Visualising Larger State Spaces in ProB

Michael Leuschel[1,2] and Edd Turner[1]

[1] Department of Electronics and Computer Science
University of Southampton
Highfield, Southampton, SO17 1BJ, UK
{mal,ent03r}@ecs.soton.ac.uk
[2] Institut für Informatik, Heinrich-Heine Universität Düsseldorf

Abstract. ProB is an animator and model checker for the B method. It also allows to visualise the state space of a B machine in graphical way. This is often very useful and allows users to quickly spot whether the machine behaves as expected. However, for larger state spaces the visualisation quickly becomes difficult to grasp by users (and the computation of the graph layout takes considerable time). In this paper we present two relatively simple algorithms to often considerably reduce the complexity of the graphs, while still keeping relevant information. This makes it possible to visualise much larger state spaces and gives the user immediate feedback about the overall behaviour of a machine. The algorithms have been implemented within the ProB toolset and we highlight their potential on several examples. We also conduct a thorough experimentation of the algorithm on 47 B machines and analyse the results.
Keywords: Formal Methods, B-Method, Tool Support, Model Checking, Animation, Visualisation, Logic Programming.

1 Introduction

The B-method, originally devised by J.-R. Abrial [1], is a theory and methodology for formal development of computer systems. It is used by industries in a range of critical domains, most notably railway control. B is based on the notion of *abstract machine* and the notion of *refinement*. The variables of an abstract machine are typed using set theoretic constructs such as sets, relations and functions. The invariant of a machine is specified using predicate logic. Operations of a machine are specified as *generalised substitutions*, which allow deterministic and nondeterministic assignments to be specified. There are two main proof activities in B: *consistency checking*, which is used to show that the operations of a machine preserve the invariant, and *refinement checking*, which is used to show that one machine is a valid refinement of another. These activities are supported by industrial strength tools, such as Atelier-B [24] and the B-toolkit [5].

ProB [18] is an animation and model checking tool for the B method. ProB's animation facilities allow users to gain confidence in their specifications, and unlike the animator provided by the B-Toolkit, the user does not have to guess the right values for the operation arguments or choice variables. The undecidability of animating B is overcome by restricting animation to finite sets and integer

ranges, while efficiency is achieved by delaying the enumeration of variables as long as possible. ProB also contains a model checker [9] and a constraint-based checker, both of which can be used to detect various errors in B specifications.

ProB shows the user a graphical view of the state space the model checker has already explored. For this ProB makes use of the Dot tool of the graphviz package [4]. This feedback is very beneficial to the understanding of the specification since human perception is good at identifying structural similarities and symmetries [10]. Such a feature works well for small state spaces, but in practice specifications under analysis often consume thousands of states, which severely limits the usefulness of the graph.

Take the following example machine (distributed with ProB).

```
MACHINE phonebook
SETS  Name  ; Code = {c1,c2,c3}
VARIABLES  db
DEFINITIONS   scope_Name == 1..3
INVARIANT
      db : Name +-> Code
INITIALISATION
      db := {}
OPERATIONS
  cc <-- lookup(nn) =  PRE nn : Name & nn : dom(db) THEN
               cc:=db(nn) END;
 add(nn,cc) = PRE nn:Name & cc:Code & nn /: dom(db) THEN
              db := db \/ { nn |-> cc} END ;
 delete(nn,cc) = PRE nn:Name & cc:Code & nn: dom(db) &
                                     cc: ran(db) & db(nn) = cc  THEN
              db := db - { nn |-> cc}  END
END
```

The full state space of this example (with `Name` set to cardinality 3) has 65 states and 433 transitions. As can be seen, the visualization of the state space in ProB is possible (depicted in Fig. 1; the reader is not expected to be able to read the labels, just get a general impression of the visualization) but is quite difficult to grasp by humans and certain "obvious" aspects of the state space are not be easy to identify in the visualization. For example, it is not obvious to spot what the actual enabled operations are or that one can do at most three consecutive calls to the `add` operation.

The question is whether the state space of B machines can be rendered in ways more suitable for human understanding. It turns out that there are surprisingly few tools and techniques that addressed this problem in general or for B in particular. In this paper we thus present various (complimentary) techniques to improve the visualisation of larger state spaces. These techniques have been implemented in the ProB toolset and we illustrate the performance of them on various examples. We also empirically evaluate the techniques on a large number of examples and show that the techniques can be surprisingly effective.

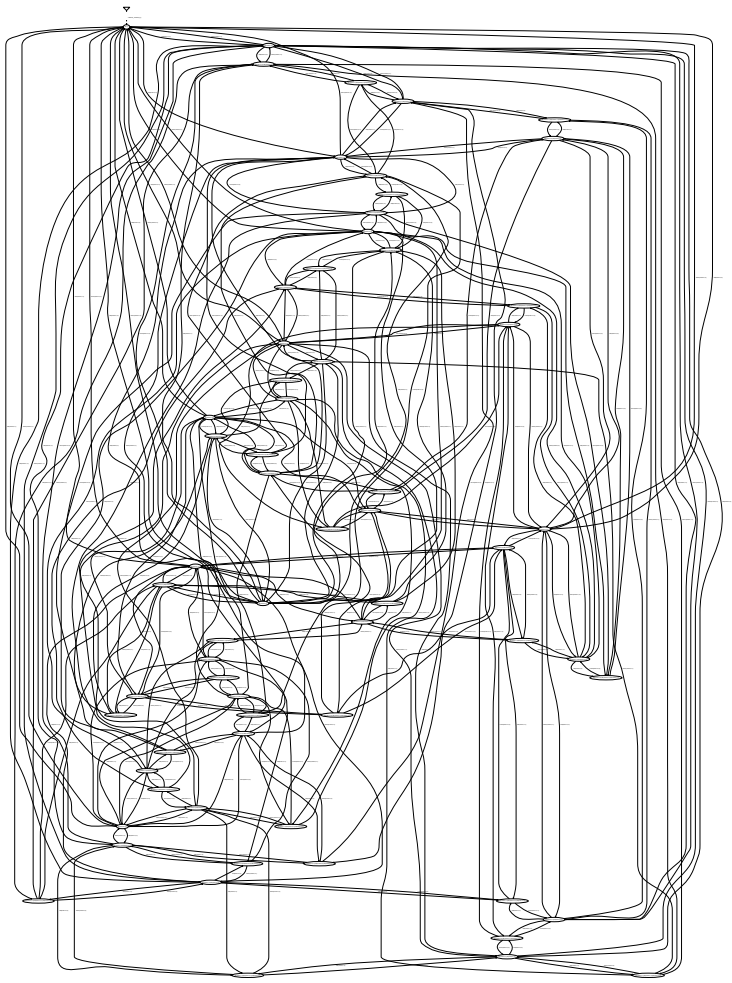

Fig. 1. Phonebook machine - Original State Space.

2 The DFA-Abstraction Algorithm

The state space generated by ProB can be viewed as non-deterministic labelled transition system (LTS), where the edges are labelled with terms of the form $op(a_1, \ldots, a_n)$ and $op(a_1, \ldots, a_n) \rightarrow r_1, \ldots, r_k$, where op is the name of the operation that has been applied and $a_1, \ldots, a_n$ are the arguments of the operation. The first form is used for operations that do not return values, whereas the sec-

ond form is used for operations that do and where $r_1, \ldots, r_k$ are the returned values.

Formally, an LTS is a 4-tuple (Q, Σ, q_0, δ) where Q is the set of states, Σ the alphabet for labelling the transitions, q_0 the initial state and $\delta \subseteq Q \times \Sigma \times Q$ is the transition relation. By $q \rightarrow_a q'$ we denote that $(q, a, q') \in \delta$. As usual, we extend this to sequences of transitions so that $q \rightarrow_{a_1,\ldots,a_k} q'$ denotes the fact that there exists a sequence of states $q_0, \ldots, q_k$ such that $q_0 = q$, $q_k = q'$ and $q_i \rightarrow_{a_i} q_{i+1}$. The set of *reachable states* of an automaton is defined to be the set $\{q \in Q \mid q_0 \rightarrow_\gamma q \text{ for some sequence of states } \gamma\}$. Finally, the *traces* of an automaton L is the set of sequences $traces(L) = \{\gamma \in \Sigma^* \mid q_0 \rightarrow_\gamma q \text{ for some } q \in Q\}$.

One way to reduce the complexity of an LTS is to abstract away from certain details of the labelling function. For example, the user may not be interested in seeing (all) the arguments of (all) the operations. To this end we now define abstraction functions and a way to apply them to construct simplified LTS.

Definition 1. *An* abstraction function *for an LTS (Q, Σ, q_0, δ) is a function α from Σ to some new alphabet Σ'.*

The α-abstraction of the LTS is then defined to be a new LTS $(Q, \Sigma', q_0, \delta')$ where $\delta' = \{(q, \alpha(a), q') \mid (q, a, q') \in \delta\}$.

For the experiments later in the paper we have used $\alpha(op(a_1, \ldots, a_n)) = op/n$ for operations without return values and $\alpha(op(a_1, \ldots, a_n) \rightarrow r_1, \ldots, r_k) = op/n \rightarrow k$ for operations that return values, but any other abstraction (or even the identity function) can be used instead.[1] This encodes a common perspective where the user is interested in seeing which operations can be applied, but is not interested in the actual arguments.

Now, the α-abstraction on its own is not yet very useful, as we have not yet diminished the number of states (even though we may have reduced the number of transitions). The first thing that comes to mind in that respect is the classical minimization algorithm for Deterministic Finite Automaton (DFA) [2, 14]. Indeed, a finite LTS can be viewed as a Non-Deterministic Finite Automaton (NFA) simply by marking all states as final states (basically the only difference between an NFA and an LTS is the notion of final states). We can then convert this NFA into a DFA using another classical algorithm [2, 14], to then apply the minimization algorithm. This is exactly what we have done in our first so-called *DFA-Abstraction* Algorithm, which we have implemented and integrated into the ProB toolset. In summary, the *DFA-Abstraction* Algorithm computes

- the α-Abstraction of an LTS
- then determinizes the resulting intermediate LTS by converting sets of reachable states of the NFA into single states of the DFA,

[1] We have decided to show the number of arguments n and the number of return values k in our abstracted graphs. This is largely a matter of taste and $\alpha(op(a_1, \ldots, a_n) \rightarrow r_1, \ldots, r_k) = op$ could have been used instead (as one is not allowed to have two different operations with the same name anyway).

– before minimizing it by computing maximal equivalence classes of the DFA, yielding the result LTS.

The algorithm is shown on a small example in Fig. 2.

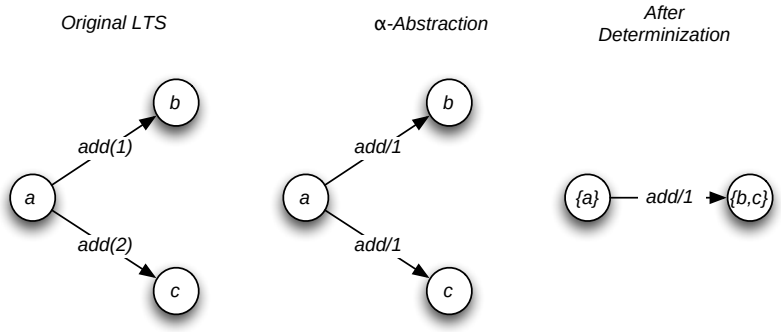

Fig. 2. Illustrating the DFA-Abstraction Algorithm.

This algorithm was was primarily applied as a control: something to which other algorithms could be compared. We were also aware that it had the potential to collapse symmetrical subgraphs. It hence turns out to be very useful in some cases, while in other cases increasing the size of the graph (as is well-known, the NFA to DFA conversion can lead to an exponential blow-up, even though this is rarely observed in practice).

How to read DFA-Abstracted graphs Every node in the graph corresponds to a set of states of the animated B machine. Obviously, we lose information from the α-abstraction, i.e., we loose the operation arguments. The DFA conversion and minimization algorithms preserve the set of traces that can be performed. However, because of determinization, multiple B states are put together into a single node. Hence, if a node in the DFA-abstracted graph has an outgoing edge marked with op/n this does not guarantee that the operation can be applied in *all* B states covered by this node. Thus, to make the graphs more informative, our LTS visualization algorithm checks whether an outgoing edge can be performed in all covered B states: if it does the edge is drawn solid, otherwise the edge is dashed.

In Fig. 3 you can see the effect of our algorithm on the full state space of Fig. 1. The reduction is considerable, and the graph can now be easily digested by a human. Furthermore, even though we have lost the operation arguments, the reduced graph still contains a lot of useful information (especially since all edges are solid). For example, one can see that it is only possible to add three entries into the phonebook. It is also clear that one can only perform a delete or

lookup after something has been added to the phonebook. One can also deduce that add and delete are state changing operations.

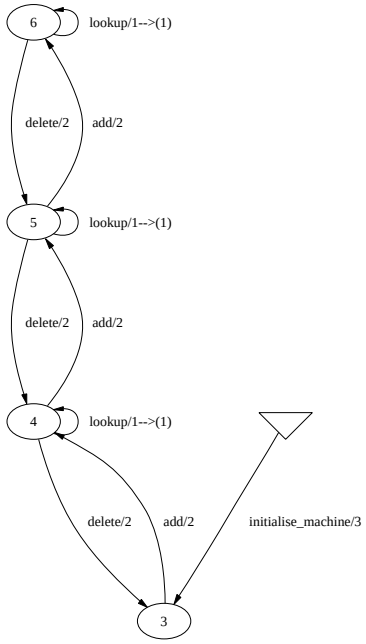

Fig. 3. Phonebook machine - DFA.

An alternative approach to using a DFA-Abstraction would be to minimize the NFA – which is attractive considering that it is possible for an NFA to be exponentially smaller in size when compared to an equivalent DFA. However, the problem of minimizing NFAs is computationally intractable [16, 20], and we have hence decided not to go down this route.

Another approach, documented in [15], attempts to reduce, and not minimize, the size of an NFA while retaining language equivalence. Our experiments so far have shown that the reductions gained are not as useful as our DFA-Abstraction or Signature Merge approach described in the next section.

3 Merge States With Same Outgoing Transitions

This technique was devised after studying a collection of graphs produced by ProB. It works by merging all states with the same enabled operations and so

it may produce an automaton that is not equivalent (as far as the traces of possible operations are concerned) to the original one. However, the technique can achieve a big reduction in the size of the automaton while still preserving the information about which B operations are enabled in a particular state (i.e. the traces of length 1). To do this, we first introduce the concept of a signature of a state, which represents the operations (i.e., transition labels) that can be performed in that state.

Definition 2. *Let (Q, Σ, q_0, δ) be an LTS. We define the signature of a node $q \in Q$, denoted by $signature(q)$, as follows: $signature(q) = \{a \mid q \rightarrow_a q' \text{ for some } q' \in Q\}$.*

If $signature(q) = \emptyset$ then we say that q is *deadlocked*. An automaton is said to *deadlock* iff there is a reachable state that is deadlocked. If $a \in signature(q)$ then we say that a is *enabled* in q. An automaton is said to be *quasi-live for transition a* iff there exists a reachable state where a is enabled.

Definition 3. *Let (Q, Σ, q_0, δ) be an LTS. The* Signature-Merge *of the LTS is defined to be a new LTS $(Q^s, \Sigma, q_0^s, \delta')$ where $Q^s = \{signature(q) \mid q \in Q\}$, $q_0^s = signature(q_0)$, and $\delta^s = \{(signature(q), a, signature(q')) \mid (q, a, q') \in \delta\}$.*

Basically, the effect of a signature-merge is to merge all states which have a common signature. This ensures that at least for traces of length 1 we do not lose any precision. There are a few more properties that are preserved by the Signature-Merge:

Proposition 1. *3 Let $L = (Q, \Sigma, q_0, \delta)$ be an LTS and L_S its Signature-Merge. Then L deadlocks iff L_S deadlocks. Also, for any $a \in \Sigma$, L is quasi-live for a iff L_S is quasi-live for a. Finally, $traces(L) \subseteq traces(L_S)$.*

The last property means that if a certain sequence is not possible in the Signature-Merge then it cannot be performed in the original LTS either.

The overall algorithm we have now implemented is to first compute the α-abstraction of an LTS and then perform the Signature-Merge on the abstracted LTS.

How to read Signature-Merge graphs As with the DFA-Abstracted graphs, every node in the graph corresponds to a set of states of the animated B machine. However, if a node has an outgoing edge marked with op/n we are not sure that this particular edge can be taken in *all* B states covered by this node: we only know that there is at least one covered state where this edge can be followed. Hence, contrary to the DFA-conversion and minimization, signature merging does not preserve the set of possible traces. However, all the states associated with the node have the same signature: so we at least know that the operation *op* is possible in *all* B states covered by the node. We can also apply Proposition to deduce information about deadlocks and about traces that are not possible in the original machine.

To facilitate the interpretation of the Signature-Merge graphs, we will actually differentiate the edges according to whether an edge is definitely possible in all states that have been merged together. Such edges are called *definite*, as formally defined below, and will be drawn as solid lines, while edges which are not definite will be drawn as dashed lines. This gives the user clear visual feedback and allows to infer more properties about the underlying B machine.

Definition 4. *Let* (Q, Σ, q_0, δ) *be an LTS and* $(Q^s, \Sigma, q_0^s, \delta')$ *its Signature-Merge. A transition* $(signature(q), a, signature(q')) \in \delta'$ *is called* definite *iff* $\forall p \in Q$ *such that* $signature(p) = signature(q)$*:* $\exists p'$ *with* $signature(p') = signature(q')$ *and* $(p, a, p') \in \delta$*. In other words, for all other nodes* p *that have been merged together with* q*, we can also perform the transition* a *leading to the same state (in the Signature-Merge graph).*

In Fig. 4 you can see the effect of this algorithm on the full state space of Fig. 1. The reduction is considerable, and the graph can now be easily digested by a human. Note that the reduced graph will not change even if we allow more entries to be added into the phonebook (e.g., by changing the cardinality of the set `Name`). So, in principle, one could even visualise the machine for an unbounded set `Name`. This is not the case for the DFA (where if we allow 100 entries the DFA will have 100 nodes). However, some of the precision of the DFA visualization is lost: we can no longer spot how many entries can be added; all we can see is that we can add at least two entries, but not exactly how many. Still, the signature based approach has managed to keep relevant information. For example, it is still obvious from the graph that we can only lookup or delete entries after adding an entry, and we can see that it is possible to reach a state where it is no longer possible to add entries.

Extending the Algorithm We can make the algorithm more precise by diminishing the α-abstraction, e.g., by not abstracting away certain arguments. This could be guided by the user and also applies to the DFA-Abstraction algorithm. Second, the signature of a node basically corresponds to all the traces of length 1 that can be performed from that node. We could thus extend the notion of a signature and compare all the traces of length 2,3,....[2]

On the other hand we can make the algorithm less precise and achieve more reduction in several ways. First, one could make α more aggressive, e.g., by mapping several operations together (e.g., maybe the user is not interested in some of the operations). Second, instead of merging nodes if they have exactly the same signature, we could merge them if the signatures are sufficiently close (e.g., they are the same except for one element or we only look at the signature as far as a certain number of operations of interest is concerned).

In practice it may be good to combine both approaches: e.g. the user could type a certain number as a target for the ideal number of nodes (say 20) and then the graph is progressively made less or more precise to approach that number.

[2] In the limit we obtain the classical equivalence preserving minimization algorithm.

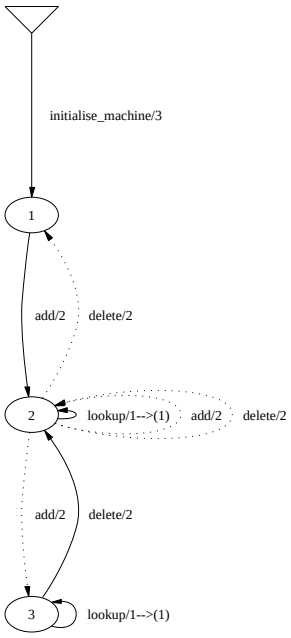

Fig. 4. Phonebook machine - Signature Merge.

4 Two more complicated examples

Figures 5, 6, and 7 show the behaviour of our algorithms for the "scheduler" example taken from [7, 3]. Again, both algorithms perform very well, providing clear graphs about the overall behaviour of the system.

Another example is taken from our ABCD[3] project where we have developed various B models for a distributed online travel agency, through which users can make hotel and car rental bookings. Here is one of the (partial) models where the DFA algorithm works extremely well: basically, the original graph is unreadable due to the large number of nodes and transitions, while Fig. 8 is quite clear and provides interesting feedback about the system.

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

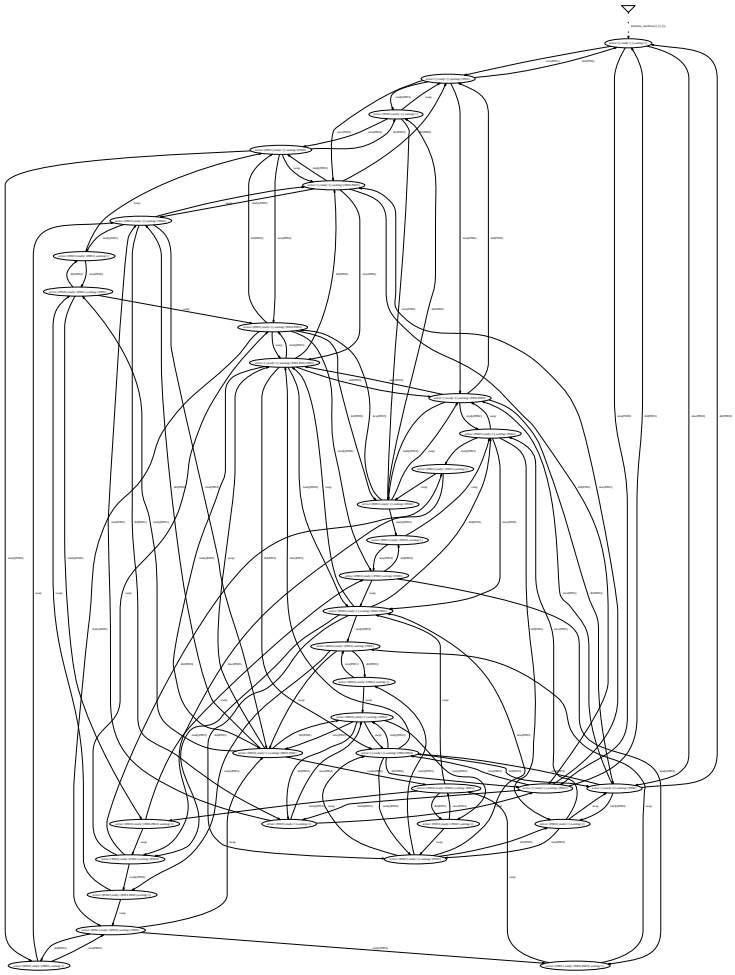

Fig. 5. Scheduler machine - Original State Space.

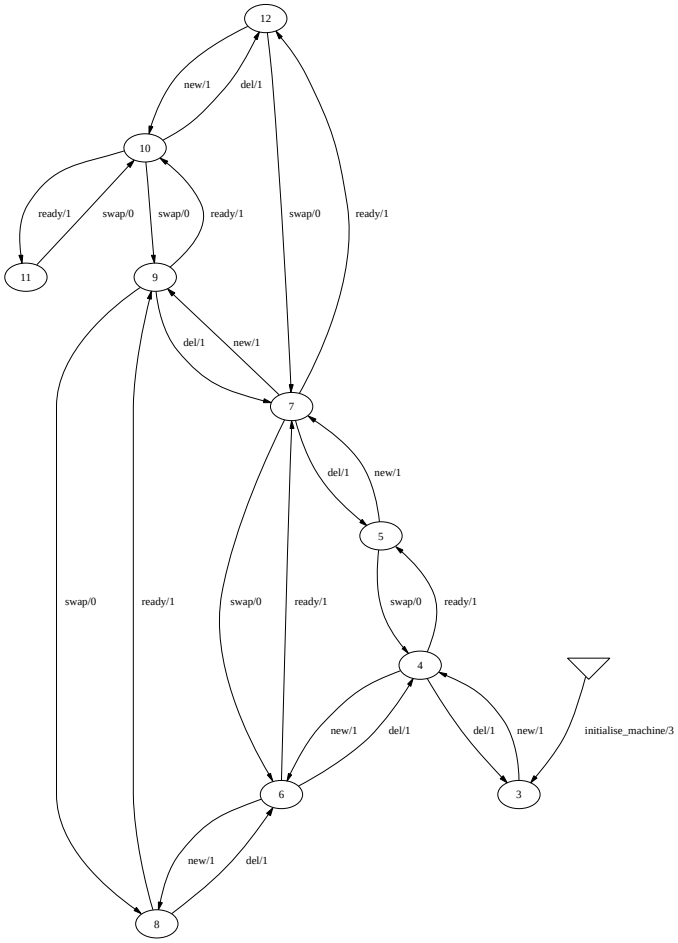

Fig. 6. Scheduler machine - DFA.

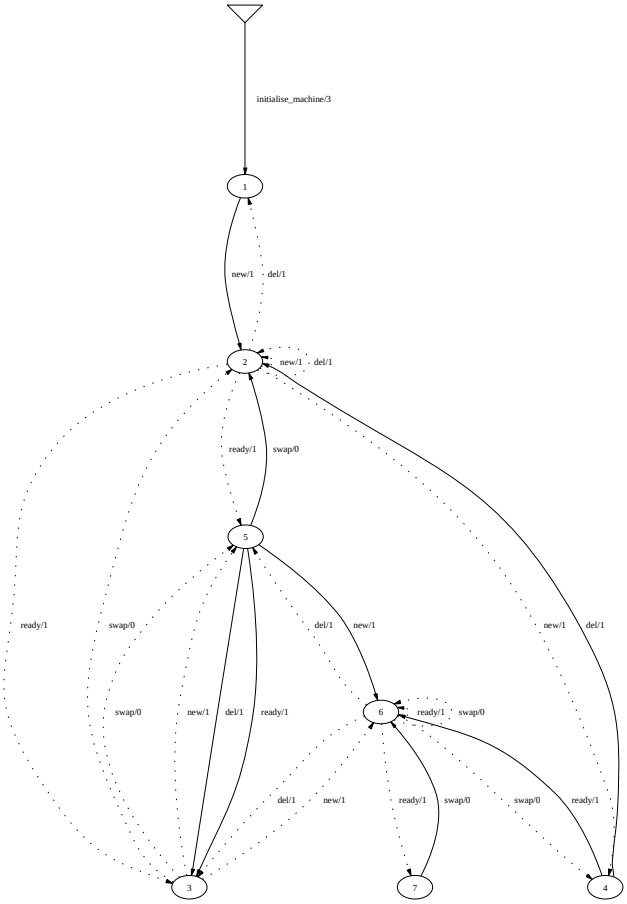

Fig. 7. Scheduler machine - Signature Merge.

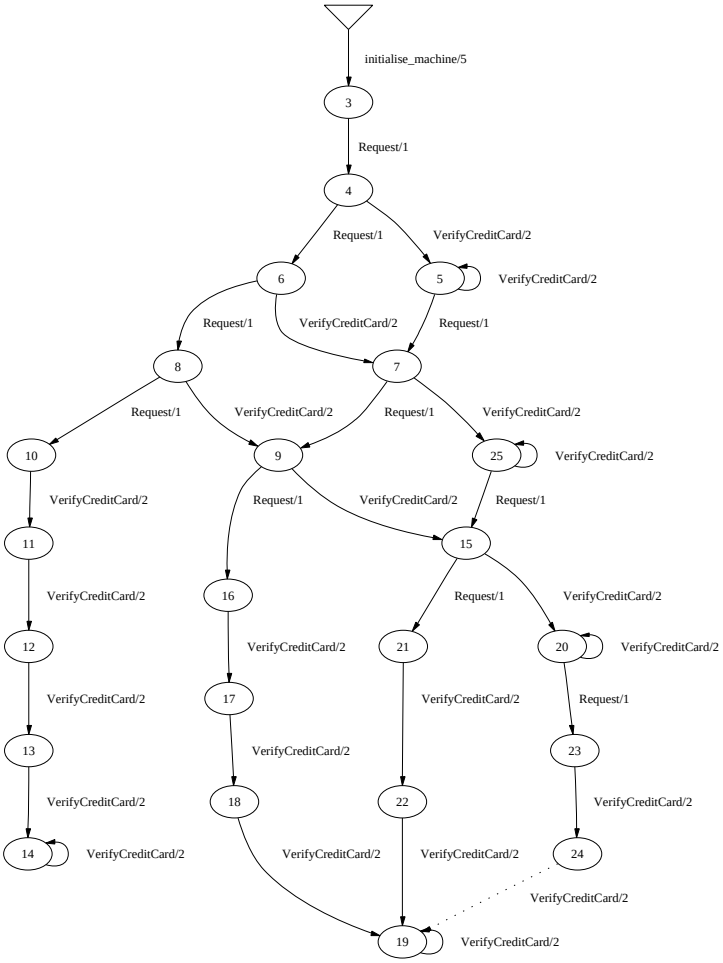

Fig. 8. TravelProB machine - DFA.

5 Empirical Evaluation

The DFA-Abstraction and Signature-Merge algorithms described in this document have been implemented within ProB and are available in ProB 1.1 and later.

We have conducted both an empirical evaluation of our algorithms, with concrete numbers on the size reductions achieved, and a more informal evaluation. Some of the examples of the latter are found in the various figures of this paper (notably in the preceding section). A more extensive list of figures is presented in an accompanying technical report [19]. This informal evaluation suggests that the algorithms are often surprisingly efficient at deriving informative graphs. However, on some examples they fail to help the user, but overall they are a very useful addition to the ProB toolset. The precise numbers presented in the rest of this section underline this more informal evaluation.

Tables 1 and 2 below show key statistics obtained after applying the *Signature-Merge* and the *DFA-Abstraction* algorithms on 47 arbitrary state spaces that had been previously model checked with the ProB model checker: Table 1 shows percentages of states and transitions compared to the original state space[4] and Table 2 shows the overall statistics.

Signature-Merge produced the best results, reducing the number of states by at least 85% and the number of transitions by at least 87% in half of the state spaces tested. Moreover, 80% of the graphs had at least 43% fewer states and 59% fewer transitions than the original. The best case produced a graph with approximately 99% fewer states and transitions. The *DFA-Abstraction* technique also gave good results; half of the graphs having at least 40% fewer states and at least 64% fewer transitions, and the best case again reduced the number of states and transitions by 99%. The worst case didn't follow the trend of producing a reduction, but in fact increased the size of the original graph by approximately ten times. A result like this should not come as a surprise since, after all, it is possible for a DFA to be exponentially greater in size than an equivalent NFA. However, only a small proportion of the applications of this technique had this effect; approximately 80% of the tests produced a reduction.

Table 1. Size of state space compared to original (%).

	Sig. Merge		**DFA-Abstr.**	
Machine Name	**States**	**Transitions**	**States**	**Transitions**
Ambulances	0.24	0.02	0.86	0.10
Baskets	6.33	2.02	21.52	8.59
B_Clavier_code	100.00	42.11	133.33	42.11
bibliotheque	73.33	58.49	93.33	75.47

[4] Some of the machine names in Table 1 appear more than once, however their implementations differ.

Table 1 Size of state space (%) – continued from previous page.

Machine Name	Sig. Merge States	Sig. Merge Transitions	DFA-Abstr. States	DFA-Abstr. Transitions
B_Site_central	60.00	12.50	80.00	12.50
CarlaTravelAgency	9.09	30.09	60.61	78.76
CarlaTravelAgencyErr	13.33	43.17	67.5	71.22
countdown	0.13	0.10	7.97	7.77
Cruise	29.54	18.19	1203.97	901.35
CSM	83.12	86.60	101.30	100.00
DAB	40.00	4.88	80.00	7.32
dfa	75.00	57.14	150.00	100.00
dijkstra	42.86	33.33	100.00	66.67
DSP0	12.24	10.61	16.33	12.12
Fermat	11.76	3.70	58.82	20.99
FinalTravelAgency	0.93	0.57	7.69	6.12
FunLaws	1.95	0.63	14.79	6.49
FunLaws	4.28	2.45	20.23	15.39
GAME	8.97	5.30	32.79	20.45
GSM_revue	36.36	28.57	63.64	50.00
Inscription	25.93	16.03	33.33	19.08
inst_adapted	1.07	0.41	17.17	6.68
Jukebox	15.00	4.53	1225.00	616.83
Level0	0.26	0.03	1.43	0.16
m0	100.00	99.98	150.77	150.29
Main	100.00	100.00	150.00	100.00
mm0	3.55	2.44	43.65	40.52
monitor	9.88	3.59	39.51	18.90
phonebook7	6.15	1.62	9.23	2.31
Queues	42.86	22.22	57.14	22.22
Results	66.67	45.45	83.33	45.45
Rubik2	0.09	0.10	100.03	100.00
RussianPostalPuzzle	2.04	1.71	27.21	22.33
rw	90.00	94.59	105.00	100.00
scheduler	22.22	14.05	33.33	20.66
SensorNode	60.00	18.18	80.00	18.18
SeqLaws	15.79	22.41	71.05	101.72
SetLaws	1.23	0.72	17.40	11.78
station	25.00	14.61	28.57	14.61
Teletext	16.00	5.71	48.00	35.71
Teletext	21.43	9.84	107.14	100.00

Table 1 Size of state space (%) – continued from previous page.

Machine Name	Sig. Merge States	Sig. Merge Transitions	DFA-Abstr. States	DFA-Abstr. Transitions
TheSystem	14.04	43.09	72.81	69.92
TransactionsSimple	16.79	33.33	76.34	83.01
TravelAgency	9.09	34.55	59.66	62.83
TravelAgency_trace_check	0.33	0.93	38.75	40.92
TravelProB	0.80	0.26	3.83	0.95
UndefinedFunctions	29.41	13.99	70.59	37.82

Statistic	Signature Merge States	Signature Merge Transitions	NFA to DFA States	NFA to DFA Transitions
Minimum	0.09	0.02	0.86	0.10
Maximum	100.00	100.00	1225.00	901.35
Median	15.00	12.50	59.66	35.71
Average	27.77	22.23	107.76	73.33
80th Percentile	**56.57**	**40.6**	**100.02**	**96.6**
Std. Dev.	31.05	27.95	241.50	155.05

Table 2. Statistics of reductions on 47 arbitrary state spaces

6 Discussion and Related Work

Tables 1 and 2 show some encouraging results. The often considerable reduction of the original state space by the DFA-Abstraction algorithm can be explained by its ability of finding regular behaviour amongst abstracted transitions, and collapsing duplicated instances of it into a single path. A good example of this is shown in the original Phonebook example (Figure 1) and the DFA reduced Phonebook example (Figure 3).

The Signature-Merge algorithm gave better reductions than the DFA-Abstraction reduction, producing non-equivalent graphs to the original that do not show the exact behaviour. However, they remain useful since they can still be used to check many properties (e.g., to check whether a certain execution path may exist in the full state space).

The three algorithms, DFA minimization [2, 14], Computing Small NFAs [15] and the Minimal Unambigous ε-transition NFAs [17] were also tested but were found to be less effective than the two mentioned above. One reason for this is that they do not implement any α-abstraction — hence future testing will attempt to incorporate this.

In addition to the two main algorithms, several other approaches for improving the visualization of state spaces were implemented and tested, and are documented in the following subsections.

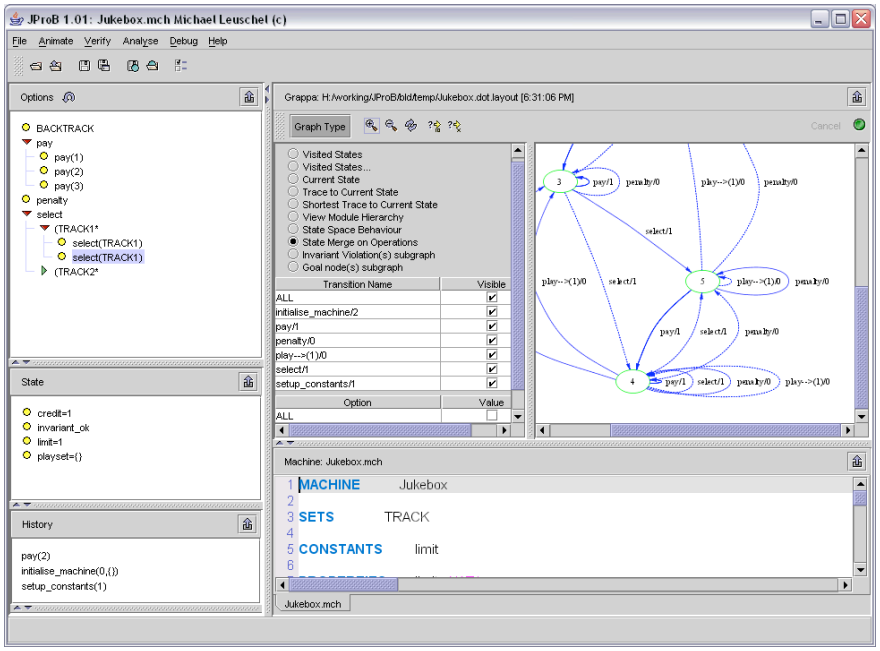

Fig. 9. Screenshot of Java version of ProB

Integrated Java/Swing Visualizer Fig. 9 shows a version of ProB that has been developed using Java to take advantage of its cross platform compatability and rich graphical user interface library. Various panes in the main window present the user with information relating to the current state; including the variables and values of the current state, a history of operations executed, a hierarchical expansion of enabled operations (top left pane) and a state space visualization. There is also an integrated specification editor to facilitate any changes necessary. As can be seen in the central pane of the screenshot, the user has several choices of visualization to choose from – some of which allow operations to be selectively removed from the visualization e.g., to remove self loops and improve clarity.

User Defined Constraints Through previous experience gained with model checkers, it was proposed that a better understanding of the system might be gained if the user were able to directly query the state space. Therefore we extended our tool by enabling the user to define constraints on system variables and on values of operation arguments, and to subsequently view a graph of all states in which these hold, and the relationship between them, if any. This is generally useful when the user is interested in exposing some subtle aspect of the state space, which a more general algorithm would be unlikely to reveal without

user intevention. It should be noted that the effectiveness of reducing state spaces using this technique depends largely on the user's literacy in the specification language and their understanding of the system; however its potential makes it a feature worth keeping and extending in the future. As mentioned, it is also possible for the user to selectively turn off visible operations in the visualization, to further reduce the size of the graph: see the tick boxes in the middle of Figure 9.

Subgraphs Another method of reducing the size of the graph is to show only part of it: a subgraph – hence our system has the option to view the subgraph that connects one or more states. This is particularly useful when one wants to view all paths that lead to a state that violates the system invariant.

More Related Work

In addition to considering algorithms and techniques that produce smaller graphs, with the goal of finding a more effective visualization, we must also consider the other aspects that affect this. These are outside the main scope of the present paper, but the interested reader is referred to [21], [8], [11], and [13].

The final aspect regards the influence of the size of a graph on the efficiency of the graph layout algorithm. This issue is somewhat orthogonal to the issues addressed in the present paper. Few layouting techniques can claim to deal effectively with thousands of nodes even though graphs of this size appear frequently in application domains, including model checking. The size of a graph can make a normally good layout algorithm completely unusable. Therefore many visualization techniques attempt to reduce the size of the graph to display. A large quantity of the important techniques are documented in [13], one of which appears particularly relevant to our overall goal: that of *clustering*. A clustering layout algorithm generally assigns nodes of a graph that satisfy some condition, into the same *cluster* (the condition may be an equivalence relation). Edges between clusters are displayed to represent the relation between the nodes of one cluster with those of another. Some good results have been witnessed and tested for large graphs containing thousands of vertices [12]. However, these graphs were representing deterministic protocols; it would be interesting to see if one could find a suitable clustering technique for the elements of the state space of a nondeterministic B model.

Finally, one can view the work in abstraction-based model checking, where abstractions are applied during exploration, as very related to our work. For example, the data abstraction of [9] is similar to our α-abstraction. However, the purpose of all these model checking works (e.g., [22, 23, 6]) is to obtain more efficient model checking, and not visualization by humans.

Acknowledgements

We would like to thank Michael Butler for stimulating discussions and feedback on the paper. We are also very grateful to anonymous referees of ZB'2005 for their very useful comments.

References

1. J.-R. Abrial. *The B-Book*. Cambridge University Press, 1996.
2. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers — Principles, Techniques and Tools*. Addison-Wesley, 1986.
3. F. Ambert, F. Bouquet, S. Chemin, S. Guenaud, B. Legeard, F. Peureux, M. Utting, and N. Vacelet. BZ-testing-tools: A tool-set for test generation from Z and B using constraint logic programming. In *Proceedings of FATES'02, Formal Approaches to Testing of Software*, pages 105–120, August 2002. Technical Report, INRIA.
4. AT&T Labs-Research. Graphviz - open source graph drawing software. Obtainable at `http://www.research.att.com/sw/tools/graphviz/`.
5. B-Core (UK) Limited, Oxon, UK. *B-Toolkit, On-line manual.*, 1999. Available at http://www.b-core.com/ONLINEDOC/Contents.html.
6. S. Bensalem, Y. Lakhnech, and S. Owre. Computing abstractions of infinite state systems compositionally and automatically. In *Proceedings of CAV'98*, LNCS, pages 319–331. Springer-Verlag, 1998.
7. F. Bouquet, B. Legeard, and F. Peureux. CLPS-B - a constraint solver for B. In J.-P.Katoen and P.Stevens, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, LNCS 2280, pages 188–204. Springer-Verlag, 2002.
8. S. Casner and J. Larkin. Cognitive Efficiency Considerations for Good Graphic Design. In *11th Annual Conf. of the Cognitive Science Society*, Ann Arbor, Michigan, August 1989.
9. E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.
10. H. W. F. Ham and J. Wijk. Visualization of State Transition Graphs. In *IEEE Symposium on Information Visualization*, pages 59–63, San Diego, CA, USA, October 2001.
11. M. Fitter and T. Green. When Do Diagrams Make Good Programming Languages? *Int. Journal of Man-Machine Studies*, pages 235–261, 1979.
12. J. Groote and F. Ham. Large State Space Visualization. In *TACAS*, volume 2619 of *Lecture Notes in Computer Science*, pages 585–590. Springer, 2003.
13. I. Herman, G. Melanon, and M. S. Marshall. Graph Visualization and Navigation in Information Visualization: A Survey. *IEEE Transactions on Visualization and Computer Graphics*, 6(1):24–43, 2000.
14. J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.
15. L. Ilie and S. Yu. Algorithms for Computing Small NFAs. In *MFCS*, volume 2420 of *Lecture Notes in Computer Science*. Springer, 2002.
16. T. Jiang and B. Ravikunar. Minimal NFA Problems are Hard. *SIAM Journal on Computing*, 22(6):1117–1141, 1993.
17. S. A. John. Minimal Unambigous ε-NFA. In *9th International Conference on Implementation and Application of Automata (CIAA-2004)*, Kingston, Ontario, Canada, July 2004.

18. 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.
19. M. Leuschel and E. Turner. Visualizing larger states spaces in ProB. Technical report, School of Electronics and Computer Science, University of Southampton, January 2005.
20. A. Malcher. Minimizing Finite Automata is Computationally Hard. *Springer-Verlag Berlin Heidelberg*, 2710:386–397, August 2003.
21. N. L. N. Dulac, T. Viguier and M.-A. Storey. On the use of Visualization in Formal Requirements Specification. In *IEEE Joint International Conference on Requirements Engineering*, pages 71–81, Essen, Germany, September 2002.
22. V. Rusu and E. Singerman. On proving safety properties by integrating static analysis, theorem proving and abstraction". In *Tools and Algorithms for the Construction and Analysis of Systems TACAS '99*, LNCS. Springer-Verlag, 1999.
23. H. Sadi and N. Shankar. Abstract and model check while you prove. In *Proceedings of CAV'99*, LNCS, pages 319–331, Trento, Italy, July 1999. Springer-Verlag.
24. Steria, Aix-en-Provence, France. *Atelier B, User and Reference Manuals*, 1996. Available at http://www.atelierb.societe.com/index_uk.html.