | Kodkod |
|---|---|
| $A \in B$ | $\mathcal{T}(A)$ `in` $\mathcal{T}(B)$ |
| $A \subseteq B$ | $\mathcal{T}(A)$ `in` $\mathcal{T}(B)$ |
| $A \times B$ | $\mathcal{T}(A)$ `->` $\mathcal{T}(B)$ |
| $A \mapsto B$ | $\mathcal{T}(A)$ `->` $\mathcal{T}(B)$ |
| $A \cap B$ | $\mathcal{T}(A)$ `&` $\mathcal{T}(B)$ |
| $A \cup B$ | $\mathcal{T}(A)$ `+` $\mathcal{T}(B)$ |
| $A \setminus B$ | $\mathcal{T}(A)$ `-` $\mathcal{T}(B)$ |
| $A[B]$ | $\mathcal{T}(B)$`.`$\mathcal{T}(A)$ |
| $A(B)$ | $\mathcal{T}(B)$`.`$\mathcal{T}(A)$ |
| $A \mathbin{<\!\!+} B$ | $\mathcal{T}(A)$`++`$\mathcal{T}(B)$ |
| $A^{-1}$ | `~`$\mathcal{T}(A)$ |
| $A^{+}$ | `^`$\mathcal{T}(A)$ |

(a) direct translation

| B | Kodkod |
|---|---|
| $\mathrm{dom}(A)$ | `prj[1:`$\mathcal{A}(\alpha)$`](`$\mathcal{T}(A)$`)` |
| | with $A$ being of type $\mathbb{P}(\alpha \times \beta)$ |
| $\mathrm{ran}(A)$ | `prj[`$\mathcal{A}(\alpha)$`+1:`$\mathcal{A}(A)$`](`$\mathcal{T}(A)$`)` |
| | with $A$ being of type $\mathbb{P}(\alpha \times \beta)$ |
| $\mathrm{prj}_1(A)$ | $\mathcal{T}(\mathrm{dom}(A))$ |
| $\mathrm{prj}_2(A)$ | $\mathcal{T}(\mathrm{ran}(A))$ |
| $A \lhd B$ | `(`$\mathcal{T}(A)$ `->` $\mathcal{T}(\beta)$`)` `&` $\mathcal{T}(B)$ |
| | with $B$ being of type $\mathbb{P}(\alpha \times \beta)$ |
| $A \mathbin{\lhd\!\!\!-} B$ | `((univ-`$\mathcal{T}(A)$`)->`$\mathcal{T}(\alpha)$`)` `&` $\mathcal{T}(B)$ |
| | with $B$ being of type $\mathbb{P}(\alpha \times \beta)$ |
| $\mathrm{bool}(P)$ | `if` $\mathcal{T}(P)$ `then` $\mathcal{T}(\mathrm{TRUE})$ `else` $\mathcal{T}(\mathrm{FALSE})$ |
| $f \in A \mathbin{+\!\!\!\rightarrow} B$ | `pfunc(`$\mathcal{T}(f), \mathcal{T}(A), \mathcal{T}(B)$`)` |
| $f \in A \rightarrow B$ | `func(`$\mathcal{T}(f), \mathcal{T}(A), \mathcal{T}(B)$`)` |
| $f \in A \rightarrowtail B$ | `func(`$\mathcal{T}(f), \mathcal{T}(A), \mathcal{T}(B)$`)` `&&` |
| | `(`$\mathcal{T}(f)$`.~`$\mathcal{T}(f)$`)` `in iden` |

(b) more complex rules

Figure 5.3: Examples for translation rules

variable to represent an integer, we have to convert explicitly between a relation that describes the value and an integer expression (see Section 5.3.3).

**Relational operators.** Many operators that act on sets and relations can be translated easily to Kodkod. Figure 5.3a shows a list of operators that have a direct counterpart in Kodkod. With $\mathcal{T}(A)$ we denote the translated version of the expression $A$. Please note that the expressions $A \in B$ and $A \subseteq B$ are translated to the same expression in Kodkod. This is due to the fact that single values are just a special case in Kodkod where a set contains just one element. The same effect can be found at the Cartesian product ($A \times B$) and a pair ($A \mapsto B$) and at the relational image ($A[B]$) and the function application ($A(B)$).

Other operators need a little bit more work. They can be expressed by combining other operators. Figure 5.3b shows a selection of such operators. In the table, we use an operator $\mathcal{A}(E)$ to denote the arity of the relation that represents the expression $E$. Again we can see that different operators in B (e.g. $\mathrm{dom}$ and $\mathrm{prj}_1$) lead to the same result.

### 5.3.3 Integer handling in Kodkod

Kodkod provides only a very limited support for integers. The reason for this is twofold. Since SAT solvers are used as the underlying technology, integers are encoded by binary

| | | | | | | | binary numbers | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | for integer sets | | | | | | | | | | |
| atom | ... | $i_0$ | $i_1$ | $i_2$ | $i_3$ | | $i_4$ | $i_5$ | $i_6$ | $i_7$ | $i_8$ | ... |
| associated integer value | ... | $-1$ | 0 | 3 | 5 | | 1 | 2 | 4 | 8 | $-16$ | ... |

Figure 5.4: Mapping atoms to integer values

numbers. Operations like addition then have to be encoded as boolean formulas. This makes the use of integers ineffective and cumbersome. Another reason is that the designers of Alloy – where Kodkod has its origin – argue [Jac02] that integers are often not very useful and an indication of lack of abstraction when modeling systems.

Our intention is to make our tool applicable to as many specifications as possible, and many of the B specifications we tried contained some integer expressions. Indeed, integers are used to model sequences in B or multi-sets in Z.

When using Kodkod with integers, we have to specify the number of bits used in integer expressions. Integer overflows are silently ignored, e.g. the sum of two large naturals can be negative when the maximum integer size is exceeded. Thus we need to ensure that we use only integers in the specified range to prevent faulty results.

Kodkod distinguishes between relational and integer expressions. An integer expression is for example a constant integer or the sum of two integer expressions. Comparison of integer expressions like "less than" is also supported. In case we want a relation (i.e. a variable) that represents an integer, we must first assign values to some atoms. Figure 5.4 shows an example with a universe consisting of 9 atoms $i_0, \ldots, i_8$ that represent integers.

We have basically two options when we want represent integers by a relation:

- We can represent sets of integers in the interval $a\,..\,b$ by having an atom for each number in $a\,..\,b$. Then the relation simply represents the integers of its atoms.

  E.g. with the universe in Figure 5.4, we can represent arbitrary subsets of $-1\,..\,5$ by using a relation that is bounded to the atoms $i_0, \ldots, i_4$.

  The downside of this approach is that the number of atoms can become easily very large.

- Single integers can be represented more compactly by using a binary number. E.g. with the universe in Figure 5.4, we can represent a number of the interval $-16\,..\,15$ by using a relation that is bounded to the atoms $i_4\,..\,i_8$. A relation that consists of the atoms $i_4$, $i_6$, $i_8$ would represent the sum $1 + 4 + (-16) = -9$. Kodkod provides an operator to summarise the atoms of a relation, yielding an integer expression.

  With this approach large numbers can be handled easily. The downside is that we cannot represent sets of numbers.

The atoms in the universe seen in Figure 5.4 are ordered in a way that we can use both approaches to represent integers in the same specification.

It can be seen that we need an exact knowledge of the possible size of integer expressions in the specification. To get the required information, a static analysis is applied to the specification before the translation. See below in Section 5.3.4 for details of the analysis.

Another problem that arises from having two kinds of integer representations, is that we have to ensure the consistency of formulas that use integer expressions. We briefly describe the problem in Section 5.3.5.

### 5.3.4 Predicate Analysis

In case that integers are used in the model, we need to know how large they can get in order to translate the expressions. To get this information we apply a static analysis on the given problem.

The first step of the analysis is that we create a graph that describes a constraint problem. For each expression in the abstract syntax tree, we create some of nodes depending on the expression's type that contain relevant information associated to the syntax node. E.g. this might be the possible interval for integers, the interval of the cardinality for sets or the interval in which all elements of a set lie.

By applying pattern matching on the syntax tree, we add rules that describe the flow of information in the graph. E.g. if we have a predicate $A \subset B$, we can propagate all information about elements of $B$ to nodes that contain information about $A$. We evaluate all rules until a fixpoint or a maximum number of evaluation steps is reached.

**Example.** Let's take the predicate $A \subseteq 3\,..\,6 \wedge card(A) > 1$. For each integer node ($1$, $3$, $6$, $card(A)$) we create a node containing the integer range. For each of the sets $A$ and $3\,..\,6$ we create two nodes: One contains the range of the set's cardinality, the other describes the integer range of the set's elements. Figure 5.5 shows the resulting graph. In the upper part of each node the expression is shown, in the lower part the kind of information that is stored in the node. The edges without any labels denote rules that pass information just from one node to another. Those which are labeled with $\leq$, $\geq$, $<$, $>$ express a relation between integer ranges of the source and target node. There is a special rule marked $i \rightarrow c$ ("interval to cardinality") which deduces a maximum cardinality from the allowed integers in a set. E.g. if all elements of a set $I$ are in the range $x\,..\,y$, we know that $card(I) \leq y - x + 1$. The graph in Figure 5.5 shows the information we have about each node after the analysis. In particular, we know the bounds of all integer expressions.

Currently the analysis is limited to integer intervals and cardinality, because this was the concrete use case given by our translation to Kodkod. We plan to re-use the analysis for other aspects of *P*roB. E.g. if we can deduce the interval of a quantified integer variable, *P*roB can limit the enumeration of values to that range if it must test a predicate for all possible values of that variable.

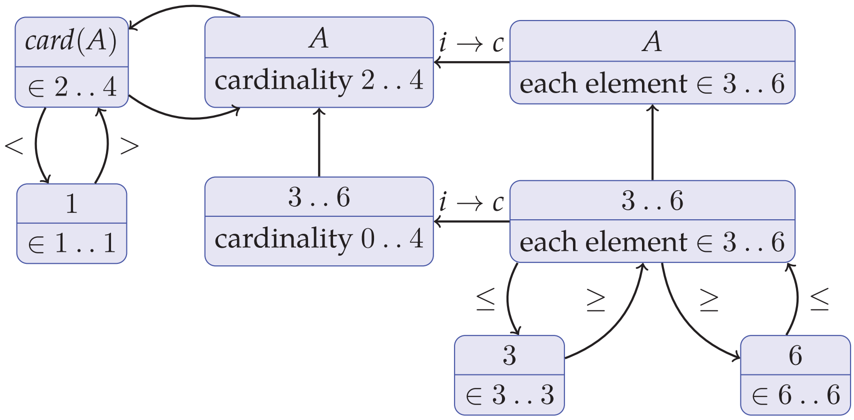


Figure 5.5: Constraint system for $A \subseteq 3..6 \wedge card(A) > 1$

Other types of information nodes are also of interest. For instance, we could infer the information if an expression is a function or sequence to assist $P$roB when evaluating predicates.

### 5.3.5 Integer representations

We have seen above in Section 5.3.3 that we have two distinct forms of representing integers. Additionally we have Kodkod's integer expressions when we want to compare, add, subtract or multiply them. During the translation process we must ensure that the correct representation is chosen for each expression and that the representations are consistently used. If the take a simple equality $A = B$ with $A$, $B$ being integers as an example, we must ensure that both sides use the same representation.

Internally, the result of the translation is an abstract syntax tree that describes the formula. Some expressions in this syntax tree have annotations about the needed integer representation. E.g., if the original expression in B is a set of integers, it has the annotation that the integer set representation and not the the binary number representation must be used. We now impose a kind of type checking on this syntax tree to infer if conversions between different integer representations have to be inserted in the formula.

### 5.3.6 Extent of the translation: Partitioning

Theoretically we can take any translatable sub-predicate of a specification and replace it with a call to Kodkod. But usually, the overhead due to the communication between the processes can easily get so large that incorporating Kodkod has no advantage over using $P$roB alone.

A more sensible approach for a specification that is a conjunction of predicates $P_1 \wedge \ldots \wedge P_n$ is to apply the translation to every $P_i$. All translatable predicates are then replaced by one single call to Kodkod. But even here we made the experience that the communication overhead can become large if not all predicates are translated.

Our current approach is to create a partition of the predicates $P_1, \ldots, P_n$. Two predicates are then in the same set of the partition if they both use the same variable. We translate only complete partitions to keep the communication overhead small. There is one exception: We do not translate simple equations where one side is a variable and the other side an easy to compute constant. Such deterministic equations are computed first by the constraint solver, so the value for such a variable will be computed before the call to Kodkod is made. This keeps the translated formula small even for a large amount of data.

## 5.4 Experiments

We have chosen a number of problems to compare the performance of *P*roB's constraint solving technique and Kodkod's SAT solving approach. We have only used problems that can be completely translated. Completely translated models are still fully integrated into ProB, the results are converted to *P*roB's internal format and can be used for further animation and model checking. The results can be seen in Table 5.1.[1] All experiments were conducted on a dual-core Intel i7 2.8 GHz processor running under Linux. MiniSat was used as a SAT solver. The measured times do not contain time for starting up *P*roB and for loading, parsing and type-checking the model. We measured the time to compute all solutions to each problem. For Kodkod, we measured two different times: The "total time" includes the translation of the problem, the communication between the two processes and the time needed by the solver to produce solutions. The "solver time" is the time that the Kodkod solver itself needs to find solutions, without the overhead of translation and communication between the processes.

### 5.4.1 Analysis

**Relational operators** Let's have a look at those problems where Kodkod is much faster than *P*roB. These are "crew allocation", "loop detection" and "WRSPM". In all these problems we search for instances for sets or relations and the problem is described by relational operators and universal quantification. In this scenario, *P*roB sometimes starts to enumerate possible instances for the sets or relations which leads to a dramatic decrease of performance.

**Arithmetic and large relations** The arithmetic problems ("Send More Money", "Eratosthenes' Sieve") are solved by ProB much faster than by Kodkod. The first two problems deal with arithmetic. *P*roB uses internally a very efficient finite domain solver (CLP/FD) to tackle such problems. On the other side, arithmetic is one of the weaknesses of Kodkod, as we already pointed out.

[1] The source code of the examples are available in the technical report at: `http://www.stups.uni-duesseldorf.de/w/Special:Publication/PlaggeLeuschel_Kodkod2012`

| | *P*roB | Kodkod | |
|---|---|---|---|
| Model | | total | solver |
| Who Killed Agatha? | 177 | 123 | 12 |
| Crew Allocation | timeout* | 297 | 112 |
| 20–Queens | 110 | 8223 | 8076 |
| Graph Colouring (integer sets) | 50 | 2323 | 1859 |
| Graph Colouring (enumerated sets) | 50 | 1037 | 818 |
| Graph Isomorphism | 13 | 553 | 379 |
| Loop detection in control flow | 23037 | 117 | 12 |
| SAT instance | 11830 | 4143 | 588 |
| Send More Money | 7 | 1773 | 1578 |
| Eratosthenes' sieve (1 step) | 7 | 5833 | 5712 |
| Union of two sets (2000 elements) | 33 | 4880 | 4659 |
| Requirements WRSPM model | timeout* | 333 | 89 |
| BPEL deadlock check | 20 | 337 | 68 |

*: interrupted after 60 seconds

Table 5.1: Comparing *P*roB and Kodkod (in milliseconds)

Kodkod does not seem to scale well when encountering large relations (e.g. "union of two sets"). This has only been relevant for certain applications of *P*roB, such as the property verification on real data [LFFP09].

The graph colouring, graph isomorphism and 20-Queens problems are clearly faster solved by *P*roB. The structure of the problem is somehow fixed (by having e.g. total functions) and constraint propagation is very effective.

**Room for optimization** It can be seen that the graph colouring problem needs less than half the time when it is encoded with enumerated sets instead of integers. This indicates that the translation is not yet as effective as it should be. For the "SAT" problem, the translation and communication takes six time as long as the computation of the problem itself. This shows that we should investigate if we can optimize the communication.

### 5.4.2 SMT and other tools

Very recently, an Event-B to SMT-Lib converter has become available for the Rodin platform [DFGV12]. This makes it possible to use SMT solvers (such as veriT, CVC and Z3 [dMB08]; we used version 3.2 of the latter within the Rodin SMT Solvers Plug-in 0.8.0r14169 in our experiments below) on Event-B proof obligations. We have experimented with the translator on the examples from Table 5.1.[2] This is done by adding a theorem `1=2` to the model: this generates an unprovable proof obligation which in turn produces a satisfiable SMT formula encoding the problem. For "Send More Money" from Table 5.1 Z3 initially reported "unknown". After rewriting the model (making the inequalities explicit), Z3 was able to determine the solution after about 0.250 seconds. It is thus faster than Kodkod, but still slower than *P*roB. Surprisingly, Z3 was unable to solve the SMT-Lib translations for most of the other examples, such as the "Who killed Agatha" example, the "Set Union" example or the "Graph Colouring" example. Similarly, for the "Crew Allocation" example, Z3 was unable to find a solution already for three flights.[3] Furthermore, for the constraint solving tasks related to deadlock checking, Z3 was not able to solve the translations of the simpler examples from [HL11]. It is too early for a conclusive result, but it seems that more work needs to be put into the B to SMT-Lib translator for this approach to be useful for model finding, animation or constraint-based checking.

Other tools for B are AnimB [Ani], Brama and BZTT [LPU02]. They all have much weaker constraint-solving capabilities (see [LFFP09, LFFP11]) and are unable to solve most of the problems in Table 5.1. Another tool is TLC [YML99] for TLA+. It is very good at model checking, but constraints are solved by pure enumeration. As such, TLC is unable to solve, e.g., a 20 variable SAT problem, the NQueens problem for N>9 and takes more than 2 hours for a variation of the graph isomorphism problem from Table 5.1.

[2] Apart from "loop" which cannot be easily translated to Event-B due to the use of transitive closure.

[3] *P*roB solves this version in 0.06 seconds; Table 5.1 contains the problem for 20 flights.

## 5.5 More Related Work, Discussion and Conclusion

### 5.5.1 Alternative Approaches

Before starting our translation to Kodkod, we had experimented with several other alternate approaches to solve constraints in *P*roB. [WL04] offers the user a Datalog-like language that aims to support program analysis. It uses BDDs to represent relations and compute queries on these relations. In particular, one has to represent a state of the model as a bit-vector and events have to be implemented as relations between two of those bit-vectors. These relations have to be constructed by creating BDDs directly with the underlying BDD library (JavaBDD) and storing them into a file. Soon after starting experimenting with BDDBDDB it became apparent that due to the lack of more abstract data types than bit vectors, the complexity of a direct translation from B to BDDBDDB was too high, even for small models, and this avenue was abandoned.

SAL [SAL] is a model-checking framework combining a range of tools for reasoning about systems. The SAL tool suite includes a state of the art symbolic (BDD-based) and bounded (SAT-based) model checkers. Some first results were encouraging for a small subset of the Event-B language, but the gap between B and SAL turned out to be too big in general and no realistic way was found to handle important B operators.[4] More details about these experiments can be found in [PLLR09]. For Z, there is an ongoing attempt to use SAL for model checking Z specifications [DNS08, DNS11]. The examples presented in [DNS08, DNS11] are still relatively simple and pose no serious challenge in constraint solving. As the system is not publicly available, it is unclear how it will scale to more complicated specifications and constraints.

### 5.5.2 More Related Work

The first hand-translation of B to Alloy was undertaken in [MB02]. The paper [MMS08] contains first experiments in translating Event-B to Alloy; but the work was also not pursued. Later, [MGL10] presented a prototype Z to Alloy converter. The current status of this system is available at the website `http://homepages.ecs.vuw.ac.nz/~petra/zoy/`; the applicability seems limited by the lack of type inference and limited support for schemas. In contrast to these works, we translate directly to Kodkod and have a fully developed system, covering large subsets of B, Event-B, Z and TLA$^+$ and delegating the rest to the *P*roB kernel.

A related system that translates a high-level logic language based on inductive definitions to SAT is IDP [WMD10]. Another recent addition is Formula from Microsoft [JLB11], which translates to the SMT solver Z3 [dMB08].

### 5.5.3 Future Work

Currently our translation is only applicable for finding constraints satisfying the axioms as well as for constraint based deadlock checking. We are, however, working to also

[4] Private communication from Alexei Iliasov and Ilya Lopatkin, March 6th, 2012.

make it available for computing enabled events as well as for more general constraint-based testing and invariant checking.

Another avenue is to enlarge the area of applicability to some recurrent patterns of higher-order predicates. For example, many B specifications use total functions of the type `f : DOM -> POW(RAN)` which cannot be translated as such to Kodkod. However, such functions can often be translated to relations of the form `fr : DOM <-> RAN` by adapting the predicates accordingly (e.g., translating `f(x)` to `fr[{x}]`). More work is also needed on deciding automatically when to attempt the Kodkod translation and when predicates should be better left to the existing *P*roB kernel. Finally, inspired by the experiments, we also plan to improve the *P*roB kernel for better solving constraints over relational operators such as composition and closure.

### 5.5.4 Conclusion

After about three years of work and several attempts our translation to Kodkod is now mature enough to be put into practice and has been integrated into the latest version of the *P*roB toolset. The development required a considerable number of subsidiary techniques to be implemented. As our experiments have shown that the translation can be highly beneficial for certain kinds of constraints, and as such opens up new ways to analyze and validate formal specifications in B, Z and TLA$^+$. However, the experiments have also shown that the constraint logic programming approach of *P*roB can be superior in a considerable number of scenarios; the translation to Kodkod and down to SAT is not (yet) the panacea. The same can be said of the existing translations from B to SMT. As such, we believe that much more research is required to reap the best of both worlds (SAT/SMT and constraint programming). An interesting side-effect of our work is that the *P*roB toolset now provides a double-chain (relying on technology developed independently and using different programming languages and paradigms) of validation for first-order predicates, which should prove relevant in high safety integrity level contexts.

### Acknowledgements

We want to thank Sebastian Krings for helping us in the SMT experiments. We also thank Alexei Iliasov and Ilya Lopatkin for their earlier experiments involving translating Event-B to SAL and the insights gained in that work. We are also grateful for discussions with Daniel Jackson and Emina Torlak, in particular relating to encoding integers in Kodkod.

# 6 Conclusion

The leitmotif of the articles presented in this work is to simplify the life of an engineer who is concerned with validation and verification of formal models. We have shown how the work presented in the four articles had an impact on validation:

- The *P*roZ extension and the support for animation of refinements made animation applicable to formalisms where such support was very limited before. We have explained how animation itself has an impact to validation by making a system understandable to the engineer and even to domain experts that have no expert knowledge in formal methods. It also makes it possible to check whether a system has desired properties.
- The LTL extension allows to formalise additional requirements about the system and to check automatically whether the system under development fulfills them.
- The translation to Kodkod which is hardly noticed by the user makes animation applicable to a broader range of specifications and improves its performance.

Verification is affected by

- supporting the engineer when he is trying to proof properties of the system by detecting errors through explicit and constraint-based model-checking which is both based on the ability to animate the underlying formalism. This is realized in the articles about *P*roZ and about animation support for refinements.
- The model checker for LTL provides a way to verify if a model holds the specified properties.
- Through the impact to animation the Kodkod translation also improves *P*roB support in animation and analysation of specifications and specific problems (e.g. in the disprover) which again is used for support of the engineers verification process.

All extensions are implemented as extensions to the *P*roB tool in such a way that they can be easily applied by the engineer.

# Bibliography

[ABC+02] 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 FATES'02*, pages 105–120, 2002, technical Report, INRIA.

[ABH06] Jean-Raymond Abrial, Michael Butler and Stefan Hallerstede, An open extensible tool environment for Event-B. In Liu and He [LH06], pages 588–605.

[Abr96] Jean-Raymond Abrial, *The B-Book*. Cambridge University Press, 1996.

[Abr10] Jean-Raymond Abrial, *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, 2010.

[ACM05] Jean-Raymond Abrial, Dominique Cansell and Dominique Méry, Refinement and reachability in Event-B. In Treharne et al. [TKHS05], pages 222–241.

[AH07] Jean-Raymond Abrial and Stefan Hallerstede, Refinement, decomposition, and instantiation of discrete models: Application to Event-B. *Fundamenta Informaticae*, volume 77(1–2):pages 1–28, 2007.

[ALSU07] Alfred V. Aho, Monica S. Lam, Ravi Sethi and Jeffrey D. Ullman, *Compilers. Principles, Techniques, and Tools (Second Edition)*. Addison Wesley, 2007.

[Ani] `http://www.animb.org/index.xml`. AnimB Homepage.

[ASA08] Idir Aït-Sadoune and Yamine Aït Ameur, Animating Event B models by formal data models. In Tiziana Margaria and Bernhard Steffen, editors, *ISoLA*, volume 17 of *Communications in Computer and Information Science*, pages 37–55, Springer, 2008.

[ASM80] J.-R. Abrial, S. A. Schuman and B. Meyer, Specification language. In R. M. McKeag and A. M. Macnaghten, editors, *On the Construction of Programs: An Advanced Course*, pages 343–410, Cambridge University Press, 1980.

[B-C02] B-Core (UK) Ltd, Oxon, UK, B-Toolkit, online manual. `http://web.archive.org/web/20041012141220/http://www.b-core.com/ONLINEDOC/BToolkit.html`, 2002, the archived link was accessed February 2015.

[Bac89] Ralph-Johan Back, Refinement calculus, part II: Parallel and reactive programs. In J. W. de Bakker, Willem P. de Roever and Grzegorz Rozenberg, editors, *REX Workshop*, volume 430 of *Lecture Notes in Computer Science*, pages 67–93, Springer, 1989.

[Bar11] Janet Elizabeth Barnes, Experiences in the industrial use of formal methods. *ECEASST*, volume 46, 2011, URL `http://journal.ub.tu-berlin.de/eceasst/article/view/680`.

[BB02] Héctor Ruíz Barradas and Didier Bert, Specification and proof of liveness properties under fairness assumptions in B event systems. In Michael J. Butler, Luigia Petre and Kaisa Sere, editors, *IFM*, volume 2335 of *Lecture Notes in Computer Science*, pages 360–379, Springer, 2002.

[BBBB08] Egon Börger, Michael Butler, Jonathan P. Bowen and Paul Boca, editors, *Abstract State Machines, B and Z, First International Conference, ABZ 2008, London, UK, September 16-18, 2008. Proceedings*, volume 5238 of *Lecture Notes in Computer Science*, Springer, 2008.

[BCD$^+$13] Jens Bendisposto, Joy Clark, Ivaylo Dobrikov, Philipp Körner, Sebastian Krings, Lukas Ladenberger, Michael Leuschel and Daniel Plagge, Prob 2.0 tutorial. In *Proceedings of the 4th Rodin User and Developer Workshop*, TUCS Lecture Notes, TUCS, 2013, URL `http://stups.hhu.de/ProB/index.php5/Tutorial13`.

[BDJK01] Françoise Bellegarde, Christophe Darlot, Jacques Julliand and Olga Kouchnarenko, Reformulation: A way to combine dynamic properties and B refinement. In José Nuno Oliveira and Pamela Zave, editors, *FME*, volume 2021 of *Lecture Notes in Computer Science*, pages 2–19, Springer, 2001.

[BF00] Michael J. Butler and Carla Ferreira, A process compensation language. In Wolfgang Grieskamp, Thomas Santen and Bill Stoddart, editors, *IFM*, volume 1945 of *Lecture Notes in Computer Science*, pages 61–76, Springer, 2000.

[BFL10] Jens Bendisposto, Fabian Fritz and Michael Leuschel, Developing Camille, a text editor for Rodin. In *Workshop on Tool Building in Formal Methods 2010*, pages 38–40, 2010.

[BL05] Michael Butler and Michael Leuschel, Combining CSP and B for specification and property verification. In John Fitzgerald, Ian J. Hayes and Andrzej Tarlecki, editors, *FM*, volume 3582 of *Lecture Notes in Computer Science*, pages 221–236, Springer, 2005.

[BLLS08] Jens Bendisposto, Michael Leuschel, Olivier Ligot and Mireille Samia, La validation de modèles Event-B avec le plug-in ProB pour RODIN. *Technique et Science Informatiques*, volume 27(8):pages 1065–1084, 2008.

[BLP02] Fabrice Bouquet, Bruno Legeard and Fabien Peureux, CLPS-B – a constraint solver for B. In Joost-Pieter Katoen and Perdita Stevens, editors, *TACAS*, volume 2280 of *Lecture Notes in Computer Science*, pages 188–204, Springer, 2002.

[BM13] Michael Butler and Issam Maamria, Practical theory extension in event-b. In Zhiming Liu, Jim Woodcock and Huibiao Zhu, editors, *Theories of Programming and Formal Methods - Essays Dedicated to Jifeng He on the Occasion of His 70th Birthday*, volume 8051 of *Lecture Notes in Computer Science*, pages 67–81, Springer, 2013.

[Bör03] Egon Börger, The ASM refinement method. *Formal Aspects of Computing*, volume 15:pages 237–257, 2003.

[Bow96] Jonathan P. Bowen, *Formal Specification and Documentation using Z*. International Thomson Computer Press, 1996.

[BPS05] 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 Treharne et al. [TKHS05], pages 299–318.

[BS03] E. Börger and R. Stärk, *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer, 2003.

[CCO$^+$05] Sagar Chaki, Edmund Clarke, Joël Ouaknine, Natasha Sharygina and Nishant Sinha, Concurrent software verification with states, events, and deadlocks. *Formal Aspects of Computing*, volume V17(4):pages 461–483, 2005.

[CDD$^+$98] Baoqiu Cui, Yifei Dong, Xiaoqun Du, K. Narayan Kumar, C. R. Ramakrishnan, I. V. Ramakrishnan, Abhik Roychoudhury, Scott A. Smolka and David Scott Warren, Logic programming and model checking. In Catuscia Palamidessi, Hugh Glaser and Karl Meinke, editors, *PLILP/ALP*, volume 1490 of *Lecture Notes in Computer Science*, pages 1–20, Springer, 1998.

[CES09] Edmund M. Clarke, E. Allen Emerson and Joseph Sifakis, Model checking: Algorithmic verification and debugging. *Communications of the ACM*, volume 52(11):pages 74–84, 2009.

[CGP99] Edmund M. Clarke, Orna Grumberg and Doron Peled, *Model Checking*. MIT Press, 1999.

[Chu36] Alonzo Church, An unsolvable problem of elementary number theory. *American Journal of Mathematics*, volume 58(2):pages 345–363, 1936.

[CJMB05] Samir Chouali, Jacques Julliand, Pierre-Alain Masson and Françoise Bellegarde, PLTL-partitioned model checking for reactive systems under fairness assumptions. *ACM Transactions in Embedded Computing Systems*, volume 4(2):pages 267–301, 2005.

[CW98] Ana Cavalcanti and Jim Woodcock, A weakest precondition semantics for Z. *The Computer Journal*, volume 41(1):pages 1–15, 1998.

[DEF03] Daniel Dollé, Didier Essamé and Jérôme Falampin, B dans le tranport ferroviaire. L'expérience de Siemens Transportation Systems. *Technique et Science Informatiques*, volume 22(1):pages 11–32, 2003.

[DFG$^+$12] John Derrick, John A. Fitzgerald, Stefania Gnesi, Sarfraz Khurshid, Michael Leuschel, Steve Reeves and Elvinia Riccobene, editors, *Abstract State Machines, Alloy, B, VDM, and Z - Third International Conference, ABZ 2012, Pisa, Italy, June 18-21, 2012. Proceedings*, volume 7316 of *Lecture Notes in Computer Science*, Springer, 2012.

[DFGV12] David Déharbe, Pascal Fontaine, Yoann Guyot and Laurent Voisin, SMT solvers for Rodin. In Derrick et al. [DFG$^+$12], pages 194–207.

[DG07] Jim Davies and Jeremy Gibbons, editors, *Integrated Formal Methods, 6th International Conference, IFM 2007, Oxford, UK, July 2-5, 2007, Proceedings*, volume 4591 of *Lecture Notes in Computer Science*, Springer, 2007.

[dMB08] Leonardo Mendonça de Moura and Nikolaj Bjørner, Z3: An efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340, Springer, 2008.

[DNS06] John Derrick, Siobhán North and Tony Simons, Issues in implementing a model checker for Z. In Liu and He [LH06], pages 678–696.

[DNS08] John Derrick, Siobhán North and Anthony J. H. Simons, Z2SAL - building a model checker for Z. In Börger et al. [BBBB08], pages 280–293.

[DNS11] John Derrick, Siobhán North and Anthony J. H. Simons, Z2SAL: a translation-based model checker for Z. *Formal Aspects of Computing*, volume 23(1):pages 43–71, 2011.

[DP01] Giorgio Delzanno and Andreas Podelski, Constraint-based deductive model checking. *Software Tools for Technology Transfer*, volume 3(3):pages 250–270, 2001.

[DS04] John Derrick and Graeme Smith, Linear temporal logic and Z refinement. In Charles Rattray, Savi Maharaj and Carron Shankland, editors, *AMAST'04*, volume 3116 of *Lecture Notes in Computer Science*, pages 117–131, Springer, 2004.

[Dun04] Steve Dunne, Understanding Object-Z operations as generalised substitutions. In *IFM*, pages 328–342, Springer, 2004.

[DVR96] Ben L. Di Vito and Larry W. Roberts, Using formal methods to assist in the requirements analysis of the space shuttle GPS change request. Technical report, NASA Langley, 1996.

[EB02] Neil Evans and Michael Butler, A proposal for records in Event-B. In Lars-Henrik Eriksson and Peter A. Lindsay, editors, *FM*, volume 2391 of *Lecture Notes in Computer Science*, pages 221–235, Springer, 2002.

[EBM$^+$12] Andrew Edmunds, Michael Butler, Issam Maamria, Renato Silva and Chris Lovell, Event-B code generation: type extension with theories. In Derrick et al. [DFG$^+$12], pages 365–368.

[ED07] Didier Essamé and Daniel Dollé, B in large-scale projects: The Canarsie line CBTC experience. In Julliand and Kouchnarenko [JK06], pages 252–254.

[EL02] Lars-Henrik Eriksson and Peter A. Lindsay, editors, *FME 2002: Formal Methods – Getting IT Right, International Symposium of Formal Methods Europe, Copenhagen, Denmark, July 22-24, 2002, Proceedings*, volume 2391 of *Lecture Notes in Computer Science*, Springer, 2002.

[Els11] 2011 impact factors computer science. `https://web.archive.org/web/20130128000823/http://about.elsevier.com/impactfactor/aauthor-reports-4506138/webpage/author-webpage-4506138.html`, 2011, the archived link was accessed January 2015.

[FGG07] Roozbeh Farahbod, Vincenzo Gervasi and Uwe Glässer, CoreASM: An extensible ASM execution engine. *Fundamenta Informaticae*, volume 77(1–2):pages 71–103, 2007.

[FGK$^+$10] Marc Frappier, Uwe Glässer, Sarfraz Khurshid, Régine Laleau and Steve Reeves, editors, *Abstract State Machines, Alloy, B and Z, Second International Conference, ABZ 2010, Orford, QC, Canada, February 22-25, 2010. Proceedings*, volume 5977 of *Lecture Notes in Computer Science*, Springer, 2010.

[FHB$^+$14] Andreas Fürst, Thai Son Hoang, David Basin, Krishnaji Desai, Naoto Sato and Kunihiko Miyazaki, Code generation for Event-B. In Elvira Albert and Emil Sekerinski, editors, *IFM*, volume 8739 of *Lecture Notes in Computer Science*, pages 323–338, Springer, 2014.

[FL04] Berndt Farwer and Michael Leuschel, Model checking object Petri nets in Prolog. In Eugenio Moggi and David Scott Warren, editors, *PPDP*, pages 20–31, ACM Press, 2004.

[FM] Rodin handbook. `https://web.archive.org/web/20150222225756/http://handbook.event-b.org/current/html/`, the archived link was accessed February 2015.

[fS94] International Organization for Standardization, *ISO 8402: 1994: Quality Management and Quality Assurance : Vocabulary*. International Organization for Standardization, 1994.

[Goe13] René Goebbels, *Worksheet für die Interaktion mit ProB*. Master's thesis, Heinrich-Heine-Universität Düsseldorf, Lehrstuhl für Softwaretechnik und Programmiersprachen, 2013.

[Gro06a] Julien Groslambert, A JAG extension for verifying LTL properties on B event systems. In Julliand and Kouchnarenko [JK06], pages 262–265.

[Gro06b] Julien Groslambert, Verification of LTL on B event systems. In Julliand and Kouchnarenko [JK06], pages 109–124.

[Hal90] Anthony Hall, Seven myths of formal methods. *IEEE Software*, volume 7(5):pages 11–19, 1990.

[Hal96] Anthony Hall, Using formal methods to develop an ATC information system. *IEEE Software*, volume 13:pages 66–76, 1996, reprinted in Industrial-Strength Formal Methods in Practice, M.G. Hinchey & J.P. Bowen, Springer, 1999.

[Hal02] Anthony Hall, Correctness by construction: Integrating formality into a commercial development process. In Eriksson and Lindsay [EL02], pages 224–233.

[Hal09a] Stefan Hallerstede, Proving Quicksort Correct in Event-B. *Electronic Notes in Theoretical Computer Science*, volume 259:pages 47–65, 2009.

[Hal09b] Stefan Hallerstede, A (small) improvement of Event-B? In *Proceedings of Dagstuhl Seminar on Refinement Based Methods for the Construction of Dependable Systems (09381)*, pages 48–52, Dagstuhl, 2009.

[HD01] John Hatcliff and Matthew B. Dwyer, Using the Bandera tool set to model-check properties of concurrent Java software. In Kim Guldstrand Larsen and Mogens Nielsen, editors, *CONCUR*, volume 2154 of *Lecture Notes in Computer Science*, pages 39–58, Springer, 2001.

[HH07] Stefan Hallerstede and Thai Son Hoang, Qualitative probabilistic modelling in Event-B. In Davies and Gibbons [DG07], pages 49–63.

[HHS86] Jifeng He, C. A. R. Hoare and Jeff W. Sanders, Data refinement refined. In Bernard Robinet and Reinhard Wilhelm, editors, *ESOP*, volume 213 of *Lecture Notes in Computer Science*, pages 187–196, Springer, 1986.

[HL09] Stefan Hallerstede and Michael Leuschel, How to explain mistakes. In Jeremy Gibbons and José Nuno Oliveira, editors, *TFM*, volume 5846 of *Lecture Notes in Computer Science*, pages 105–124, Springer, 2009.

[HL11] Stefan Hallerstede and Michael Leuschel, Constraint-based deadlock checking of high-level specifications. *Theory and Practice of Logic Programming*, volume 11(4–5):pages 767–782, 2011.

[HL12] Dominik Hansen and Michael Leuschel, Translating TLA+ to B for validation with ProB. In John Derrick, Stefania Gnesi, Diego Latella and Helen Treharne, editors, *IFM*, volume 7321 of *Lecture Notes in Computer Science*, pages 24–38, Springer, 2012.

[HLP10] Stefan Hallerstede, Michael Leuschel and Daniel Plagge, Refinement-animation for Event-B - towards a method of validation. In Frappier et al. [FGK$^+$10], pages 287–301.

[HLP13] Stefan Hallerstede, Michael Leuschel and Daniel Plagge, Validation of formal models by refinement animation. *Science of Computer Programming*, volume 78(3):pages 272–292, 2013, doi: 10.1016/j.scico.2011.03.005.

[Hoa85] C.A.R. Hoare, *Communicating Sequential Processes*. Prentice-Hall, 1985.

[Hol97] Gerard J. Holzmann, The model checker Spin. *IEEE Transactions on Software Engineering*, volume 23(5):pages 279–295, 1997.

[HST98] D. Hazel, P. Strooper and O. Traynor, Requirements engineering and verification using specification animation. In *Proceedings of the 13th IEEE international conference on Automated software engineering*, ASE '98, page 302, IEEE Computer Society Press, 1998.

[ID93] C. Norris Ip and David L. Dill, Better verification through symmetry. In David Agnew, Luc J. M. Claesen and Raul Camposano, editors, *CHDL*, volume A-32 of *IFIP Transactions*, pages 97–111, North-Holland, 1993.

[IST06] Wilson Ifill, Steve A. Schneider and Helen Treharne, Augmenting B with control annotations. In Julliand and Kouchnarenko [JK06], pages 34–48.

[Jac97] Jonathan Jacky, *The Way of Z: Practical Programming with Formal Methods*. Cambridge University Press, 1997, URL `http://www.radonc.washington.edu/prostaff/jon/z-book/`.

[Jac02] Daniel Jackson, Alloy: A lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology*, volume 11:pages 256–290, 2002.

[Jac12] Daniel Jackson, *Software Abstractions*. MIT Press, 2012.

[Jia95] Xiaoping Jia, An approach to animating Z specifications. In *Proceedings of COMPSAC'95*, pages 108–113, IEEE Computer Society Press, 1995.

[JK06] Jacques Julliand and Olga Kouchnarenko, editors, *B 2007: Formal Specification and Development in B 7th International Conference of B Users, Besançon, France, January 17-19, 2007. Proceedings*, volume 4355 of *Lecture Notes in Computer Science*, Springer, 2006.

[JLB11] Ethan K. Jackson, Tihamer Levendovszky and Daniel Balasubramanian, Reasoning about metamodeling with formal specifications and automatic proofs. In Jon Whittle, Tony Clark and Thomas Kühne, editors, *MoDELS*, volume 6981 of *Lecture Notes in Computer Science*, pages 653–667, Springer, 2011.

[Jon90] Cliff B. Jones, *Systematic Software Development Using VDM*. Prentice-Hall, 2nd edition, 1990.

[KBL14] 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.

[Lam02] Leslie Lamport, *Specifying Systems, The TLA$^+$ Language and Tools for Hardware and Software Engineers*. Addison Wesley, 2002.

[LB03] Michael Leuschel and Michael J. Butler, ProB: A model checker for B. In Keijiro Araki, Stefania Gnesi and Dino Mandrioli, editors, *FME*, volume 2805 of *Lecture Notes in Computer Science*, pages 855–874, Springer, 2003.

[LB05] Michael Leuschel and Michael Butler, Automatic refinement checking for B. In Kung-Kiu Lau and Richard Banach, editors, *ICFEM*, volume 3785 of *Lecture Notes in Computer Science*, pages 345–359, Springer, 2005.

[LB08] Michael Leuschel and Michael J. Butler, ProB: an automated analysis toolset for the B method. *Software Tools for Technology Transfer*, volume 10(2):pages 185–203, 2008.

[LBST06] Michael Leuschel, Michael Butler, Corinna Spermann and Edd Turner, Symmetry reduction for B by permutation flooding. In Julliand and Kouchnarenko [JK06], pages 79–93.

[LBST07] Michael Leuschel, Michael Butler, Corinna Spermann and Edd Turner, Symmetry reduction for B by permutation flooding. In Julliand and Kouchnarenko [JK06], pages 79–93.

[LCB09] Michael Leuschel, Dominique Cansell and Michael Butler, Validating and animating higher-order recursive functions in B. In Jean-Raymond Abrial and Uwe Glässer, editors, *Rigorous Methods for Software Construction and Analysis*, volume 5115 of *Lecture Notes in Computer Science*, pages 78–92, Springer, 2009.

[Led97] Yves Ledru, Specification and animation of a bank transfer using KIDS/VDM. *Automated Software Engineering*, volume 4(1):pages 33–51, 1997.

[Leu08] Michael Leuschel, The high road to formal validation. In Börger et al. [BBBB08], pages 4–23.

[LF08] Michael Leuschel and Marc Fontaine, Probing the depths of CSP-M: A new FDR-compliant validation tool. In Shaoying Liu, T. S. E. Maibaum and Keijiro Araki, editors, *ICFEM'2008*, volume 5256 of *Lecture Notes in Computer Science*, pages 278–297, Springer, 2008.

[LFFP09] Michael Leuschel, Jérôme Falampin, Fabian Fritz and Daniel Plagge, Automated property verification for large scale B models. In Ana Cavalcanti and Dennis Dams, editors, *FM*, volume 5850 of *Lecture Notes in Computer Science*, pages 708–723, Springer, 2009.

[LFFP11] Michael Leuschel, Jérôme Falampin, Fabian Fritz and Daniel Plagge, Automated property verification for large scale B models with ProB. *Formal Aspects of Computing*, volume 23(6):pages 683–709, 2011.

[LH06] Zhiming Liu and Jifeng He, editors, *Formal Methods and Software Engineering, 8th International Conference on Formal Engineering Methods, ICFEM 2006, Macao, China, November 1-3, 2006, Proceedings*, volume 4260 of *Lecture Notes in Computer Science*, Springer, 2006.

[LM00] Michael Leuschel and Thierry Massart, Infinite state model checking by abstract interpretation and program specialisation. In Annalisa Bossi, editor, *LOPSTR'99*, volume 1817 of *Lecture Notes in Computer Science*, Springer, 2000.

[LM07] Michael Leuschel and Thierry Massart, Efficient approximate verification of B via symmetry markers. In *Proceedings International Symmetry Conference*, pages 71–85, Edinburgh, UK, 2007.

[LMC01] Michael Leuschel, Thierry Massart and Andrew Currie, How to make FDR spin: LTL model checking of CSP by refinement. In José Nuno Oliveira and Pamela Zave, editors, *FME*, volume 2021 of *Lecture Notes in Computer Science*, pages 99–118, Springer, 2001.

[LP85] Orna Lichtenstein and Amir Pnueli, Checking that finite state concurrent programs satisfy their linear specification. In Mary S. Van Deusen, Zvi Galil and Brian K. Reid, editors, *POPL'85*, pages 97–107, ACM Press, 1985.

[LP07] Michael Leuschel and Daniel Plagge, Seven at one stroke: LTL model checking for high-level specifications in B, Z, CSP, and more. In Yamine Aït Ameur, Frédéric Boniol and Virginie Wiels, editors, *ISoLA*, volume RNTI-SM-1 of *Revue des Nouvelles Technologies de l'Information*, pages 73–84, Cépaduès-Éditions, 2007.

[LPU02] B. Legeard, F. Peureux and Mark Utting, Automated boundary testing from Z and B. In Eriksson and Lindsay [EL02], pages 21–40.

[LS95] François Laroussinie and Ph. Schnoebelen, A hierarchy of temporal logics with past. *Theoretical Computer Science*, volume 148(2):pages 303–324, 1995.

[LT05] Michael Leuschel and Edward Turner, Visualizing larger states spaces in ProB. In Treharne et al. [TKHS05], pages 6–23.

[MB02] Leonid Mikhailov and Michael J. Butler, An approach to combining B and Alloy. In Didier Bert, Jonathan P. Bowen, Martin C. Henson and Ken Robinson, editors, *ZB*, volume 2272 of *Lecture Notes in Computer Science*, pages 140–161, Springer, 2002.

[MGL10] Petra Malik, Lindsay Groves and Clare Lenihan, Translating Z to Alloy. In Frappier et al. [FGK$^+$10], pages 377–390.

[Mil82] Robin Milner, *A Calculus of Communicating Systems*. Springer, 1982.

[MMS08] Paulo J. Matos and João Marques-Silva, Model checking Event-B by encoding into Alloy. In Börger et al. [BBBB08], page 346.

[MS11] Dominique Méry and Neeraj Kumar Singh, Automatic code generation from Event-B models. In *Proceedings of the Second Symposium on Information and Communication Technology*, SoICT ’11, pages 179–188, ACM Press, 2011.

[MU05] Petra Malik and Mark Utting, CZT: A framework for Z tools. In Treharne et al. [TKHS05], pages 65–84.

[NL00] Ulf Nilsson and Johan Lübcke, Constraint logic programming for local and symbolic model-checking. In John W. Lloyd, Verónica Dahl, Ulrich Furbach, Manfred Kerber, Kung-Kiu Lau, Catuscia Palamidessi, Luís Moniz Pereira, Yehoshua Sagiv and Peter J. Stuckey, editors, *CL*, volume 1861 of *Lecture Notes in Computer Science*, pages 384–398, Springer, 2000.

[Oxf10] Oxford University Computing Laboratory, *Failures-Divergence Refinement — FDR2 User Manual*. 9th edition, 2010.

[Par00] Benoit Parreaux, *Vérification de systèmes d’événements B par model-checking PLTL*. Thèse de Doctorat, LIFC, Université de Franche-Comté, 2000.

[PL07] Daniel Plagge and Michael Leuschel, Validating Z specifications using the ProB animator and model checker. In Davies and Gibbons [DG07], pages 480–500.

[PL10] 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*, volume 12(1):pages 9–21, 2010.

[PL12] Daniel Plagge and Michael Leuschel, Validating B, Z and TLA$^+$ using ProB and Kodkod. In Dimitra Giannakopoulou and Dominique Méry, editors, *FM 2012*, volume 7436 of *Lecture Notes in Computer Science*, pages 372–386, Springer, 2012.

[PL14] Daniel Plagge and Michael Leuschel, A practical approach for validation with Rodin theories. In *Proceedings of the 5th Rodin User and Developer Workshop, June 2–3, 2014, Toulouse*, 2014.

[PLLR09] Daniel Plagge, Michael Leuschel, Ilya Lopatkin and Alexander Romanovsky, SAL, Kodkod, and BDDs for validation of B models. lessons and outlook. In *AFM 2009*, pages 16–22, 2009, available at `http://fm.csl.sri.com/AFM09/`.

[Pou03] Guilhem Pouzancre, How to diagnose a modern car with a formal B model? In Didier Bert, Jonathan P. Bowen, Steve King and Marina A. Waldén, editors, *ZB'2003*, volume 2651 of *Lecture Notes in Computer Science*, pages 98–100, Springer, 2003.

[PR00] Robert L. Pokorny and C. R. Ramakrishnan, Model checking linear temporal logic using tabled logic programming. *Proceedings Tabling in Parsing and Deduction TAPD 2000*, 2000.

[Ros99] A. W. Roscoe, *The Theory and Practice of Concurrency*. Prentice-Hall, 1999.

[Ros05] A. W. Roscoe, On the expressive power of CSP refinement. *Formal Aspects of Computing*, volume 17(2):pages 93–112, 2005.

[RRR$^+$97] Y. S. Ramakrishna, C. R. Ramakrishnan, I. V. Ramakrishnan, Scott A. Smolka, Terrance Swift and David Scott Warren, Efficient model checking using tabled resolution. In Orna Grumberg, editor, *CAV*, volume 1254 of *Lecture Notes in Computer Science*, pages 143–154, Springer, 1997.

[SAL] Symbolic Analysis Laboratory (SAL) website. `http://sal.csl.sri.com/`.

[Sch99] Steve Schneider, *Concurrent and Real-time Systems: The CSP Approach*. Wiley, 1999.

[Sch01] Steve Schneider, *The B-Method*. Cornerstones of Computing, Palgrave Macmillan, 2001.

[SCI15a] SCImago Journal Ranking: International Journal on Software Tools for Technology Transfer. `https://web.archive.org/web/20150130215227/http://www.scimagojr.com/journalsearch.php?q=19271&tip=sid`, 2015, the archived link was accessed January 2015.

[SCI15b] SCImago Journal Ranking: Science of Computer Programming. `https://web.archive.org/web/20150130214000/http://www.scimagojr.com/journalsearch.php?q=28416&tip=sid&clean=0`, 2015, the archived link was accessed January 2015.

[Ser07] Thierry Servat, BRAMA: A new graphic animation tool for B models. In Julliand and Kouchnarenko [JK06], pages 274–276.

[SGE00] A. Prasad Sistla, Viktor Gyuris and E. Allen Emerson, SMC: a symmetry-based model checker for verification of safety and liveness properties. *ACM Transactions on Software Engineering and Methodology*, volume 9(2):pages 133–166, 2000.

[SL08] Corinna Spermann and Michael Leuschel, ProB gets Nauty: Effective symmetry reduction for B and Z models. In *TASE*, pages 15–22, IEEE Computer Society Press, 2008.

[Spi92] J. M. Spivey, *The Z Notation: a Reference Manual*. Prentice-Hall, 1992.

[Spi00] J. M. Spivey, *The Fuzz Manual*. 2000, available at `http://spivey.oriel.ox.ac.uk/mike/fuzz`.

[SSW94] Konstantinos F. Sagonas, Terrance Swift and David Scott Warren, XSB as an efficient deductive database engine. In Richard T. Snodgrass and Marianne Winslett, editors, *SIGMOD*, pages 442–453, ACM Press, 1994.

[Ste94] Leon Sterling, *The Art of Prolog*. MIT Press, 1994.

[Ste09] France Steria, Aix-en-Provence, *Atelier B, User and Reference Manuals*. 2009, available at `http://www.atelierb.eu/`.

[Sys10] CSK Systems, VDMTools tool homepage. 2010, `http://www.vdmtools.jp/en/index.php`.

[Tar72] Robert Endre Tarjan, Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, volume 1(2):pages 146–160, 1972.

[Tar94] Dir. Quentin Tarantino, *Pulp Fiction*. 1994, movie, Miramax Films.

[TJ07] Emina Torlak and Daniel Jackson, Kodkod: A relational model finder. In Orna Grumberg and Michael Huth, editors, *TACAS'07*, volume 4424 of *Lecture Notes in Computer Science*, pages 632–647, Springer, 2007.

[TKHS05] Helen Treharne, Steve King, Martin C. Henson and Steve A. Schneider, editors, *ZB 2005: Formal Specification and Development in Z and B, 4th International Conference of B and Z Users, Guildford, UK, April 13-15, 2005, Proceedings*, volume 3455 of *Lecture Notes in Computer Science*, Springer, 2005.

[TLSB07] Edd Turner, Michael Leuschel, Corinna Spermann and Michael Butler, Symmetry reduced model checking for B. In *TASE*, pages 25–34, IEEE Computer Society Press, 2007.

[Tol11] Andriy Tolstoy, *Visualisierung von LTL-Gegenbeispielen*. Master's thesis, Heinrich-Heine-Universität Düsseldorf, Lehrstuhl für Softwaretechnik und Programmiersprachen, 2011.

[TS00] Helen Treharne and Steve Schneider, How to drive a B machine. In Jonathan P. Bowen, Steve Dunne, Andy Galloway and Steve King, editors, *ZB'2000*, volume 1878 of *Lecture Notes in Computer Science*, pages 188–208, Springer, 2000.

[Tur36] Alan M. Turing, On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, volume 2(42):pages 230–265, 1936.

[Utt00] Mark Utting, Data structures for Z testing tools. In *FM-TOOLS 2000 conference*, 2000, in TR 2000-07, Information Faculty, University of Ulm.

[Var01] Moshe Y. Vardi, Branching vs. linear time: Final showdown. In Tiziana Margaria and Wang Yi, editors, *TACAS'01*, volume 2031 of *Lecture Notes in Computer Science*, pages 1–22, Springer, 2001.

[WD96] Jim Woodcock and Jim Davies, *Using Z: Specification, Refinement, and Proof*. Prentice-Hall, 1996.

[WDK98] Michael Winikoff, Philip Dart and Ed. Kazmierczak, Rapid prototyping using formal specifications. In *Proceedings of the 21st Australasian Computer Science Conference*, pages 279–294, Springer, 1998.

[WGMM05] Bram De Wachter, Alexandre Genon, Thierry Massart and Cédric Meuter, The formal design of distributed controllers with $_{d}$sl and Spin. *Formal Aspects of Computing*, volume 17(2):pages 177–200, 2005.

[WKR$^{+}$09] 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 Manuel Núñez, Paul Baker and Mercedes G. Merayo, editors, *TestCom/FATES*, volume 5826 of *Lecture Notes in Computer Science*, pages 179–194, Springer, 2009.

[WL04] John Whaley and Monica S. Lam, Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In William Pugh and Craig Chambers, editors, *PLDI*, pages 131–144, ACM Press, 2004.

[WMD10] Johan Wittocx, Maarten Mariën and Marc Denecker, Grounding FO and FO(ID) with bounds. *Journal of Artificial Intelligence Research*, volume 38:pages 223–269, 2010.

[YML99] Yuan Yu, Panagiotis Manolios and Leslie Lamport, Model checking TLA$^{+}$ specifications. In Laurence Pierre and Thomas Kropf, editors, *Proceedings CHARME'99*, volume 1703 of *Lecture Notes in Computer Science*, pages 54–66, Springer, 1999.