SAL, Kodkod, and BDDs for Validation of B Models

Lessons and Outlook

Daniel Plagge Michael Leuschel

Lehrstuhl Softwaretechnik und Programmiersprachen
Institut für Informatik
University of Düsseldorf
{plagge,leuschel}@cs.uni-duesseldorf.de

Ilya Lopatkin Alexei Iliasov
Alexander Romanovsky

School of Computing Science
Newcastle University
{Ilya.Lopatkin, Alexei.Iliasov,
Alexander.Romanovsky}@newcastle.ac.uk

Abstract

ProB is a model checker for high-level B and Event-B models based on constraint-solving. In this paper we investigate alternate approaches for validating high-level B models using alternative techniques and tools based on using BDDs, SAT-solving and SMT-solving. In particular, we examine whether ProB can be complemented or even supplanted by using one of the tools BDDBDDB, Kodkod or SAL.

Categories and Subject Descriptors I.2.3 [*Artificial Intelligence*]: Deduction and Theorem Proving—Logic Programming; D.2.4 [*Software Engineering*]: Software/Program Verification—Model Checking; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs—Mechanical verification; Specification techniques

General Terms Languages; Verification

Keywords Logic Programming; formal methods; model checking; verification; animation.[1]

1. Introduction

Event-B is a formal method for state-based system modelling and analysis evolved from the B-method [1]. The B-method itself is derived from Z and based upon predicate logic with set theory and arithmetic, and provides a wide array of sophisticated data structures (sets, sequences, relations, higher-order functions) and operations on them (set union, difference, function composition to name but a few).

Event-B has a tool support in a form of the Rodin Platform [2], which is extensible with plugins. The platform supports the mechanisms, essential for rigorous model development among which model checking (exemplified by ProB) plays an important role in ensuring the model correctness and understanding the system behaviour.

[1] This research is partially supported by the EU funded FP7 project 214158: DEPLOY (Industrial deployment of advanced system engineering methods for high productivity and dependability).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

AFM'09, June, 2009, Grenoble, France.

Copyright © 2009 ACM 978-1-60558-117-0/08/07. . . $5.00.

ProB [14, 16] is an animator for B and Event-B built in Prolog using constraint-solving technology. It incorporates optimisations such as symmetry reduction and has been successfully applied to several industrial case studies.

Still, sometimes the performance of the tool is suboptimal, especially in the context of complicated properties over unknown variables, which is the reason we have investigated alternate approaches for animating or model checking B specifications. More precisely, we investigate whether model checking of B/Event-B could by done by mapping to the input languages for other tools making use of either SAT/SMT solving techniques or of BDDs. If so, what is the performance difference compared to ProB's current constraint solving approach? Can the constraint solving approach be combined with the alternate approaches? These are the questions we try to answer in this paper. In Section 2 we provide some background information about ProB and its constraint solving approach. In Section 3 we elaborate on our experience in trying to use the BDDBDDB package [28], which provides a Datalog/relational interface to BDDs and has been successfully used for scalable static analysis of imperative programs. In Section 4 we present our experience and first results in mapping B to Kodkod [26], a high-level interface to SAT-solvers used by Alloy [12]. In Section 5, we turn our investigation to SAL [24], based on the SMT-solver Yices. We conclude with a discussion about related and future work in Section 7.

2. ProB

ProB [14, 16] is an animator and model checker for the B-method based on the constraint solving paradigm. Constraint-solving is used to find solutions for B's predicates. As far as model checking technology is concerned, ProB is an explicit state model checker with symmetry reduction [15, 27, 19]. While the constraint solving part of ProB is developed in Prolog, the new LTL model checking engine [20] is encoded in C.

Some of the distinguishing features of ProB are

- the support for almost full syntax of B, integration into Atelier B and Rodin,
- the support for Z [23] and Event-B, building on the same kernel and interpreter as for "classical" B,
- the support for other formalisms such as CSP [18] via custom Prolog interpreters which can be linked with the B interpreter [7].

However, in this paper we will concentrate on B and Event-B. ProB uses a custom built constraint solver over the datatypes of B. The base types of B are booleans, integers and user-defined base

sets, while the composed types can be constructed using cartesian product and the power set constructions. As such, B caters for sets, relations, functions and sequences and provides many custom operators on these datatypes (e.g., computing the inverse of a relation, composing relations, ...). Note that the relations and functions can be higher-order (and often are).

The performance of PROB is often good for animation and it has been successfully applied to a variety of industrial specifications. Recently, PROB has been extended to also deal with very large sets and relations (with tens of thousands of elements or more) and it is planned that PROB will be used by Siemens in production in 2009 for validating assumptions about rail network topologies [17].

In some real life scenarios, PROB can actually be more efficient than, e.g., Spin or SMV working on equivalent lower-level models (see [13] or or [11]). Still, there are many scenarios where the performance is not (yet) adequate.

For example, PROB's performance can be disappointing when values for variables have to be found which satisfy complicated predicates, and those constraints allow little deterministic constraint propagation to occur (see, e.g., the example in Section 4.3). For example, finding values for the constants of a B model which satisfy complicated constants can be very challenging. Another scenario with similar characteristics is the use of PROB as a disprover [4] for proof obligations: here one wants to find values which make the hypotheses of the proof true but the consequent false.

In this paper we investigate whether PROB can be complemented by other technologies in order to improve its performance. We also study whether the entire constraint solving engine could be replaced, if other technologies turn out to be universally superior.

3. BDDs via Datalog

Symbolic model checking with binary decision diagrams (BDDs) has become very popular since the very successful applications on hardware models [6]. We investigated, if and how we could use this approach for Event-B or B models.

BDDBDDB [28] offers the user a Datalog-like language that aims to support program analysis. It uses BDDs to represent relations and compute queries on these relations. We wanted to use the tool to find states that violate the invariant of a model, using Datalog queries that follow the schema

```
check(S) :- init(I),do_events(I,S),inv_violated(S).
do_events(A,A).
do_events(A,B) :- step(A,C),do_events(C,B).
step(A,B) :- event_X(A,B).
step(A,B) :- event_Y(A,B).
```

`check(S)` should return a reachable state that violates the invariant. To find such a state, we start in an initial state `I`, do zero or more operations from `I` to `S` via `do_events` and check if the resulting state violates the invariant with `inv_violated`. `do_events(A,B)` is specified by doing either zero steps (`A` and `B` are the same) or doing one step to an intermediate state `C` and continuing recursively. `step` again models the transition between two states by an event, here e.g. `event_X` and `event_Y`.

What can not be seen in the query above is how an initial state, a state that violates the invariant or an event is specified. To do that, one has to represent a state of the model as a bit-vector and events have to be implemented as relations between two of those bit-vectors. These relations have to be constructed by creating BDDs directly with the underlying BDD library (JavaBDD) and storing them into a file.

If we take e.g. a model that contains two integers a and b, and an event with the action $a := a + b$, we have to define a boolean formula that specifies for every bit of a in the new state how it correlates with the bits of a and b in the original state and for all other bits that they stay the same.

Soon after starting experimenting with BDDBDDB it became apparent that due to the lack of more abstract data types than bit vectors, the complexity of a direct translation from B to BDDBDDB was too high, even for small models. So we abandoned this approach, especially as there are other tools like SAL or Kodkod that give us the possibility to use symbolic approaches but offer a more powerful interface to define the models.

4. SAT Solving via Kodkod

Kodkod [26] is a constraint solver for first order logic that offers an extensive set of operations on relations. It uses an underlying SAT-solver (like *minisat*) to find solutions to a given problem.

It seems to be much more suitable for our purpose than the previously mentioned low-level approach using BDDs, because sets and relations (Kodkod considers sets as unary relations) are heavily used in B specifications. To use Kodkod, one basically has to provide four things to find solutions for a problem:

- An *universe* of atoms.
- n-ary *relations* between the atoms.
- A predicate, called *formula*, is constructed as an abstract syntax tree and can refer to previously defined relations.
- *Bounds* on the relations define which atoms can be in each relation.

Kodkod then provides possible instances for the relations.

Kodkod itself does not define an input language, but comes as a Java library and the user defines the components of the problem via the API. We used this library to implement a component of PROB that runs in a separate process. We do not replace a whole B specification by a kodkod problem description but we rather replace single predicates. This allows us to mix Kodkod and PROB's constraint solving technique, which is particular useful if components of the specification are not translatable.

4.1 Translation from B to Kodkod

In a first approach, we restricted ourselves to expressions that used only deferred sets or enumerated sets as data types. To translate a B predicate into a Kodkod problem, we do the following:

First, we construct the atoms of Kodkod's universe by creating an atom for each element of an enumerated set. Deferred sets can be treated the same way, because PROB assigns a fixed finite cardinality to each deferred set.

Then we can translate the given (i.e. enumerated or deferred) set into an unary relation which contains exactly all atoms that belong to its elements. The information that the relation contains exactly those atoms is specified as a bound on the relation. We translate each element of an enumerated set to an unary singleton relation which contains exactly the associated atom.

For every variable (or constant) that is referenced in the predicate we create a Kodkod relation. If the variable's type is a set or relation, the translation is straight-forward, if the variable's type is an element of a deferred or enumerated set, we have to add the predicate that the relation is a singleton.

B expressions often have a direct Kodkod counterpart: E.g. the cartesian product, set union, intersection and difference, reverse and closure of a relation, the relational image. Some expressions can be translated using other existing constructs. E.g. the domain restriction $S \lhd R$ with $S \subseteq A$ and $R \in A \leftrightarrow B$ can be rewritten into $S \lhd R = R \cap (S \times B)$, that again can be translated directly. Identifiers are translated to references to the according relations, as described above. Also many B predicates like conjunction, disjunc-

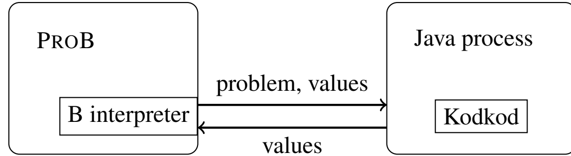

Figure 1. ProB and Kodkod

tion, negation, universal and existential quantification, set membership, subsets, etc. can be directly translated.

Fig. 3 shows an example of a predicate in B syntax and the corresponding Kodkod formula. The formula is taken from the output of the `toString` method of the corresponding Java object and only slightly modified to improve readability. One can see that it's very similar to the original B predicate, the additional `one lentry` in the beginning states that `lentry` is a singleton set. `&&` is the logical conjunction, `.` the relational image, `&` the intersection, `~` the inverse and `->` the cartesian product.

Limitations of the translation Currently the data type of each expression that we translate must be an element of a given set, a subset of a given set, or a relation between those types. Our approach does *not* support types like sets of sets, as those cannot be translated to Kodkod easily.

Another limitation is that we currently do not consider integers. Kodkod has limited support for integers, one can specify integers and sets of integers and use basic operations like addition or multiplication on them. Internally, Kodkod maps each integer to an atom (at least if a set of integers is used) and uses a bit encoding with a fixed number of bits for the integer operations. Integer overflows are silently ignored. We are currently working on a version which first does a static analysis on the B predicate to determine the possible intervals of each expression. This is needed to pass a maximum bit width to Kodkod that guarantees the integer operations to be correct. Additionally, we need the intervals to create an atom for each possible integer if a set of integers is used in an expression.

4.2 Interaction between ProB and Kodkod

To make use of Kodkod's features in ProB, a predicate in the internal syntax tree is replaced by a special syntax element that describes a Kodkod representation of the predicate. When the interpreter encounters this element the first time, the description of the problem is sent to a separate Java process (see Fig. 1). After this initialisation phase, each time the interpreter evaluates the predicate, the currently known values of used variables are sent to the process, which in turn returns possible values (calculated by Kodkod) for the remaining variables of the predicate.

In the example above, ProB would send the value of $succ$, and the Java process returns the 7 possible values for L and $lentry$.

4.3 Performance comparison

For now, we do not have an exhaustive performance comparison between ProB and Kodkod. Evaluating two extreme examples suggests that depending on the problem, either approach can show its strength.

Kodkod and large relations Kodkod does not seem to scale well when encountering large relations. This has only been relevant for certain applications of ProB, such as the property verification on real data [17]. The log-log plot in Figure 2 contains a small experiment where the performance of the set-difference operation is analysed. ProB scales linearly, while Kodkod exhibits an exponential growth (slope of the Kodkod curve > 1).

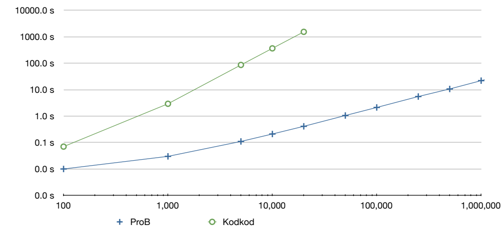

Figure 2. Performance of ProB vs Kodkod on large sets

$$Blocks = \{b_1, b_2, b_3, b_4, b_5, b_6, b_{entry}, b_{exit}\}$$
$$succs = \{b_{entry} \mapsto b_1, b_1 \mapsto b_2, b_2 \mapsto b_3, b_3 \mapsto b_3, b_3 \mapsto b_4, b_4 \mapsto b_2, b_4 \mapsto b_5, b_5 \mapsto b_6, b_6 \mapsto b_6, b_6 \mapsto b_{exit}\}$$

The predicate in B syntax

$$lentry \in L \wedge succs^{-1}[L \setminus \{lentry\}] \subseteq L \wedge \forall l.(l \in L \Rightarrow lentry \in (L \lhd succs \rhd L)^{+}[\{l\}])$$

and encoded in Kodkod

```
one lentry && lentry in L &&
((L-lentry) . ~succs) in L &&
all l: one Blocks | (l in L =>
  lentry in (l.^(((L->Blocks)&succs)&(Blocks->L))))
```

Figure 3. Finding loops in a control flow graph

Finding loops in a control flow graph For a certain class of problems, Kodkod is much faster. The problem given in Fig. 3 is encoded in a B-machine by defining the basic blocks $Blocks$ as an enumerated set, the relation $succs$ as a constant and the predicate as a precondition for an operation with two parameters L and $lentry$.

ProB needs 683 sec to find all 7 solutions without Kodkod-support. When the predicate is replaced by a call to the Kodkod-process, the computation time reduces to 10 ms.

4.4 Conclusion and ongoing work

For a certain problems, the use of a SAT solver via Kodkod seems very promising. We do not see that this approach might replace the constraint solving mechanism of ProB, because there still will be constructs in specifications that are very hard to translate.

The lack of support for integers turned out to be a hurdle for trying out more examples. So we are currently working on supporting integers and will then continue the evaluation of this approach.

Replacing single predicates in the syntax tree has the advantage that we can mix both techniques and that we have no restrictions on the elements that can be used in specifications. However, this approach restricts us to explicit state model checking. For making symbolic techniques like bounded model-checking available, we must be able to translate the whole specification. We are currently thinking about ways to circumvent this restriction by using predicate abstraction [8].

5. SAL

SAL [24] is a model-checking framework combining a range of tools for reasoning about information systems, in particular concurrent systems, in a rigorous manner. The core of SAL is a language for constructing system specifications in a compositional way. The

SAL tool suite includes a state of the art symbolic (BDD-based) and bounded (SAT-based) model checkers.

The overall aim of our work is to investigate the potential applications of SAL in a combination with the Rodin platform, a development tool for the Event-B formalism. Unlike SAL, Rodin relies mostly on theorem proving for model analysis. We are looking for a way of complementing the Rodin platform with a plugin that would automate some of the more difficult tasks such as liveness, deadlock freeness and reachability analysis.

5.1 Event-B to SAL translation

In this section we describe our ongoing work on translating Event-B models into the input language of SAL. We report the results on initial experiments on verifying Event-B models in the SAL framework. The benchmark for our efforts is ProB[14, 16]. Since ProB is Prolog-based, we had started with an expectation of achieving some performance advantage for a considerable subset of problems.

Table 1 and Figure 4 present the comparative performance of the ProB Event-B model checker and SAL [2] run on the result of transforming the same model into the input language of SAL. The first model is a synthetic benchmark based on bubble sort algorithm. In this model a single event swaps neighbouring elements of an array if they are in a wrong order. The other four models are the examples bundled with ProB distribution. These demonstrate the translation and the performance of some of the most "inconvenient" parts of Event-B syntax for SAL: sets, functions, relations and operators on them such as union, intersection, cardinality and etc.

In the comparison tables "SAL run" stands for the time of iterating through the state space, and "SAL total" is the total time including generation of model checker. On the charts we show ProB and total SAL times with values connected by lines where there is an evident dependency between the size of state space and corresponding timings. During model checking the Club specification we were changing several parameters, and each of them had its effect on the size of state space and total model checking time. Thus we show results for this model as a set of non-connected points. All figures are meaningful for comparative analysis only: they could change substantially depending on the operating system, compiler distribution, and, obviously, the performance of a machine on which tests are run.

These results are obtained with default settings in both model checkers. By enabling hash or nauty *symmetries*, one is able to obtain much better results in ProB (see also [13]) , although this requires some understanding on when these options should be enabled. In some cases, symmetry reduction may slow down the process but mostly it reduces the model checking time. For example, with hash symmetry enabled, ProB can model check the Scheduler specification with parameter set to 5 in just 0.25s compared to 1.7s with default settings (see Table 2). Also, for Life with set size 20, ProB only generates 232 states with symmetry and the model checking time goes down to 2.2 seconds. Finally, for Club with MaxInt = 20 and set size 4, the number of states goes down to 682 from 3597, and the model checking time goes down to roughly 6 seconds.

Reasoning on timings we obtained during our experiments, we drew preliminary conclusions about efficiency of SAL model checker on our models:

- SAL verification stage alone can be more than 10x up to 1000x faster than ProB (without symmetry) for a large class of models;

Sort				
Parameters	*States*	ProB	*SAL run*	*Sal total*
array[3] of 1..10	1000	18 sec	0.08 sec	0.2 sec
array[4] of 1..10	10000	5 min 41 sec	0.1 sec	0.3 sec
array[6] of 1..10	1000000	> 30 min	13 sec	14 sec
array[6] of 1..12	2985984	> 30 min	29.5 sec	30.7 sec
Life				
Parameters	*States*	ProB	*SAL run*	*Sal total*
set size = 5	243	0.9 sec	0.06 sec	0.28 sec
set size = 7	2187	11.8 sec	0.06 sec	0.42 sec
set size = 8	6561	72.5 sec	0.06 sec	0.8 sec
set size = 20	$> 10^9$	> 20 min	0.28 sec	2.7 sec
Max				
Parameters	*States*	ProB	*SAL run*	*Sal total*
MaxInt = 7	256	1.4 sec	0.06 sec	0.22 sec
MaxInt = 8	512	3.3 sec	0.05 sec	0.25 sec
MaxInt = 9	1024	7.8 sec	0.08 sec	0.26 sec
MaxInt = 10	2048	20.7 sec	0.08 sec	0.28 sec
MaxInt = 11	4096	61 sec	0.09 sec	0.25 sec
MaxInt = 12	8192	3 min 18 sec	0.2 sec	0.6 sec
MaxInt = 30	$> 10^9$	> 30 min	0.4 sec	2 sec
MaxInt = 50	$> 10^{15}$	> 30 min	2.2 sec	7.4 sec
Scheduler				
Parameters	*States*	ProB	*SAL run*	*Sal total*
set size = 3	35	0.24 sec	0.05 sec	0.47 sec
set size = 4	124	0.7 sec	0.1 sec	1.8 sec
set size = 5	437	1.7 sec	0.1 sec	55 sec
Club				
Parameters	*States*	ProB	*SAL run*	*Sal total*
capacity = 1..2 set size = 3 MaxInt = 10	368	1.8 sec	0.03 sec	0.6 sec
capacity = 1..2 set size = 4 MaxInt = 10	958	4.6 sec	0.03 sec	5 sec
capacity = 1..2 set size = 4 MaxInt = 30	3358	11.8 sec	0.03 sec	5.5 sec
capacity = 1..2 set size = 4 MaxInt = 50	5758	30.3 sec	0.05 sec	6 sec
capacity = 1..2 set size = 4 MaxInt = 10000	119758	> 30 min	0.05 sec	6 sec
capacity = 1..2 set size = 5 MaxInt = 10	2408	9.5 sec	0.05 sec	4 min 29 sec
capacity = 1..3 set size = 4 MaxInt = 20	3597	14.2 sec	0.03 sec	5.4 sec

Table 1. Comparison on Event-B and SAL specifications

- even in the case of enabled symmetry in ProB, SAL shows either better or comparable performance;
- the bottleneck of SAL model checking performance is in the pre-verification analysis and checker generation stages. In particular, unfolding quantifiers may take $\sim 95\%$ of generation time;
- SAL model checking time strongly depends on the complexity of theorems, complex computations in theorems dramatically reduce the overall performance.

In these models, we used a classical representation of sets in predicates adopted from [25]. Let us look at an excerpt from a B model which uses sets:

[2] All timings are obtained using *sal-smc* model checker

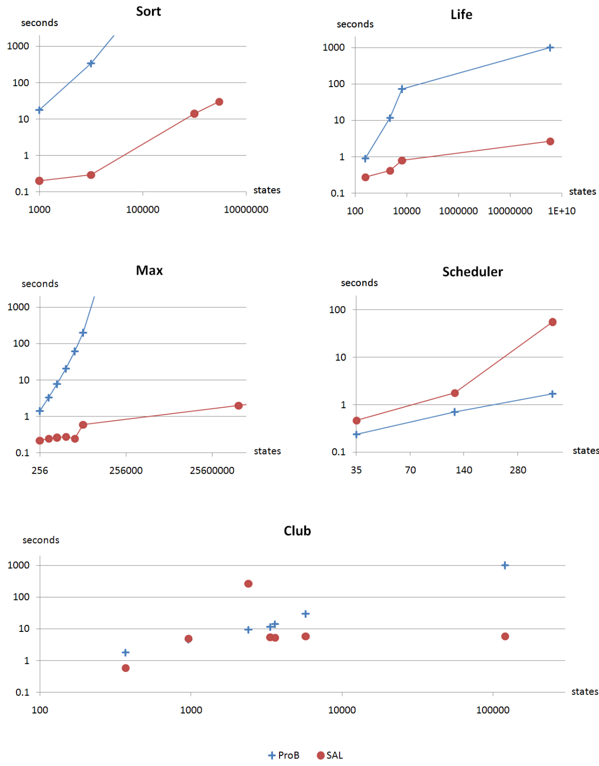

Figure 4. Comparison on Event-B and SAL specifications

```
SETS s ...
CONSTANTS total ...
VARIABLES a ...
INVARIANT a⊂s & card(a)≤total ...
OPERATIONS
 add(b) = WHEN b∈s & b∉a THEN a:=a∪{b} END;
 ...
```

The corresponding SAL specification is

```
modelname: CONTEXT =
BEGIN
 ntype: TYPE = {n: prob!NAT1 | n <= prob!MaxSetSize};
 s: CONTEXT = set{ntype;};
 main: MODULE =
 BEGIN
   LOCAL a: s!Set ...
   TRANSITION [
   ([](b:ntype): add:
     NOT s!contains(a, b) --> a' = s!union(a, b))...
 END
 th: theorem main ⊢
      G(EXISTS(n: prob!SizeType):
           s!size(a, n) AND n<=total);
END
```

And the implementation of cardinality test is

```
size(s: Set, n: natural): boolean =
 (n=0 and equals(s, Empty)) or
  (n>0 and exists (f: [[1..n] -> T]) :
   (forall (x1, x2: [1..n]): f(x1)=f(x2) => x1=x2) and
      (forall (y: T): s(y) <=>
           (exists (x: [1..n]) : f(x)=y)));
```

Such calculation of cardinality involves an excessive use of quantifiers. In [9] a brute-force approach is proposed which iterates through a function from a set type to boolean. We believe that this can still be improved upon. Since SAL natively supports arrays, we encoded sets as arrays of boolean type with an index being mapped to values of a set type. Essentially, a set (and its derivatives such as relation and function) is represented as a characteristic function. The approach brings a number of advantages. For instance, cardinality calculation is realised with a relatively efficient recursive function:

```
sizesofar(s: Set, n: SetArrayType): prob!SizeType =
 if s[n] then 1 else 0 endif +
 if n>1 then sizesofar(s, n-1) else 0 endif;
size(s: Set): prob!SizeType = sizesofar(s, prob!MaxSetSize);
```

As a further optimisation technique, for each set we introduce an auxiliary variable storing the current set size. This variable is updated each time the set is changed.

With our new style of set translation, the example above is transformed into the following:

```
modelname: CONTEXT =
BEGIN
 s: CONTEXT = set;
 main: MODULE =
 BEGIN
   LOCAL a: s!Set, a_size: prob!SizeType ...
   INITIALIZATION ...; n = set!size(a); ...
   TRANSITION [
   ([](b:set!SetArrayType): add:
     NOT s!contains(a, b) -->
       a' = s!union(a, b); a_size' = set!size(a');)...
 END
 th: theorem main ⊢ G(a_size <= total);
END
```

With this approach an invariant (SAL safety theorem) does not involve the heavy calculation of a set size. Unsurprisingly, this results in a significant performance benefit for the models operating on sets. We present the comparison of these timings in Table 2 and Figure 5.

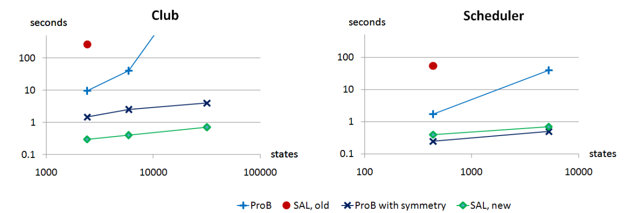

Figure 5. The comparative results for the efficient set translation (Sal, new)

5.2 Ongoing work on the plugin

Our ongoing work is focusing on incorporating the SAL model checker into the Rodin platform. A number of steps are being performed to achieve this. We are aiming at developing a nearly complete mapping of Event-B to the SAL input language. We do not consider it practical to attempt to cover the whole of the

Club				
Parameters	*States*	PROB	*Sal, old*	*SAL, new*
capacity = 1..2 set size = 5 MaxInt = 10	2408	9.5 sec	4 min 29 sec	0.3 sec
with symmetry	216	$\approx$ 1.5 sec		
capacity = 1..2 set size = 6 MaxInt = 10	5857	39.4 sec		0.4 sec
with symmetry	245	$\approx$ 2.5 sec		
capacity = 1..2 set size = 8 MaxInt = 10	31738	>10 min		0.7 sec
with symmetry	288	$\approx$ 4 sec		
Scheduler				
Parameters	*States*	*ProB*	*Sal, old*	*SAL, new*
set size = 5	437	1.7 sec	55 sec	0.4 sec
with symmetry	21	0.25 sec		
set size = 7	5231	39.2 sec		0.7 sec
with symmetry	36	0.5 sec		

Table 2. The comparative results for the efficient set translation (Sal, new)

Event-B mathematical language. Instead, we intend for our tool to cooperate with the PROB model checker so that models that cannot be handled with SAL are automatically handled by PROB.

The result of developing the language mapping would be an automated translation of Event-B models into SAL. The next step is in providing a user with a meaningful feedback from the tool.

The summary of our translation approach is given in Table 3. It covers the main Event-B model elements such as events, invariant, variables, etc.

Since SAL requires variable types to be predefined and finite, all variables of an Event-B model must be automatically constrained to finite (and small) ranges. In Event-B models, definition of a variable type is a part of invariant. Therefore, constraints on variables can be obtained by analysing the invariant. In case of unbound type the model translator would use predefined ranges either specified by user or taken by default.

SAL supports enumerated types which can be used for encoding Event-B given sets. However, considering our general implementation of sets we see reasonable to map enumerated values into a range of integers at the translation level. The result of a SAL model checking would be traced back to the Event-B enumeration and given to a user as a feedback in terms of the initial model.

Along with solutions to translating Event-B models into SAL, we have identified a number of challenges:

- the use of cartesian products, relational composition and related operators often leads to a state explosion even in examples with modest model state space;
- it is apparent that some constructs of Event-B, such as closure, set comprehension and others, are very hard to translate into anything that would not preclude checking of interesting model properties;
- for some language constructs, it is easier to do partial translation. That is, we choose to assume that some properties hold without checking them to gain a performance benefit. This, for example, happens when accessing a function - there is no check of well-formedness of a function construct. Our intention is to benefit for the Rodin platform static checker and theorem prover to simplify translation by relying on the properties of a model already demonstrated by a static checker or a theorem prover.

General scheme	
Event-B	*SAL*
Model + context	Single module within a single SAL context
Events	Named guarded commands in TRANSITION block of a module
Event guards	Transition guards
Non-deterministic choice (ANY)	Variable becomes a parameter of a guarded command, the predicate becomes a part of a guard. The event is splitted into two transitions if necessary.
Invariants	Constraints on variables and sets go into type definition, remaining becomes a theorem
Contexts	
Carrier sets	Enumerated types or arrays
Constants	Constants
Named properties	Part of type definition, and guard of initialization transition
Types	
Basic types	Subranges of equivalent SAL types. Subranging depends on model being translated and model checking parameters.
Sets + operations	Arrays of reasonable size and operations on them
Total functions + operations	Total functions
Partial functions, injections, surjection + operations	Either definition in predicates or optimized using arrays

Table 3. Translation scheme

6. Related Work

Preliminary experience with translating Event-B to Alloy is reported in [22], but empirical results are not available yet. The authors also encountered the problem encoding complicated expressions of B in Alloy:

> "Expressions are the hardest part to encode. There is not only a myriad of complex expressions in Event-B but given that Alloy uses only flat relations, some Event-B expressions that introduce relations with nested sets generate many (and potentially large) Alloy expressions." [22]

Note that Daniel Jackson's dream was to work directly in Z, whose notation of expressions and predicates is very similar to B, but he abandoned that goal and developed the Alloy language

much more suitable for automated analysis.[3] Hence, it is no wonder that translating Z or B into Alloy is not trivial. The Z2SAL [10] project has similar goal as the translation to SAL shown here. Initial translations were not very efficient (see discussion in [20]), but more recent translations seem to perform better.

7. Conclusion and Future Work

We presented three approaches to model check B or Event-B specifications by translating them to other formalisms. A first naïve attempt to use low-level methods like BDDs soon turned out to be impractical and has no apparent advantage over using a more sophisticated tool.

SAL and Kodkod both are very promising. There is still room for improvements of the translation to support more expressions and to enhance the efficiency of the translated model. For some scenarios, using those translations gives us a much faster model check than PROB currently offers. On the other hand, for some tasks the constraint-solving approach of PROB is much more efficient (e.g., when working on large relations). As such, we believe there is considerable advantage in trying to combine these approaches (rather than one approach supplanting the other).

For SAL there is still the problem in translating more complicated Event-B data structures efficiently. Indeed, we have found that the SAL input language imposes a number of restrictions on what can be translated and how it is translated. For some objects, such as functions, the semantic gap between native Event-B formulae and SAL translation makes the interpretation of SAL output a bigger challenge than it could be. In addition, with a complex translation one has to worry about validity of the translation rules. As a way to overcome these limitations we are considering the possibility of translating a subset of Event-B mathematical language directly into Yices. The resulting tool could be used to improve the performance of PROB model checker and also to build a more capable disprover plugin [21].

References

[1] J.-R. Abrial. *The B-Book.* Cambridge University Press, 1996.

[2] J.-R. Abrial, M. Butler, and S. Hallerstede. An open extensible tool environment for Event-B. In *ICFEM06*, LNCS 4260, pages 588–605. Springer, 2006.

[3] 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.

[4] 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.

[5] E. Börger, M. Butler, J. P. Bowen, and P. Boca, editors. *Abstract State Machines, B and Z, First International Conference, ABZ 2008, London, UK, September 16-18, 2008. Proceedings*, volume 5238 of *Lecture Notes in Computer Science*. Springer, 2008.

[6] 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.

[7] 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.

[8] S. Das and D. L. Dill. Successive approximation of abstract transition relations. In *Proceedings of the Sixteenth Annual IEEE Symposium on Logic in Computer Science*, 2001. June 2001, Boston, USA.

[9] J. Derrick, S. North, and A. J. H. Simons. Z2sal - building a model checker for z. In Börger et al. [5], pages 280–293.

[10] J. Derrick, S. North, and T. Simons. Issues in implementing a model checker for Z. In Z. Liu and J. He, editors, *ICFEM*, LNCS 4260, pages 678–696. Springer, 2006.

[11] T. Hörne and J. A. van der Poll. Planning as model checking: the performance of prob vs nusmv. In R. Botha and C. Cilliers, editors, *SAICSIT Conf.*, volume 338 of *ACM International Conference Proceeding Series*, pages 114–123. ACM, 2008.

[12] D. Jackson. Alloy: A lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology*, 11:256–290, 2002.

[13] M. Leuschel. The high road to formal validation. In Börger et al. [5], pages 4–23.

[14] 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.

[15] M. Leuschel, M. Butler, C. Spermann, and E. Turner. Symmetry reduction for B by permutation flooding. In *Proceedings B2007*, LNCS 4355, pages 79–93, Besancon, France, 2007. Springer-Verlag.

[16] M. Leuschel and M. J. Butler. ProB: an automated analysis toolset for the B method. *STTT*, 10(2):185–203, 2008.

[17] M. Leuschel, J. Falampin, F. Fritz, and D. Plagge. Automated property verification for large scale b models. submitted, 2009.

[18] M. Leuschel and M. Fontaine. Probing the depths of CSP-M: A new FDR-compliant validation tool. In *Proceedings ICFEM 2008*, LNCS, pages 278–297. Springer-Verlag, 2008.

[19] M. Leuschel and T. Massart. Efficient approximate verification of B via symmetry markers. In *Proceedings International Symmetry Conference*, pages 71–85, Edinburgh, UK, January 2007.

[20] 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. [3], pages 73–84.

[21] O. Ligot, J. Bendisposto, and M. Leuschel. Debugging Event-B Models using the ProB Disprover Plug-in. *Proceedings AFADL'07*, Juni 2007.

[22] P. J. Matos and J. Marques-Silva. Model checking event-b by encoding into alloy. In Börger et al. [5], page 346.

[23] D. Plagge and M. Leuschel. Validating Z Specifications using the ProB Animator and Model Checker. In J. Davies and J. Gibbons, editors, *Proceedings IFM 2007*, LNCS 4591, pages 480–500. Springer-Verlag, 2007.

[24] The SAL website. `http://sal.csl.sri.com`.

[25] G. Smith and L. Wildman. Model checking Z specifications using SAL. In *ZB*, pages 85–103, 2005.

[26] E. Torlak and D. Jackson. Kodkod: A relational model finder. In O. Grumberg and M. Huth, editors, *TACAS*, LNCS 4424, pages 632–647. Springer, 2007.

[27] E. Turner, M. Leuschel, C. Spermann, and M. Butler. Symmetry reduced model checking for B. In *Proceedings Symposium TASE 2007*, pages 25–34, Shanghai, China, June 2007. IEEE.

[28] J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 131–144, New York, NY, USA, 2004. ACM Press.

[3] Invited talk "Recent Advances in Alloy" at iFM 2007 in Oxford.