DFG funded Project
Gepavas

(Gerichtete und Parallele Verifikation von abstrakten Spezifikationen)

Project Report

Work Conducted Until June 2010

Jens Bendisposto, Markus Borgermans, Michael Leuschel

Institut für Informatik, Universität Düsseldorf
Universitätsstr. 1, D-40225 Düsseldorf
{ `bendisposto, leuschel` } `@cs.uni-duesseldorf.de`

Table of Contents

1 Management Summary

1.1 Overview

Workpackages T1 (case studies), E1 (empirical evaluation) have been completed. A directed model checking algorithm has been implemented and empirically evaluated. Parts of workpackage T3 and to some extend also I3 have been started much earlier than planned, and have already generated a scientific paper. Indeed, two avenues of research have been identified as particularly promising: proof-directed model checking and flow-analysis guided model checking. This explains that more resources were put into pursuing these avenues. This shift of focus has been triggered by the further emergence of Event-B and the Rodin platform, where a model checker has direct access to proof obligations and the various provers.

Workpackage I1 (parallel prototype) is nearing completion. However, due to the rising availability of multi-core systems, we have concentrated our efforts on a shared memory approach. Other approaches will be investigated in the next phase of the project. The project is thus progressing very well.

1.2 Published papers and presentations

The following papers have been published within the project:

– Michael Leuschel, The High Road to Formal Validation, Proceedings ABZ 2008, LNCS 5238, p. 4–23.
– Jens Bendisposto, Michael Leuschel: Proof Assisted Model Checking for B, Proceedings ICFEM 2009, LNCS 5885 , p. 504–520.
– Jens Bendisposto and Michael Leuschel. Parallel Model Checking of Event-B Specifications with ProB. Preliminary Proceedings PDMC 2009, Eindhoven, November 2009.
– Mireille Samia, Harald Wiegard, Jens Bendisposto, Michael Leuschel : High-Level versus Low-Level Specifications: Comparing B with Promela and ProB with Spin, Proceedings TFM-B 2009, Nantes, June, 2009, APCB. ISBN:2951246102.

In addition, two papers have been submitted for publication "Automatic Flow Analysis for Event-B" and "Directed Model Checking for B: An Evaluation and New Techniques".

1.3 Outlook

The outcome of the empirical evaluation is that we believe that proof-directed and flow-analysis-directed model checking to be the most promising avenues for further work. We believe that a genetic library is less likely to yield good results.[1]

[1] In the project description we promised we would evaluate whether this approach is likely to be fruitful or not.

On the other hand, an avenue that we had discarded in the original description now seems more more likely to be fruitful: (dynamic) partial order reduction. This change is again triggered by the rise of Event-B, which with its simpler events, provides a much better basis for partial order reduction. Furthermore, Event-B has been designed for modelling reactive systems, and as such there is a much bigger need but also potential for partial order reduction.

Hence, in phase two of the project we plan to:

– pursue proof- and flow-analysis-directed model checking much more concretely, and
– develop and experiment with (dynamic) partial order reduction techniques for Event-B.

2 Workpackage T1: Case Studies

2.1 Overview of Case Studies

We have chosen a variety of case studies for evaluating the effectiveness of existing and new techniques for model checking, in particular directed and parallel model checking. All models are either classical B models or Rodin Event-B models. We have included several industrial specifications (some stemming from various EU projects, such as Rodin and Deploy[2]), as well as academic specifications of various intricate algorithms. There are a few artificial benchmarks as well, testing specific aspects of the model checking algorithm. We have also included some classical puzzles as well, in particular to test directed model checking.

The case studies have been partitioned into four classes:

1. Models with invariant violations,
2. Models with deadlocks,
3. Models with no errors (i.e., no deadlocks or invariant violations), but where a particular GOAL predicate is to be found. Indeed, in ProB the user can define a particular GOAL predicate and ask the model checker to find states which make the predicate true. The main difference with point 1 is that the goals are often much more precise (sometimes a concrete particular state) than the invariant violations.
4. Models with no errors, and where the full state space needs to be explored.

Below we briefly describe the individual models. They are used extensively in the empirical evaluation E1 (Section 3), as well as in the later tasks of the project. Note that new models will be added as the project progresses. We then describe in subsection 2.6 tool developments that we have made, so that the performance of a particular model checking technique can be automatically evaluated for all those models.

2.2 Models with Invariant Violations

– **Scheduler_err**
  The process scheduler from [34] for 5 processes, where an error has been added to the model.
– **Simpson_Four_Slot**
  A model of Simpson's four slot algorithm. This B model only represents the individual steps of the algorithm. It is intended to be used in conjunction with a CSP model to describe the sequencing of the steps. Here, the B model on its own is model checked (thus leading to invariant violations).
– **TravelAgency**
  A B model of a distributed online travel agency, through which users can make hotel and car rental bookings. It consists of 6 pages of B and was developed within the ABCD[3] project.

[2] EU funded FP7 research project 214158: DEPLOY (Industrial deployment of advanced system engineering methods for high productivity and dependability).

[3] "Automated validation of Business Critical systems using Component-based Design," EPSRC grant GR/M91013.

– **Peterson_err**
  Peterson is the specification of the mutual exclusion protocol for n processes as defined in [47]. Here we have used 4 processes and have introduced an error in the protocol.
– **SecureBuilding**
  The model of a secure building equipped with access control; see [**?**].
– **NastyVendingMachine**
  The model of a simple ticket vending machine. Note that we have compared the performance of ProB with Spin in [35] (also in Appendix ?).
– **Alstom_axl3**
  A train model by Alstom (confidential). This is not the final model but an intermediate one which still contains errors.
– **dfcheck_houseset**
  A simple model (derived from Schneider's houseset example [51]), where an operation can add new house numbers to a set. An invariant violation occurs when the set exceeds a certain limit.
– **BreadthFirstTest, DepthFirstTest , DepthFirstTest2** Three artificial models, to test certain performance aspects of depth-first and breadth-first search.
– **Abrial_Press_m2_err**
  A larger development of a mechanical by press by Abrial [2]. The development of the mechanical press started from a very abstract model and went through several refinements. The final model contained "about 20 sensors, 3 actuators, 5 clocks, 7 buttons, 3 operating devices, 5 operating modes, 7 emergency situations, etc." [2]. This model is a variation of the second refinement, where an error has been introduced.
– **SAP_M_Partner**
  An Event-B model of a business process generated by SAP from a MCM choreography model. This model describes the behaviour of an individual partner. See [56].
– **Needham-Schroeder**
  The Needham-Schroeder public key protocol is an authentication protocol for creating a secure connection over a public network [44]. The model consists of a network with the two normal users called Alice and Bob, an attacker named Eve and the keyserver. The first version of this protocol, developed in 1978, contains an error which was found in 1995 [41].
  This model is a slightly simplified version (reducing the messages sent by Eve), to allow the model checker to more quickly find a counter example. The shortest counter example has 14 steps.

2.3 Models with Deadlocks

– **Abrial_Earley_3_v3, Abrial_Earley_3_v5, Abrial_Earley_4_v3**
  A model developed by Jean-Raymond Abrial, with the help of Dominique Cansell. The purpose was to formally derive the Earley parsing algorithm in

Event-B and to establish its correctness. The model contains four levels of refinement and very complicated guards. Every event corresponds to a step in the parsing algorithm. The purpose was to animate the model for a particular grammar and to reproduce the sequence in `http://en.wikipedia.org/wiki/Earley_parser`. However, ProB did locate deadlocks (in fully proven models). For more details see [11].

- **Alstom_axl3_deadlock**
  The same model as above; but this time we only look for deadlocks.
- **Alstom_exemple7**
  Another model by Alstom (confidential). This is again not the final model but an intermediate one which still contains errors. One counter example trace has length 22.
- **Bosch_CrsCtrl**
  The second level of refinement of a Bosch model of an adaptive cruise control system, developed within Deploy.
- **SAP_MChoreography**
  An Event-B model of a business process generated by SAP from a MCM choreography model. This model describes the behaviour of an global system. See [56].
- **Dining**
  The classical Dining philosophers problem, with 8 philisophers.
- **CXCC0**
  CXCC (Cooperative Crosslayer Congestion Control) [50] is a cross-layer approach to prevent congestion in wireless networks. The key concept is that, for each end-to-end connection, an intermediate node may only forward a packet towards the destination after its successor along the route has forwarded the previous one. The information that the successor node has successfully retrieved a package is gained by active listening. The model is described in [10]. The invariants used in the model are rather complex.

2.4 Models with GOAL to be found

- **Wegenetz**
  The problem is to find a target state within a graph. The shortest solution needs 14 moves.
- **RussianPostalPuzzle**
  This is a B model of a cryptographic puzzle. (see, e.g., [23]). The shortest solution needs 10 moves.
- **TrainTorchPuzzle**
  The shortest solution needs 7 moves.
- **BlocksWorld**
  A model of blocksworld, with five blocks; the goal being to put all blocks in the right-order on top of each other. The shortest solution needs 6 moves.
- **Farmer**
  The Farmer/Fox/Goose/Grain puzzle. The shortest solution needs 9 moves.

- **Hanoi**
  The well-known towers of Hanoi puzzle. The shortest solution needs 33 moves.
- **Puzzle8**
  The well-known eight puzzle.[4] The goal is to arrive at a configuration where the eight tiles are in the correct order. The shortest solution needs 17 moves.
- **RushHour**
  The Rush Hour puzzle.[5] This is the hardest puzzle (number 40 in the regular version of the game). The shortest solution needs 83 moves.
- **Abrial_Press_m13**
  This is the last level of refinement of [2]; already described above. The goal is the guard of a partcular event (traiter_arret_moteur_2).
  The shortest solution has length 3.
- **Abrial_Queue_m1**
  Level 1 of a non-blocking concurrent Queue algorithm, derived by Abrial and Cansell in [4]. The goal is to find a particular configuration of the datastructures of the algorithm
  (`#pp.(pp:PROCESS & pp:dom(tld) & tld(pp)=hdd(pp) & tld(pp)=Tail))`).
  The shortest solution has length 4.
- **SystemOnChip_Router**
  A system-on-chip router developed by Satpathy. The shortest solution has length 4.

2.5 Models without Errors

- **Scheduler1**
  Another version of the process scheduler, for 5 processes. The first level of refinement from [37].
- **Volvo_Cruise**
  Volvo Vehicle Function. The B specification machine has 15 variables, 550 lines of B specification, and 26 operations. The invariant consists of 40 conjuncts. This B specification was developed by Volvo as part of the European Commission IST Project PUSSEE (IST-2000-30103).
- **USB4**
  USB is a specification of a USB protocol, developed by the French company ClearSy.
- **Nokia_Nota**
  A model developed by Nokia within the RODIN Project[6] for the validation and verification of Nokia's NoTA hardware platform; see [45].
- **HuffmanM** Event-B model of a Huffman encoder/decoder.
- **Cansell_Contention**
  A Firewire-Leader election protocol by Dominique Cansell, see also [48].

[4] See, e.g., `http://en.wikipedia.org/wiki/Fifteen_puzzle`.
[5] See `http://en.wikipedia.org/wiki/Rush_Hour_(board_game)`.
[6] `http://rodin.cs.ncl.ac.uk/`

- **DeMoney_GS_R1** Part of a model of an electronic purse by Trusted Logic, developed within the SecSafe project. See for example [12].
- **SystemOnChip_Router1**
  See above, but not searching for goal.
- **Mondex_m2, Mondex_m3**
  The mechanical verification of the Mondex Electronic Purse was proposed for the repository of the verification grand challenge in 2006. We use an Event-B model developed at the University of Southampton [17]. We have chosen two refinements from the model, m2 and m3. The refinement m2 is a rather big development step while the second refinement m3 was used to prove convergence of some events introduced in m2, in particular, m3 only contains gluing invariants.
- **Siemens_ATP0**
  This Siemens Mini Pilot was developed within the Deploy Project. It is a specification of a fault-tolerant automatic train protection system, that ensures that only one train is allowed on a part of a track at a time. The model contains a single refinement level and rather complex invariants.
- **SSF_obsw1**
  The Space Systems Finland example is a model of a subsystem used for the ESA BepiColombo mission. The BepiColombo spacecraft will start in 2013 on its journey to Mercury. The model is a specification of parts of the BepiColombo On-Board software, that contains a core software and two subsystems used for tele command and telemetry of the scientific experiments, the Solar Intensity X-ray and particle Spectrometer (SIXS) and the Mercury Imaging X-ray Spectrometer (MIXS). The model was a mini pilot of the Deploy project.
- **ETH_elevator12**
  This is the twelfth revienement of an elevator model by ETH Zürich.
- **Echo**
  The Echo algorithm [19] is designed to find the shortest paths in a network topology. A start node sends an explore-message to all neighbors. Each node is marked with red, when it receives an explore-message for the first time. Moreover, it memorizes the nodes, from which it received the message, as a shortest path to the initialization node. It also sends, in turn, explore-messages to its other neighbors. Whenever the node receives either an explore-message or an echo-message from all its neighbors, to which it sent one of such messages, the node will be marked green and sends an echo-message to the nodes, from which it had first received an explore-message. When all nodes are marked green, the cycle is finished.
  In the B model, every type of message type has one corresponding operation. The operations are active, as soon as a node sends the appropriate message. The execution of the operation reflects the receipt of the message by the node's recipient. By the non-deterministic order, the selection of the active operations is assured that any various long message runtime in the channel of the model is taken into account. It is possible that many messages are simultaneously on the channel. The edges are depicted as functions between

the nodes. With the proper invariants, it is easy to verify if a protocol ensures that all nodes are marked green, when no message is in the channel. Furthermore, all shortest paths must be known as soon as all nodes are marked green.

2.6 Tool developments

The ProB command-line version has been extended so that parameters and preferences can be set via the command-line. Also, a logging facility has been developed, which allows to write the results of experiments into a log file. This log file contains a series of Prolog facts, and can be analysed by ProB itself, which can either generate gnuplot graphs or Excel (csv) spreadsheets.

Using make, we have written a script which runs ProB for a given configuration on all the benchmarks above, and stores the result in a log file. This was used to generate the outputs and graphs in the following section.

A sample entry in the log file is as follows:

```
start_logging(1271961480507,'log/heuristic.log').
version(1271961480507,1,3,2,beta10,'5115:5120M',
 '$LastChangedDate: 2010-03-30 17:57:13 +0200 (Tue, 30 Mar 2010) $').
options(1271961480507,[mc(1000),comment(dfbf75),log('log/heuristic.log'),timeout(180000)],
 ['examples/EventBPrologPackages/ProofDirected/benchmarks/siemens_mch_0.eventb']).
date(1271961480507,datime(2010,4,22,20,38,0)).
loading(1271961480507,'examples/EventBPrologPackages/ProofDirected/benchmarks/siemens_mch_0.eventb').
start_animation(1271961480507).
starting_model_check(1271961480507,1000).
model_check(1271961480507,1000,2290,no).
prob_finished(1271961480507).
```

3 Workpackage E1: Empirical Evaluation

3.1 Initial Motivation: Model Checking High-level versus Low-level Specifications

Most model checking tools work on relatively low-level formalisms. E.g., the model checker SMV [43, 13] works on a description language well suited for specifying hardware systems. The model checker Spin [28, 30, 9] accepts the Promela specification language, whose syntax and datatypes have been influence by the programming language C. Recently, however, there have also been model checkers which work on higher-level formalisms, such as ProB [36, 38] which accepts B [1]. Other tools working on high-level formalisms are, for example, FDR [25] for CSP and ALLOY [33] for a formalism of the same name (although they both are strictly speaking not model checkers).

It is relatively clear that a higher level specification formalism enables a more convenient modelling. On the other hand, conventional wisdom would dictate that a lower-level formalism will lead to more efficient model checking. However, our own experience has been different. During previous teaching and research activities, we have accumulated anecdotal evidence that using a high-level formalism such as B can be much more productive than using a low-level formalism such as Promela. Furthermore, quite surprisingly, it turned out that the use of a high-level model checker such as ProB was much more effective in practice than using a very efficient model checker such as Spin on the corresponding low-level model.

3.2 A small empirical study

We first tried to put this anecdotal evidence on a more firm empirical footing, by systematically comparing the development and validation time of B models with that of the corresponding Promela models. [57, 49] studies the elaboration of B-models for ProB and Promela models for Spin on ten different problems. With one exception (the Needham-Schroeder public key protocol), all B-models are markedly more compact than the corresponding Promela models. On average, the Promela models were 1.85 longer (counting the number of symbols). The time required to develop the Promela models was about 2-3 times higher than for the B models, and up to 18 times higher in extreme cases. No model took less time in Promela. Some models could not be fully completed in Promela. The study also found that in practice both model checkers ProB and Spin were comparable in model checking performance, despite ProB working on a much higher-level input language and being much slower when looking purely at the number of states that can be stored and processed.

Other independent experimental evaluations also report good performance of ProB compared against SMV.

3.3 Looking for Answers

Within this project we first tried to analyse and understand the counter-intuitive behaviour described above in subsection 3.2. The results have been published in [35].

Granularity One tricky issue is the much finer granularity of low-level models. If one is not careful, the number of reachable states can explode expponentially, compared to a corresponding high-level model.

In summary, translating high-level models into Promela is often far from trivial. Additional intermediate states and additional state variables are sometimes unavoidable. When writing Promela models, for example, great care has to be taken to make use of `atomic` (or even `dstep`) primitives and resetting dead temporary variables to default values. However, restrictions of `atomic` make it sometimes very difficult or impossible to hide all of the intermediate states. More details can be found in [35].

Searching for Errors in Large State Spaces Let us disregard the granularity issue and let us look at simple problems, with simple datatypes, which can be easily translated from B to Promela, so that we have a one-to-one correspondence of the states of the models. In such a setting, it is obvious to assume that the Spin model checker for Promela will outperform the B model checker by several orders of magnitude. Indeed, Spin generates a specialised model checker in C which is then compiled, whereas ProB uses an interpreter written in Prolog. Furthermore, Spin has accrued many optimisations over the years, such as partial order reduction [31, 46] and bitstate hashing [29].

However, it is our experience that even in this setting, this potential speed advantage of Spin often does not necessarily translate into better performance in practice in real-life scenarios. Indeed—contrary to what may be expected—we show in this section that Spin sometimes fares quite badly when used as a debugging tool, rather than as verification tool. Especially for software systems, verification of infinite state systems cannot be done by model checking (without abstraction). Here, model checking is most useful as a debugging tool: trying to find errors in a very large state space.

One experiment reported on in [35] is the NastyVendingMachine (see Section 2). It has a very large state space, where there is a systematic error in one of the operations of the model (as well as a deadlock when all tickets have been withdrawn). To detect the error, it is important to enable this operation and then exercise this operation repeatedly. It is not important to generate long traces of the system, but it is important to systematically execute combinations of the individual operations. This explains why depth-first behaves so badly on this model, as it will always try to exercise the first operation of the model first (i.e., inserting the 10 cents coin). Note that a very large state space is a typical situation in software verification (sometimes the state space is even infinite).

In a corrected non-deadlocking model of the vending machine, the state space is again very large, but here the error occurs if the system runs long enough;

it is not very critical in which order operations are performed, as long as the system is running long enough. This explains why for this model breadth-first was performing badly, as it was not generating traces of the system which were long enough to detect the error.

In order to detect both types of errors with a single model checking algorithm, ProB has been using a mixed depth-first and breadth-first search [38]. More precisely, at every step of the model checking, ProB randomly chooses between a depth-first and a breadth-first step. This behaviour is illustrated in Fig. 1, where two different possible runs of ProB are shown after exploring 5 nodes of the B model from [35].

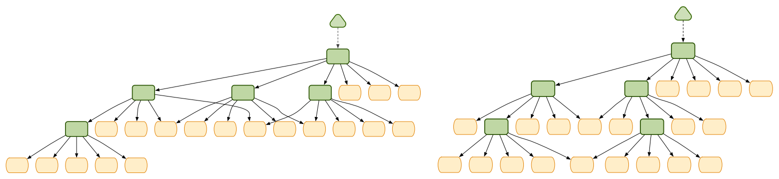

Fig. 1. Two different explorations of ProB after visiting 5 nodes of the NastyVendingmachine

The motivation behind ProB's heuristic is that many errors in software models fall into one of the following two categories:

– Some errors are due to an error in a particular operation of the system; hence it makes sense to perform some breadth-first exploration to exercise all the available functionality. In the early development stages of a model, this kind of error is very common.
– Some errors happen when the system runs for a long time; here it is often not so important which path is chosen, as long as the system is running long enough. An example of such an error is when a system fails to recover resources which are no longer used, hence leading to a deadlock in the long run.

In summary, if the state space is very large, Spin's depth-first search can perform very badly as it fails to systematically test combinations of the various operations of the system. Even partial order reduction and bitstate hashing do not help. Similarly, breadth-first can perform badly, failing to locate errors that require the system to run for very long. We have argued that ProB's combined depth-first breadth-first search with a random component does not have these pitfalls.

3.4 Depth-First versus Breadth-First: A thorough empirical Evaluation

We report on a first extensive empirical evaluation of directed model checking approaches for B and Event-B. The experiments for parallel model checking using the first prototype are still being undertaken, and will be reported on later.

First task was to compare depth-first versus breadth-first, as well as the default mixed depth-first/breadth-first approach of ProB.

The results are summarised in Tables 1–4. Relative times are computed with ProB in default settings (which up to know was a mixed depth-first/breadth-first strategy with one-third probability of doing depth-first; more on that below). The experiments were run on a MacBook Pro with a 3.06 GHz Core2 Duo processor, and ProB 1.3.2 compiled with SICStus Prolog 4.1.2.

Pure Depth-First In a considerable number of cases pure depth-first is the fastest method, e.g., for the Peterson_err, Abrial_Earely3_v5, Alstom_axl3, and BlocksWorld benchmarks.

However, we can see in Figure 1 that for some models Depth-First fares very badly:

- In Alstom_ex7 in Figure 2, pure depth-first search even fails to find the deadlock when given an hour of cputime. This real-life example thus supports our claim from [35] and subsection 3.3 that when state space is too large to examine fully, depth-first will sometimes not find a counter example. This is actually a quite common case for industrial models: they are typically (at least before abstraction) too large to handle fully.
- Another similar example is Abrial_Press_m13 in Figure 3, where pure depth-first is about 900 times slower than ProB in the reference settings.
- Another bad example is Puzzle8, where depth-first is more than 7 times slower or Simpson4Slot where it is 163 times slower than ProB in the reference settings. Finally, for the artificially constructed BFTest, depth-first search fails to find the invariant violation.

For finding goals, the geometric mean was 0.92, i.e., slightly better than the reference setting. Overall, pure depth-first seemed to fare best for the deadlocking models with a geometric mean of 0.43. For finding invariant violations, however, the geometric mean was 1.03, i.e., slightly worse than the reference setting.

The bad performance in the Huffman benchmark is actually not relevant: here not all nodes were evaluated. As such, the time to examine 10,000 nodes was measured. The pure-depth first search here encountered more complicated states, than the other approaches, explaining the additional time required for model checking.

In conclusion, the performance of pure depth-first alone can vary quite dramatically, from very good to very bad. A such, pure depth first search is not a good choice as a default setting of ProB. Note, however, that we allow the user to override the default setting and put ProB into pure depth-first mode.

Pure Breadth-First In most cases pure Breadth-First is worse than the reference setting; in some cases considerably so. The geometric mean was always above 1, i.e., worse than the reference setting.

For Alstom_ex7 pure breadth-first also fails to find the deadlock.

Peterson_err in Figure 1 gives a similar picture, Breadth-First being 134 times slower than DF and 11 times slower than the reference settings of ProB. Other examples where BF is not so good: Abrial_Earley3_v5, DiningPhil, SystemOnChip_Router, Wegenetz.

There are some more examples where it performs considerably better than pure depth-first: Puzzle8, Simpson4Slot, Abrial_Press_m13 and the "artificial" benchmarks BFTest and DFTest2.

In conclusion, breadth-first on its own is not appropriate, except in special circumstances. Note, a user can set ProB into breadth-first mode, but the default is another setting (see below).

Mixed Mixed Depth-first/Breadth-first The motivation behind ProB's mixed depth-first/breadth-first heuristic is that many errors in software models fall into one of the following two categories:

- Some errors are due to an error in a particular operation of the system; hence it makes sense to perform some breadth-first exploration to exercise all the available functionality. In the early development stages of a model, this kind of error is very common.
- Some errors happen when the system runs for a long time; here it is often not so important which path is chosen, as long as the system is running long enough. An example of such an error is when a system fails to recover resources which are no longer used, hence leading to a deadlock in the long run.

An interesting real-life benchmark is Alstom_ex7: here both pure depth-first and pure breadth-first fail to find the deadlock. However, the mixed strategy finds the deadlock.

We have experimented with four different versions of the mixed strategy: DF75, DF50, DF33, DF25. The reference setting was DF33, where there is a 33 % chance of going depth-first at each step. Best overall geometric mean is obtained when using DF50 (which is now the new default setting of ProB).

In summary, let us look at the radar plots in Figure 2, where we summarise the results for pure depth-first, pure breadth-first, the old reference setting and the new one. We can clearly see the quite erratic performance of pure depth-first (relative to the reference setting), and the less erratic but usually worse performance of pure breadth-first. We can also see that the new reference setting usually lies within the reference setting circle, smoothing out most of the (bad) erratic behaviour of the pure depth-first approach.

The Houseset benchmark clearly shows that mixed DF/BF has problem going deep if there is a large branching factor. This may indicate a possible way to improve our current algorithm, by favouring at least some very deep paths.

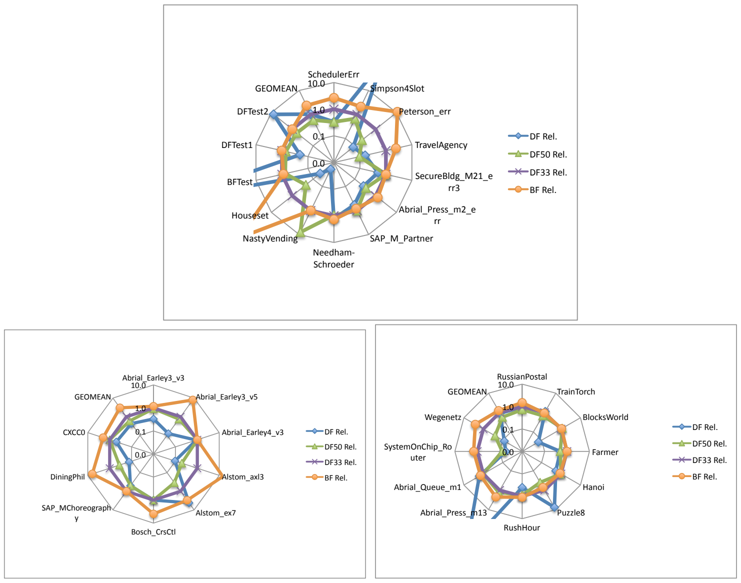

Fig. 2. Radar plot for invariant, deadlock and goal checking (DF/BF Analysis)

3.5 Evaluation the potential of using Heuristics

Directed Model Checking uses additional information about the model or the destination state in form of a heuristic that guides the model checker towards a target state. This additional information can be collected using for instance static analysis or it can be given by the modeler.

Currently the state space of ProB is stored as a Prolog fact database. Every state can be quickly accessed using its ID or using the hash-value of its state; see Figure 3 The model checker also maintains a pending list of open nodes, using two predicates: retracting a fact from one of the predicates yields the least recently added open node (for depth-first traversal) and rectracting from the other predicate yields the oldest open node (for breadth-frist traversal). This approach allowed us to implement a mixed depth-first / breadth-first approach by randomly selecting either an element from the front or the end of the pending list of open nodes.

We have implemented a priority queue in C++ using the STL (Standard Template Library) `multimap` data structure. One can thus efficiently add new open nodes with a particular weight, and then either chose the node with the lowest or highest weight.

We evaluated some strategies to assign weights to newly encountered nodes. In particular a random search, a search based on the number of successor states, a search based on the (term)size of the state as well as some custom heuristic functions written by the modeler for a particular model. The latter approach is used for models where a specific goal was known, e.g., puzzles.

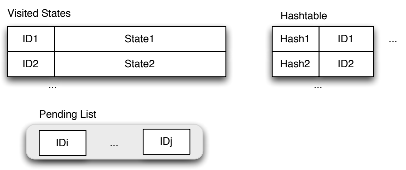

Fig. 3. Some Datastructures of the ProB Model Checker

We now describe the various heuristic functions we have investigated, as well as the result of the empirical investigation.

Random Hash The idea is simply to use the hash value of a state as the weight for the priority queue. The hope is that the hash value distributes uniformly, i.e., that this would provide a good way to randomize the treatment of pending states. The hash value is computed anyway for every state, using SICStus Prolog `term_hash` predicate.

The purpose was to use this heuristic as a base-line: a heuristic that is worse or not markedly better than this one is not worth the effort. We also want to compare this heuristic with the mixed depth-first/breadth-first approach from Section 3.4 and see whether there any notable differences. Indeed, the mixed depth-first/breadth-first search does not really permute the order of nodes in the list, and this could have an influence on the model checking performance.

Results For finding deadlocks (Figure 6) and goals (Figure 7) it is markedly better than the reference settings of ProB (except for the Bosch cruise control model; but runtimes there are very low anyway). For finding invariant violations, however, (Figure 5) it is worse (its geometric mean is greater than 1 (1.07) and in two examples it is markedly worse).

Overall, it seems to perform slightly better than our mixed DF/BF search.

We have also experimented with truly random approach, where we use a random number generator rather than the hash value for the priority. The results are rather similar, except for Alstom_ex7 where it systematically outperforms Random Hash.

Out Degree The idea is to use the out degree of a state as priority, i.e., the number of outgoing transitions. The motivation is that if we have found a state with an an out degree of 0, i.e., the highest priority, we have found a deadlock. Intuitively, the less transitions are enabled for a state, the closer we may be to a deadlock. In the implementation we actually do not know the out degree of a node until it has been processed. Hence, we use the out degree of the (first) predecessor node for the priority.

Results Indeed, for finding deadlocks this heuristic obtained the best geometric mean of 0.5. So, this simple heuristic works surprisingly well. For finding goals, this heuristic still obtains geometric mean of 0.63, but it is worse than the random hash function. For finding invariant violations it does not work at all; its geometric mean is 1.56.

A further refinement of this heuristic is to combine the out degree with the random hash heuristic, i.e., if two nodes have the same out degree (which can happen quite often) we use the hash value as heuristic to avoid a degeneration into depth-first search. This refinement leads to a further performance improvement for deadlock finding (geometric mean of 0.34 compared to 0.50), and for goal finding. But it is markedly worse for invariant violation finding.

In conclusion, the out degree heuristic, especially when combined with random hash, works surprisingly well for its intended purpose of finding deadlocks. In future work, we plan to further refine this approach, by using a static flow analysis to guide model checker into deadlocks and/or particular enablings for events.

Term Size The idea of this heuristic is to use the term size of the state (i.e., the number of constant and function symbols appearing in its representation) as priority. The motivation for this heuristic is that the larger the state is, the more complicated it will be to process (for checking invariants and computing outgoing transitions). Hence, the idea is to process simpler states first, in an attempt to maximise the number of nodes processed per time unit.

Results For finding goals this heuristic has a geometric mean of 0.85, i.e., it is better than the reference setting of ProB, but worse than random hash. For deadlock and invariant checking, it also performs worse than random hash. In summary, this heuristic does not seem worth pursuing further.

Effectiveness of custom heuristic function: In order to experiment easily with other heuristic functions, we have added the possibility for the user to define a custom heuristic function for a B model. Basically, this function can be introduced in the DEFINITIONS part of a B machine by defining `HEURISTIC_FUNCTION`. ProB now evaluates the expression `HEURISTIC_FUNCTION` in every state, and uses its value as the priority of the state. Note, the expression must return an integer value. For the BlocksWorld benchmark, we have written the following custom heuristic function:

```
ongoal == {a|->b, b|->c, c|->d, d|->e};
DIFF(A,TARGET) == (card(A-TARGET) - card(TARGET /\ A));
HEURISTIC_FUNCTION == DIFF(on,ongoal);
```

Note the machine as a variable `on` is of type `Objects +-> Objects` and the GOAL for the model checker is to find a state where `on = ongoal` is true.

In the benchmarks, we have mainly written heuristic functions which estimate the distance between a target goal state and the current state. In future, we plan to derive the definition of those heuristic functions automatically. A simple distance heuristic, can be derived if the goal of the model checking is to find specific values for certain variables of the machine (such as `on = ongoal`). Basically, for current state $s = \langle s_1, \ldots, s_n \rangle$ and a target state $t = \langle t_1, \ldots, t_n \rangle$ we use as heuristic $h(s) = \Sigma_{1 \leq i \leq n} \Delta(s_i, t_i)$ where

- $\Delta(x, target) = abs(x - target)$ if x integer
- $\Delta(x, target) = card(x - TARGET) - card(TARGET \cap A)$ if x a set
- $\Delta((x, y), (t1, t2)) = \Delta(x, t1) + \Delta(y, t2)$ for pairs,
- in all other cases: $\Delta(x, target) = 0$ if $x = target$ and 1 otherwise

If the value of a particular variable is not relevant, then we simply set $\Delta(s_i, t_i) = 0$ for that variable.

This defines a kind of Hamming distance for B states. We have applied this (manually) in the BlocksWorld example above.

We have only evaluated this approach for finding goals. Here, it obtained the best overall geometric mean of 0.34. For Puzzle8 and Abrial_press_m13, this approach yielded by far the best solution. For RussianPostal, TrainTorch, Blocksworld, Abrial_Queue_m1 it obtains the best result. There was one experiment were it is markedly worse than ProB in the reference settings: SystemOnChip_Router. Here the heuristic did not pay off at all. Indeed, here the last event changes all of the four variables, relevant for the model checking GOAL, in one step. This only confirms the fact that we are working with *heuristic* functions, which are not guaranteed to always improve the performance.

Summary of heuristic function experiments: We have summarised the main findings of our experiments in Figure 4. We can conclude that:

- for invariant checking, the random hash heuristic fared best.[7] This seems to indicate that it is maybe useful to combine some more random component into the depth-first/breadth-first techniques of Section 3.4, e.g., to also randomly permute the operation order. Indeed, the approaches from Section 3.4 always process the operations in the same order, and does not shuffle the states inside the pending list.
- for deadlock checking, the out-degree-hash heuristic is the best. It should provide a good basis for further improved deadlock checking techniques.
- for goal finding, a custom heuristic function provides (except in one case) by far the best result. The next step is to derive those heuristic functions automatically.

[7] However, note that DF50 had an overall geometric mean of 0.58, and was hence better overall than random hash.

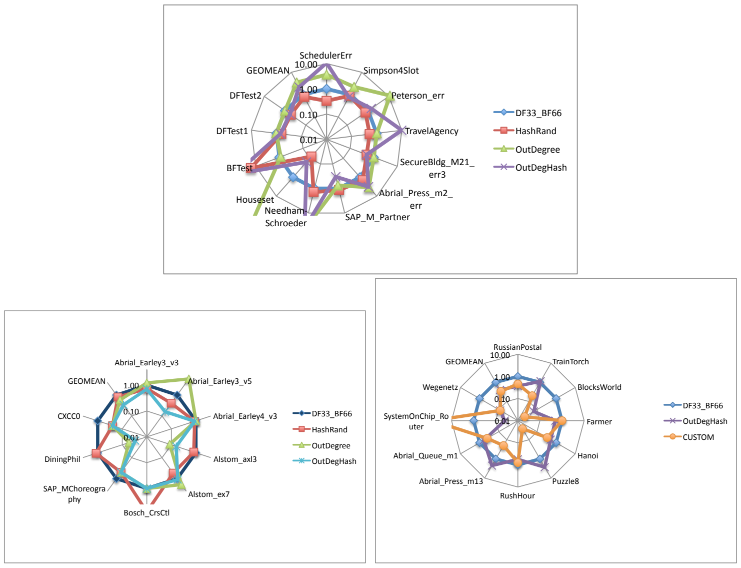

Fig. 4. Radar plot for invariant, deadlock and goal checking (Heuristic Analysis)

The out-degree-hash heuristic also provides reasonably good performance (its geometric mean is 0.41, which is better than the best mixed-depth-first one of 0.57 for DF75).

3.6 Overall Conclusions of all Experiments

- The mixed depth-breadth-first strategy is a good idea, it is much more robust than either depth-first and breadth-first. Still, it may be useful to improve the approach by randomising the order of transitions and by favouring some very deep paths.
- The use of heuristics can pay off considerably. The out-degree heuristic was successful for deadlock checking, while measuring a distance to a target state was generally very successful for goal finding tasks.
  Still, more refined heuristics are required to fully exploit the potential. It is probably a good idea to use control flow graph information to guide the model checker more precisely.

3.7 Experimental Results (Figures and Tables)

Note: we use geometric mean [24], as the arithmetic mean is useless for normalised results. Of course, the geometric mean itself should also be taken with a grain of salt (various articles also attack its usefulness). Indeed, without knowing how representative the chosen benchmarks are for the overall population of

B specifications, we can conclude little. In Section 2 we have tried to assemble a variety of benchmarks from our own experience; but this sample may be inadequate for other application scenarios.

Thus, we also provide all figures in the tables below, so that minimum, maximum relative runtimes can be seen, as well as the absolute runtime of the reference benchmark. Indeed, the relative runtimes are less reliable, when the absolute runtime of the reference benchmark is already very low.

Invariant Benchmark	DF	DF75	DF50	DF33 (abs+rel)		DF25 Rel	BF
SchedulerErr	0.33	0.33	0.33	30 ms	1.00	1.00	2.67
Simpson4Slot	163.33	0.22	0.67	90 ms	1.00	0.78	2.11
Peterson_err	0.08	0.08	0.22	360 ms	1.00	3.06	11.17
TravelAgency	0.16	0.27	0.10	630 ms	1.00	0.78	2.43
SecureBldg_M21_err3	0.50	0.50	1.00	20 ms	1.00	1.00	1.00
Abrial_Press_m2_err	0.26	3.38	0.34	880 ms	1.00	1.81	1.26
SAP_M_Partner	0.58	1.08	1.00	120 ms	1.00	0.92	0.83
NastyVending	0.02	0.08	8.00	130 ms	1.00	2.85	1.00
NeedhamSchroeder	1.28	1.52	0.95	22620 ms	1.00	0.70	1.37
Houseset	0.05	0.06	0.22	2610 ms	1.00	2.66	** 336.27
BFTest	** 15000.00	11592.75	0.75	80 ms	1.00	1.00	0.88
DFTest1	0.20	0.35	0.71	2360 ms	1.00	1.01	1.02
DFTest2	7.86	0.50	0.60	2930 ms	1.00	0.99	1.00
GEOMEAN	1.03	0.78	0.58	394 ms	1.00	1.24	2.35

DF: out of memory

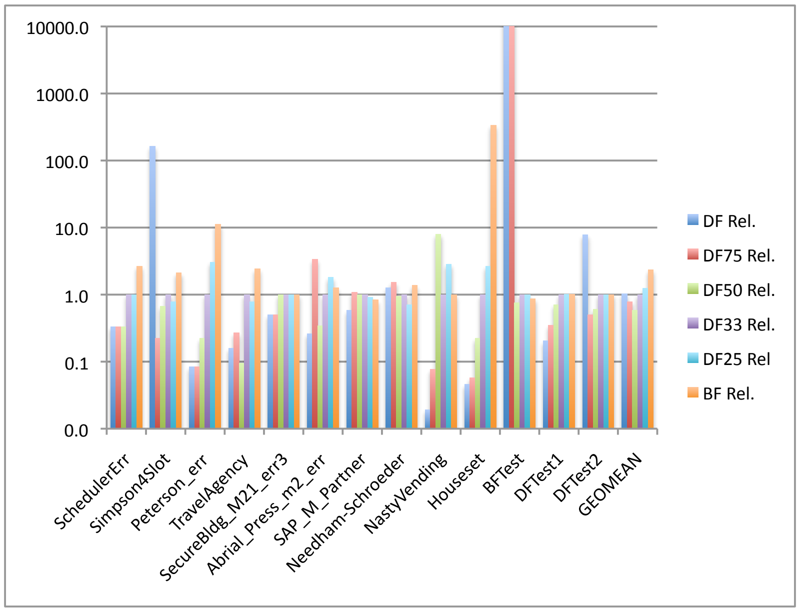

Table 1. Relative times for checking models with invariant violations (DF/BF Analysis)

Deadlock Benchmark	DF	DF75	DF50	DF33 (abs+rel)		DF25 Rel	BF
Abrial_Earley3_v3	0.33	0.44	0.93	270 ms	1.00	1.19	1.19
Abrial_Earley3_v5	0.13	0.32	0.75	4320 ms	1.00	2.18	7.95
Abrial_Earley4_v3	0.89	0.89	1.00	90 ms	1.00	1.00	1.00
Alstom_axl3	0.10	0.17	0.20	51270 ms	1.00	3.51	14.61
Alstom_ex7	** 4.20	0.23	0.35	856320 ms	1.00	** 1.21	** 3.00
Bosch_CrsCtl	1.00	1.00	1.00	3 ms	1.00	4.00	4.00
SAP_MChoreography	0.50	0.50	0.50	20 ms	1.00	1.00	1.00
DiningPhil	0.13	0.26	0.36	1690 ms	1.00	2.49	6.20
CXCC0	0.50	1.00	1.00	10 ms	1.00	2.00	2.00
GEOMEAN	0.43	0.44	0.59	540 ms	1.00	1.82	3.02
AVG	0.00	0.00	0.00	0 ms	0.00	0.00	0.00

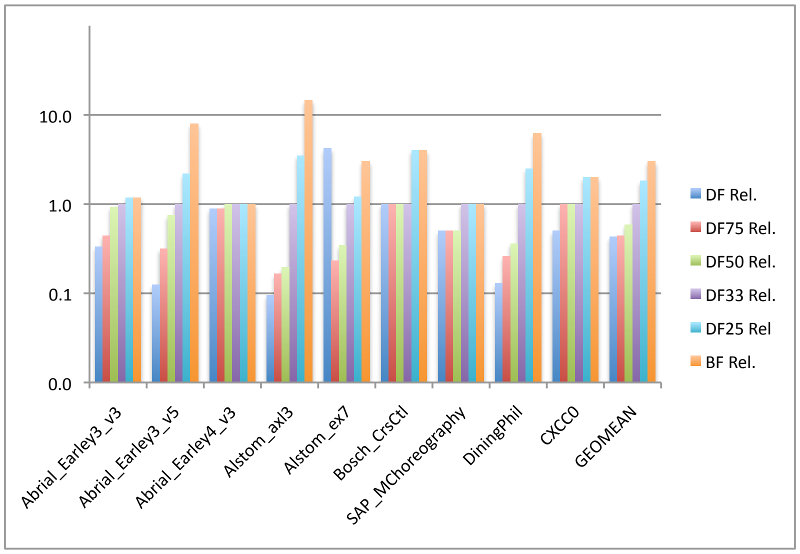

Table 2. Relative times for checking models with deadlocks (DF/BF Analysis)

Goal Benchmark	DF	DF75	DF50	DF33 (abs+rel)		DF25 Rel	BF
RussianPostal	0.95	0.14	0.73	220 ms	1.00	1.05	1.50
TrainTorch	1.14	1.17	0.69	350 ms	1.00	1.06	0.94
BlocksWorld	0.07	0.25	1.16	440 ms	1.00	1.18	1.07
Farmer	0.50	1.00	0.50	20 ms	1.00	1.00	1.00
Hanoi	0.54	0.48	1.00	500 ms	1.00	0.84	0.90
Puzzle8	**7.56**	2.86	0.40	59060 ms	1.00	0.11	0.71
RushHour	0.41	0.66	0.90	127020 ms	1.00	1.09	1.11
Abrial_Press_m13	**899.78**	1.64	1.26	800 ms	1.00	0.66	2.23
Abrial_Queue_m1	1.60	3.00	1.00	50 ms	1.00	1.60	1.40
SystemOnChip_Router	0.05	0.14	0.08	2050 ms	1.00	1.04	1.45
Wegenetz	0.08	0.08	0.25	120 ms	1.00	0.17	2.58
GEOMEAN	0.92	0.57	0.58	715 ms	1.00	0.71	1.26

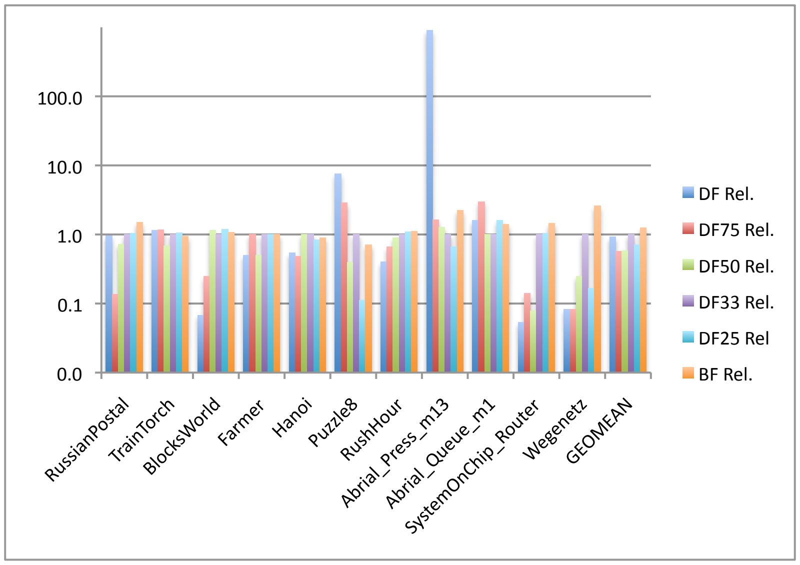

Table 3. Relative times for checking models with goals to be found (DF/BF Analysis)

No Error Benchmark	DF	DF75	DF50	DF33 (abs+rel)	DF25 Rel	BF
Scheduler1	1.00	1.00	1.00	6010 ms 1.00	1.00	1.00
Volvo_Cruise	1.01	1.01	1.01	5360 ms 1.00	1.00	1.00
USB4	1.00	1.00	1.00	3270 ms 1.00	1.00	1.00
Nokia_Nota	1.09	1.03	1.01	44220 ms 1.00	1.00	1.00
Huffman	**17.98**	1.15	1.05	11100 ms 1.00	1.00	0.98
Cansell_Contention	1.00	1.01	1.01	1490 ms 1.00	1.01	1.01
Demoney_GS_R1	1.00	1.01	1.00	1580 ms 1.00	1.00	1.00
SystemOnChip_Router	0.99	1.00	1.00	27020 ms 1.00	1.00	1.01
Mondex_m2	1.10	0.95	0.96	1700 ms 1.00	1.02	1.14
Mondex_m3	1.09	0.95	0.96	1910 ms 1.00	1.01	1.12
Siemens_ATP0	1.12	1.10	1.00	2090 ms 1.00	0.98	0.93
SSF_obsw1	1.10	1.05	1.04	22500 ms 1.00	1.00	1.00
Echo	0.98	0.98	1.00	440 ms 1.00	1.00	1.00
ETH_Elevator12	0.83	0.86	0.93	106520 ms 1.00	1.01	0.90
GEOMEAN	1.25	1.00	1.00	6687 ms 1.00	1.00	1.01

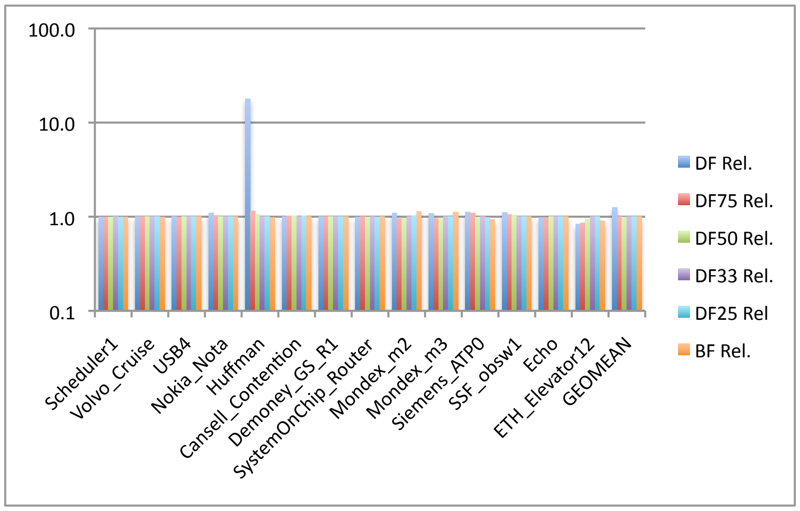

Table 4. Relative times for checking models without errors (DF/BF Analysis)

Invariant Benchmarks	DF33 (abs+rel)		HashRand	OutDegree	OutDegHash	TermSize
SchedulerErr	30 ms	1.00	0.33	3.67	10.67	2.67
Simpson4Slot	90 ms	1.00	0.89	2.22	0.78	2.22
Peterson_err	360 ms	1.00	0.75	11.25	1.42	9.14
TravelAgency	630 ms	1.00	0.52	1.02	9.98	0.49
SecureBldg_M21_err3	20 ms	1.00	0.50	1.00	0.50	0.50
Abrial_Press_m2_err	880 ms	1.00	1.45	3.38	3.19	1.25
SAP_M_Partner	120 ms	1.00	1.17	0.75	0.33	1.00
NeedhamSchroeder	22620 ms	1.00	1.40	**48.51	41.58	** 46.06
Houseset	2610 ms	1.00	0.08	** 333.53	0.16	** 338.61
BFTest	80 ms	1.00	16.00	0.88	73.00	0.88
DFTest1	2360 ms	1.00	0.64	1.02	0.69	1.02
DFTest2	2930 ms	1.00	0.53	1.00	0.57	1.66
GEOMEAN	432 ms	1.00	0.79	2.44	1.54	1.93

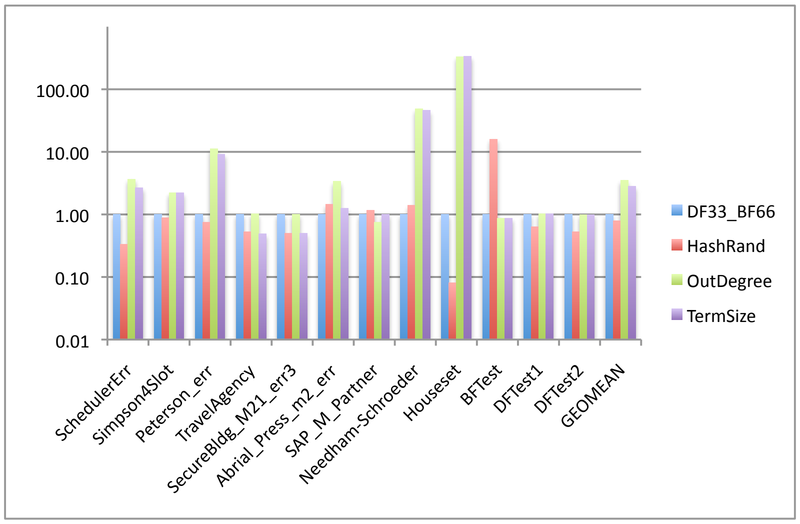

Table 5. Relative times for checking models with invariant violations (Heuristics Analysis)

Deadlock Benchmarks	DF33 (abs+rel)		HashRand	OutDegree	OutDegHash	TermSize
Abrial_Earley3_v3	270 ms	1.00	0.74	1.22	0.70	1.22
Abrial_Earley3_v5	4320 ms	1.00	0.41	5.88	0.16	7.95
Abrial_Earley4_v3	90 ms	1.00	0.89	1.00	0.89	1.00
Alstom_axl3	51270 ms	1.00	0.82	0.09	0.16	0.08
Alstom_ex7	856320 ms	1.00	0.55	** 1.91	1.10	**1.57
Bosch_CrsCtl	3 ms	1.00	8.00	1.00	1.00	4.00
SAP_MChoreography	20 ms	1.00	0.50	0.50	0.50	1.00
DiningPhil	1690 ms	1.00	1.16	0.05	0.03	1.59
CXCC0	10 ms	1.00	0.25	0.25	0.25	0.25
GEOMEAN	432 ms	1.00	0.80	0.58	0.34	1.07

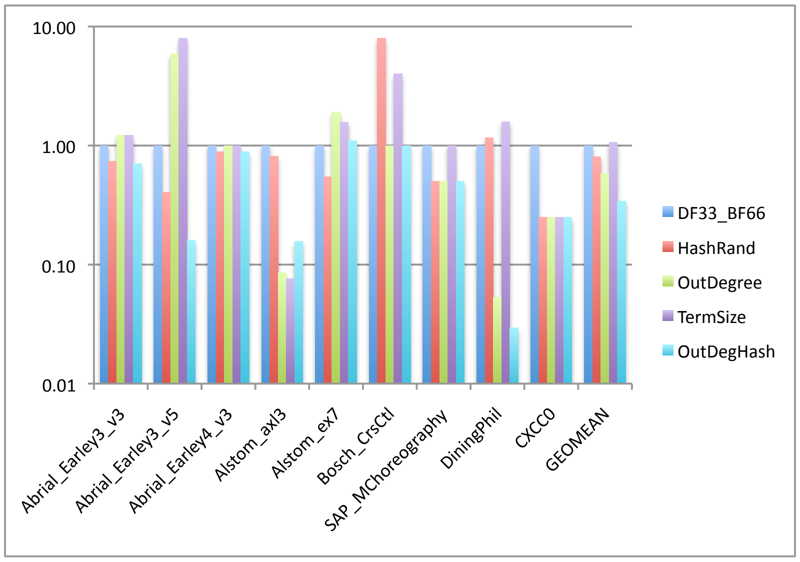

Table 6. Relative times for checking models with deadlocks (Heuristics Analysis)

Goal Benchmarks	DF33 (abs+rel)		HashRand	OutDegree	OutDegHash	TermSize	CUSTOM
RussianPostal	220 ms	1.00	0.45	0.77	0.36	1.18	0.45
TrainTorch	350 ms	1.00	0.97	0.26	1.14	0.20	0.20
BlocksWorld	440 ms	1.00	1.16	0.07	0.07	1.20	0.02
Farmer	20 ms	1.00	1.00	1.00	0.50	1.00	1.00
Hanoi	500 ms	1.00	0.52	0.92	0.52	0.90	0.34
Puzzle8	59060 ms	1.00	20.54	1.31	2.69	0.71	0.03
RushHour	127020 ms	1.00	0.42	0.60	0.56	1.12	0.79
Abrial_Press_m13	800 ms	1.00	0.49	2.36	2.21	2.48	0.20
Abrial_Queue_m1	50 ms	1.00	0.40	16.60	0.60	2.80	0.40
SystemOnChip_Router	2050 ms	1.00	0.10	0.05	0.04	1.50	72.42
Wegenetz	120 ms	1.00	0.17	0.33	0.08	0.08	0.08
GEOMEAN	715 ms	1.00	0.64	0.63	0.41	0.85	0.34

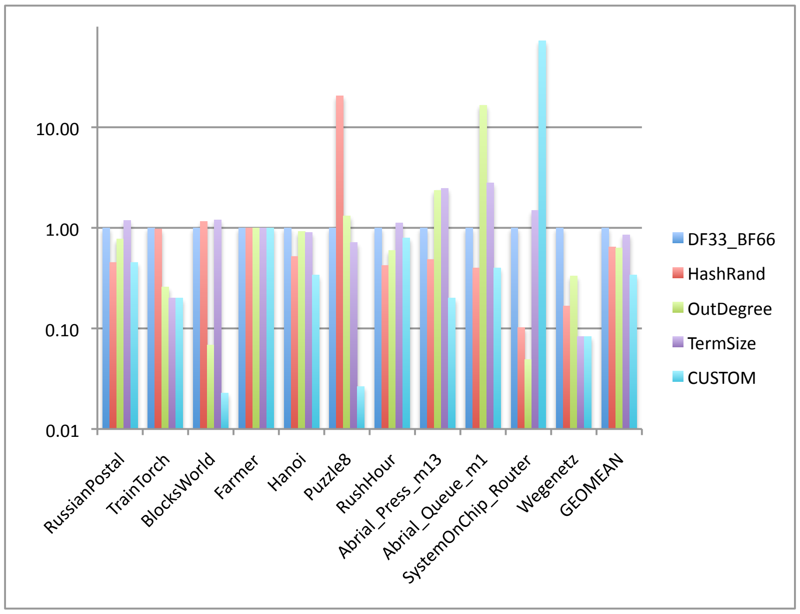

Table 7. Relative times for checking models with goals to be found (Heuristics Analysis)

noerror	DF33 (abs+rel)	HashRand	OutDegree	OutDegHash	TermSize
Scheduler1	6010 ms 1.00	1.00	1.00	1.00	1.02
Volvo_Cruise	5360 ms 1.00	1.01	1.00	1.01	1.01
USB4	3270 ms 1.00	1.00	0.99	1.00	1.00
Nokia_Nota	44220 ms 1.00	1.10	0.90	0.92	1.12
Huffman	11100 ms 1.00	1.64	1.04	2.09	1.03
Cansell_Contention	1490 ms 1.00	1.00	1.01	1.01	1.02
Demoney_GS_R1	1580 ms 1.00	1.00	0.99	1.01	1.03
SystemOnChip_Router	27020 ms 1.00	1.00	0.98	1.00	1.02
Mondex_m2	1700 ms 1.00	1.03	1.44	1.27	0.97
Mondex_m3	1910 ms 1.00	1.02	1.39	1.24	0.96
Siemens_ATP0	2090 ms 1.00	1.81	1.03	0.92	0.93
SSF_obsw1	22500 ms 1.00	1.12	1.01	1.02	1.19
ETH_Elevator12	106520 ms 1.00	1.09	0.54	0.56	0.66
GEOMEAN	6687 ms 1.00	1.12	1.00	1.04	0.99

Table 8. Relative times for checking models without errors (Heuristics Analysis)

4 Workpackage I1: Generic Parallelisation Prototype

We investigated different approaches for building a parallel version of ProB. Because Multicore platforms become more and more popular we decided to not only investigate a distributed version of ProB, but also a version that works on multicore platforms. However the lack of thread support in SICStus Prolog makes the implementation of a multicore version more difficult. In this report we describe some attempts we tried (i.e., using shared memory for inter process communication, using a database) and the lessons we learnt from these prototypes.

4.1 Interfacing with SICStus Prolog

Because SICStus Prolog does not yet support multithreading it is necessary to parallelize the model checking using different processes. To share data among these processes we considered the following approaches:

- Share data using socket communication to a Java process.
- Share data using the SICStus to C interface.
- Share data using direct socket communication between the Prolog Processes.

We use the first approach for interaction of the ProB core and our Plug-in for the Rodin tool. It has the advantage that we could reuse a large number of frameworks for distributed computing. However when developing the Rodin Plug-in we noticed that the communication has a performance impact that is tolerable when animating a model but it could become a problem when it comes to more frequent interaction between Java and Prolog during model checking. The Jasper Java interface of SICStus Prolog has turned out to be quite hard to setup and fragile across different platforms. The second approach was intensively pursued in the first project phase. It is required in particular when using shared memory to communicate between processes running on the same platform. We report on this approach in section 4.2. The last method was used in the first prototype that used a tuple space for communication. The prototype turned out to be not very robust against changes of the environment, i.e., a process is getting killed or a network connection is lost. Also the blackboard approach does only scale until a certain amount of traffic because of the client server structure. Implementing a more sophisticated concept in pure Prolog does not seems to be a good idea because it requires to write a lot of functionality from scratch. Instead we want to investigate to use Java as the control layer while Prolog processes share the data directly.

4.2 Share data using the SICStus C-Interface

ProB uses two main facts to represent the state space. To represent a state it asserts a `visited_expression/2` fact containing a unique id of the node and the state itself in a Prolog encoding. We represent a transition using a `transition/3` fact containing the ids of the source and the destination state and a representation of the transition event, i.e., an encoding of the opetartion's name and its parameters. We also have a few additional predicates as flags, e.g. `invariant_violated/1` to mark all stated that violate the invariant. The reason for this is that SICStus Prolog does not support multi indexing, i.e., finding all states violating the invariant would be inefficient if this information is incorporated into the `visited_expression` fact. However our C representation allows multi indexing therefore we can join all information about a particular state into a single structure. In addition we gain the possibility to efficiently search for incoming transitions. Without duplicating the `transition` facts this is not possible using SICStus Prolog.

However the representation in C has also some drawbacks. It is difficult to support sharing of data structures. Let us consider the following Prolog predicate.

```
share :- fact(A), fact(B), R=tree(A,B), assert(fact(R)).
```

If we call `share` Prolog will fetch the data structures A and B from the database, combine them and store them as a new structure. Notice that Prolog does not have to copy A and B. The new fact tree(A,B) contains pointers to the original structures. Discovering these structural sharing using the foreign interface is difficult because the internal representation is hidden from the interface.

Our prototype implementation stores states and transitions in the memory controlled by the C extension rather than Prolog together with some counters and index structures to provide a fast lookup of transitions. To save memory a symbol table mapping atoms to integer values was implemented. Is is also stored in the shared memory. However for the prototype the memory architecture is implemented as a static structure and has to be configured manually before compiling ProB.

Big segments of memory are allocated at startup of ProB and will be filled when model checking starts. ProB uses two segments for storing structs of transitions and visited expressions, with some meta information and a pointer to the term associated with these expressions, the terms are stored in two other chunks, containing the serialized terms as a string. For indexing transitions by source and destination three segments are allocated, a src- and dst-array pointing to an index element list in the third segment. Another segment stores some counter and the last one is used to store the symbol table.

Prolog terms are represented as a tree and can be accessed on the C-level with a hand full of functions provided by SICStus Prolog. Using these functions it is possible to determine how many arguments a functor has and to get the type and value of any atom. To flatten a term a recursive function in the extension is called and saves the term representing atoms as ids from the symbol table and arity of functors in a string. Saving the additional information of arity is used to deserialize the string to a tree in a single run.

4.3 Using ProB with shared memory

On Linux and OS X systems we can use the shared memory implementation for inter process communication (IPC) to share information among processes. The implementation provides functions to allocate, attach and detach shared memory segments and to create semaphores for synchronization. The idea is to use distributed shared memory to increase the power of ProB by using more CPUs and to have more RAM available. Indeed our prototype implementation of the Prolog to C interface already uses shared memory. In combination with OSS we can use the same implementation to get a version of ProB that can be used in a cluster.

4.4 Using Berkely DB

SICStus Prolog contains an interface to the Berkely Database(bdb) Because this interface is built in we wanted to evaluate if its performance is sufficient to share the state space among instances of the ProB process. We had the hope that SICStus can use faster techniques (i.e., direct copying of memory blocks) than a foreign interface can. We developed a simple prototype to compare the performance of asserting facts in the Prolog database and storing the facts in a database. We also tried to access the database concurrently with multiple threads. Storing a million facts into the Berkeley database is about 40 times slower than asserting them in Prolog. This performance loss is not surprising because the database uses disk I/O while Prolog uses memory. But the approach does not scale at all. Using two processes (one only reading and one reading and writing) the performance of the database collapses writing 100000 facts takes about 7 seconds using a single process. If we add a process that only reads facts from the database writing the same facts takes more than 2 minutes. Because we cannot control the way the database is used by Prolog we cannot fine-tune it, i.e., we cannot exploit the fact that we never remove entries from the state space. We will not consider using the SICStus bdb library in the future, but it might be an option to use a distributed database together with the Prolog to C interface instead of OSS.

4.5 Combining ProB and Spin

We did investigate using Spin, using Spin's C-interface to link the ProB interpreter via Prolog's C-interface. However, after discussions with Gerhard Holzmann and Dragan Bosnacki, this avenue seems very tricky as our Prolog interpreter is inherently non-deterministic.

4.6 Future Work

We currently evaluate if it is reasonable to use the shared memory approach for a multicore version of ProB. We require that the additional costs of storing the state space in the shared memory can be compensated by sharing the computation among a reasonable low number (i.e., two) processes. Luckily, first results have shown that this is the case for the majority of the benchmarks, but there are also several models where we clearly fail. In some cases the version that uses shared memory is up to 16 times slower than the Prolog version. We suspect that our rather naive encoding of the state (and in particular the symbol table) is the reason for these problems. However, we are confident that these issues can be solved. The next steps are therefore the optimization of the symbol table and a better memory management. It is clearly necessary to not allocate huge blocks of memory. Instead we want to implement our own page table and manage smaller chunks to get a more scalable system. Furthermore, we plan to port ProB to a distributed memory system using OSS. In addition we started to work on a prototype that uses Java to control communicating Prolog processes. We will not further pursue the database approach unless the OSS version turns out to be not sufficient and we will also not consider to combine ProB and Spin.

5 Workpackages T2/I2: Abstract Interpretation of B

During the evolution of the project, we realised that in addition to a "classical" abstract interpretation, we would need a flow analysis. The latter can actually also be viewed as a predicate-abstraction-based abstract interpretation. This is described in Workpackage T3.

As far as classical abstract interpretation is concerned, we have developed and implemented a particular predicate analysis technique to infer intervals for integers and cardinalities. We describe this in Section 5.1. We have also developed a generic abstract interpretation framework, which approximates the effects of B operations. We describe this in Section 5.2.

5.1 Predicate Analysis

A static analysis of predicates provides information that can be used in several contexts.

– Sometimes the constraint solver has to enumerate all possible values of variables to find values that satisfy a predicate. The additional information can be used to restrict the search to a subset of such values.
– Some variables can be represented more effectively when additional info about the possible values is known. E.g., a relation could be stored as a map if we known that it is a function.

We also plan to use our analysis to enable translation of B predicates to SAT solvers via Kodkod. Finally, we also plan to use our analysis to detect erroneous settings of MAXINT, MININT in ProB.

The analysis we implemented as a prototype is easily extensible for different kinds of information and works as follows. For each node in the abstract syntax tree of a predicate we store an arbitrary number of "inferred information points", depending of the node's type. E.g. for an integer expression we store the integer interval of possible values, identified by `intval`. For a set, we store the integer interval of possible cardinalities, identified by `card`.

Each type of information point can also be used for all elements of a set (`elem`). E.g. for a set of integers, we can store the cardinality of the set (`card`) and the integer interval in wich all elements are located (`elem:intval`).

We generate constraints between those information points by applying pattern matching to the syntax tree. E.g. for a predicate $x \in S$ we can propagate the information of each point `elem:I`(S) to an information point `I`(x). For a predicate $x < y$ we can propagate the minimum of `intval`(x) to `intval`(y) and the maximum of `intval`(y) to `intval`(x).

The prototype's constraint solving technique is a naïve approach where information is propagated until a fixpoint or a maximum number of inference steps is reached. The appliction of modern constraint solving techniques should be future work.

Example. For the predicate $x \in 1 \mathinner{..} 10$ we generate the following "inferred information points" with their initial values: $\texttt{intval}(x) = (-\infty, \infty)$, $\texttt{intval}(1) = [1, 1]$, $\texttt{intval}(10) = [10, 10]$, $\texttt{card}(1 \mathinner{..} 10) = [0, \infty)$, $\texttt{elem:intval}(1 \mathinner{..} 10) = (-\infty, \infty)$. For $x \mathinner{..} y$ we can propagate the minimum resp. maximum of $\texttt{intval}(x)$ and $\texttt{intval}(y)$ to the node $\texttt{elem:intval}(1 \mathinner{..} 10)$. Additionally we can use the information $\texttt{elem:intval}(1 \mathinner{..} 10)$ to limit the maximum cardinality $\texttt{card}(1 \mathinner{..} 10)$. With the inference rule of membership as presented above, we can derive the information: $\texttt{intval}(x) = (1, 10)$, $\texttt{card}(1 \mathinner{..} 10) = [0, 10]$, $\texttt{elem:intval}(1 \mathinner{..} 10) = [1, 10]$. Whereas the example seems trivial, the generality of the presented approach can be easily applied to more complex problems.

5.2 Generic Abstract Interpretation Framework for B

We have developed a framework for applying abstract interpretation to B and Event-B machines.

1. The state of a machine is represented using abstract domains. Our *framework* currently supports three domains: Boolean domains, integer domains and set domains. The set domain is approximated by an interval domain representing bounds of the set cardinality.
2. We have abstract counterparts for the B functions, such as union ($\cup$), addition ($+$), ... They take abstract values and produce a new abstract value.
3. We have abstract counterpart for the B predicates, such as membership ($\in$), $<$, ... They take abstract values and succeed if the predicate may be true. They can also narrow down the abstract values of its argument: e.g., after checking that a value is v is greater than zero ($v > 0$) we now definitely know in the remainder of the abstract interpretation that v must be greater than zero.
4. We have abstract counterparts for the B substitutions. In general these behave exactly like the concrete B substitutions; the only difference being that the environment contains abstract rather than concrete values.

   In the implementation we reuse the ProB interpreter-part for substitutions almost without modification. Note that non-determinism may arise for substitutions such as the IF-THEN-ELSE, in case the abstract values are not precise enough to decide which branch is taken. This situation does, however, not a pose a problem for the interpreter: it was already designed to work with partially instantiated states, where this same situation also arises.
5. The abstract interpretation procedure starts off with an abstract representation of the initial states and then repeatedly applies the abstract counterpart for all operations of the machine. This gives rise to new abstract states. In the mono-variant version of the procedure, only a single abstract state is retained and all states are merged using the leastupper-bound operator. In the polyvariant version, multiple abstract states are allowed. A widening operator can be applied to ensure convergence of the procedure.

5.3 Flow Analysis for B

We have also started to investigate the foundations of a new kind of flow analysis for Event-B. Flow analysis answers questions like "Can event h take place after event g was observed? If so, under which conditions?". We report on this work later in Section 6.2, together with its application to directed model checking.

6 Workpackage T3: Intelligent Techniques for Directed and Parallel Model Checking

Thus far we have concentrated within this workpackage on directed model checking. In particular we incorporated static information gained from a prover into the process of consistency checking (i.e., checking that all reachable states satisfy the invariant). Section 6.1 describes how we can use information about invariant preservation to reduce the size of the invariant we have to check, while section 6.2 describes an approach to reduce the number of guards evaluations using automatic flow analysis. We have also developed some more potential use cases, where flow analysis can help to improve model checking. We have a prototype implementation that shows that flow analysis is feasible for many Event-B models. A paper on automatic flow analysis has been submitted. We believe that flow analysis can help to find intelligent strategies for directed model checking and we want to continue to improve this kind of analysis in this work package.

6.1 Proof supported and proof-directed model checking

The Rodin platform for the formal Event-B notation provides an ideal framework for integrating model checking and prove techniques. Indeed, Rodin is based on the extensible Eclipse platform and as such it is easy for provers, model checkers and other arbitrary tools to interact. We report how we ca make use of this tight interaction inside Rodin to improve the ProB [36, 38] model checking algorithm by using information provided by the various Rodin provers.

More concretely, we show how we can optimize the *consistency checking* of Event-B and B models, i.e., checking whether the invariants of the model hold in all reachable states. The key insight is that from the proof information we can deduce that certain events are guaranteed to preserve the correctness of specific parts of the invariant. By keeping track of which events lead to which states, we can avoid having to check a (sometimes considerable) amount of invariants.

The Rodin platform interactively checks a model, generates and discharges proof obligations for Event-B. These proof obligations deal with different aspects of the correctness of a model. In this report we only deal with proofs that are related to invariant preservation, i.e., if the invariant holds in a state and we observe an event, the invariant still holds in the successor state:

$$I(v) \wedge G(v,t) \wedge S_{BA}(v,t,v') \Longrightarrow I(v')$$

By $S_{BA}(v,t,v')$ we mean the substitution S expressed as a Before-After predicate. The primed variables refer to the state after the event happened, the unprimed variables to the state before the event happened. In our small example, $S_{BA}(v,t,v')$ is the predicate $x' = x + ab$. If we want to express, that x is a positive integer, i.e. $x \in \mathbb{N}_1$, we need to prove:

$$x \in \mathbb{N}_1 \wedge a \in \mathbb{N} \wedge b \in \mathbb{N} \wedge x' = x + ab \Longrightarrow x' \in \mathbb{N}_1$$

This implication is obviously very easy to prove, in particular, it is possible to automatically discharge this obligation using the Rodin tool.

For each pair of invariant and event the Rodin Proof Obligation Generator, generates a proof obligation (PO) that needs to be discharged in order to prove correctness of a model as mentioned before. A reasonable number of these POs are discharged fully automatically by the tool. If an obligation is discharged, we know that if we observe an event and the invariant was valid before, then it will be valid afterwards. Before generating proof obligations, Rodin statically checks the model. Because this also includes type checking, the platform can eliminate a number of proof obligations that deal with typing only. For instance the invariant $x \in \mathbb{Z}$ does not give rise to any proof obligation, its correctness is guaranteed by the type checker.

The propagation and exploitation of this kind of proof information to help the model checker is the key concept of the combination of proving and model checking presented in this report.

Consistency checking One core application of ProB is the *consistency checking* of a B model, i.e., checking whether the invariant of a B machine is satisfied in all initial states and whether the invariant is preserved by the operations of the machine. ProB achieves this by computing the *state space* of a B model, by

– computing all possible initializations of a model and
– by computing for every state all possible ways to enable events and computing the effects of these events (i.e., computing all possible successor states).

Graphically, the state space of a B model looks like in Figure 5. Note that the initial states are represented as successor states of a special root node.

ProB then checks the invariant for every state in the state space. (Note that ProB can also check assertions, deadlock absence and full LTL properties [39].)

Another interesting aspect is that ProB uses a mixture of depth-first and breadth-first evaluation of the state space, which can lead to considerable performance improvements in practice [35].

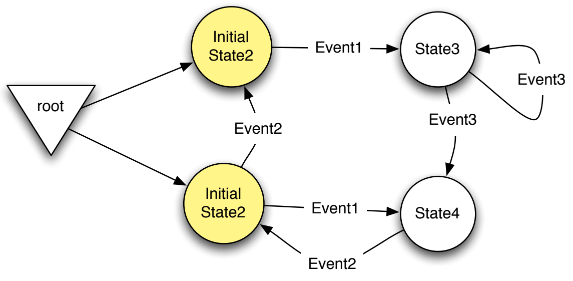

Fig. 5. A simple state space with four states

Proof-Supported Consistency Checking The status of a proof obligation carries valuable information for other tools, such as a model checker. As described, ProB does an exhaustive search, i.e. it traverses the state space and verifies that the invariant is preserved in each state. This section describes how we incorporate proof information from Rodin into the ProB core.

Assuming we have a model, that contains the invariant $[I_1, I_2, I_3]$[8] and we follow an event *evt* to a new state. If we would, for instance, know that *evt* preserves I_1 and I_3, there would be no need to check these invariants. This kind of knowledge, which is precisely what we get from a prover, can potentially reduce the cost of invariant verification during the model checking.

[8] Sometimes it is handier to use a list of predicates rather than a single predicate, we use both notations equivalently. If we write $[P_1, P_2, \ldots, P_n]$, we mean the predicate $P_1 \wedge P2 \wedge \ldots \wedge P_n$.

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

Using all this information, we determine an individual invariant for each event that is defined in the machine. Because we only remove proven conjuncts, this specialized invariant is a subset of the model's invariant. When encountering a new state, we can evaluate the specialized invariant rather than the machine's full invariant.

As an example we can use the Event-B model shown in Figure 6. The full state space of this model and the proof status delivered by the automatic provers of the Rodin tool are shown in Figure 7.

VARIABLES
  f, x
INVARIANTS
  $inv1: \ f \in \mathbb{N} \mapsto \mathbb{N}$
  $inv2: \ x > 3$
EVENTS
Initialisation
$f := \{1 \mapsto 100\} || x := 10$
Event a $\widehat{=}$
$f := \{1 \mapsto 100\} || x := f(1)$
Event b $\widehat{=}$
$f := f \cup \{1 \mapsto 100\} || x := 100$

Fig. 6. Example for intersection of invariants

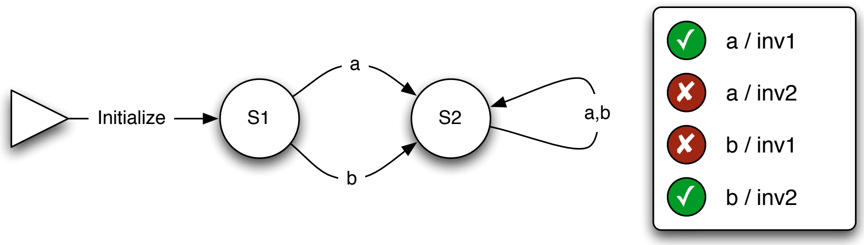

Fig. 7. State space of the model in figure 6

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

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

This is of course only a tiny example but it demonstrates, that using proof information we are able to reduce the number of invariants for each event significantly, and sometimes by combining proof information from different events, we are able to get rid of the whole invariant. We actually have evidence that this is not only a theoretical possibility, but happens in real world specifications (see Section 7.2).

Algorithm 8 describes ProB's consistency checking algorithm, we will justify it formally in section 6.1. The algorithm employs a standard queue data structure to store the unexplored nodes. The key operations are:

– Computing the successor states, i.e., "$state \rightarrow_{evt} succ$".
– Verification of the invariant "$\exists inv_i \in Inv(state)$ $s.t.$ inv_i is $false$"
– Determining whether "$succ \notin$ Visited"

The algorithm terminates when there are no further queued states to explore or when an error state is discovered. The underlined parts highlight the important differences with the algorithm in [38].

In contrast to the algorithm, the actual implementation does the calculation of the intersection ($Inv(succ) := Inv(succ) \cap Unproven(op)$) in a lazy manner, i.e., for each $state \notin Visited$, we store the event names as a list. As soon as we evaluate the invariant of a state, we calculate and evaluate the intersection on the fly. The reason is, that storing the invariant's predicate for each state is typically more expensive than storing the event names.

Verification To show, that our approach is indeed correct, we developed a formal model of an abstraction of algorithm 8. We omitted few technical details, such as the way the state space is traversed by the actual implementation and also we omitted the fact, that our implementation always uses all available information. Instead, we have proven correctness for any traversal and any subset of the available information. Our model was developed using Event-B and fully proven in Rodin. The model is available as a Rodin 1.0 archive from `http://deploy-eprints.ecs.soton.ac.uk/152/`. In this report we present only some parts of the model and some lemmas, without their proofs. All the proofs can be found in the file, we thus refer the reader to the Rodin model.

```
while $Queue$ is not empty do
  if random(1) $< \alpha$ then
    $state$ := pop_from_front($Queue$); /* depth-first */
  else
    $state$ := pop_from_end($Queue$); /* breadth-first */
  end if
  if $\underline{\exists inv_i \in Inv(state) \text{ s.t. } inv_i \text{ is false}}$ then
    return counter-example trace in $Graph$
               from $root$ to $state$
  else
    for all $succ$,$evt$ such that $state \rightarrow_{evt} succ$ do
      $Graph := Graph \cup \{state \rightarrow_{evt} succ\}$
        if $succ \notin Visited$ then
            push_to_front($succ$, $Queue$);
            $Visited := Visited \cup \{succ\}$
            $\underline{Inv(succ) := Unproven(evt)}$
        else
            $\underline{Inv(succ) := Inv(succ) \cap Unproven(evt)}$
        end if
      end if
    end for
od
return ok
```

Fig. 8. Proof-Supported Consistency Checking

We used three carrier sets *STATES*, *INVARIANTS* and *EVENTS*. We assume, that these sets are finite. For invariants and events this is true by definition in Event-B, but the state space can in general be unbounded. However, the assumption of only dealing with finite state spaces is reasonable in the context of our particular model, because we can interpret the *STATES* set as the subset of all states that can be traversed by the model checker within some finite number of steps.[9] The following definitions are used to prove some properties of Event-B:

$truth \subseteq STATES \times INVARIANTS$
$trans \subseteq STATES \times STATES$
$preserve = \{s \mid \{s\} \times INVARIANTS \subseteq truth\}$
$violate = STATES \setminus preserve$
$label \subseteq trans \times EVENTS$
$discharged \subseteq EVENTS \times INVARIANTS$

The model also contains a set *truth*: pair of a state s and an invariant i is in *truth* if and only if i holds in s. The set *preserve* is defined as the set of states where each invariant holds, the relations *trans* and *label* describe, how two states are related, i.e. a triple $(s \mapsto t) \mapsto e$ is in *label* (and therefore $s \mapsto t \in trans$) if and only if t can be reached from s by executing e. The observation that is the foundation of all theorems we proved and is the following assumption:

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

The assumption is, that if we reach a state t from a state s where all invariants hold by executing an event e and we know, that the invariant i is preserved by e, we an be sure, that i holds in t. This statement is what we prove by discharging an invariant proof obligation in Event-B, thus it is reasonable to assume that it holds.

We are now able to prove a lemma, that will capture the essence of our proposal; it is enough to find for each invariant i one event that preserves this invariant leading from a consistent state into a state t to prove, that all invariants hold in t.

Lemma 1. $\forall t \cdot t \in STATES \land (\forall i \cdot i \in INVARIANTS \land (\exists s, e \cdot s \in preserve \land e \in EVENTS \land (s \mapsto t) \in trans \land (s \mapsto t) \mapsto e \in label \land e \mapsto i \in discharged)) \Rightarrow t \in preserve$

Proof. All proofs have been done using Rodin and can be found in the model archive. □

[9] Alternatively, we can remove this assumption from our Rodin models. This only means that we are not be able to prove termination of our algorithm; all other invariants and proofs remain unchanged.

We used five refinement steps to prove correctness of our algorithm. We will describe the first three steps, the last two steps are introduced to prove termination of new events. The first refinement step *mc0* contains two events *check_state_ok* and *check_state_broken*. The events take a yet unprocessed state and move it either into a set containing consistent or inconsistent states. Algorithm 9 shows the *check_state_ok* event, *check_state_broken* is defined analogously, except that it has the guard $s \notin preserve$ and it puts the state into the set *inv_broken*.

```
event check_state_ok
  any s
  where
    $s \in open$
    $s \in preserve$
  then
    $inv\_ok := inv\_ok \cup \{s\}$
    $open := open \setminus \{s\}$
end
```

Fig. 9. Event *check_state_ok* from *mc0*

At this very abstract level this machine specifies that our algorithm separates the states into two sets. If they belong to *preserve*, the states are moved into the set *inv_ok*. Otherwise, they are moved into *inv_broken*. Lemma 2 guarantees, that our model always generate correct results.

Lemma 2. *mc0 satisfies the invariants*

1. $inv\_ok \cup inv\_broken = STATES \setminus open$
2. $open = \varnothing \Rightarrow inv\_ok = preserve \wedge inv\_broken = violate$

The next refinement strengthens the guard and removes the explicit knowledge of the sets *preserve* and *violate*, the resulting proof obligation leads to lemma 3.

Lemma 3. *For all* $s \in open$

$$\{s\} \times INVARIANTS \setminus discharged[label[inv\_ok \lhd trans \rhd \{s\}]]) \subseteq truth$$
$$\Leftrightarrow s \in preserve$$

The third refinement introduces the algorithm. We introduce a new relation *invs_to_verify* in this refinement. The relation keeps track of those invariants, that need to be checked, in the initialization, we set $invs\_to\_verify := STATES \times INVARIANTS$.

The algorithm has three different phases. It first selects a state that has not been processed yet then it checks if the invariant holds and moves the state into either *inv_ok* or *inv_broken*. Finally, it uses the information about discharged proofs to remove some elements from *invs_to_verify* as shown in algorithm 10.

```
event mark_successor
  any p s e
  where
    $p \in inv\_ok$
    $s \in trans[\{p\}]$
    $(p \mapsto s) \mapsto e \in label$
    $(p \mapsto s) \mapsto e \notin marked$
    $ctrl = mark$
  then
    $invs\_to\_verify := invs\_to\_verify \lhd\!\!\!- (\{s\} \times (invs\_to\_verify[\{s\}] \cap unproven[\{e\}]))$
    $marked := marked \cup \{(p \mapsto s) \mapsto e\}$
end
```

Fig. 10. Event *mark_successor* from *mc2*

We take some state s and event e, where we know that s is reachable via e from a state p, where all invariants hold. Then we remove all invariants but those, that are not proven to be preserved by e. This corresponds to the calculation of the intersection in algorithm 8.

The main differences between the formal model and our implementation are, that the model does not explicitly describe how the states are chosen and the algorithm uses all available proof information while the formal model can use any subset. In addition, the model does not stop if it detects an invariant violation. We did not specify these details because it causes technical difficulties (e.g., we need the transitive closure of the *trans* relation) but does not seem to provide enough extra benefit.

Correctness of algorithm 8 is established by the fact that the outgoing edges of a *state* are added to the *Graph* only *after* the invariants have been checked for *state*. Hence, the removal of a preserved invariant only occurs *after* it has been established that the invariant is true before applying the event. This corresponds to the guard $p \in inv\_ok$. However, the proven proof obligations for an event only guarantee *preservation* of a particular invariant, not that this invariant is established by the event. Hence, if the invariant is false before applying the event, it could be false after the event, *even* if the corresponding proof obligation is proven and true. If one is not careful, one could easily set up cyclic dependencies and our algorithm would incorrectly infer that an incorrect model is correct.

Proof-Assisted Consistency Checking for Classical-B In the setting of Event-B and the Rodin platform, ProB can rely on the other tools for providing type inference and as we have seen the proof information.

In the context of classical B, we are working on a tighter integration with Atelier B [54]. However, at the moment ProB does not have access to the proof information of classical B models.

ProB does perform some additional analyses of the model and annotates the AST (Abstract Syntax Tree) with additional information. For instance for each event we calculate a set of variables that are possibly modified. For instance if we analyze the operation[10]

$$Operation1 = BEGIN\ x := z \mid\mid y := y \wedge \{x \mapsto z\}\ END$$

the analysis will discover that the set of variables that could potentially influence the truth value of the invariant is $\{x, y\}$.

This analysis was originally used to verify the correct usage of `SEES` in the classical B-Method. The `SEES` construct was used in the predecessor of Event-B, so-called classical B, to structure different models. In classical B a machine can see another machine, i.e., it is allowed to call operations that do not modify the state of the other machine. To support this behavior, it was necessary to know if an operation has effect on state variables, that is the set of modified variables is the empty set. It turned out, that the information is more valuable than originally thought, as it is equivalent to some proof obligation:

If u and v are disjoint sets of state variables, and the substitution of an operation is $S_{BA}(v, t, v')$ we know that $u = u'$ and thus a simplified proof obligation for the preservation of an invariant $I(u)$ over the variables u is

$$I(u) \wedge G(u \cup v, t) \wedge S_{BA}(v, t, v') \Rightarrow I(u)$$

which is obviously true. These kind of proof obligations are not generated by any of the proving environments for B we are aware of. In particular Rodin does

[10] Operations are the equivalent of events in classical B.

not generate them. For a proving environment, this is a good idea as they do not contain valuable information for the user and they can be filtered out by simple syntax analysis. But for the model checker these proofs are very valuable; in most cases they allow us to reduce the number of invariants we need to check. As this type of proof information can be created from the syntax, we can use them even if we do not get proof information from Rodin, i.e., when working on classical B machines. As such, we were able to use Algorithm 8 also for classical B models and also obtain improvements of the model checking performance (although less impressive than for Event-B).

Conclusion and Future Work First of all, we never found a model where using proof information significantly reduced the performance, i.e., the additional costs for calculating individual invariants for each state are rather low. Using proof information is the new default setting in ProB.

We got a number of models, in particular those coming from industry, where using the proof information has a high impact on the model checking time. In other cases, we gained only a bit or no improvement. This typically happens if the invariant is rather cheap to evaluate compared to the costs of calculating the guards of the events. We used an out-of-the-box version of Rodin[11] to produce our experimental results. Obviously, it is possible to further improve them by adding manual proof effort. In particular, it gives the user a chance to influence the speed of the model checker by proving invariant preservation for those parts that are difficult to evaluate, i.e., those predicates that need some kind of enumeration.

Related Work A similar kind of integration of theorem proving into a model checker was previously described in [**?**]. In their work Pnueli and Shahar introduced a system to verfify CTL and LTL properties. This system works as a layer on top of CMU SMV and was sucessfully applied to fragments of the Futurebus+ system [**?**]. SAL is a framework and tool to combine different symbolic analysis [52], and can also be viewed as an integration of theorem proving and model checking. Mocha [5] is another work where a model checker is complemented by proof, mostly for assume-guarantee reasoning. Some more works using theorem proving and model checking together are [20, 7, 21, 26].

In the context of B, the idea of using a model checker to assist a prover has already been exploited in practice. For example, in previous work [11] we have already shown how a model checker can be used to complement the proving environment, by acting as a disprover. In [11] it was also shown that sometimes the model checker can be used as a prover, namely when the underlying sets of the proof obligation are finite. This is for example the case for the vehicle function mentioned in [36]. Another example is the Hamming encoder in [18], where Dominique Cansell has used ProB to prove certain theorems which are difficult to prove with a classical prover (due to the large number of cases).

[11] For legal reasons, it is necessary to install the provers separately

Future Work We have done but a first step towards exploiting the full potential for integrating proving and model checking. For instance, we may feed the theorem prover with proof obligations generated by the model checker in order to speed up the model checking. A reasonable amount of time is spent evaluating the guards. If the model checker can use the theorem prover to prove that an event e is guaranteed to be disabled after an event f occurs, we can reduce the effort of checking guards. We may need to develop heuristics to find out when the model checker should try to get help from the provers.

Also we might feed information from the model checker back into the proving environment. If the state space is finite and we traverse all states, we can use this as a proof for invariant preservation. ProB restricts all sets to finite sets [38] to overcome the undecidability of B, so this needs to be handled with care. We need to ensure, that we do not miss states because ProB restricted some sets. Also we need to ensure that all states are reachable by the model checker, thus we may need some additional analysis of the model.

We also think of integrating a prover for classical B, to exploit proof information. The integration is most likely not as seamless as in Rodin and the costs of getting proof information is higher.

Although the cost of calculating the intersections of the invariants for each state is too low to measure it, the stored invariants take some memory. It might be possible to find a more efficient way to represent the intersections of invariants.

6.2 Automatic Flow Analysis

Event-B [3] has only very limited ways to express ordering of events. In particular it lacks a notion of sequential composition, or other ways to explicitly describe the ordering of events. If we specify software or systems that include software in Event-B, we often have some implicit algorithmic structure.[12] Unfortunately this information is implicit only and therefor not directly usable by tools nor directly visible to users. We report on a method to uncover this implicit algorithmic structure. This information can be useful for analyzing or comprehending models and for automatic code generation. We also show how to use this information to improve model checking. We use Event-B to illustrate our approach and present an implementation for Event-B inside the animator an model checker ProB. Our method, however, is not limited to that particular specification language and can also be used for a range of other formalisms such as TLA$^+$.

Preliminaries We follow the style of [3] of expressing variables and substitution in formulas. In particular, let $v = v_1, \dots, v_n$ be a sequence of n distinct variables, $t = t_1, \dots, t_n$ 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 $v_1, \dots, v_n$. Once the formula $F(v)$ has been introduced, we denote by $F(t)$ the formula $F[t/v]$ with v replaced by t.

In Event-B a state consists of a set of variables that are modified by events. The values of the variables are constrained by invariants $I(v)$. Each event is composed of a *guard* $G(t, v)$ and an *action* $S(t, v)$, where t are *parameters* of the event. We will only consider events of the form

$$\begin{aligned} evt \mathrel{\widehat{=}}\ & \textsf{any } t \\ & \textsf{when } G(t, v) \\ & \textsf{then } v_{i_1}, \dots, v_{i_k} := E_1(v, t), \dots, E_k(v, t) \textsf{ end} \end{aligned}$$

for some $i_j \in i_1, \dots, i_n$. Note that t can be empty and $G(t, v)$ can be *true*. Also note that k can be 0, in which case we write the action part as skip.

All assignments of an action $S(t, v)$ occur simultaneously. Variables $v_{j_1}, \dots, v_{j_l}$ that do not appear on the left-hand side of an assignment of an action are not changed by the action. The effect of an assignment can be described by a before-after predicate:

$$S(v, t, v') \ \widehat{=}\ v'_{i_1} = E_1(v, t) \wedge \dots v'_{i_k} = E_k(v, t) \wedge v'_{j_1} = v_{j_1} \wedge \dots v'_{j_l} = v_{j_l}$$

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'.

[12] To order events in Event-B the usual method is to introduce abstract program counters.

Note that Event-B also allows non-deterministic actions of the form $x :\in E(t, v)$ or $x :\mid Q(t, v, x')$. To simplify the presentation of our method, and without loss of generality, we assume that those are rewritten to the above form using new parameters, one for every non-deterministic action which denotes the chosen element. For instance, we rewrite

$$\textsf{any } max \textsf{ when } max > 10 \textsf{ then } x :\in 1..max \textsf{ end}$$

into

$$\textsf{any } max, choice \textsf{ when } max > 10 \land choice : 1..max \textsf{ then } x := choice \textsf{ end}$$

Dependency Between Events We are interested in how events influence each other. The motivations are multiple: either we may try to understand the dynamic behavior of our model, we may wish to generate code by determining the control flow or we may wish to improve the performance of model checking...

Suppose we have an event g with action $x, y := x + 1, 0$. There are various ways it can influence another event:

1. it can disable another event. E.g., the event h with guard $y > 0$ will for sure be disabled after executing g.
2. it can enable another event. E.g., the event h' with guard $y = 0$ would for sure be enabled after executing g.
3. it can be independent of another event. For example, the enabling of the event h'' with guard $z > 0$ would not be modified by executing g, i.e., it will be enabled after g if and only if it was enabled before. (Note that, depending on the action part of h'', the effect of h'' could have been modified.)

In cases 1 and 2 the enabling or disabling may depend on the current state of the model. Take for example the event h''' with guard $y = 0 \land x > 1$. Then h''' would be enabled after g if $x > 0$ holds in the state before executing g, and disabled otherwise. The predicate $x > 0$ is what we call an enabling predicate, and which we define as follows:

Definition 1 (Enabling predicate). *The predicate P is called enabling predicate for an event h after an event g, denoted by $g \rightsquigarrow_{P(v,t,s)} h$, if and only if the following holds*

$$I(v) \land G(v, t) \land S(v, t, v') \Rightarrow (P(v, t, s) \Leftrightarrow H(v', s))$$

where $I(v)$ is the invariant of the machine, $G(v, t)$ is the guard of g with parameters t and $S(v, t, v')$ the before-after predicate of its action part, and where $H(v, s)$ is the guard of h with parameters s.

In the absence of non-deterministic actions, an equivalent definition can be obtained using the weakest precondition notation:

$$I(v) \land G(v, t) \Rightarrow (P(v, t, s) \Leftrightarrow [S(t, v)]H(v, s))$$

where $[S]P$ denotes the weakest precondition which ensures that after executing the action S the predicate P holds.

Note that it is important for us that the action part $S(t, v)$ of an event does not contain any non-determinism (i.e., that all non-determinism has been lifted to the parameters t; see Section 6.2). Indeed, in the absence of non-determinism, the negation of an enabling predicate is a disabling predicate, i.e., it guarantees that the event h is disabled after g if it holds (together with the invariant) before executing g. However, if we have non-determinism the situation is different. There may even exist no solution for $P(v, t, s)$ in Def. 1, as the following example shows.

Example 1. Take $x :\in \{1, 2\}$ as the action part of an event g with no parameters and the guard $true$ and $x = 1$ as the guard of h. Then $[S(t, v)]H(v) \equiv false$ as there is no way to guarantee that h is enabled after g. Indeed, there is no predicate over x that is equivalent to $x' = 1$ in the context Def. 1 : the before after predicate $S(v, t, v')$ is $x' \in \{1, 2\}$ and does not link x and x'. Similarly, there is no way to guarantee that h is disabled after g. In particular, $\neg[S(t, v)]H(v) \equiv true$ is not a disabling predicate.

Note that if $I(v) \wedge G(v) \wedge [S(t, v)]H(v, s)$ is inconsistent, then any predicate $P(v, t, s)$ is an enabling predicate, i.e., in particular $P(v, t, s) \equiv false$.

How can we compute enabling predicates? Obviously, $[S(t, v)]H(v)$ always satisfies the definition of an enabling predicate. What we can do, is simplify it in the context of $I(v) \wedge G(v)$.[13] We will explain later in Sect. 6.2 exactly how we compute enabling predicates.

Example 2. Take for instance a model of a for loop that iterates over an array and increments each value by one. Assuming the array is modeled as a function $f : 0..n \rightarrow \mathbb{N}$ and we have a global counter $i : 0..(n + 1)$, we can model the for loop (at a certain refinement level) using two events $terminate$ and $loop$.

$$terminate \mathrel{\widehat{=}} \textsf{when } i > n \textsf{ then skip end}$$

$$loop \mathrel{\widehat{=}} \textsf{when } i \leq n \textsf{ then } f(i) := f(i) + 1 || i := i + 1 \textsf{ end}$$

We can now try to find enabling predicates for each possible combination of events. Table 9 shows the proof obligations from Def. 1 and simplified predicates P which satisfy it.

The directed graph on the left in Figure 11 is a graphical representation of Table 9. Every event is represented by a node and there for every enabling predicate $first \rightsquigarrow_P second$ from Table 9 there is an edge between the corresponding nodes.

The right picture shows the same graph if we take independence of events into account, i.e., if an event g cannot change the guard of another event h, we

[13] This is similar to equivalence preserving rewriting steps within sequent calculus proofs, where $I(v), G(v)$ are the hypothesis and $[S(t, v)]H(v)$ is the goal of the sequent.

Event Pairs (first $\rightsquigarrow_P$ second)	Enable Predicate Definition (wp notation)	Simplified P
terminate $\rightsquigarrow_P$ terminate	$i > n \implies (P \iff i > n)$	true
loop $\rightsquigarrow_P$ loop	$i \leq n \implies (P \iff (i+1) \leq n)$	$(i+1) \leq n$
loop $\rightsquigarrow_P$ terminate	$i \leq n \implies (P \iff (i+1) > n)$	$(i+1) > n$
terminate $\rightsquigarrow_P$ loop	$i > n \implies (P \iff i \leq n)$	false

Table 9. Enable Predicates for a simple model

do not insert an edge between g and h. In particular, as *terminate* does not modify any variables, it cannot modify the truth value of any guard. On first sight it seems as if we may have also lost some information, namely that after the execution of *terminate* the event *loop* is certainly disabled. We will return to this issue later and show that for the purpose of reducing model checking and other application, this is actually not relevant.

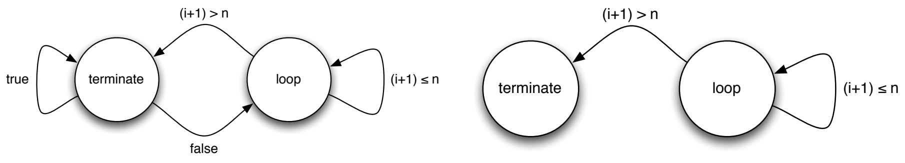

Fig. 11. Graph Representations of Dependence for a Simple Model

In Event-B models of software components independence between events occurs very often, e.g., if an abstract program counter is used to activate a specific subset of the events at a certain point in the computation. We can formally define independence as follows.

Definition 2 (Independence of events). *Let g and h be events. We say that h is independent from g — denoted by $g \not\rightsquigarrow h$ — if the guard of h is invariant under the substitution of g, i.e., iff the following holds:*

$$I(v) \land G(v,t) \land S(v,t,v') \implies (H(v,s) \iff H(v',s))$$

Our first observation is that an event g can only influence the enabledness of an event h (we do not require $g \neq h$) if g modifies some variables that are read in the guard of h. We denote the set of variables used in the guard of h by $read(h)$ and the set of variables modified by g by $write(g)$. If $write(g)$ and $read(h)$ are disjoint, then h is trivially independent from g:

Lemma 4. *For any two events h and g we have that $read(h) \cap write(g) = \varnothing \Rightarrow g \not\rightsquigarrow h$.*

This happens in our loop example, because $read(terminate) = \varnothing$, and hence all events (including *terminate* itself) are independent from *terminate*.

However, $read(h) \cap write(g) = \varnothing$ is sufficient for independence of events but not necessary. Take for instance the events from figure 12. Event g clearly modifies variables that are read by h and therefor $read(h) \cap write(g) \neq \varnothing$ but g can not enable or disable h.

```
event g                 event h
 begin                   when
   x := x + 1             x + y > 5
   y := y - 1            then
 end                     end
```

Fig. 12. Independent events

The trivial independence can be decided by simple static analysis, i.e., by checking if $read(h) \cap write(g) = \varnothing$. Non trivial independence is in general undecidable. In practice, it is a good idea to try to prove that two events are independent in the sense of Def. 2, as it will result in a graph representation with fewer edges. However, it is not crucial for our method that we detect all independent events.

As we have seen in the right side of Fig. 11, the information we gain about enabling and independence can be represented as a directed graph, now formally defined as follows.

Definition 3 (Enable Graph). *An Enable Graph for an Event-B model is a directed edge labeled graph $G = (V, E, L)$. The vertices V of the graph are the events of the model. Two events can be linked by an edge if they are not independent, i.e., $(g \mapsto h) \notin E \Rightarrow g \not\rightsquigarrow h$. Each existing edge $g \mapsto h$ is labeled with the enabling predicate, i.e., $g \rightsquigarrow_{L(g \mapsto h)} h$.*

Above we define a family of enable graphs, depending on how precise our information about independence is. Below, we often talk about the enable graph for a model, where we assume a fixed procedure for computing independence information.

Aside. There is another representation of the graph that is sometimes more convenient for human readers. We can represent the graph as a forest where each tree has one event as its root and only the successor nodes as leafs. The alternate representation is shown in figure 13 for a small example.

Computing the Enabling Predicates As mentioned in section 6.2, the weakest precondition $[S(t, v)]H(v, t, s)$ obviously satisfies the property of enabling predicates. We can use the syntax tree library of the Rodin tool to calculate the weakest precondition. However, these candidates for enabling predicates need to be simplified, otherwise they are as complicated as the original guard and we will gain no benefit from them. therefor we simplify the candidate, in the context of the invariant $I(v)$ and the guard of the preceding event $G(v, t)$.

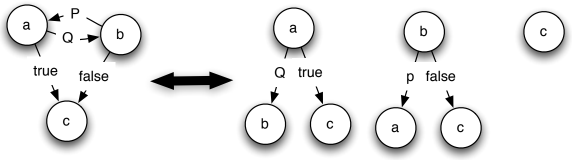

Fig. 13. Representations of the Enable Graph

Consider the model shown in figure 6.2. The weakest precondition for g preceding h is $[x := x+2]x = 1$ which yields $x = -1$. This contradicts the invariant $x > 0$ and thus h can never be executed after g took place. In the context of the invariant, $x = -1$ is equivalent to false.

```
invariant x > 0
event g                 event h
 begin                   when x = 1
   x := x + 2           end
 end
```

Fig. 14. Simplification

The simplification of the predicates is an important step in our method, deriving an enabling predicate $P(v,t,s)$ from the weakest precondition $[S(t,v)]H(v,s)$. Recall, that we simplify the predicate $[S(t,v)]H(v,s)$ in the context of the invariant $I(v)$ (and the guard $G(v,t)$)

$$I(v) \wedge G(v,t) \Rightarrow (P(v,t,s) \Leftrightarrow [S(t,v)]H(v,s))$$

A very important requirement in our setting is that the simplifier never increases the number of conjuncts. We have to keep the input for our enable and flow graph constructions small to prevent exponential blowup. Our simplifier shall find out if a conjunct is equivalent to true or false. In the first case the conjunct can be removed from the predicate in the second case the whole predicate is equivalent to false.

We have implemented a first prototype simplifier in Prolog that uses a relatively simple approach. The core algorithm is presented in Fig. 15. First it normalizes all negated formulas; this is part of the *conjuncts* function, which also translates the formula into a set of conjuncts. For example, the formula $x \neq false$ is normalized to $x = true$ and $\neg(a \leq b)$ is normalized to $a > b$. Then we successively try to add the conjuncts of the formulas to a set of current conjuncts K. If we try to add the conjunct C we can observe two special cases:

1. c is already a member of K: then we skip c because we know that it is *true* in the context of K.

```
$K_I := closure(conjuncts(I(v)))$ (is precomputed)
$K_g := closure(conjuncts(G(v,t)))$ (is precomputed for every event $g$)
$K := closure(K_I \cup K_g)$
$WP := conjuncts([S(t,v)]H(v,s))$
$P := \varnothing$
while $WP \neq emptyset$ do
  choose $C$ and remove from $WP$
  if $C \notin K$ then
    $K := closure(K \cup \{C\})$
    if inconsistent(K) then
     $P := \{\text{false}\}$; $WP := \varnothing$
    else
     $P := P \cup \{C\}$
    end if
  end if
od
$P$ is the simplified version of $[S(t,v)]H(v,s)$
```

Fig. 15. Algorithm for simplifying the weakest precondition in the context of the invariant and guard of g

2. c is inconsistent with a member of K: then the enabling predicate is $false$.

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

```
inconsistent_fact(not_equal,X,X).
inconsistent_fact(less,X,X).
inconsistent_fact(less,X,Y) :- value(X), value(Y), X >= Y.
inconsistent_fact(equal,X,Y) :-  value(X), value(Y), X \= Y.
inconsistent_fact(member,_,empty_set).
% some more rules [...]
```

The first rule states that $x \neq x$ is inconsistent, the second one that $x < x$ is inconsistent, the third one that $x < y$ is inconsistent if x and y are known values with $x \geq y$, and so on.

In any other case we add the conjunct and calculate the closure of K using some rules that combine two formulas, computing new logical consequences (e.g., deriving from transitivity of the arithmetic operators, i.e., $x \leq y$ and $y < z$ implies $x < z$).

For example, assume we have built the set of conjuncts $K = \{x < 2, y = 5\}$ and we now try to add $x = y$. Combining $x = y$ with $y = 5$ yields $x = 5$ which we add to K.

Assume we have only the rules mentioned above. Now we combine $x = 5$ and $x < 2$, yielding the new fact $5 < 2$. This triggers the third Prolog rule detecting an inconsistency.

The actual implementation does not always recompute everything from scratch. As outlined in algorithm 15 we precompute a closure K_I from the conjuncts of the invariant. This set can be reused for the whole model. For each event g we precompute a closure K_g from the guards. This set can be reused for all enabling predicates where g is the first event.If the algorithm has not stopped because of a contradiction the enabling predicate is the conjunction of the formulas stored in P.

Using the Enable Graph for Model Checking The enable graph contains valuable information for a model checker. In this section we describe how it can be used within ProB. When checking the consistency of an Event-B model, ProB traverses the state space of the model starting from the initialization and checks the model's invariant for each state it encounters. The cost for checking a state is the sum of the cost of evaluating the invariant for the state and the calculation of the successors. Finding successor states requires to find solutions for the guards of each event. A solution means that the event is applicable and we can find some parameter values. ProB then applies the actions to the current state using the parameter values resulting in some successor states. In some cases the enable graph can be used to predict the outcome of the guard evaluation. The special case of an enabling predicate $P = false$ is very important. It means that no matter how we invoke g we can omit the evaluation of the guard of h because it will be false after observing g. In other words it is a proof that the property *h is disabled* holds in any state that is reachable using g.

When encountering a new state s via event e, we look up e in the enable graph. We can safely skip evaluation of the guards of all events f that have an edge (e,f) which is labeled with $false$ in the Enabled Graph. We can even go a step further if we have multiple ways to reach s. When considering an event to calculate successor states we can arbitrary chose one of the incoming events and use the information from the enable graph. For instance, if we have four events a, b, c and d and we know that a disables c and b disables d. Furthermore we encounter a state s via a but do not yet calculate the successors. Later we encounter s again, this time via b. When calculating the successors we can skip both, c and d.

The reason is that we have a proof for *c is disabled* because the state was reachable using event a and a proof that *d is disabled* because the state was reachable using event b. Thus the conjunction *c and d are disabled* is also true.

Because we use the invariant when simplifying the enabling predicate (see Section 6.2), the invariant must hold in the previous state in order to use the flow information. However we believe this is reasonable because most of the time we are hunting bugs and thus we stop at a state that violates the invariant. The implementation must take this into account and in case of an invariant violation it must not use the information gained by flow analysis. Also it needs to check not only the invariant but also the theorems if they are used in the simplifier.

Enable Graph Case study In this section, we will apply the concept to a model of the extended GCD algorithm taken from [27]. The model consists of a refinement chain, where the last model consists of two loops. The first loop builds a stack of divisions. The second loop calculates the result from this stack. The last refinement level contains five events excluding the initialization. The events *up* and *dn* are the loop bodies, the events *upini*, *dnini* initialize the loops and *gcd* is the end of the computation. The event *init* is the *INITIALISATION* of the model.

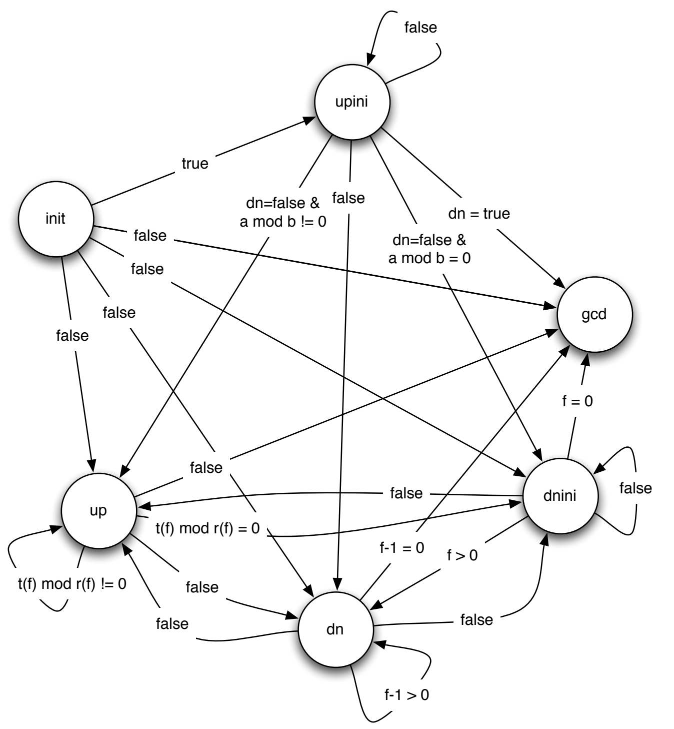

Fig. 16. Enable graph of the extended GCD example

The first step is to extract the read and write sets for each event; the result is shown in Table 10. Then we construct the enable graph. We calculate the weakest precondition for each pair of independent events and simplified them. Both steps were done manually but they were not very difficult. For instance, the most complicated weakest precondition was $[S_{up}]G_{dnini}$. In the presentation below we left out all parts of the guard and substitution that do not contain shared identifiers, e.g., the guard contains $up = TRUE$ but the substitution does not modify *up*. The next step is calculating the weakest precondition mechanically, finally we simplify the relational override using the rule $(r \mathbin{\lhd\!\!\!-} a \mapsto b)(a) = b$.

event	read(event)	write(event)
init	$\varnothing$	$\{a, b, d, u, v, up, f, s, t, q, r, uk, vk, dn, dk\}$
upini	$\{up\}$	$\{up, f, s, t, q, r\}$
up	$\{up, r, f, dn\}$	$\{f, s, t, r, q\}$
gcd	$\{up, f, dn\}$	$\{d, u, v\}$
dnini	$\{up, dn, r, f\}$	$\{dn, dk, uk, vk\}$
dn	$\{dn, f\}$	$\{uk, vk, f\}$

Table 10. Read and write sets

$$\begin{aligned}[S_{up}]G_{dnini} &= [f := f+1, r \lhd\!\!- \{f+1 \mapsto f(t) \bmod r(f)\}]\ (r(f) = 0)\\ &= (r \lhd\!\!- \{f+1 \mapsto t(f) \bmod r(f)\})(f+1) = 0\\ &= t(f) \bmod r(f) = 0\end{aligned}$$

The other simplification were much easier, for example, replacing $dn = TRUE \wedge dn = FALSE$ by $false$. The constructed graph is shown in figure 16.

The enable graph can be used by the model checker to reduce the number of guard evaluations. Let us examine one particular run of the algorithm for fixed input numbers. The run will start with *init* and *upini* then contain a certain number of *up* events, say n. This will be followed by *dnini* and then exactly n *dn* events and will finish with one *gcd* event. In all, the calculation takes $2n + 4$ steps. After each step, the model checker needs to evaluate 5 event guards (one for each event, except for the guard of the initialization which does not need to be evaluated) yielding $10n+20$ guard evaluations in total. Using the information of the enable graph we only need a total of $4n+4$ guard evaluations. For example, after observing *up*, we only need to check the guards of *up* and *dnini*: they are the only outgoing edges of *up* in Fig. 16 which are not labelled by $false$.

Flow Construction Beside the direct use in ProB the enable graph can be used to construct a flow. A flow is an abstraction of the model's state space where an abstract state represents a set of concrete states. Each abstract state is characterized by a set of events, representing all those concrete states where those (and only those) events are enabled.

The flow graph is a graph where the vertices are labeled with sets of events, i.e., the set of enabled events. The edges are labeled with an event and a predicate composed from the enable predicates for this event. The construction of the flow graph takes the enable graph as its input. Starting from the state where only the initialization event is enabled the algorithm unfolds the enable graph. We will describe the unfolding in a simple example, an algorithm is shown in Figure 18 and 19.

Figure 17 shows a simple flow graph construction. On the left side the enable graph for the events *init*, a and b are shown. The graph reveals that b always disables itself while it does not change the enabledness of a. The event a keeps itself enabled if and only if P holds and it enables b if and only if Q holds. The *init* event enables a and disables b.

We start the unfolding in the state labeled with $\{init\}$. In this case we do not have a choice but to execute *init*. From the enable graph on the left hand side we know that after *init* occurs a is the only enabled event. Therefor we have to execute a. We know that if P is true then a will be enabled afterwards and analogously if Q holds then b will be enabled. Combining all combination of P and Q and their negations, we get the new states $\{\}, \{b\}$ and $\{a, b\}$. If we continue, we finally get the graph shown on the right hand side. If more than one event is enabled, we add edges for each event separately. We can combine edges by disjunction of the predicates. In our case we did that for the transition from $\{a, b\}$ to $\{a\}$ which can be used by either executing b or a.

The algorithm in Figure 19 calculates for a given event e the successors in the flow graph by combining all possible configurations. The algorithm also uses a list of independent events that are enabled in the current state and therefor they are also enabled in any new state. The algorithm in Figure 18 produces the flow graph starting form the state $\{init\}$.

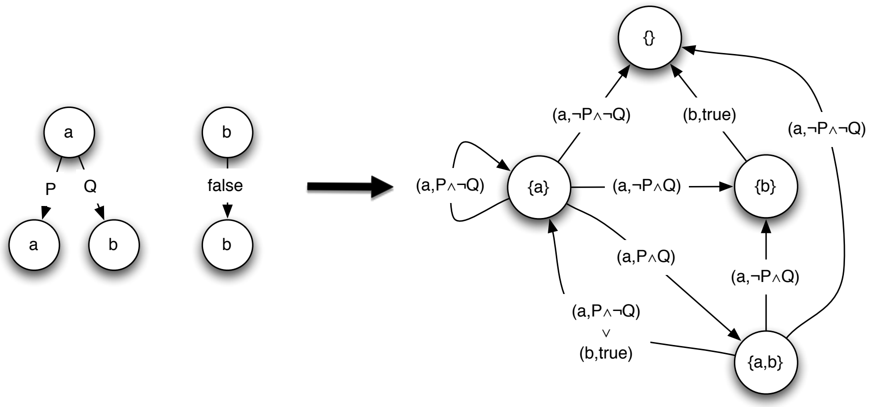

Fig. 17. Simple Flow Graph Construction

Generating the Flow Graph can be infeasible because the graph can be of size $O(2^{\#Events})$. However, in cases where constructing the flow graph is feasible, we gain a lot of information about the algorithmic structure and we can generate code if the model is deterministic enough. We will discuss applicability and restrictions of the methods in section 6.2.

Flow Graph Case Study If we apply the flow construction to the example graph shown in figure 16 we get the flow graph shown in figure 21. Compared

```
$todo := \{\{init\}\}$
$done := \varnothing$
$flow := \varnothing$
**while** $todo \neq emptyset$ **do**
  choose $node$ from $todo$
  **foreach** $e \in node$ **do**
    $keep := node \cap independent(e)$
    $atoms := expand(e, keep)$
    $todo := (todo \cup ran(atoms))$
    $flow := flow \cup \{node \mapsto atoms\}$
  **od**
  $done := done \cup \{node\}$
  $todo := todo - done$
**od**
```

Fig. 18. Algorithm for constructing a Flow Graph

```
Given: enable graph as $EG : (Events \times Events) \nrightarrow Predicate$
**def** $expand(e, keep) =$
  $true\_pred := \{f \mapsto true | (e \mapsto f) \in dom(EG) \wedge EG(e \mapsto f) = true\}$
  $maybe\_pred := \{f \mapsto p | (e \mapsto f) \in dom(EG) \wedge EG(e \mapsto f) = p \wedge p \neq false\}$
  $result := \varnothing$
    **foreach** $s \subseteq node$ **do**
      $targets := dom(true\_pred) \cup dom(s) \cup keep$
      $predicate := \bigwedge ran(s) \wedge \neg(\bigvee ran(s \lhd maybe\_pred)$
      $result := result \cup \{predicate \mapsto targets\}$
    **od**
 **return** $result$
**end def**
```

Fig. 19. Algorithm for expanding the Enable Graph (i.e., computing successor configurations)

to the structured model developed by Hallerstede in [27] shown in 20 we see a very similar shape.

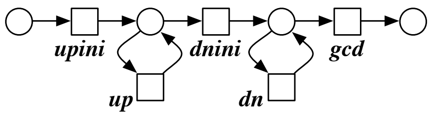

Fig. 20. Structural model from [27]

However, the automatic flow analysis helped us to discover an interesting property. The flow graph contains a state that corresponds to concrete states where no event is enabled, i.e., states where the system deadlocks. Thus the model contains a potential deadlock. Inspection showed that the deadlock actually does not occur. The reason why the flow graph contains the deadlock state is a guard that is too strong. The guards of *dn* and *gcd* only cover $f \geq 0$. The invariant implicitly prevents the system from deadlocking by restricting the values of f.

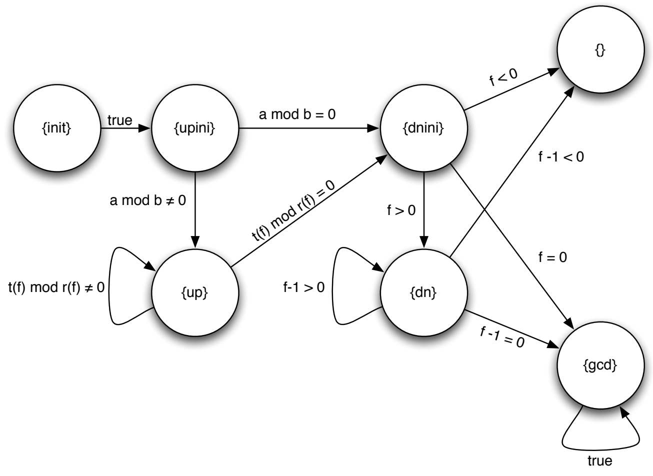

Fig. 21. Example for a relation between abstract and concrete states

In Figure 21 we can see that is is possible to automatically generate sequential code from a flow graph. The events *up* and *dn* can be translated into while loops and *upini* and *dnini* are $if - then - else$ statements. In the particular case the termination of the computation was encoded into the *gcd* event.

Applicability and Restrictions An important question is when to apply a method and maybe even more important when not to apply it. It is clear that flow analysis is probably not applicable if the model does not contain an algorithmic structure. In the worst case for flow construction, any combination of events can be enabled in some state, leading to $2^{card(Events)}$ states, where $card(Events)$ is the number of events. However in case of software developments it is very likely that eventually the model will contain events that are clustered, i.e, at each point during the computation a hopefully small set of events is enabled. We conjecture that the more concrete a model is, the better are results from simplification.

Constructing the enable graph is relatively efficient; it requires to calculate $O(card(Events)^2)$ enabling predicates. In case of software specifications generating the enable graph and using the information gained for guard reduction is probably worth trying. We can also influence the graph interactively. For instance, suppose the enable graph contains an edge labeled with $card(x) > 0$. Suppose we know that after the first event $x = \varnothing$ but the simplifier was too weak to figure it out, i.e., the empty set is written down in a difficult way, let's say $x := S \cap T$ where S and T are disjoint. By specifying (and proving) a theorem that helps the simplifier, e.g., $x = \varnothing$, we can interactively improve the graph. We believe that expressing these theorems does not only improve the graph but also our understanding of a model because we explicitly formalize properties of the model that are not obvious (at least not for the automatic simplifier).

Constructing the flow graph is much more fragile; it can blow up very fast. It is crucial to inspect the enable graph and try to reduce the size of the predicates as much as possible. However our experience is that Event-B models of software at a sufficient low level of refinement typically have some notion of an abstract program counter that implicitly control the flow in a model. These abstract program counter are not very complicated and therefor it is likely that they are exploited by the simplifier.

Related and Future Work

Inferring Flow Information Model checking itself explores the state space of a model, and as such infers very fine-grained flow information. For Event-B, the ProB model checker [36, 38] can be used for that purpose. However, it is quite rare that the complete state space of a model can be explored. When it is possible, the state space can be very large and flow information difficult to extract. Still, the work in [40] provides various algorithms to visualize the state space in a condensed form. The signature merge algorithm from [40] merges all states with the same signature, and as such will produce a picture very similar to the flow graph. However, the arcs are not labelled by predicates and the construction requires prior traversal of the state space.

Specifying Flow Information There is quite a lot of related work, where flow information is provided explicitly by the modeler (rather than being deduced automatically, as in our method). For example, several works use CSP to specify the sequencing of operations of B machines [55, 14, 16] or of Z specifications [22, 42, 53, 8].

In the context of Event-B, there are mainly three other approaches that are related to our flow analysis. Hallerstede introduced in [27] a new approach to support refinement in Event-B that contains information about the structure of a component. Also Butler showed in [15] how structural information can be kept during refinement of a component. Both approaches have the advantage to incorporate the information about structure into the method, yielding in better precision. However both methods require the developer to use the methods from the beginning while automatic flow analysis can be applied to existing projects. In particular automatic flow analysis can actually be used to discover properties of a model such as liveness and feasibility of events. Hallerstede's structural refinement approach does not fully replace our automatic flow analysis. Both methods overlap to some extend but we think that they can can be combined, such that the automatic flow analysis uses structural information to ease the generation of the flow graph and in return our method can suggest candidates for the intermediate predicates used during structural refinement. Actually the enable predicated can be used as candidates.

The third approach is yet unpublished but implemented as a plug-in for Rodin [32]. It allows the developer to express flow properties for a model and to verify them using proofs.

Future Work The next step is to fully integrate our method into the next release of ProB, and use it to improve the model checking procedure and help the user in analyzing or comprehending models. We also plan to use the technique to develop a new algorithm for test-case generation. In [56] we have introduce a first test-case generation algorithm for Event-B, tailored towards event coverage. One issue is that quite often it is very difficult to cover certain events. Here the flow analysis will hopefully help guide the model checker towards enabling those difficult events.

Conclusion In summary, we have developed techniques to infer algorithmic structure from a formal specification. From an Event-B model, we have derived the enable graph, which contains information about independence and dependence of events. This graph can be used for model comprehension and to improve model checking. We have described a more sophisticated flow analysis, which derives a flow graph from an Event-B model. It can again be used for model comprehension, model checking but also for code generation.

7 Workpackage I3: Parallel and Directed ProB

The techniques mentioned in Section 3.4 and Section 6.1 have been integrated into the core ProB codebase and are now available in the latest distributions. The heuristic model checking algorithm has been integrated into the ProB codebase, but is currently not available in the compiled versions (due to issues generating a shared library version for the C++ priority queue under Windows). The implementation of the flow analysis from Section 6.2 is still ongoing. Below, we describe an empirical evaluation of the proof directed model checking technique from Section 6.1.

7.1 Implementation and Evaluation of Proof-Directed MC

To show, that our approach is indeed correct, we developed a formal model of the Algorithm used in ProB. The model was developed in Event-B and fully proven in Rodin. We have shown that the proof supported model checking algorithm is a refinement of the original algorithm. Thus the new algorithm accepts a state if and only if the old one does.

7.2 Experimental results

To verify that the combination of proving and model checking results in a considerable reduction of model checking effort, we prepared an experiment consisting of specifications we got from academia and industry. The specification are taken from the benchmarks chosen in T1. In addition we prepared a constructed example as one case, where the prover has a very high impact on the performance of the model checker. It basically contains an event, that increments a number x and an invariant $\forall a, b, c.a \in \mathbb{N} \land b \in \mathbb{N} \land c \in \mathbb{N} \Rightarrow (a = a \land b = b \land c = c \land x = x)$. Because the invariant contains the variable modified by the event, we cannot simply remove it. But Rodin can automatically prove that the event preserves the invariant, thus our tool is able to remove the whole invariant. Without proof information, ProB needs to enumerate all possible values for a,b and c which results in an expensive calculation.

Except for the case of the Siemens specification, we removed all interactive proofs from the models and used only those proof information, that Rodin was able to automatically generate using default settings. In the case of the Siemens model, we used both, a version with automatic proofs only and a development version with few additional interactive proofs; the development version was not fully proven.

The results have been gathered using a Mac Book Pro, 2.4 GHz Intel Core 2 Duo Computer with 4 GB RAM running Mac OS X 10.5. We collected 5 samples for each model and calculated the average and standard deviation of the times measured in milliseconds.. The result of the experiment is shown in tables 11 and 12. The proof assisted model checking algorithm requires to pre-calculate specialized invariants for each event. We tried to measure the time required for this calculation but we could not find a case where the pre-calculation had significant impact on the total time so it can be neglected.

We believe that proof information can be used as a heuristic to direct the model checker in order minimize its work when verifying a model or in order to optimize the search when finding an invariant violation. If we think that our model is correct we use the observation that if we have multiple events leading to a state, the invariant we only need to check is the intersection of the sets unproven invariants of these events. Instead of ProB's mixed depth and breadth first search, we can do a best fist search based on the total number of remaining invariants. This approach does not reduce the number of states but can significantly reduce the number of invariant evaluations.

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

Table 11. Experimental results (multiple refinement level check)

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

Table 12. Number of invariants evaluated (single refinement level check).

References

1. J.-R. Abrial. *The B-Book*. Cambridge University Press, 1996.
2. J.-R. Abrial. Case study of a complete reactive system in Event-B: A mechanical press controller. In *Tutorial at ZB'2005*, 2005. Available at `http://www.zb2005.org/`.
3. J.-R. Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, 2010.
4. J.-R. Abrial and D. Cansell. Formal construction of a non-blocking concurrent queue algorithm (a case study in atomicity). *J. UCS*, 11(5):744–770, 2005.
5. R. Alur, T. A. Henzinger, F. Y. C. Mang, S. Qadeer, S. K. Rajamani, and S. Tasiran. Mocha: Modularity in model checking. In A. J. Hu and M. Y. Vardi, editors, *CAV*, volume 1427 of *Lecture Notes in Computer Science*, pages 521–525. Springer, 1998.
6. Y. A. Ameur, F. Boniol, and V. Wiels, editors. *ISoLA 2007, Workshop On Leveraging Applications of Formal Methods, Verification and Validation, Poitiers-Futuroscope, France, December 12-14, 2007*, volume RNTI-SM-1 of *Revue des Nouvelles Technologies de l'Information*. Cépaduès-Éditions, 2007.
7. K. Arkoudas, S. Khurshid, D. Marinov, and M. C. Rinard. Integrating model checking and theorem proving for relational reasoning. In R. Berghammer, B. Möller, and G. Struth, editors, *RelMiCS*, volume 3051 of *Lecture Notes in Computer Science*, pages 21–33. Springer, 2003.
8. D. A. Basin, E.-R. Olderog, and P. E. Sevinç. Specifying and analyzing security automata using csp-oz. In F. Bao and S. Miller, editors, *ASIACCS*, pages 70–81. ACM, 2007.
9. M. Ben-Ari. *Principles of the Spin Model Checker*. Springer, 2008.
10. J. Bendisposto, M. Jastram, M. Leuschel, C. Lochert, B. Scheuermann, and I. Weigelt. Validating Wireless Congestion Control and Realiability Protocols using ProB and Rodin. *FMWS 2008: Workshop on Formal Methods for Wireless Systems*, Aug. 2008.
11. J. Bendisposto, M. Leuschel, O. Ligot, and M. Samia. La validation de modèles Event-B avec le plug-in ProB pour RODIN. *Technique et Science Informatiques*, 27(8):1065–1084, 2008.
12. D. Bert, M.-L. Potet, and N. Stouls. Genesyst: A tool to reason about behavioral aspects of B event specifications. application to security properties. In *ZB 2005*, pages 299–318, 2005.
13. J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, Jun 1992.
14. M. Butler. csp2B: A practical approach to combining CSP and B. *Formal Aspects of Computing*, 12:182–198, 2000.
15. M. Butler. Decomposition structures for event-b. In M. Leuschel and H. Wehrheim, editors, *IFM*, volume 5423 of *Lecture Notes in Computer Science*. Springer, 2009.
16. M. Butler and M. Leuschel. Combining CSP and B for specification and property verification. In *Proceedings of Formal Methods 2005*, LNCS 3582, pages 221–236, Newcastle upon Tyne, 2005. Springer-Verlag.
17. M. Butler and D. Yadav. An incremental development of the Mondex system in Event-B. *Formal Aspects of Computing*, 20(1):61–77, 2008.

18. D. Cansell, S. Hallerstede, and I. Oliver. UML-B specification and hardware implementation of a hamming coder/decoder. In J. Mermet, editor, *UML-B Specification for Proven Embedded Systems Design*. Kluwer Academic Publishers, Nov 2004. Chapter 16.
19. E. J. Chang. Echo algorithms: Depth parallel operations on general graphs. *IEEE Transactions on Software Engineering*, 8(4), 1982.
20. D. Dams, D. Hutter, and N. Sidorova. Using the inka prover to automate safety proofs in abstract interpretation - a case study. In F. Bellegarde and O. Kouchnarenko, editors, *Workshop on Modelling and Verification*, C.I.S., Besançon, France, 1999. Alternative title: Combining Theorem Proving and Model Checking - A Case Study.
21. P. Dybjer, Q. Haiyan, and M. Takeyama. Verifying haskell programs by combining testing, model checking and interactive theorem proving. *Information & Software Technology*, 46(15):1011–1025, 2004.
22. C. Fischer. Combining object-z and csp. In A. Wolisz, I. Schieferdecker, and A. Rennoch, editors, *FBT*, volume 315 of *GMD-Studien*, pages 119–128. GMD-Forschungszentrum Informationstechnik GmbH, 1997.
23. S. Flannery. *In Code: A Mathematical Adventure*. Profile Books Ltd, 2001.
24. P. J. Fleming and J. J. Wallace. How not to lie with statistics: the correct way to summarize benchmark results. *Commun. ACM*, 29(3):218–221, 1986.
25. Formal Systems (Europe) Ltd. *Failures-Divergence Refinement — FDR2 User Manual (version 2.8.2)*.
26. E. L. Gunter and D. Peled. Model checking, testing and verification working together. *Formal Asp. Comput.*, 17(2):201–221, 2005.
27. S. Hallerstede. Structured Event-B Models and Proofs. In *ABZ 2010*, LNCS. Springer-Verlag, 2010.
28. G. J. Holzmann. The model checker Spin. *IEEE Trans. Software Eng.*, 23(5):279–295, 1997.
29. G. J. Holzmann. An analysis of bitstate hashing. *Formal Methods in System Design*, 13(3):289–307, 1998.
30. G. J. Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, 2004.
31. G. J. Holzmann and D. Peled. An improvement in formal verification. In D. Hogrefe and S. Leue, editors, *FORTE*, volume 6 of *IFIP Conference Proceedings*, pages 197–211. Chapman & Hall, 1994.
32. A. Iliasov. Flows Plug-In for Rodin. `http://wiki.event-b.org/index.php/Flows#Flows_plugin`.
33. D. Jackson. Alloy: A lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology*, 11:256–290, 2002.
34. B. Legeard, F. Peureux, and M. Utting. Automated boundary testing from Z and B. In *Proceedings FME'02*, LNCS 2391, pages 21–40. Springer-Verlag, 2002.
35. M. Leuschel. The high road to formal validation. In E. Börger, M. Butler, J. P. Bowen, and P. Boca, editors, *ABZ*, volume 5238 of *Lecture Notes in Computer Science*, pages 4–23. Springer, 2008.
36. M. Leuschel and M. Butler. ProB: A model checker for B. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *FME 2003: Formal Methods*, LNCS 2805, pages 855–874. Springer-Verlag, 2003.
37. M. Leuschel and M. Butler. Automatic refinement checking for B. In K.-K. Lau and R. Banach, editors, *Proceedings ICFEM'05*, LNCS 3785, pages 345–359. Springer-Verlag, 2005.

38. M. Leuschel and M. J. Butler. ProB: an automated analysis toolset for the B method. *STTT*, 10(2):185–203, 2008.
39. M. Leuschel and D. Plagge. Seven at a stroke: LTL model checking for high-level specifications in B, Z, CSP, and more. In Ameur et al. [6], pages 73–84.
40. M. Leuschel and E. Turner. Visualizing larger states spaces in ProB. In H. Treharne, S. King, M. Henson, and S. Schneider, editors, *Proceedings ZB'2005*, LNCS 3455, pages 6–23. Springer-Verlag, April 2005.
41. G. Lowe. An attack on the Needham-Schroeder public key authentication protocol. *Information Processing Letters*, 56(3):131–136, November 1995.
42. B. P. Mahony and J. S. Dong. Blending object-z and timed csp: An introduction to tcoz. In *ICSE*, pages 95–104, 1998.
43. K. L. McMillan. *Symbolic Model Checking*. PhD thesis, Boston, 1993.
44. R. Needham and M. Schroeder. Using encryption for authentication in large networks of computers. 21(12):993–999, Dezember 1978.
45. I. Oliver. Experiences in using b and uml in industrial development. In J. Julliand and O. Kouchnarenko, editors, *B'2007*, volume 4355 of *Lecture Notes in Computer Science*, pages 248–251. Springer, 2007.
46. D. Peled. Combining partial order reductions with on-the-fly model-checking. In D. L. Dill, editor, *CAV*, LNCS 818, pages 377–390. Springer, 1994.
47. G. Peterson. Myths about the mutual exclusion problem. *Inf. Process. Lett.*, 12(3):115–116, 1981.
48. J. Rehm and D. Cansell. Proved development of the real-time properties of the ieee 1394 root contention protocol with the event b method. In Ameur et al. [6], pages 179–190.
49. M. Samia, H. Wiegard, J. Bendisposto, and M. Leuschel. High-Level versus Low-Level Specifications: Comparing B with Promela and ProB with Spin. In Attiogbe and Mery, editors, *Proceedings TFM-B 2009*, pages –. APCB, June 2009.
50. B. Scheuermann, C. Lochert, and M. Mauve. Implicit hop-by-hop congestion control in wireless multihop networks. *Ad Hoc Networks*, 2007. doi: 10.1016/j.adhoc.2007.01.001.
51. S. Schneider. *The B-Method: An introduction*. Palgrave Macmillan, 2001.
52. N. Shankar. Combining theorem proving and model checking through symbolic analysis. In C. Palamidessi, editor, *CONCUR*, LNCS 1877, pages 1–16. Springer, 2000.
53. G. Smith and J. Derrick. Specification, refinement and verification of concurrent systems-an integration of object-z and csp. *Formal Methods in System Design*, 18(3):249–284, 2001.
54. F. Steria, Aix-en-Provence. *Atelier B, User and Reference Manuals*, 1996. Available at `http://www.atelierb.societe.com`.
55. H. Treharne and S. Schneider. How to drive a B machine. In J. P. Bowen, S. Dunne, A. Galloway, and S. King, editors, *ZB'2000*, LNCS 1878, pages 188–208. Springer, 2000.
56. S. Wieczorek, V. Kozyura, A. Roth, M. Leuschel, J. Bendisposto, D. Plagge, and I. Schieferdecker. Applying Model Checking to Generate Model-based Integration Tests from Choreography Models. In *Proceedings TESTCOM/FATES 2009*, page to appear. Springer-Verlag, 2009.
57. H. Wiegard. A comparison of the model checker ProB with Spin. Master's thesis, Institut für Informatik, Universität Düsseldorf, 2008. Bachelor's Thesis.