Validating B,Z and TLA^+ using ProB and Kodkod

(Technical Report, March 2012, *Rev* : 8536)

Daniel Plagge and Michael Leuschel

Institut für Informatik, Universität Düsseldorf[**]
Universitätsstr. 1, D-40225 Düsseldorf
{plagge,leuschel}@cs.uni-duesseldorf.de

Abstract. We present the integration of the Kodkod high-level interface to SAT-solvers into the kernel of ProB. As such, predicates from B, Event-B, Z and TLA^+ can be solved using a mixture of SAT-solving and ProB's own constraint-solving capabilities developed using constraint logic programming: the first-order parts which can be dealt with by Kodkod and the remaining parts solved by the existing ProB kernel. We also present an extensive empirical evaluation and analyze the respective merits of SAT-solving and classical constraint solving. We also compare to using SMT solvers via recently available translators for Event-B.
Keywords: B-Method, Z, TLA, Tool Support, SAT, SMT, Constraints.

1 Introduction and Motivation

TLA^+ [10] are B [1] and Z are all state-based formal methods rooted in predicate logic, combined with arithmetic and set theory. The animator and model checker ProB [12] can be applied to all of these formalisms and is being used by several companies, mainly in the railway sector for safety critical control software [13, 14]. At the heart of ProB is a kernel dealing with the basic data types of these formalisms, i.e., integers, sets, relations, functions and sequences. An important feature of ProB is its ability to solve constraints; indeed constraints can arise in many situations when manipulating a formal specification: the tool needs to find values of constants which satisfy the stipulated properties, the tool needs to find acceptable initial values of a model, the tool has to determine whether an event or operation can be applied (i.e., is there a solution for the parameters which make the guard true) or whether an quantified expression is true or not. Other tasks involve more explicit constraint solving, e.g., finding counter examples to invariant preservation or deadlock freedom proof obligations [7]. While ProB is good at dealing with large data structures and also at solving certain kinds of complicated constraints [7], it can fare badly on certain other constraints,

[**] Part of this research has been sponsored by the EU funded FP7 projects ADVANCE and 214158: DEPLOY (Industrial deployment of advanced system engineering methods for high productivity and dependability).

in particular relating to relational composition and transitive closure. (We will illustrate this later in the paper.)

Another state-based formalism is Alloy [8] with its associated tool which uses the Kodkod [21] library to translate its relational logic predicates into propositional formulas which can be fed into SAT solvers. Alloy can deal very well with complicated constraints, in particular those involving relational composition and transitive closure. Compared to B, Z and TLA$^+$, the Alloy language and the Kodkod library only allow first-order predicates, e.g., they do not allow relations over sets or sets of sets.

The goal of this work is to integrate Kodkod into ProB, providing an alternative way of solving B, Z and TLA$^+$ constraints. Note that we made sure that the animation and model checking engine as well as the user interface of ProB are agnostic as to how the underlying constraints are solved. Based on this integration we also conduct a thorough empirical evaluation of the performance of Kodkod compared to solving constraints with the existing constraint logic programming approach of ProB. As we will see later in the paper, this empirical evaluation provides some interesting insights. Our approach also ensures that the whole of B is covered, by delegating the untranslatable higher-order predicates to the existing ProB kernel.

2 B, Z, TLA$^+$ and Kodkod in comparison

ProB can support Z and TLA$^+$ by translating those formalisms to B, because these formalisms have a common mathematical foundation. In the case of TLA$^+$ a readable B machine is actually generated, whereas a Z specification is translated to ProB's internal representation because some Z constructs did not have a direct counterpart in B's syntax. In the next sections we refer only to B, but because all three notations share the same representation in ProB, all presented techniques can be applied likewise to the two other specification languages.

If we specify a problem in B, we basically have a number of variables, each of a certain type and a predicate. The challenge for ProB is then to find values for the variables that fulfil the predicate. For simplicity, we ignore B's other concepts like machines, refinement, etc.

Kodkod provides a similar view on a problem. We have to specify a number of relations (these correspond to our variables in B) and a formula (which corresponds to a predicate in B) and Kodkod tries to find solutions for the relations.

From this point of view, the main difference between B and Kodkod is the type system: Instead of having some basic types and operations like power set and Cartesian product to combine these, Kodkod has the concept of an universe consisting of atoms. To use Kodkod, we must define a list of atoms and for each relation we must specify a *bound* that determines a range of atoms that can be in the relation.

The bound mechanism can also be used to assign an exact value to an relation. This is later useful when we have already computed some values by ProB.

3 Architecture

3.1 Overview

We use a small example to illustrate the basic mechanism how Kodkod and SAT solving is integrated into ProB's process to find a model for a problem. Details about the individual components are presented below after this overview.

Our small problem is taken from the "dragon book" [2] and formalised in B. The aim is to find loops in a control flow graph of a program (see Figure 1).

We model the basic blocks as an enumerated set *Blocks* with the elements b_1, b_2, b_3, b_4, b_5, b_6, b_{entry}, b_{exit}. The successor relation is represented by a variable *succs*, the set of the nodes that constitute the loop by L and the entry point of the loop by *lentry*. The problem is described by the B predicate:

$$\begin{aligned}
&\mathit{succs} = \{b_{entry} \mapsto b_1, b_1 \mapsto b_2, b_2 \mapsto b_3, b_3 \mapsto b_3,\\
&\quad b_3 \mapsto b_4, b_4 \mapsto b_2, b_4 \mapsto b_5,\\
&\quad b_5 \mapsto b_6, b_6 \mapsto b_6, b_6 \mapsto b_{exit}\}\\
\wedge\ &\mathit{lentry} \in L\\
\wedge\ &\mathit{succs}^{-1}[L \setminus \{\mathit{lentry}\}] \subseteq L\\
\wedge\ &\forall l.(l \in L \Rightarrow \mathit{lentry} \in (L \lhd \mathit{succs} \rhd L)^{+}[\{l\}])
\end{aligned}$$

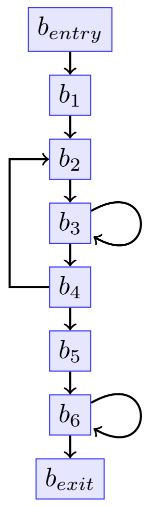

Fig. 1: Control flow graph of a program

In total, there are seven different solutions to this problem, for instance $L = \{b_2, b_3, b_4\}$ with $\mathit{lentry} = b_2$.

After parsing and type checking the predicate, we start a static analysis (the box "Analysis" in Fig. 2) to determine the integer intervals of all integer expressions. In our simple case, the analysis is not necessary. In Section 3.4 we describe how this analysis works and under which circumstances it is needed.

In the next phase, we try to translate the formula from B to Kodkod ("Translation" in Fig. 2). First we have a look at the used variables and their types: *succs* is of type $\mathbb{P}(\mathit{Blocks} \times \mathit{Blocks})$, L of type $\mathbb{P}(\mathit{Blocks})$ and *lentry* of type *Blocks*. *Blocks* is here the only basic type that is used. Thus we have to reserve 8 atoms in the Kodkod universe to represent this type; each atom in the universe corresponds directly to a block b_i. The variables can be represented by binary (*succs*) and unary (L and *lentry*) relations, where we have to keep in mind that the relation for *lentry* must contain exactly one element. The B predicate can be completely translated to a Kodkod problem. In Section 3.2 we will describe the translation in more detail. It can be useful to keep a part of the formula untranslated: since the part $\mathit{succs} = \{\ldots\}$ is very easy to compute by ProB, we leave it untranslated. The translated formula has the form:

```
one lentry &&
lentry in L &&
```

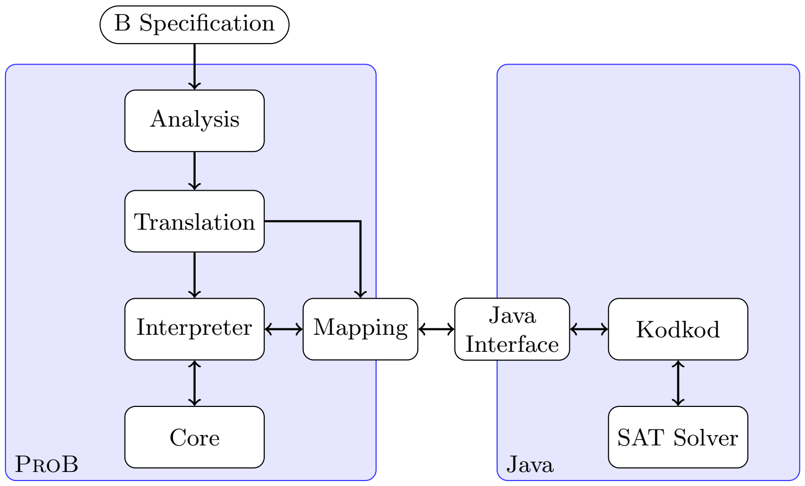

Fig. 2: Overview of the architecture

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

The translated description of the formula is then stored and a mapping between ProB's internal representation and Kodkod's representation of values is constructed ("Mapping" in Fig. 2). The message "new problem with following properties..." is sent to the Java process.

The Kodkod problem gets a unique identifier and the translated part of the B predicate is replaced by a reference to the problem, i.e., $succs = \{\ldots\} \wedge$ `kodkod`(ID), and then given to the B interpreter of ProB.

When the ProB interpreter starts to evaluate the predicate, it prioritises which parts should be computed first. It chooses $succs = \{\ldots\}$ because it can be computed deterministically by ProB's core and finds a value for *succs*. Then a message "We have these values for *succs*, try to find values for the other variables" is sent to the Java process.

The Java process has now a complete description of the problem. It consists of the universe (with 8 atoms) and relations for the variables and the type *Blocks* itself. The bounds define the value of *succs* and *Blocks* and ensure that all relations contain only atoms that match their corresponding types. This information is then given together with the formula to the solver of the Kodkod library ("Kodkod" in Fig. 2) that translates the problem into a SAT problem and passes this to the SAT solver.

The SAT solver finds solutions that are transformed by Kodkod to instances of the relations that fulfil the given formula The values of the previously unknown relations that represent L and *lentry* are sent back in an answer to the ProB process. The answer is then mapped to ProB's internal representation of values.

The B interpreter can now continue with the found values. Now all predicates have been evaluated and the solutions can be presented to the user.

3.2 Translation

Representing values It turns out that the available data types in Kodkod are the main limitation when trying to translate a problem described in B. Let's first have a look at the available data types in B and how they can be translated to Kodkod. We have the basic data types:

Enumerated Sets Enumerated sets can directly be translated to Kodkod. For each element of the set, we add an atom to the universe and create an unary relation that contains exactly that atom. The relation is needed in case the element is referred in an expression. We create another unary relation for the whole set that contains exactly all atoms of the enumerated set.

Deferred Sets Deferred Sets in B can have any number of elements that are not further specified. For animation, ProB chooses a fixed finite cardinality for the set, either by an analysis of the axioms or by using user preferences. Then we can treat deferred sets just like a special case of enumerated sets.

Booleans The set of Booleans is a special case of an enumerated set with two elements TRUE and FALSE.

Integer Integers in B represent mathematical numbers, they can be arbitrary large. It is possible to represent integer values in Kodkod, but the support is very limited and special care has to been taken. We describe the handling of integers in Section 3.3 in detail.

Thus, we can map a B variable of a basic data type to a Kodkod relation. Since Kodkod treats every relation as a set, we must ensure explicitly that the relations for such variables contain exactly one element.

Example. Let's assume that we use two types in our specification, an enumerated set $E = \{a, b, c\}$ and BOOL. Treating the Booleans as enumerated set $\text{BOOL} = \{\text{TRUE}, \text{FALSE}\}$, we have the following universe with five atoms:

	E			BOOL	
B value	a	b	c	TRUE	FALSE
atom	0	1	2	3	4

We can now represent a variable of type E by an unary relation `r1` whose elements are bounded to be a subset of the atoms $0 \mathinner{..} 2$. We also have to add the Kodkod formula `one r1`.

In B, two or more basic types can be combined with the Cartesian product. Variables of such a type can be represented by a relation.

Example. If we have a variable of type $(E \times E) \times \text{BOOL}$, we can represent it by a ternary relation `r2` whose elements are bound to subsets of the atoms $0 \mathinner{..} 2 \times 0 \mathinner{..} 2 \times 3 \mathinner{..} 4$. Like in the example above, we have to add the condition `one r2`.

We can construct the power set $\mathbb{P}(\alpha)$ for any type α in B. A variable of type $\mathbb{P}(\alpha)$ can be mapped to a Kodkod relation if α is itself not a set. A relation for $\mathbb{P}(\alpha)$ is defined exactly as a relation for α but without the additional restriction that it must contain exactly one element.

Finally, let's have a look at what we *cannot* translate. All "higher-order" data-types, i.e. sets of sets are not translatable. E.g a function $f \in A \nrightarrow \mathbb{P}(B)$ cannot be handled.

It turned out that unary and binary relations are handled very well. With relations of a higher arity we encounter the problem that many operators in Kodkod are restricted to binary relations. Thus it is not as easy to translate many properties using these data types.

Translating predicates and expressions One of the central tasks in combining ProB and Kodkod is the translation of the B predicate that specifies the problem to a Kodkod formula. Many of B's most common operators can be directly translated to Kodkod, especially when basic set theory and relational algebra is used. It is not strictly necessary to cover all operators that B provides, because we always have the possibility to fall back to ProB's own constraint solving technique. Of course, we strive to cover as many operators as possible.

Operators on Predicates. The basic operators that act on predicates like conjunction, disjunction, etc. have a direct counterpart in Kodkod. This includes also universal and existential quantification.

Arithmetic operators. Addition, subtraction and multiplication of integers can also directly translated, whereas division is not supported by Kodkod. Other supported integer expressions are constant numbers and the cardinality of a set. If we want a variable to represent an integer, we have to convert explicitly between a relation that describes the value and an integer expression (see Section 3.3).

Relational operators. Many operators that act on sets and relations can be translated easily to Kodkod. Figure 3a shows a list of operators that have a direct counterpart in Kodkod. With $\mathcal{T}(A)$ we denote the translated version of the expression A. Please note that the expressions $A \in B$ and $A \subseteq B$ are translated to the same expression in Kodkod. This is due the fact that single values are just a special case in Kodkod where a set contains just one element. The same effect can be found at the Cartesian product $(A \times B)$ and a pair $(A \mapsto B)$ and at the relational image $(A[B])$ and the function application $(A(B))$.

Other operators need a little bit more work. They can be expressed by combining other operators. Figure 3b shows a selection of such operators. In the table, we use an operator $\mathcal{A}(E)$ to denote the arity of the relation that represents the expression E. Again we can see that different operators in B (e.g. dom and prj_1) lead to the same result.

B	Kodkod
$A \in B$	$\mathcal{T}(A)\ \texttt{in}\ \mathcal{T}(B)$
$A \subseteq B$	$\mathcal{T}(A)\ \texttt{in}\ \mathcal{T}(B)$
$A \times B$	$\mathcal{T}(A)\ \texttt{->}\ \mathcal{T}(B)$
$A \mapsto B$	$\mathcal{T}(A)\ \texttt{->}\ \mathcal{T}(B)$
$A \cap B$	$\mathcal{T}(A)\ \texttt{\&}\ \mathcal{T}(B)$
$A \cup B$	$\mathcal{T}(A)\ \texttt{+}\ \mathcal{T}(B)$
$A \setminus B$	$\mathcal{T}(A)\ \texttt{-}\ \mathcal{T}(B)$
$A[B]$	$\mathcal{T}(B)\texttt{.}\mathcal{T}(A)$
$A(B)$	$\mathcal{T}(B)\texttt{.}\mathcal{T}(A)$
$A \mathbin{\lhd\!\!\!+} B$	$\mathcal{T}(A)\texttt{++}\mathcal{T}(B)$
A^{-1}	$\texttt{\~{}}\mathcal{T}(A)$
A^{+}	$\texttt{\^{}}\mathcal{T}(A)$

(a) direct translation

B	Kodkod
$\mathrm{dom}(A)$	$\texttt{prj[1:}\mathcal{A}(\alpha)\texttt{](}\mathcal{T}(A)\texttt{)}$
	with A being of type $\mathbb{P}(\alpha \times \beta)$
$\mathrm{ran}(A)$	$\texttt{prj[}\mathcal{A}(\alpha)\texttt{+1:}\mathcal{A}(A)\texttt{](}\mathcal{T}(A)\texttt{)}$
	with A being of type $\mathbb{P}(\alpha \times \beta)$
$\mathrm{prj}_1(A)$	$\mathcal{T}(\mathrm{dom}(A))$
$\mathrm{prj}_2(A)$	$\mathcal{T}(\mathrm{ran}(A))$
$A \lhd B$	$(\mathcal{T}(A)\ \texttt{->}\ \mathcal{T}(\beta))\ \texttt{\&}\ \mathcal{T}(B)$
	with B being of type $\mathbb{P}(\alpha \times \beta)$
$A \mathbin{\lhd\!\!\!-} B$	$\texttt{((univ-}\mathcal{T}(A)\texttt{)->}\mathcal{T}(\alpha)\texttt{)}\ \texttt{\&}\ \mathcal{T}(B)$
	with B being of type $\mathbb{P}(\alpha \times \beta)$
$\mathrm{bool}(P)$	$\texttt{if}\ \mathcal{T}(P)\ \texttt{then}\ \mathcal{T}(\mathrm{TRUE})\ \texttt{else}\ \mathcal{T}(\mathrm{FALSE})$
$f \in A \mathrel{+\!\!\!\to} B$	$\texttt{pfunc(}\mathcal{T}(f), \mathcal{T}(A), \mathcal{T}(B)\texttt{)}$
$f \in A \to B$	$\texttt{func(}\mathcal{T}(f), \mathcal{T}(A), \mathcal{T}(B)\texttt{)}$
$f \in A \rightarrowtail B$	$\texttt{func(}\mathcal{T}(f), \mathcal{T}(A), \mathcal{T}(B)\texttt{)}\ \texttt{\&\&}$
	$(\mathcal{T}(f)\texttt{.\~{}}\mathcal{T}(f))\ \texttt{in iden}$

(b) more complex rules

Fig. 3: Examples for translation rules

3.3 Integer handling in Kodkod

Kodkod provides only a very limited support for integers. The reason for this is twofold. Since SAT solvers are used as the underlying technology, integers are encoded by binary numbers. Operations like addition then have to be encoded as boolean formulas. This makes the use of integers ineffective and cumbersome. Another reason is that the designers of Alloy – where Kodkod has its origin – argue [8] that integers are often not very useful and an indication of lack of abstraction when modeling systems.

Our intention is to make our tool applicable to as many specifications as possible, and many of the B specifications we tried contained some integer expressions. Indeed, integers are used to model sequences in B or multi-sets in Z.

When using Kodkod with integers, we have to specify the number of bits used in integer expressions. Integer overflows are silently ignored, e.g. the sum of two large naturals can be negative when the maximum integer size is exceeded. Thus we need to ensure that we use only integers in the specified range to prevent faulty results.

Kodkod distinguishes between relational and integer expressions. An integer expression is for example a constant integer or the sum of two integer expressions. Comparison of integer expressions like "less than" is also supported. In case we want a relation (i.e. a variable) that represents an integer, we must first assign values to some atoms. Figure 4 shows an example with an universe consisting of 9 atoms $i_0, \ldots, i_8$ that represent integers.

We have basically two options when we want represent integers by a relation:

					binary numbers					
	for integer sets									
atom …	i_0	i_1	i_2	i_3	i_4	i_5	i_6	i_7	i_8	…
associated integer value …	-1	0	3	5	1	2	4	8	-16	…

Fig. 4: Mapping atoms to integer values

- We can represent sets of integers in the interval $a \mathinner{..} b$ by having an atom for each number in $a \mathinner{..} b$. Then the relation simply represents the integers of its atoms.
  E.g. with the universe in Figure 4, we can represent arbitrary subsets of $-1 \mathinner{..} 5$ by using a relation that is bounded to the atoms $i_0, \ldots, i_4$.
  The downside of this approach is that the number of atoms can become easily very large.
- Single integers can be represented more compactly by using a binary number. E.g. with the universe in Figure 4, we can represent a number of the interval $-16 \mathinner{..} 15$ by using a relation that is bounded to the atoms $i_4 \mathinner{..} i_8$. A relation that consists of the atoms i_4, i_6, i_8 would represent the sum $1 + 4 + (-16) = -9$. Kodkod provides an operator to summarise the atoms of a relation, yielding an integer expression.
  With this approach large numbers can be handled easily. The downside is that we cannot represent sets of numbers.

The atoms in the universe seen in Figure 4 are ordered in a way that we can use both approaches to represent integers in the same specification.

It can be seen that we need an exact knowledge of the possible size of integer expressions in the specification. To get the required information, a static analysis is applied to the specification before the translation. See below in Section 3.4 for details of the analysis.

Another problem that arises from having two kinds of integer representations, is that we have to ensure the consistency of formulas that use integer expressions. We briefly describe the problem in Section 3.5.

3.4 Predicate Analysis

In case that integers are used in the model, we need to know how large they can get in order to translate the expressions. To get this information we apply a static analysis on the given problem.

The first step of the analysis is that we create a graph that describes a constraint problem. For each expression in the abstract syntax tree, we create some of nodes depending on the expression's type that contain relevant information associated to the syntax node. E.g. this might be the possible interval for integers, the interval of the cardinality for sets or the interval in which all elements of a set lie.

By applying pattern matching on the syntax tree, we add rules that describe the flow of information in the graph. E.g. if we have a predicate $A \subset B$, we can propagate all information about elements of B to nodes that contain information about A. We evaluate all rules until a fixpoint or a maximum number of evaluation steps is reached.

Example. Let's take the predicate $A \subseteq 3\mathinner{..}6 \wedge card(A) > 1$. For each integer node (1, 3, 6, $card(A)$) we create a node containing the integer range. For each of the sets A and $3\mathinner{..}6$ we create two nodes: One contains the range of the set's cardinality, the other describes the integer range of the set's elements. Figure 5 shows the resulting graph. In the upper part of each node the expression is shown, in the lower part the kind of information that is stored in the node. The edges without any labels denote rules that pass information just from one node to another. Those which are labeled with $\leq, \geq, <, >$ express a relation between integer ranges of the source and target node. There is a special rule marked $i \rightarrow c$ ("interval to cardinality") which deduces a maximum cardinality from the allowed integers in a set. E.g. if all elements of a set I are in the range $x\mathinner{..}y$, we know that $card(I) \leq y - x + 1$. The graph in Figure 5 shows the information we have about each node after the analysis. In particular, we know the bounds of all integer expressions.

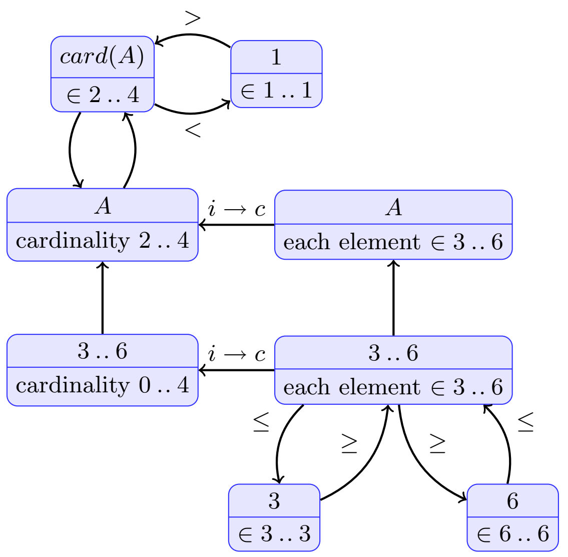

Fig. 5: Constraint system for $A \subseteq 3\mathinner{..}6 \wedge card(A) > 1$

Currently the analysis is limited to integer intervals and cardinality, because this was the concrete use case given by our translation to Kodkod. We plan to re-use the analysis for other aspects of ProB. E.g. if we can deduce the interval of a quantified integer variable, ProB can limit the enumeration of values to that range if it must test a predicate for all possible values of that variable.

Other types for information nodes are also of interest. For instance, we could infer the information if an expression is a function or sequence to assist ProB when evaluating predicates.

3.5 Integer representations

We have seen above in Section 3.3 that we have two distinct forms of representing integers. Additionally we have Kodkod's integer expressions when we want to compare, add, subtract or multiply them. During the translation process we must ensure that the correct representation is chosen for each expression and that the representations are consistently used. If the take a simple equality $A = B$ with A, B being integers as an example, we must ensure that both sides use the same representation.

Internally, the result of the translation is an abstract syntax tree that describes the formula. Some expressions in this syntax tree have annotations about the needed integer representation. E.g. if the original expression in B is a set of integers, it has the annotation that the integer set representation and not the the binary number representation must be used. We now impose a kind of type checking on this syntax tree to infer if conversions between different integer representations have to be inserted in the formula.

3.6 Extent of the translation: Partitioning

Theoretically we can take any translatable sub-predicate of a specification and replace it with a call to Kodkod. But usually, the overhead due to the communication between the processes can easily get so large that incorporating Kodkod has no advantage over using ProB alone.

A more sensible approach for a specification that is a conjunction of predicates $P_1 \wedge \ldots \wedge P_n$ is to apply the translation to every P_i. All translatable predicates are then replaced by one single call to Kodkod. But even here we made the experience that the communication overhead can become large if not all predicates are translated.

Our current approach is to create a partition of the predicates $P_1, \ldots, P_n$. Two predicates are then in the same set of the partition if they both use the same variable. We translate only complete partitions to keep the communication overhead small. There is one exception: We do not translate simple equations where one side is a variable and the other side an easy to compute constant. Such deterministic equations are computed first by the constraint solver, so the value for such a variable will be computed before the call to Kodkod is made. This keeps the translated formula small even for a large amount of data.

4 Experiments

We have chosen a number of problems to compare the performance of ProB's constraint solving technique and Kodkod's SAT solving approach. We have only used problems that can be completely translated. The results can be seen in Table 1.[1] All experiments where conducted on a dual-core Intel i7 2.8 GHz processor running under Linux. MiniSat was used as a SAT solver. The measured times do not contain time for starting up ProB and for loading, parsing and type-checking the model. We measured the time to compute all solutions to each problem. For Kodkod, we measured two different times: The "total time" includes the translation of the problem, the communication between the two processes and the time needed by the solver to produce solutions. The "solver time" is the time that the Kodkod solver itself needs to find solutions, without the overhead of translation and communication between the processes.

	ProB	Kodkod	
Model		total	solver
Who Killed Agatha?	177	123	12
Crew Allocation	timeout*	297	112
20–Queens	110	8223	8076
Graph Colouring (integer sets)	50	2323	1859
Graph Colouring (enumerated sets)	50	1037	818
Graph Isomorphism	13	553	379
Loop detection in control flow	23037	117	12
SAT instance	11830	4143	588
Send More Money	7	1773	1578
Eratosthenes' sieve (1 step)	7	5833	5712
Union of two sets (2000 elements)	33	4880	4659

*: interrupted after 60 seconds

Table 1: Comparing ProB and Kodkod (in milliseconds)

4.1 Analysis

ProB *enumerates relations* Let's have a look at those problems where Kodkod is much faster than ProB. These are "crew allocation" and "loop detection". In both problems we search for instances for sets or relations and the problem is described by relational operators and universal quantification. In this scenario, ProB basically starts to enumerate possible instances for the sets or relations which leads to a dramatic decrease of performance.

Arithmetic and large relations The arithmetic problems ("Send More Money", "Eratosthenes' Sieve") are solved by ProB much faster then by Kodkod. The

[1] The source code of the examples are available in the technical report at: `http://www.stups.uni-duesseldorf.de/w/Special:Publication/PlaggeLeuschel_Kodkod2012`.

first two problems deal with arithmetic. ProB uses internally a very efficient finite domain solver (CLP/FD) to tackle such problems. On the other side is arithmetic one of the weaknesses of Kodkod, as we already pointed out.

Kodkod does not seem to scale well when encountering large relations (e.g. "union of two sets"). This has only been relevant for certain applications of ProB, such as the property verification on real data [13].

The graph colouring, graph isomorphism and 20-Queens problems are clearly faster solved by ProB. The structure of the problem is somehow fixed (by having e.g. total functions) and constraint propagation is very effective.

Room for optimization It can be seen that the graph colouring problem needs less then half the time when it is encoded with enumerated sets instead of integers. This indicates that the translation is not yet as effective as it should be. For the "SAT" problem, the translation and communication takes six time as long as the computation of the problem itself. This shows that we should investigate if we can optimize the communication.

4.2 SMT and other tools

Very recently, an Event-B to SMT-Lib converter has become available for the Rodin platform [4]. This makes it possible to use SMT solvers (such as veriT, CVC and Z3 [3]; we used version 3.2 of the latter within the Rodin SMT Solvers Plug-in 0.8.0r14169 in our experiments below) on Event-B proof obligations. We have experimented with the translator on the examples from Table 1.[2] This is done by adding a theorem `1=2` to the model: this generates an unprovable proof obligation which in turn produces a satisfiable SMT formula encoding the problem. The SAT problem was solved quicky, in 0.135 seconds; the time to generate the SMT-lib translation has not been measured. For "Send More Money" from Table 1 Z3 initially reported "unknown". After rewriting the model (making the inequalities explicit), Z3 was able to determine the solution after about 0.250 seconds. It is thus faster than Kodkod, but still slower than ProB. Surprisingly, Z3 was unable to solve most of the other examples, such as the "Who killed Agatha" example, the "Set Union" example or the "Graph Colouring" example. Similarly, for the CrewAllocation example, Z3 was unable to find a solution already for three flights.[3] As such, the current SMT translators are not yet powerful enough to solve the constraints we aim to attack with our Kodkod translation. Furthermore, for the constraint solving tasks related to deadlock checking, Z3 was not able to solve the translations of the simpler examples from [7]. It is too early for a conclusive result, but it seems that more work needs to be put into the B to SMT translator for this approach to be useful for model finding, animation or constraint-based checking.

Other tools for B are AnimB [17], Brama and BZTT [11]. They all have much weaker constraint-solving capabilities; see [13, 14] and are unable to solve most of

[2] Apart from "loop" which cannot be easily translated to Event-B due to the use of transitive closure.

[3] ProB solves this version in 0.06 seconds; Table 1 contains the problem for 20 flights.

the problems in Table 1. Another tool is TLC [24] for TLA+. It is very good at model checking, but constraints are solved by pure enumeration. As such, TLC is unable to solve, e.g., a 20 variable SAT problem, the NQueens problem for N>9 and takes e.g. more than 2 hours for a variation of the graph isomorphism problem from Table 1.

5 More Related Work, Discussion and Conclusion

5.1 Alternative Approaches

Before starting our translation to Kodkod, we had experimented with several other alternate approaches to solve constraints in ProB. [22] offers the user a Datalog-like language that aims to support program analysis. It uses BDDs to represent relations and compute queries on these relations. In particular, one has to represent a state of the model as a bit-vector and events have to be implemented as relations between two of those bit-vectors. These relations have to be constructed by creating BDDs directly with the underlying BDD library (JavaBDD) and storing them into a file. Soon after starting experimenting with bddbddb it became apparent that due to the lack of more abstract data types than bit vectors, the complexity of a direct translation from B to bddbddb was too high, even for small models, and this avenue was abandoned.

SAL [20] is a model-checking framework combining a range of tools for reasoning about systems. The SAL tool suite includes a state of the art symbolic (BDD-based) and bounded (SAT-based) model checkers. Some first results were encouraging for a small subset of the Event-B language, but the gap between B and SAL turned out to be too big in general and no realistic way was found to handle important B operators.[4] More details about these experiments can be found in [19]. For Z, there is an ongoing attempt to use SAL for model checking Z specifications [6, 5]. The examples presented in [6, 5] are still relatively simple and pose no serious challenge in constraint solving. As the system is not publicly available, it is unclear how it will scale to more complicated specifications and constraints.

5.2 More Related Work

The first hand-translation of B to Alloy was undertaken in [18]. The paper [16] contains first experiments in translating Event-B to Alloy; but the work was also not pursued. Later, [15] presented a prototype Z to Alloy converter. The current status of this system is available at the website `http://homepages.ecs.vuw.ac.nz/~petra/zoy/`; the applicability seems limited by the lack of type inference and limited support for schemas. In contrast to these works, we translate directly to Kodkod and have a fully developed system, covering large subsets of B, Event-B, Z and TLA$^+$ and delegating the rest to the ProB kernel.

[4] Private communication from Alexei Iliasov and Ilya Lopatkin, March 6th, 2012.

A related system that translates a high-level logic language based on inductive definitions to SAT is IDP [23]. Another recent addition is Formula from Microsoft [9], which translates to the SMT solver Z3 [3].

5.3 Future Work

Currently our translation is only applicable for finding constraints satisfying the axioms as well as for constraint based deadlock checking. We are, however, working to also make it available for computing enabled events as well as for more general constraint-based testing and invariant checking.

Another avenue is to enlarge the area of applicability to some recurrent patterns of higher-order predicates. For example, many B specifications use total functions of the type `f : DOM --> POW(RAN)` which cannot be translated as such to Kodkod. However, such functions can often be translated to relations of the form `fr : DOM <-> RAN` by adapting the predicates accordingly (e.g., translating `f(x)` to `fr[{x}]`). More work is also needed on deciding automatically when to attempt the Kodkod translation and when predicates should be better left to the existing ProB kernel. Finally, inspired by the experiments, we also plan to improve the ProB kernel for better solving constraints over relational operators such as composition and closure.

5.4 Conclusion

After about three years or works and several attempts our translation to Kodkod is now mature enough to be put into practice and has been integrated into the latest version of the ProB toolset. The development required a considerable number of subsidiary techniques to be implemented. As our experiments have shown, the translation can be highly beneficial for certain kinds of constraints, and as such opens up new ways to analyze and validate formal specifications in B, Z and TLA$^+$. However, the experiments have also shown that the constraint logic programming approach of ProB can be superior in a considerable number of scenarios; the translation to Kodkod and down to SAT is not (yet) the panacea. The same can be said of the existing translations from B to SMT. As such, we believe that much more research required to reap the best of both worlds (SAT/SMT and constraint programming). An interesting side-effect of our work is that the ProB toolset now provides a double-chain (relying on technology developed independently and using different programming languages and paradigms) of validation for first-order predicates, which should prove relevant in high safety integrity level contexts.

References

1. J.-R. Abrial. *The B-Book.* Cambridge University Press, 1996.
2. A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers. Principles, Techniques, and Tools (Second Edition).* Addison Wesley, 2007.

3. L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In C. R. Ramakrishnan and J. Rehof, editors, *TACAS*, LNCS 4963, pages 337–340. Springer, 2008.
4. D. Deharbe, P. Fontaine, Y. Guyot, and L. Voisin. SMT solvers for Rodin. In *Proceedings ABZ'2012*, LNCS. Springer. to appear.
5. J. Derrick, S. North, and A. Simons. Z2SAL: a translation-based model checker for Z. *Formal Aspects of Computing*, 23(1):43–71, 2011.
6. J. Derrick, S. North, and A. J. H. Simons. Z2SAL - building a model checker for Z. In E. Börger, M. Butler, J. P. Bowen, and P. Boca, editors, *Proceedings ABZ 2008*, LNCS 5238, pages 280–293, 2008.
7. S. Hallerstede and M. Leuschel. Constraint-based deadlock checking of high-level specifications. *TPLP*, 11(4–5):767–782, 2011.
8. D. Jackson. Alloy: A lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology*, 11:256–290, 2002.
9. E. K. Jackson, T. Levendovszky, and D. Balasubramanian. Reasoning about metamodeling with formal specifications and automatic proofs. In J. Whittle, T. Clark, and T. Kühne, editors, *MoDELS*, LNCS 6981, pages 653–667. Springer, 2011.
10. L. Lamport. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.
11. B. Legeard, F. Peureux, and M. Utting. Automated boundary testing from Z and B. In L.-H. Eriksson and P. Lindsay, editors, *Proceedings FME'02*, LNCS 2391, pages 21–40. Springer-Verlag, 2002.
12. 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.
13. M. Leuschel, J. Falampin, F. Fritz, and D. Plagge. Automated property verification for large scale B models. In A. Cavalcanti and D. Dams, editors, *Proceedings FM 2009*, LNCS 5850, pages 708–723. Springer-Verlag, 2009.
14. M. Leuschel, J. Falampin, F. Fritz, and D. Plagge. Automated property verification for large scale B models with ProB. *Formal Asp. Comput.*, 23(6):683–709, 2011.
15. P. Malik, L. Groves, and C. Lenihan. Translating Z to Alloy. In M. Frappier, U. Glässer, S. Khurshid, R. Laleau, and S. Reeves, editors, *ABZ'2010*, LNCS 5977, pages 377–390. Springer, 2010.
16. P. J. Matos and J. Marques-Silva. Model checking Event-B by encoding into Alloy. In *ABZ*, LNCS 5238, page 346, 2008.
17. C. Métayer. *AnimB 0.1.1*, 2010. Available at `http://wiki.event-b.org/index.php/AnimB`.
18. L. Mikhailov and M. J. Butler. An approach to combining B and Alloy. In D. Bert, J. P. Bowen, M. C. Henson, and K. Robinson, editors, *ZB*, LNCS 2272, pages 140–161. Springer, 2002.
19. D. Plagge, M. Leuschel, I. Lopatkin, and A. Romanovsky. SAL, Kodkod, and BDDs for Validation of B Models. Lessons and Outlook. *Proceedings AFM 2009*, pages 16–22, 2009.
20. Symbolic Analysis Laboratory (SAL) website. `http://sal.csl.sri.com/`.
21. E. Torlak and D. Jackson. Kodkod: A relational model finder. In O. Grumberg and M. Huth, editors, *Proceedings TACAS'07*, LNCS 4424, pages 632–647. Springer-Verlag, 2007.
22. J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In W. Pugh and C. Chambers, editors, *Proceedings PLDI'04*, pages 131–144, New York, NY, USA, 2004. ACM Press.
23. J. Wittocx, M. Mariën, and M. Denecker. Grounding FO and FO(ID) with bounds. *J. Artif. Intell. Res. (JAIR)*, 38:223–269, 2010.

24. Y. Yu, P. Manolios, and L. Lamport. Model checking TLA$^+$ specifications. In L. Pierre and T. Kropf, editors, *Proceedings CHARME'99*, LNCS 1703, pages 54–66. Springer-Verlag, 1999.

A The Models used in the Experiments

A.1 CrewAllocation

```
MACHINE CrewAllocationConstantsLarge
DEFINITIONS
 NRF == 20;
 FLIGHTS == 1..NRF;
 CONSTR1 == (!f.(f:FLIGHTS => speaks[assign[{f}]] = LANGUAGE));
  /* all languages must be represented on all flights */

 CONSTR2 == (!f.(f:FLIGHTS => male[assign[{f}]] = BOOL));
  /* both sexes must be on all flights */

 CONSTR3 ==
  (!(f,p).(f:FLIGHTS & f<NRF-1 & p:PERSONNEL &  f|->p:assign & (f+1)|->p:assign
                => (f+2)|->p /: assign));

 CONSTR4 == (ran(assign) = PERSONNEL);

 SET_PREF_MAXINT == 255;
 SET_PREF_MAX_INITIALISATIONS == 1
SETS
 PERSONNEL = {tom, david, jeremy, carol, janet, tracy};
 LANGUAGE = {french,german,spanish}
CONSTANTS male, speaks, assign
PROPERTIES
  male : PERSONNEL --> BOOL &
  speaks : PERSONNEL <-> LANGUAGE &
  ran(male) = BOOL &
  ran(speaks) = LANGUAGE &

  male = { tom|->TRUE, david|->TRUE, jeremy|->TRUE,
           carol|->FALSE, janet|->FALSE, tracy|->FALSE} &
  speaks = { tom|->german, david |-> french, jeremy |-> german,
             carol |-> spanish, janet |-> french, tracy |-> spanish }
 &
  assign: FLIGHTS <-> PERSONNEL
  & CONSTR1 & CONSTR2 & CONSTR3 & CONSTR4
ASSERTIONS
  CONSTR1; CONSTR2; CONSTR3; CONSTR4
END
```

A.2 Send More Money

```
MACHINE SendMoreMoney
/* Solution found when clpfd_interface is used in < 0.01 seconds;
                     without it: 32.67 seconds to find solution */
CONSTANTS S,E,N,D, M,O,R, Y
PROPERTIES
```

```
   S : 1..9 & E : 0..9 & N : 0..9 &
   D : 0..9 & M : 1..9 & O : 0..9 &
   R : 0..9 & Y : 0..9 &
   S >0 & M >0 &
   card({S,E,N,D, M,O,R, Y}) = 8 &
   S*1000 + E*100 + N*10 + D +
   M*1000 + O*100 + R*10 + E =
   M*10000 + O*1000 + N*100 + E*10 + Y
OPERATIONS
  r <-- GetS = BEGIN r := S END;

  s,e,n,d, m,o,r, y <-- GetSol =
    BEGIN s,e,n,d, m,o,r, y := S,E,N,D, M,O,R, Y END
END
```

A.3 Who killed Agatha

```
MACHINE WhoKilledAgatha
/*
$ Who killed agatha? (The Dreadsbury Mansion Murder Mystery) in Essence'.
$
$ This is a standard benchmark for theorem proving.
$
$ http://www.lsv.ens-cachan.fr/~goubault/H1.dist/H1.1/Doc/h1003.html
$
$ Someone in Dreadsbury Mansion killed Aunt Agatha.
$ Agatha, the butler, and Charles live in Dreadsbury Mansion, and
$ are the only ones to live there. A killer always hates, and is no
$ richer than his victim. Charles hates noone that Agatha hates. Agatha
$ hates everybody except the butler. The butler hates everyone not richer
$ than Aunt Agatha. The butler hates everyone whom Agatha hates.
$ Noone hates everyone. Who killed Agatha?
*/
SETS
 Persons = {Agatha, butler, Charles}
CONSTANTS hates, richer, killer /* Typing : 786,432 candidate solutions */
PROPERTIES
 hates : Persons <-> Persons &
 richer : Persons <-> Persons &  richer /\ richer~ = {} &
 richer /\ id(Persons) = {} &
 !(x,y,z).(x|->y:richer & y|->z:richer => x|->z:richer) &
 !(x,y).(x:Persons & y:Persons & x/=y => (x|->y:richer <=> y|->x /: richer)) &

 killer : Persons &
    /* killer /= Agatha & */
 killer|->Agatha : hates &
   /* A killer always hates his victim */
 killer|->Agatha /: richer &
   /* and is no richer than his victim */
 hates[{Agatha}] /\ hates[{Charles}] = {} &
   /* Charles hates noone that Agatha hates. */
 hates[{Agatha}] = Persons - {butler} &
   /* Agatha hates everybody except the butler. */
 !x.( x|->Agatha /: richer => butler|->x : hates) &
   /* The butler hates everyone not richer than Aunt Agatha */
 hates[{Agatha}] <: hates[{butler}] &
   /* The butler hates everyone whom Agatha hates.  */
 !x.(x:Persons => hates[{x}] /= Persons)
   /* Noone hates everyone. */
END
```

A.4 Union of two sets

```
MACHINE SimpleComputation
CONSTANTS limit, a, b, x
PROPERTIES
 a = 1..limit & b = (limit+2 .. limit+limit) &
 x = a \/ b &

 limit = 1000
ASSERTIONS
 card(x) = 2*limit-1
OPERATIONS
 c <-- GetCard = c:=card(x)
END
```

A.5 Loop detection

```
MACHINE Loop
SETS
 Blocks={b1,b2,b3,b4,b5,b6,entry,exit}
CONSTANTS succs, lentry, L
PROPERTIES
 succs: Blocks <-> Blocks &
 succs = {entry |-> b1, b1|-> b2, b2|-> b3, b3 |-> b3, b3|-> b4,
         b4 |-> b2, b4 |-> b5, b5 |-> b6, b6 |-> b6, b6 |-> exit}
          /* Figure 8.9, page 530 of DragonBook */
 & L <: Blocks
 & lentry : L
 & succs~[L-{lentry}] <: L
 & !l.(l:Blocks /\ L => lentry : closure1(L <| succs |> L)[{l}])
END
```

A.6 Sat Problem

This the SATLib example flat200-90 translated to B.

The header of the cnf file is as follows:

```
c File: flat200_3_0.col
c
c SOURCE: Joseph Culberson (joe@cs.ualberta.ca)
c DESCRIPTION: Quasi-random coloring problem
c              generated with flatness = 0
c                             probability = 0.036
c                             known coloring = 3
c                             random seed = 1090
c              Creation Date:  Wed Feb 24 10:40:19 1999
c Maximum degree = 8 Minimum degree = 1
c COLOR VERIFICATION: Using the permuted order
c under simple greedy yields the specified
c coloring number
c Color = 3 specified partitions = 3
c p edge 200 479
c cnf created by edge2cnf
p cnf 600 2237
```

The B encoding was generated automatically from that file:

```
MACHINE SAT
/* CNF Translated to B */
/* 600 Boolean variables , 2237 Clauses */
CONSTANTS
   x1, x2, x3, x4, x5, x6, x7, x8, x9, x10,
   ...
PROPERTIES
   x1:BOOL & x2:BOOL & x3:BOOL & x4:BOOL & x5:BOOL &
   ...
   x596:BOOL & x597:BOOL & x598:BOOL & x599:BOOL & x600:BOOL &

    (  x1=FALSE  or  x2=FALSE  ) &
   (  x1=FALSE  or  x3=FALSE  ) &
   (  x2=FALSE  or  x3=FALSE  ) &
   (  x1=TRUE  or  x2=TRUE  or  x3=TRUE  ) &
   (  x4=FALSE  or  x5=FALSE  ) &
   (  x4=FALSE  or  x6=FALSE  ) &
   (  x5=FALSE  or  x6=FALSE  ) &
   (  x4=TRUE  or  x5=TRUE  or  x6=TRUE  ) &
   ...
   (  x600=FALSE  or  x597=FALSE  ) &
 1=1
END
```

A.7 Graph Colouring

```
MACHINE GraphColouringNAT
/* The graph colouring problem; adapting examples delivered with idp */
/* http://dtai.cs.kuleuven.be/krr/software/ */
DEFINITIONS
```

```
 "Graph_small_40_200_0.def"
CONSTANTS Vertexes, Edges, maxnocol, colour
PROPERTIES
 Vertexes = Vtx & /* get data from .def file */
 Edges = Edge &  /* get data from .def file */
 Edges: Vertexes <-> Vertexes &
 maxnocol : 1..6 &  /* Note: in contrast to the IPD example we vary the number of
   colours allowed; we thus find a solution with 5 colours first */
 colour: Vertexes --> 1..maxnocol &
 !(x,y).(x|->y : Edges => colour(x) /= colour(y)) &
 colour(1) = 1
OPERATIONS /* just to get the colour of the nodes; not really necessary */
  c <-- Get(yy) = PRE yy:Vertexes THEN c:= colour(yy) END
END
```

The file `Graph_small_40_200_0.def` contains a graph with 40 vertices and 200 edges:

```
DEFINITIONS
Vtx == 1..40;
Edge == {13|->3, 26|->14, 19|->24, 2|->13, 1|->18, 10|->20, 15|->8,
 9|->8, 4|->19, 7|->3, 23|->27, 9|->7, 23|->2, 29|->17, 35|->14,
 8|->1, 27|->2, 9|->36, 34|->26, 7|->26, 34|->23, 14|->20, 39|->3,
 7|->31, 5|->21, 23|->11, 35|->30, 6|->11, 9|->28, 18|->30, 19|->33
 25|->28, 39|->4, 2|->19, 9|->24, 8|->26, 10|->6, 22|->25, 32|->25,
 23|->40, 30|->17, 2|->4, 5|->24, 27|->36, 20|->38, 31|->38, 35|->9,
 19|->5, 8|->16, 18|->15, 35|->4, 19|->21, 15|->37, 34|->24, 6|->8,
 8|->36, 2|->1, 23|->13, 13|->35, 36|->25, 26|->20, 32|->36, 24|->2,
 9|->17, 38|->27, 18|->38, 36|->20, 34|->32, 8|->5, 5|->1, 28|->7,
 33|->8, 5|->22, 31|->9, 30|->40, 26|->33, 32|->1, 6|->19, 14|->5,
 8|->18, 40|->22, 4|->5, 5|->13, 34|->40, 12|->15, 25|->14, 3|->35,
 10|->23, 18|->26, 31|->15, 13|->38, 13|->18, 20|->22, 18|->9, 11|->13,
 40|->25, 40|->5, 28|->20, 37|->28, 3|->26, 38|->4, 3|->12, 5|->6, 30|->26,
 32|->26, 7|->17, 31|->32, 22|->37, 38|->26, 3|->23, 34|->3, 6|->35, 34|->30,
 23|->4, 23|->15, 10|->17, 12|->37, 40|->37, 28|->34, 38|->5, 16|->29, 5|->25,
 21|->30, 37|->39, 32|->7, 7|->13, 15|->20, 39|->13, 26|->36, 18|->12, 4|->6,
 21|->39, 21|->7, 29|->36, 11|->21, 20|->11, 22|->36, 24|->23, 38|->24, 4|->10,
 20|->23, 38|->36, 16|->23, 12|->30, 17|->6, 29|->10, 10|->31, 7|->37, 40|->19,
 27|->18, 12|->16, 6|->7, 8|->30, 25|->27, 38|->21, 27|->31, 4|->31, 5|->9,
 23|->29, 35|->8, 11|->27, 17|->21, 26|->37, 3|->6, 5|->27, 9|->6, 26|->27,
 5|->12, 14|->30, 35|->29, 10|->11, 38|->8, 36|->28, 1|->14, 31|->37, 13|->34,
 26|->2, 12|->7, 34|->5, 3|->19, 15|->16, 20|->39, 19|->10, 12|->23, 6|->30,
 11|->2, 25|->34, 24|->10, 40|->38, 24|->13, 35|->37, 37|->2, 33|->2, 31|->22,
 15|->11, 22|->29, 9|->34, 34|->8, 17|->12, 1|->29}
```

The enumerated sets version of the model simply uses an enumerated set for the vertexes rather than the interval `1..40`:

```
 SETS
 Vertexes = {e1,e2,e3,e4,e5,e6,e7,e8,e9,e10,e11,e12,e13,e14,e15,e16,e17,e18,e19,
        e20,e21,e22,e23,e24,e25,e26,e27,e28,e29,e30,e31,e32,e33,e34,e35,e36,
        e37,e38,e39,e40}
```

The rest of the model is adapted accordingly.

A.8 NQueens 20

```
MACHINE NQueens
CONSTANTS n,queens
PROPERTIES
 n = 20 &
 queens : 1..n >-> 1..n /* for each column the row in which the queen is in */
 &
 !(q1,q2).(q1:1..n & q2:2..n & q2>q1
    => queens(q1)+(q2-q1) /= queens(q2) & queens(q1)+(q1-q2) /= queens(q2))
DEFINITIONS
      SET_PREF_MAX_INITIALISATIONS == 10;
      SET_PREF_MAX_OPERATIONS == 10;
OPERATIONS
  r<--Get(yy) = PRE yy:1..n THEN r:= queens(yy) END
END
```

A.9 Graph Isomorphism

```
MACHINE CheckGraphIsomorphism2
/* A machine where we use ProB to check if two graphs are isomorphic */
/* Note: number of permutations: 10! = 3,628,800 */
SETS
  Nodes = {a,b,c,d,e, x,y,z,v,u}
DEFINITIONS
 G1 == {a|->b,a|->c,a|->d ,  b|->c, b|->d, c|->e, d|->e};
 G2 == {x|->v, x|->u, x|->z, y|->v, y|->u, z|->v, z|->u}
CONSTANTS graph1, graph2, relevant
PROPERTIES
 graph1: Nodes <-> Nodes &
 graph2: Nodes <-> Nodes &
 graph1 = G1\/G1~ &
 graph2 = G2 \/ G2~ &
 relevant = (dom(graph1)\/dom(graph2)\/ran(graph1)\/ran(graph2)) &
 #p.(p: relevant >->>relevant &
    !(x,y).(x:relevant & y:relevant =>
              (x|->y:graph1 <=> p(x)|->p(y) : graph2)))
END
```

A.10 Eratosthenes' sieve

```
MACHINE SieveStep1
/* First step of Sieve Algorithm */
CONSTANTS all, odd_plus2, limit, cur
```

```
PROPERTIES
 all = 2..limit &
 cur = 2 &
 odd_plus2 = all - {n|n:3..limit & #y.(y:2..limit & n=y+y)} &
 limit = 200
OPERATIONS
 c <-- GetCard = c := card(odd_plus2)
END
```