Heinrich Heine
Universität
Düsseldorf

INSTITUT FÜR INFORMATIK
Softwaretechnik und Programmiersprachen
Universitätsstr. 1 D–40225 Düsseldorf

Boolean Encoding of Statically Finite Sets in B Machines

Bachelorarbeit

im Studiengang Informatik
zur Erlangung des akademischen Grades

Bachelor of Science (B.Sc.)

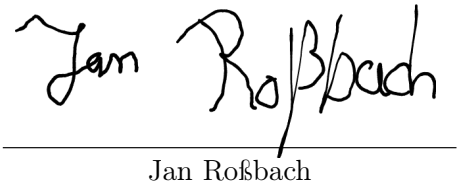
vorgelegt von
Jan Roßbach

Beginn der Arbeit: 13. Dezember 2021
Abgabe der Arbeit: 14. März 2022

Erstgutachter: Prof. Dr. Michael Leuschel
Zweitgutachter: Prof. Dr. Stefan Conrad

Selbstständigkeitserklärung

Hiermit versichere ich die vorliegende Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.



Düsseldorf, den 14. März 2022

Jan Roßbach

Abstract

In classical B machines, manual data refinement of high-level constructs, such as set and relational variables, into low-level boolean variables, is a tedious process.

This thesis provides a method for automating such refinement, that uses intermediate, fully expanded, predicate matrices to represent set expressions, similar to other relational model finders, such as KodKod.

The goal of this translation, is to provide a more fine-grained machine, which might work better in static analysis before the ProB partial order reduction algorithm. This is because it might uncover new independence relations, hidden in the high-level nature of the B language.

An implementation prototype for this, that uses Clojure and the *lisb* library, is also provided and analyzed for performance and practicality.

We find that the method is feasible for specific data types, but performance sensitive to set size and operation parameter count.

Acknowledgements

I want to thank my advisor Philipp Körner, the God of LaTeX. His passion for teaching and academic achievement is a genuine inspiration.

I also want to thank my mother for being patient and always believing in me, when nobody else would.

Contents

1 Introduction

The B-Method [ALN+91], and its successor Event-B [Abr10] are formal methods that allow one to model and specify requirements for a given software system in very high-level terms, such as set theory and first order logic. They can guarantee adherence to those requirements for the final implementation, either by formal mathematical proof, or via model checking. This method finds its use mainly in safety critical environments, such as railway systems [BKK+20].

A problem, that these systems face regularly, is that of state space explosion. During model checking, the state space is searched for potential errors, but even for relatively small systems, the amount of potential program states can get too large for a computer to check in a reasonable amount of time [Val98].

One method trying to alleviate this issue is partial order reduction (POR) [Pel93]. It attempts to reduce the state space by exploiting symmetries in the interleaving of operations. And while the method has proven to be very effective for low-level formalisms, such results have yet to be achieved in B machines [DL14, DL16, Dob17].

This might be, because the high-level constructs B provides, such as non-determinism, sets, relations and sequences, hide independence relations between operations. If that is the case, then that refining the high-level constructs in the machine, might decrease computation time and time-out for POR.

To check this hypothesis, one would need to refine the high-level B data structures into low-level SAT logic, and check for improvements of the algorithm in the resulting machine. But manual refinement, especially this kind, can be exceedingly tedious on larger scales.

For this reason, we propose an algorithm for automatically translating finite high-level relational expressions in B machines, into a mostly propositional SAT encoding, with boolean variables that act as a semantic bit vectors for set type variables.

We also provide an implementation of the algorithm, which is done using the Clojure Programming Language [Hic08] and *lisb*, a Clojure interface for B machines that builds on the classical B animator and model checker ProB [LB03, LB08].

In Section 2, we look at related work, particularly related translations from set expressions into boolean representations. Then, in Section 3, we will go over the theoretical details of the translation and potential limitations that come from that.

In Section 4, we will look at the details of how the system has been implemented and mistakes in direction that were made along the way. Then, we will draw conclusions from our findings in Section 5.

2 Related Work

In this section we elaborate more on the scientific context, the thesis is embedded in and how such translation might be useful in practice.

2.1 The B Language

The idea behind the B method, is to model system operations in abstract mathematical terms and refine them into a specific functional implementation. Each step is validated either by formal proof of correctness or by model checking. While model checking is the far more convenient method to use in practice, due to being fully automated, it has very practical limitations.

There mainly two different B dialects in use in the industry. Classical B, which is mainly used for software-development and Event-B, which more focuses on system modeling.

The dialect that is used in this work, is the classical B derivative used by the ProB tool [1].

The state explosion problem [CKNZ12] describes the fact, that even for moderately sized systems, the state-space can get very large, which can make it hard to check the entirety of it and lead to time-outs.

2.2 The ProB Animator and Model Checker

The ProB tool provides powerful model checking along with an animator, that visualizes how the system would function in action, based on the given machine. One can, for instance, traverse the state space manually or evaluate LTL formula.

While it is written in SICStus Prolog [CM12], a general Java API [KBD+20] is also provided in order to interface with other tooling, like for instance the Rodin [ABH+10] platform for Event-B, which is written in Java.

The implementation will later build on this API to, among other reasons, avoid the re-implementation of a type-checker for B.

2.3 Partial Order Reduction

There are several attempts to solve the state explosion problem present in the model checking literature. One of them is POR, which tries to leverage symmetry of transitions.

[1] Overview over the syntax can be found at https://prob.hhu.de/w/index.php?title=Summary_of_B_Syntax

Some transitions are symmetric in the sense, that they can be executed in any order without affecting the outcome. This makes parts of the state-space, namely all the permutations of the symmetric operations, redundant to check. The challenge is in finding those symmetries in the system operations.

For this purpose the ProB implementation of POR relies heavily on the comparison of the read and write sets of the operations in question.

If a conflict is found, they are automatically classed as dependent and excluded from further analysis. While being a very fast heuristic, this approach seems hastily exclude operations leading to many false positives for dependence in real world applications. This seems to be due to the fact, that most real world applications have a relatively small number of variables that most operations access.

Due to the high level nature of the B language, many variables are sets and therefore hold a high amount of application state. One operation might access one part of this state another operation a different part. But due to them being in the same variable, this would lead to a conflict.

Recently there have been new successful attempts to improve this approach by doing static analysis of the machine before POR trying to find these kinds of independence. This while effective in reducing the amount of false positives for dependence, has proven to be too slow for general use.

This work aims to aid these efforts by providing a translation from high level B expressions to lower level expressions, using more atomic variable Types, and avoiding state accumulation in few variables. This has the potential to substantially speed up the static analysis.

2.4 Other Translations

Translations from first-order formula into quantifier-free boolean formula aren't new and have been done by many, including Daniel Jackson [Jac00, Jac98] and implemented in model finding tools like Alloy [Jac02] and Kodkod [TJ07, TJ06]

A translation from ProB to Kodkod has also been presented [HL12], showing a way to encode B language constructs into Kodkod's relational constructs in order to use its model finding capabilities in the back-end. Although not all functionality was retained, the principal concept of translating B into a relational format was demonstrated. Those problems will still be significant here.

Among other significant limitations, it was shown that it is possible but difficult to encode integer variables, Here this is an even more significant limitation, because the requirement of sets to be finite makes it difficult to work with the natural numbers, an infinite set.

These works use a relational user language and do the boolean translation behind the scenes

in order to provide it to a SAT solver. The exact nature of the translation is hidden from the user and can not be used to do further calculations like feeding it to POR.

In this work, we aim to do the translation on the back end and then provide a rewritten form of the machine to the user. This sort of translation can be considered a form of automatic data refinement which has been implemented before by tools like BART [Req08].

Another difference to previous translations is, that normal boolean representations only have to be compatible with other boolean variables and logic predicates. In our translation we are still on the level of a B machine, which supports high level constructs like relations and functions.

This allows us to use classical B constructs such as if expressions to translate the boolean variables back to their original set form and provide a solid foundation for the implementation of the theory.

3 Machine Translation

In this section, we will discuss the theory behind the translation algorithm. That is, how we are going to transform the B machines into an equivalent, more fine-grained representation.

We will work in-place, meaning we will not create a refinement of the old machine, but rather an entirely new machine that has the following features.

1. An equivalent state-space to the old machine.
2. All feasible set- and relational variables are replaced by boolean variables.
3. Operations with parameters are replaced by new operations, where each new operation represents a unique parameter choice of the old operation.
4. Expressions are simplified wherever possible.

The last requirement is needed for us to get any benefits from the translation. If we had to touch all boolean variables everywhere one of the old variables was used, we would see no difference in read- and write-sets and therefor no benefits in the static analysis.

To achieve this translation, we will intro three translation functions that will be defined by tables of the throughout this chapter. Notation references can be found in the appendix.

The application of those rules will be demonstrated on short examples, for which we will work with the following classical B machine, from here on called the demo machine (see Listing 1). All references to B code will be in this context.

```
1:  MACHINE demo
2:  SETS S={s1,s2}
3:  VARIABLES a,r
4:  INVARIANT a:POW(S) & r:S<->S &
5:  a/={} &
6:  a <: dom(r) &
7:  !(x).(x:S => x : a\/dom(r))
8:  INITIALISATION a := {s1} || r := {s1|->s2,s2|->s1}
9:  OPERATIONS
10: conj(x) = SELECT x : S  THEN a := a \/ {x} END;
11: disj(x) =  SELECT x : a & a/={x} THEN a:= a \ {x} END;
12: override(x) = SELECT x:S
13:                THEN
14:                    ANY y
15:                    WHERE y:S
16:                    THEN
17:                        r := r <+ {x|->y}
18:                    END
19:                END
20: END
```

Listing 1: B Machine For Demonstration of the Algorithm (Demo Machine)

3.1 Static Context

Classical B machines, in contrast to Event-B machines, directly contain sections that introduce static identifiers into the machine.

3.1.1 Deferred and Enumerated Sets

In B, the type system contains very few primitive data types. Only boolean, whole numbers, real numbers and strings are supported. These can be combined via the power-set ($\mathbb{P}$) and the Cartesian Product, to form arbitrarily complex combinations. But this is still not very expressive.

For that purpose B provides a way to introduce new custom primitive data types via the `SETS` clause. There exist two kinds of basic sets that can be specified in the `SETS` section, enumerated sets and deferred sets. Enumerated sets are sets that list their elements at the start, and deferred sets leave this to be done at a later point, e.g. on a subsequent refinement.

Since our translation requires our sets to have a finite and known number of elements, because we actually have to create this number of new variables, we have to force deferred sets to specify their elements.

For this, we can consider them to be special cases of enumerated sets with a fixed number of elements k that have automatically generated names. We chose as a general naming scheme, the identifier of the set, followed by numbers, starting from 1.

For example, if `S` in the demo machine was a deferred set, and not an enumerated Set, the `SETS` clause after the translation with $k = 2$, would look like this.

```
SETS S={S1,S2}
```

3.1.2 Constants

For the sake of completeness, we need to mention the other two static sections, `CONSTANTS`, that introduces new identifiers, and `PROPERTIES` which types those constants and maps them to specific values that can't be changed later.

We can leave these clauses unchanged, since we can always expand the relevant set-type constants dynamically into their values and treat them in the same way as enumeration-sets, meaning a set, that explicitly lists its elements.

There are other clauses that we will not explicitly list in the rest of the section.

3.2 Variables

In B, variables are identifiers that store the state of the machine. They are assigned a fixed type in the `INVARIANTS` section. There are only two kinds of variables, that we will translate (see Item 2). Namely, subsets of finite carrier-sets and relations between finite carrier sets.

More formally, let $S = \{s_1, ..., s_n\}$ and $T = \{t_1, ..., t_m\}$ be finite carrier sets. A variable can be translated, if it has type $\mathbb{P}(S)$ for sets, or $\mathbb{P}(S \times T)$ for relations.

In case of sets, let x be a set-variables $x \in \mathbb{P}(S)$, we can translate it by replacing x with n new variables $xs_1, ..., xs_n$ of type `BOOL` that satisfy

$$\forall_i(s_i \in x \Leftrightarrow xs_i = \texttt{TRUE}), i \in \{1, ..., n\}$$

In the relational case, let y be a relational variable withe $y \in \mathbb{P}(S \times T)$, we can instead replace y with $n * m$ new variables $ys_1t_1, ..., ys_nt_m$ of type `BOOL` that satisfy

$$\forall_{i,j}(s_i \mapsto t_j \in y \Leftrightarrow ys_it_j = \texttt{TRUE}), i \in \{1, ..., n\}, j \in \{1, ..., m\}$$

This shows how the new variables are related to the old ones, meaning they would be the gluing invariant, if we made a traditional refinement to the machine. We must always keep this property to ensure Item 1.

For our demo machine that means, that the `VARIABLES` section changes to

```
VARIABLES as1,as2,rs1s1,rs1s2,rs2s1,rs2s2
```

since `a` is a set-variable and `r` a relation.

3.3 Expressions

Variables in B, form the basic building blocks of, and are exclusively used in, expressions. An expression in B is any B construct, that can be evaluated into a value. These are used in one of two contexts, namely predicates and substitutions.

In this section we will deal with both in turn. First, we look at predicates and develop, using a specific example, a way to intermediately represent set-expressions in a way that generalizes to relations and is re-usable for assignments. We will then take a brief look at assignments and show the way this is done specifically.

3.3.1 Predicates

In B predicates are clearly differentiated from expressions that return a boolean value, e.g. `TRUE` or `FALSE`, but there are ways to translate between them. One way to generate a predicate that is equivalent `TRUE` and `FALSE`, is by writing `TRUE=TRUE` ($\top$ from now on) and `TRUE=FALSE` ($\bot$) respectively. The other direction, from predicate back to value, can be achieved with the `bool` function.

For our translation, we are interested only in those predicates, that depend on the state of variables. Those can primarily be found in the invariants and operation guards, but the assertion and constraint clauses can be viewed similarly. Consider the invariants as an example. In our demo machine we can see that we have three invariant clauses. One typing and two non-typing invariants. The typing one

```
a:POW(S) & r:S<->S
```

can be replaced by the types of the new variables, e.g. `BOOL`.

```
as1:BOOL & as2 :BOOL &
rs1s1:BOOL & rs1s2:BOOL & rs2s1:BOOL & rs2s2:BOOL
```

The other two invariant predicates will serve as examples of the general predicate translation principle. Let us first consider the predicate

```
a/={}
```

As an invariant, this predicate ensures that `a` is never the empty set. So what we are saying in terms of the new boolean variables, is that they can never all equal false at the same time. So a first naive way of rewriting the predicate could be like this.

```
not(as1=FALSE & as2=FALSE)
```

While this way of rewriting the predicate is correct, it does not generalize very well. What if instead of `a` we had a more complex expression? We need some way of recursively analyze these expressions and generate a solution from that. So, let us first introduce some general theory on how we can solve this problem and then come back to this specific example, solving it again with the theory in mind.

Equality Example When we consider equality of set-expressions more generally, we can see that the predicate $A = B$ for arbitrary set-expressions A and B of the same type $S = \{s_1, ..., s_n\}$ satisfies

$$A = B \equiv \forall_i (s_i \in A \Leftrightarrow s_i \in B), i \in \{1, ..., n\}$$

Let $Pred$ be the set of all B predicates and $\vec{a} = [a_1, ..., a_n] \in Pred^n$, $\vec{b} = [b_1, ..., b_n] \in Pred^n$ two predicate vectors from $Pred^n$, the n-dimensional vector-space that is defined over $Pred$ with $\vee$ as vector addition and $\wedge$ as scalar multiplication, and that satisfy

$$\forall_i (a_i \Leftrightarrow s_i \in A), i \in \{1, ..., n\}$$

and

$$\forall_i (b_i \Leftrightarrow s_i \in B), i \in \{1, ..., n\}$$

Combining the equations yields,

$$A = B \equiv \bigwedge_{i=0}^{n} (a_i \Leftrightarrow b_i)$$

This means, that we can use these predicate vectors, for any expression of type $\mathbb{P}(S)$, to express the notion of set equality.

If we can also translate any set-expression into a predicate vector, where each predicate indicates that the element corresponding to the index is in the set, then we can use that to construct the new and equivalent equality predicate.

Formally, let $Expr$ be the set of B set expressions, and $T_e : Expr \to Pred^n$ be a function that, for any set expression X of type $S = \{s_1, ..., s_n\}$, satisfies

$$[T_e(X)]_i \Leftrightarrow s_i \in X, i \in \{1, ..., n\}$$

We can use this to define $T_p : Pred \to Pred$, the translation function for predicates, and we write

$$A = B \equiv \bigwedge_{i=1}^{n} ([T_e(A)]_i \Leftrightarrow [T_e(B)]_i) = T_p(A = B)$$

to get our final translation rule for equality.

This approach now generalizes well into arbitrary set-expressions (Table 2) and predicates (Table 1).

Now we can return to the demo example invariant

```
a\={}
```

We can now first turn both expressions into predicate vectors using Table 2. This yields

```
[as1=TRUE,as2=TRUE]
```

for `a` and

```
[TRUE=FALSE,TRUE=FALSE]
```

for the empty set. When we now combine the two, using Table 1, as

```
not(as1 <=> TRUE=FALSE & as2 <=> TRUE=FALSE)
```

Notice, that this expression simplifies (rules for simplification used throughout this work can always be found in Table 6) easily to the naive version we looked at earlier. But now we have a general principle of how to translate arbitrarily complex B predicates into new B predicates that use propositional logic over the new boolean-variables.

Relations This approach naturally extends to relational expressions. Let us consider the next example invariant in the demo machine.

```
a <: dom(r)
```

A naive translation for this rule could be

```
(as1=TRUE => (rs1s1=TRUE or rs1s2=TRUE)) &
(as2=TRUE => (rs2s1=TRUE or rs2s2=TRUE))
```

In order to translate it properly, we, again, need a way to resolve the relation variable `r` and it's domain more generally. It has been shown, that a matrix representation for SAT-encoding of relational logic is convenient [TJ06].

Formally, consider the carrier-sets $S = \{s_1, ..., s_n\}$ and $T = \{t_1, ..., t_m\}$ and a relation $R \subseteq S \times T$. Let $r \in Pred^{n \times m}$ be the predicate matrix $r = \begin{pmatrix} r_{11} & \cdots & r_{1m} \\ \vdots & \vdots & \vdots \\ r_{n1} & \cdots & r_{nm} \end{pmatrix}$ which satisfies the equation

$$\forall_{i,j}(r_{ij} \Leftrightarrow s_i \mapsto t_j \in R), i \in \{1, ..., n\}, j \in \{1, ..., m\}$$

Now we can extend the expression translation function to relations and write

$$[T_e(R)]_{ij} = r_{ij}$$

For our relation invariant that means, that we now have a way to expand the variable `r` into a predicate matrix like this.

$$\texttt{r} = \begin{pmatrix} \texttt{rs1s1=TRUE} & \texttt{rs1s2=TRUE} \\ \texttt{rs2s1=TRUE} & \texttt{rs2s2=TRUE} \end{pmatrix}$$

Taking the domain of relation, should yield a set of type $\mathbb{P}(S)$, so we need to collapse the dimension of this matrix into a vector in order to compare it to the variable `a`. If we compare this to our naive translation, we can see that if we combine the rows of the matrix via logic or, it has the desired effect. This yields a predicate vector that can be combined via the subset rule in Table 2 to get the naive implementation exactly. For other translation rules on relational expressions, see Table 3.

Functions While functions mostly behave the in same way as relations for our translation, B provides the function application operator, which presents additional challenges. The result of the function application operator is no longer a set, but an element of the set. But most expression contexts where normal elements are used, is not something we can translate easily.

A way to get around this, is by rewriting the predicates, expressions and substitutions in terms of set logic (see Table 4)and translate it with that, but this approach is limited. We simply can not list all the ways the function application might be used in the machine.

If that is so, then we need to handle non-translatable B constructs somehow. Let us now discuss what happens if we can't translate a given B construct for one reason or another. Let a be a finite set variable of type $\mathbb{P}(S)$ with $S = \{s_1, s_2\}$ and f be a partial function of type $\mathbb{P}(\mathbb{P}(S) \times S)$. The expression $f(a)$ can then not be properly translated, since f is not translatable with our algorithm. But we can't leave it as is either, because after the translation a will no longer exists.

From Booleans Back to Sets The way we can handle this kind of expression, is by using the build in `IF` expression in classical B. We can rewrite `f(a)` as

```
f(IF as1=TRUE or TRUE=TRUE {s1} ELSE {} END
\/ IF as2=TRUE or TRUE=TRUE {s2} ELSE {} END)
```

which simplifies to

```
f({s1} \/ IF as2=TRUE {s2} ELSE {} END)
```

We can use this way of getting back to set logic, not just for function application, but in all places where we have no way of translating the expression itself for one reason or another.

In fact, we could theoretically use it to translate the entire machine and not change any of the logic. The problem with that approach would be, that we would not gain anything through simplifying and therefore have none of the benefits of the translation and only increase the size of the machine significantly. That means, that wherever possible, we want to translate the machine properly, but we can use this way of calculating the boolean variables back to set logic as a fail-safe.

Table 1: Predicate Translation

B Predicate	Translated B Predicate	Semantic
$\neg$ P	$\neg T_p(P)$	negation
P $\vee$ Q	$T_p(P) \vee T_p(Q)$	logical or
P $\Rightarrow$ Q	$T_p(P) \Rightarrow T_p(Q)$	implication
P $\Leftrightarrow$ Q	$T_p(P) \Leftrightarrow T_p(Q)$	equivalence
A = B	$\bigwedge_i([T_e(A)]_i \Leftrightarrow [T_e(B)]_i)$	equal
A $\neq$ B	$\neg T_p(A = B)$	not equal
A $\subseteq$ B	$\bigwedge_i([T_e(A)]_i \Rightarrow [T_e(B)]_i)$	subset equals
A $\subset$ B	$\bigvee_i([T_e(A)]_i \wedge \neg[T_e(B)]_i) \wedge T_p(A \subseteq B)$	strict subset
$\forall$(x).(P $\Rightarrow$ Q)	$\bigwedge_i(T_p(\beta_{x \leftarrow s_i}(P)) \Rightarrow T_p(\beta_{x \leftarrow s_i}(Q)))$	universal quantification
$\exists$(x).(P)	$\bigvee_i(T_p(\beta_{x \leftarrow s_i}(P)))$	existential quantification
$s_i \in$ E	$[T_e(E)]_i$	set member

Local Identifiers In the last example invariant, we introduce another concept into the discussion. Local identifiers. Some B expressions, including the universal quantifier, allow us to introduce a new local identifier. These are meaningful only in the context of first-order logic and disappear in our propositional representation. So we need a way to resolve them in our translation. Take the invariant

```
!(x).(x:S => x : a\/dom(r))
```

The universal quantifier can be parsed in terms of all the individual elements that are being quantified over. We ensure that the right side of the implication is true for all the possible values of x. An ideal translation looks like this

```
as1=TRUE or (rs1s1=TRUE or rs1s2=TRUE) &
as2=TRUE or (rs2s1=TRUE or rs2s2=TRUE)
```

This can be achieved if we first expand the predicate into a list of predicates where x is replaced by each possible element of x.

```
s1:S => s1 : a\/dom(r)
s2:S => s2: a\/dom(r)
```

Now we can apply our translation rules to each predicate

```
TRUE => as1=TRUE or (rs1s1=TRUE or rs1s2=TRUE)
TRUE => as2=TRUE or (rs2s1=TRUE or rs2s2=TRUE)
```

Simplifying the implication and connecting everything with logical and (logical or for existential quantification) yields the desired translation for the expression.

Formally, let $x \in S = \{s_1, ..., s_n\}$ be a local identifier, and $Cons_x \subset Cons$ the set of B constructs that contain the local identifier x. The renaming function $\beta_{x \leftarrow s_i} : Cons_x \to Cons$ takes a predicate with a local identifier x and substitutes it with the specific element s_i in that predicate. For example,

$$\beta_{x \leftarrow s_3}(x \in S) = s_3 \in S$$

This now gives us all the tools to we need for the description of the translation rules.

Table 2: Set Expression to Predicate Vector

B Expression E	Translated B Predicate Vector i $\in \{1,\ldots,$ n$\}$	Semantics
a	$[T_e(E)]_i = a_i$	set variable
$\varnothing$	$[T_e(E)]_i = \bot$	empty set
S	$[T_e(E)]_i = \top$	carrier set
{s$_j$}	$[T_e(E)]_i = \begin{cases} \top & \text{, if } i = j \\ \bot & \text{, else} \end{cases}$	enumeration set
c	$[T_e(E)]_i = \begin{cases} \top & \text{, if } s_i \in c \\ \bot & \text{, else} \end{cases}$	set constant
{x|P}	$[T_e(E)]_i = T_p(\beta_{x \leftarrow s_i}(P))$	set comprehension
A $\cup$ B	$[T_e(E)]_i = [T_e(A)]_i \vee [T_e(B)]_i$	set union
A $\cap$ B	$[T_e(E)]_i = [T_e(A)]_i \wedge [T_e(B)]_i$	set intersection
A $\setminus$ B	$[T_e(E)]_i = [T_e(A)]_i \wedge \neg[T_e(B)]_i$	set subtraction

3.3.2 Assignments

Substitutions, also called assignments, are statements that associate a variable with a new value. They are used to change the state of the machine. In the `INITIALISATION` section, all variables receive an initial value from their type, that is proven to satisfy the invariant predicate.

While there are many supported substitutions in the classical B language most are combinations of predicates and more fundamental substitutions. The most basic substitution we

[3] $[T_e(E)]_{ij}$ defines a matrix by defining how the Element in the i-th row and j-th column is calculated

Table 3: Relational expressions to Predicate list

B Expression E	Translated B Predicate Matrix[2]	Semantics
$A \times B$	$[T_e(E)]_{ij} = T_e(A)_i \wedge T_e(B)_j$	Cartesian Product
$dom(r)$	$[T_e(E)]_i = \bigvee_k (T_e(r)_i k)$	domain
$ran(r)$	$T_e(dom(r^{-1}))$	range
$id(S)$	id_n	identity relation
$s \lhd r$	$[T_e(E)]_{ij} = T_e(r)_{ij} \wedge T_e(s)_i$	domain restriction
$s \lhd\!\!\!\!- r$	$[T_e(E)]_{ij} = T_e(r)_{ij} \wedge \neg T_e(s)_i$	domain subtraction
$r \rhd s$	$T_e(s \lhd r^{-1})$	range restriction
$r \rhd\!\!\!\!- s$	$T_e(s \lhd\!\!\!\!- r^{-1})$	range subtraction
r^{-1}	$T_e(r)^T$	relational inverse
$r[A]$	$T_e(A) \cdot T_e(r)$	relational image
$r1 \lhd\!\!\!\!- r2$	$T_e(r2 \cup (dom(r2) \lhd\!\!\!\!- r1))$	relational override
$r1;\ r2$	$T_e(r1) \cdot T_e(r2)$	forward composition
$\lambda\, x.(P \mid A)$	$[T_e(E)]_{ij} = T_p(\beta_{x \leftarrow s_i}(P)) \wedge T_e(A)_j$	lambda expression

need to solve first for our translation, is the simple assign statement. Consider the following statement in the demo machine.

```
a := {s1}
```

This statement assigns variable `a` the new value {`s1`}. A translated version, that writes the new boolean variables instead and achieves an equivalent state of the machine would look like this.

```
as1:=TRUE||as2:=FALSE||as3:=FALSE
```

Though it is not obvious on the face of it, how we get to this translation for more complicated expressions. As it turns out, we can get to an equivalent statement, by reusing the predicate vector from the expression on the right-hand side. For the enumeration set {`s1`} the resulting predicate vector would look like $[\top, \bot, \bot]$ which, translated into B would be [`TRUE=TRUE`, `TRUE=FALSE`, `TRUE=FALSE`]. Notice, that the length of the predicate vector is always the same as the number of new boolean variables in our machine and each entry corresponds to exactly one variable. If we take each element and assign it to the boolean variable that corresponds to it's corresponding element, we get

```
as1:=bool(TRUE=TRUE)||as2:=bool(TRUE=FALSE)||as3:=bool(TRUE=FALSE)
```

The `bool` function is needed to translate the predicate into boolean values and makes this statement equivalent to the one above. This approach works simple assign statements, but can end up with very verbose assignments that use a bunch of functional `skip` assignments,

since every boolean variable is always written. Since our goal is to reduce write conflicts, this is not an acceptable circumstance, which means we need a process of removing the unnecessary assignments. Consider for instance the following statement, in the context of the demo machine

```
a := a \/ {s1}
```

The right-hand side expression will translate into the predicate vector

[as1=TRUE or TRUE=TRUE, as2=TRUE or TRUE=FALSE, as3=TRUE or TRUE=FALSE]

which means, that the statement translates to

```
as1 := bool(as1=TRUE or TRUE=TRUE)
as2 := bool(as2=TRUE or TRUE=TRUE)
as3 := bool(as3=TRUE or TRUE=TRUE)
```

That is a rather verbose translation, but can rather easily be simplified. First we can use the rules for logical or to reduce it to

```
as1 := bool(TRUE=TRUE)
as2 := bool(as2=TRUE)
as3 := bool(as3=TRUE)
```

which we can rewrite as,

```
as1 := TRUE
as2 := as2
as3 := as3
```

and then we can eliminate the statements that just assign a variable to itself, and reach the more reasonable final translation

```
as1:=TRUE
```

This version only contains the boolean variables, that are actually affected by the assignment and therefore need to be in the write-set of the operation, which was our goal with the rewrite to begin with.

With this way of rewriting simple assignments, we can define the translation function for substitutions. Let Sub be the set of all B Substitutions, and $T_s : Sub \rightarrow Sub$ be the

translation function for substitutions. The list of assignment translations can be found in Table 5.

Most translations are present, except return value assignment and the `VAR` operator, which are more involved operations. They require, in the case of introduced set variables, the creation of new local boolean variables which is not yet supported in the implementation.

Table 4: Function Application Translation Rules

B Statement	Rewritten Statement
`f(s1)=s2`	`{s1|->s2}:f`
`f(s1):A`	`f[{s1}]<:A`
`f(s1):=s2`	`f`$\lhd\!\!-$`{s1 | ->s2}`

Table 5: Substitution Translations

B Substitution `A`	Translated Substitution T_s(`A`)
`x:=E`	`xs1:=bool(`$[T_e$(`E`)$]_1$`)` $\|\|\ldots$ `||xsn:= bool(`$[T_e$(`E`)$]_n$`)`
`f(x):=E`	see Table 4
`x::E`	n.a. [3]
`x:(P)`	`xs1:(`T_p(`P`)`)||`$\ldots$ `||xsn:(`T_p(`P`)`)`
`x<-- OP(x)`	unsupported
`G||H`	T_s(`G`) $\|\|$ $T_s(H)$
`G;H`	T_s(`G`);T_s(`H`)
`ANY x WHERE T THEN G END`	`SELECT` $T_p(\beta_{x\leftarrow s_j}(T))$ `THEN` $T_s(\beta_{x\leftarrow s_j}$(`G`)) `END` [4]
`LET x BE x=E IN G END`	$T_s(\beta_{x\leftarrow E}$(`G`))
`VAR x,...IN G END`	unsupported
`PRE P THEN G END`	`PRE` T_p(`P`) `THEN` T_s(`G`) `END`
`ASSERT P THEN G END`	`ASSERT` T_p(`P`) `THEN` T_s(`G`) `END`
`CHOICE G OR H END`	`CHOICE` T_s(`G`) `OR` T_s(`H`) `END`
`IF P THEN G END`	`IF` T_p(`P`) `THEN` T_s(`G`) `END`
`SELECT P THEN G END`	`SELECT` T_p(`P`) `THEN` T_s(`G`) `END`

3.4 Operations

In B, operations are named, transactional substitutions, that transition the machine from one state in the state-space into another. They contain a guard, a predicate that has to return true for the operation to be enabled, an act, a substitution that is executed when the operation is activated, and they can have parameters and return values like a function.

[4] For x to be a set, E has to have an unsupported type

[5] $s_j \in S$ is here the Element that the Algorithm added on the Operation level for the Choice of x, see Section 3.4.

Table 6: Simplification Rules

Expression	Simplified Expression
`not(not(P))`	`P`
`bool(TRUE=TRUE)`	`TRUE`
`bool(TRUE=FALSE)`	`FALSE`
`P or TRUE=FALSE`	`P`
`P or TRUE=TRUE`	`TRUE=TRUE`
`P & TRUE=FALSE`	`TRUE=FALSE`
`P & TRUE=TRUE`	`P`
`TRUE=TRUE => P`	`P`
`TRUE=FALSE => P`	`TRUE=TRUE`
`P => TRUE=TRUE`	`TRUE=TRUE`
`P => TRUE=FALSE`	`not(P)`
`P <=> TRUE=FALSE`	`not(P)`
`P <=> TRUE=TRUE`	`P`
`IF TRUE=TRUE THEN A ELSE B END`	`A`

Since return values are not being ignored, and we can already translate predicates and simple substitutions, the only new concept we have to address are parameters.

As it turns out, parameters can be viewed as a special case of non-deterministic local identifiers [5]. In the previous section we brushed over some of the more complex substitutions in classical B like `ANY` or `LET`. These introduce new local identifiers that are bound to a value either deterministically, with `LET`, or non-deterministically with `ANY`.

The `LET` case is trivial to translate, we simply revert the naming by replacing all occurrences of the local identifier with it's bound value. For the nondeterministic identifiers or parameters, of type set or relation, we have multiple options on how to translate them. For instance, it would be possible to introduce new local identifiers of type boolean, similarly to what we did with variables. For instance, we could rewrite the `conj` operation in the following way.

```
conj(xs1, xs2) = SELECT xs1:BOOL &
                        xs2:BOOL &
                        xs1=TRUE <=> xs2=FALSE
                 THEN as1:=bool(as1=TRUE or xs1=TRUE) ||
                      as2:=bool(as2=TRUE or xs2=TRUE)
                 END;
```

But, writing it like this does not scale very well and more importantly, does not allow us to simplify as much. Wherever possible, we want to work with constant $\top$ or $\bot$ terms, that

[5] In Event-B all parameters are introduced with the ANY keyword

can fall away with logic simplification, and with functional `skip` operations that assign a variable to itself.

A better way to rewrite the same operation, is to first create new operations, one for each possible parameter value, and then replace the local identifier with that chosen value (Item 3). For the `conj` operation, it would look like this

```
conjs1 = SELECT s1 : S THEN a:=a \/ {s1} END;
conjs2 = SELECT s2 : S THEN a:=a \/ {s2} END;
```

where `conjs1` for example denotes the `conj` operation with `s1` instead of `x`. Now, when we translate the body of the operation, we get

```
conjs1 = SELECT TRUE=TRUE
         THEN as1:=bool(as1=TRUE or TRUE=TRUE) ||
              as2:=bool(as2=TRUE or TRUE=FALSE)
         END;
conjs2 = SELECT TRUE=TRUE
         THEN as1:=bool(as2=TRUE or TRUE=TRUE) ||
              as2:=bool(as2=TRUE or TRUE=FALSE)
         END;
```

which can be simplified to the very concise version

```
conjs1 = as1:=TRUE;
conjs2 = as2:=TRUE;
```

This version is very concise and has the additional advantage of creating more operations, with minimal read and write sets, which can be found to be independent and could potentially uncover previously hidden independence relationships for POR.

The last part that needs to be discussed, is what we do if we have multiple parameters. Consider for instance the `override` operation. It introduces two new local identifiers `x` and `y`. In this case we have to consider each possible combination. Here we have only $2 * 2 = 4$ choices, but in general the number of new combinations, and therefore new operations, grows fast with the number of set parameters and their set size.

For a given operation o, let $m \in \mathbb{N}$ be the number of set-type parameters of o and $S_i, i \in \{1, ..., m\}$ the carrier-sets those parameters are from, meaning $\mathbb{P}(S_i)$ is the type of the i-th parameter. The number of new operations, that replace o, is then $\Pi_{j=1}^{m} |S_j|$.

The translation of the four new operations works exactly the same as previously and after simplification we can now put everything together, and reach the final translation of the demo machine in Listing 3.

4 Implementation

The full implementation[6] of the previously discussed translation algorithm consists of around 750 lines of Clojure code, that leverage a combination of *lisb*[7], pattern matching and Specter[8].

In this section, we will discuss some development decisions, advantages and limitations of the approach, and we give some brief examples meant to give a sense of how the different parts fit together. They are, however, not supposed give a comprehensive overview of how the inner workings of the implementation. We will also briefly consider the implementation performance and possible implications for usability.

4.1 Clojure

One important choice made in development, is the choice of implementation language. It is possible to implement the algorithm by using logical programming and Prolog, but since this work is only a first exploration of the feasibility of the concept, we prioritized the simplicity and ease of use of Clojure, over speed and direct integration into the ProB tool, that Prolog would have brought.

Clojure is a functional general-purpose dialect Lisp that provides robust and efficient immutable data-structures and access to Java Frameworks due to strong interoperability [Hic08].

We were able to utilize those data-structures in order to internally represent and transform B machines because of *lisb*, a new and highly convenient way of working with B machines from Clojure. It provides a tree like internal representation (IR) structure, that can be recursively walked and manipulated according to our translation rules.

As a small example, consider this predicate from the demo machine.

```
a/={}
```

In *lisbs* internal representation[9], it is represented as

```
{:tag :not-equals :left :a :right #{}}
```

and the translated predicate

[6] The implementation can be found at https://github.com/JanRossbach/fset
[7] https://gitlab.cs.uni-duesseldorf.de/mager/lisb-clone
[8] https://github.com/redplanetlabs/specter
[9] Under Version 0.0.4

```
not(as1=FALSE&as2=FALSE)
```

would be represented by

```
{:tag :not :pred
 {:tag :and :preds
  '({:tag :equals :left :as1 :right false}
    {:tag :equals :left :as2 :right false})}}
```

To achieve these translations, we use recursive pattern matching with core.match[10], operating on the IR level. This keeps the definitions of the translation functions similar to the tables in the previous sections and enables easy translation to Prolog later on.

An alternative implementation with core.logic[11] would have been even easier to port, but it would be harder to write and, subjectively, less readable.

For a short, but representative, example of an implementation rule, that would translate the previous IR example, see Listing 2. Here, the function-call

```
unroll-predicate({tag :not-equals :left :a :right #{}})
```

would first match the rule for non-equals on line 5 and recursively call the nested `T` function with the argument `{:tag :not :pred {:tag :equals :left :a :right #{}}}`. That will then match with the equals rule on line 8, since both sides are set-expressions, and will the build up the return IR with the predicate vectors that are returned from recursively calling the function again on the left and right expression. Those will match the last statement on line 13 and call the `unroll-expression` function, which is the implementation of the T_e function. The capitalized functions are internal wrappers for *lisbs* IR translation functions, and serve readability.

This will then yield an IR representation of the unsimplified translation, that can be turned into the one we expected.

[10] https://github.com/clojure/core.match
[11] https://github.com/clojure/core.logic

```
(defn unroll-predicate
  [pred]
  ((fn T [e]
     (match e
            {:tag :not-equals :left l :right r}
            (NOT (T (EQUALS l r)))

            {:tag :equals
             :left  (l :guard b/setexpr?)
             :right (r :guard b/setexpr?)}
            (apply AND (map <=> (T l) (T r)))

            expr (unroll-expression expr)))
   pred))
```

Listing 2: Partial Implementation of the T_p Function

4.2 Back-end

Some translations require a broader context of the larger machine and information about the thing that is being translated.

For example, when we come across an identifier, there are several questions we need to answer before we can say what needs to be done with it. What type does it have? Do we need to expand it? If so, what boolean variables does it expand to? To answer these questions, we can either parse the IR of the machine, using specter, or we can consult the ProB Java API.

Both are done in the implementation depending on the circumstance. First, in order to find out what the identifier represents (variable, constant, carrier-set, set-element ...), we use specter.

Specter allows us to deal with the highly nested IR data-structure. It leverages a path like syntax, to achieve small functions that read similarly to database queries.

For a short example, with the function

```
(defn get-vars [ir]
  (s/select [:machine-clauses #(= (:tag :variables)) s/ALL] ir))
```

we can get a list of all variables the machine has. This can then be compared to the identifier in order to find out, if it is a variable or not.

If it is a variable, then we can find its type, and ultimately the elements that type consists of, by calling the ProB Java API.

4.3 Performance

While this particular application is not very performance sensitive, since it only needs to be executed once per machine, it is still interesting to see if our theoretical performance expectations are in line with reality.

We would expect, that the run-time grows in the same way, as the number of new operations, that are generated. This is because each operation needs to be translated individually, since there might be differences in translation depending on parameter choice.

In order to measure the translation runtime from IR to new IR we ran benchmarks using the criterium library[12] and recorded the mean runtime [13] against the deferred-set-size on three adjusted demo machines (see Appendix D). The deferred-set-size, we called it k in the translation algorithm, is a configuration option of the implementation, that allows us to adjust the amount of elements of the resulting enumerated set, when a deferred set is translated. The machines are adjusted in such a way, that they use deferred-sets in order to test the translation behavior for larger set sizes. The difference between the machines, is that the demo(1) machine has the only operations with one parameter, demo(2) has the usual two parameters with the override operation and demo(4) has even more parameters in that operation, namely four.

That means, that the number of new operations in demo(1), for a deferred-set-size of n, should grow at a rate of $\mathcal{O}(n)$, the demo(2) machine at $\mathcal{O}(n^2)$ and the demo(4) machine at $\mathcal{O}(n^4)$. And this seems to also be reflected in the measured run-times (see Figure 1).

In practice, this implies that there might be issues with the long translation times for machines with a combination of large carrier-sets and operations with a high number of parameters.

[12] https://github.com/hugoduncan/criterium
[13] Using an Intel i7-4690K processor

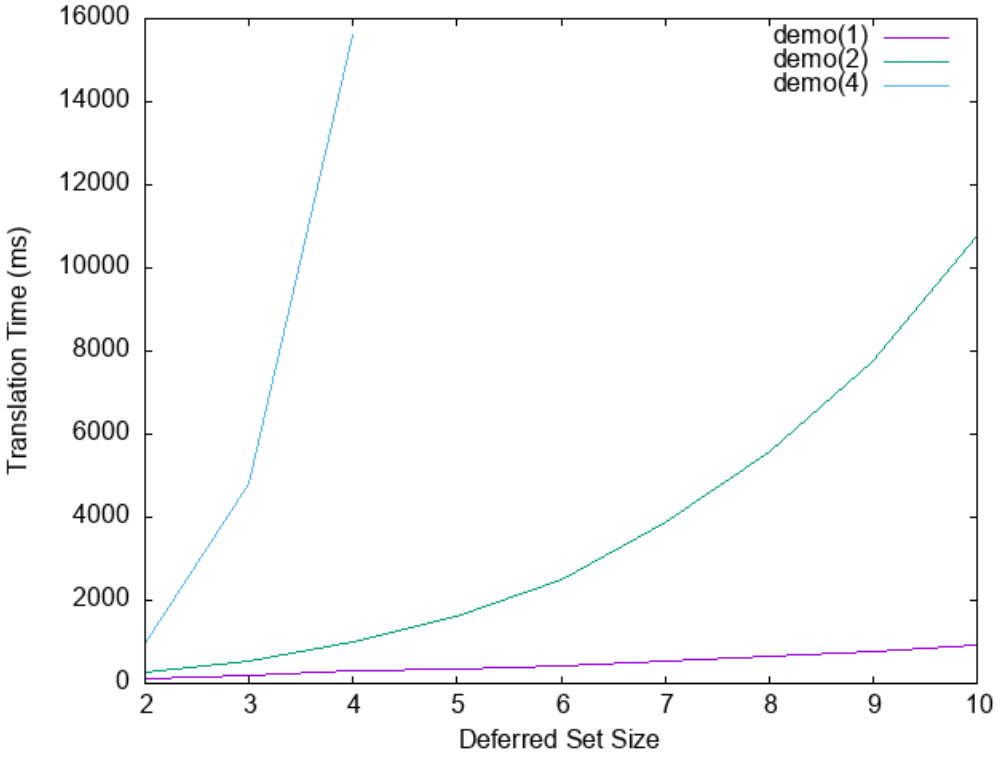


Figure 1: Encoding Performance

5 Conclusions

In conclusion, in this thesis we have presented a procedure that auto-refines certain set-variables of classical B machines into low-level semantic bit-vectors.

We expanded B expressions into predicate matrices and then used them to transform predicates and substitutions in place to their semantic equivalent.

We then presented a full Clojure implementation of the present translation rules and found its performance to be in line with our expectations from the problem domain.

5.1 Limitations and Future Work

There are still limitations to this approach. First, the performance is sensitive to parameter count and carrier-set size. This might be an issue for very large and highly complex machines, if runtime exceeds acceptable time scales, but should be fine for most applications.

A bigger limitation is that of finite types, since it prohibits, at least for now, the translation of finite integer intervals, because the type integer is not finite. It might be possible, to get around this limitation with a relational integer encoding similar to the one found in this translation from B to KodKod [HL12].

Another future project might be eventually to integrate the algorithm into ProB if the practical benefits to POR are found to be sufficient. For this, and other practical applications, a mathematical correctness proof of the translation with, since, for now, the proof of correctness is limited to the observation of equivalent state and transition numbers in the resulting state-space of the machine.

Appendices

A Notation Reference

Table 7: Notation Reference Table

Notation	Semantics
$\top$	Predicate that always returns true (e.g. `TRUE=TRUE`)
$\bot$	Predicate that always returns false (e.g. `TRUE=FALSE`)
$Expr$	Set of B expression
$Pred$	Set of B predicates
Sub	Set of B substitutions
$Cons$	B constructs $Cons = Expr \cup Pred \cup Sub$
$Cons_x$	Set of B constructs that contain the identifier x
S	Carrier Set $\{s_1, ..., s_n\}$
T	Carrier Set $\{t_1, ..., t_m\}$
a	Variable of type $\mathbb{P}(S)$
a_i	Predicate returning true if s_i is in a (e.g. `asi=TRUE`)
$T_e : Expr \to Pred^n$	Expression transformation function (Type P(S))
$T_e : Expr \to Pred^{n \times m}$	Expression transformation function (Type $\mathbb{P}(S \times T)$)
$T_p : Pred \to Pred$	Predicate transformation function
$T_s : Sub \to Sub$	Substitution transformation function
$[T_e(E)]_i$	Vector entry at the i-th index
$[T_e(E)]_{ij}$	Matrix entry in the i-th row and j-th column
$\beta_{x \leftarrow s_i} : Cons_x \to Cons$	Substitution function (replace local variable x with set-element s_i)

B Translated Demo Machine

```
1:  MACHINE demo
2:  SETS S={s1,s2}
3:  VARIABLES as1,as2,rs1s1,rs1s2,rs2s1,rs2s2
4:  INVARIANT as1:BOOL & as2:BOOL &
5:            rs1s1:BOOL & rs1s2:BOOL & rs2s1:BOOL & rs2s2:BOOL &
6:            not(as1=FALSE&as2=FALSE) &
7:            (as1=TRUE => (rs1s1=TRUE or rs1s2=TRUE)) &
8:            (as2=TRUE => (rs2s1=TRUE or rs2s2=TRUE)) &
9:            (as1=TRUE or (rs1s1=TRUE or rs1s2=TRUE)) &
10:           (as2=TRUE or (rs2s1=TRUE or rs2s2=TRUE))
11: INITIALISATION as1:=TRUE || as2 := FALSE ||
12:                rs1s1:=FALSE || rs1s2 := TRUE ||
13:                rs2s1:=TRUE || rs2s2:=FALSE
14: OPERATIONS
15: conjs1 = as1:=TRUE;
16: conjs2 = as2:=TRUE;
17: disjs1 = SELECT as1=TRUE & not(as1=TRUE & not(as2=TRUE))
18:          THEN as1 := FALSE
19:          END;
20: disjs2 = SELECT as2=TRUE & not(not(as1=TRUE) & as2=TRUE)
21:          THEN as2 := FALSE
22:          END;
23: overrides1s1 = rs1s1:=TRUE || rs1s2:=FALSE;
24: overrides1s2 = rs1s1:=FALSE || rs1s2:=TRUE;
25: overrides2s1 = rs2s1:=TRUE || rs2s2:=FALSE;
26: overrides2s2 = rs2s1:=FALSE || rs2s2:=TRUE
27: END
```

Listing 3: Translated Demo Machine

C Performance Evaluation Data

Table 8: Internal Translation Time Evaluation Data

Defferred set size	Demo(1)	Demo(2)	Demo(4)
2	134	270	946
3	201	537	4806
4	289	983	15603
5	328	1599	
6	407	2509	
7	546	3898	
8	652	5572	
9	771	7780	
10	916	10773	

D Performance Evaluation Machines

```
1:  MACHINE demo
2:  SETS S
3:  VARIABLES a,r
4:  INVARIANT a:POW(S) & r:S<->S &
5:  a/={} &
6:  a <: dom(r) &
7:  !(x).(x:S => x : a\/dom(r))
8:  INITIALISATION a := {} || r := {}
9:  OPERATIONS
10: conj(x) = SELECT x : S  THEN a := a \/ {x} END;
11: disj(x) =  SELECT x : a & a/={x} THEN a:= a \ {x} END;
12: END
```

Listing 4: Demo Machine with Linear Translation Time Demo(1)

```
MACHINE demo
SETS S
VARIABLES a,r
INVARIANT a:POW(S) & r:S<->S &
          a/={} &
          a <: dom(r) &
          !(x).(x:S => x : a\/dom(r))
INITIALISATION a := {} || r := {}
OPERATIONS
conj(x) = SELECT x : S  THEN a := a \/ {x} END;
disj(x) =  SELECT x : a & a/={x} THEN a:= a \ {x} END;
override(x) = SELECT x:S
              THEN
                 ANY y
                 WHERE y:S
                 THEN
                    r := r <+ {x|->y}
                 END
              END
END
```

Listing 5: Demo Machine with Quadratic Translation Time (Demo(2))

```
MACHINE demo
SETS S
VARIABLES a,r
INVARIANT a:POW(S) & r:S<->S &
          a/={} &
          a <: dom(r) &
          !(x).(x:S => x : a\/dom(r))
INITIALISATION a := {} || r := {}
OPERATIONS
conj(x) = SELECT x : S  THEN a := a \/ {x} END;
disj(x) =  SELECT x : a & a/={x} THEN a:= a \ {x} END;
override(x,y,v,w) = SELECT x:S & y:S & v:S & w:S
                    THEN
                      r := r <+ {x|->y,w|->v}
                    END
END
```

Listing 6: Demo Machine with Translation Time $O(n^4)$ (Demo(4))

List of Figures

List of Tables

List of Listings

References

[ABH+10] Jean-Raymond Abrial, Michael Butler, Stefan Hallerstede, Thai Son Hoang, Farhad Mehta, and Laurent Voisin. Rodin: An open toolset for modelling and reasoning in Event-B. *International journal on software tools for technology transfer*, 12(6):447–466, 2010.

[Abr10] Jean-Raymond Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, 2010.

[ALN+91] J.-R. Abrial, M.K.O. Lee, D.S. Neilson, P.N. Scharbach, and I.H. Sørensen. The B-method. In *Proceedings VDM*, pages 398–405. Springer, 1991.

[BKK+20] Michael Butler, Philipp Körner, Sebastian Krings, Thierry Lecomte, Michael Leuschel, Luis-Fernando Mejia, and Laurent Voisin. The first twenty-five years of industrial use of the b-method. In *International Conference on Formal Methods for Industrial Critical Systems*, pages 189–209. Springer, 2020.

[CKNZ12] Edmund M. Clarke, William Klieber, Miloš Nováček, and Paolo Zuliani. *Model Checking and the State Explosion Problem*, pages 1–30. Springer, 2012.

[CM12] Mats Carlsson and Per Mildner. SICStus Prolog—the first 25 years. *TPLP*, 12(1-2):35–66, 2012.

[DL14] Ivaylo Dobrikov and Michael Leuschel. Optimising the ProB model checker for B using partial order reduction. In *Proceedings SEFM 2014*, volume 8702 of *LNCS*, pages 220–234. Springer, 2014.

[DL16] Ivaylo Dobrikov and Michael Leuschel. Optimising the ProB model checker for B using partial order reduction. *Formal Aspects of Computing*, 28(2):295–323, 2016.

[Dob17] Ivaylo Miroslavov Dobrikov. *Improving Explicit-State Model Checking for B and Event-B*. PhD thesis, Universitäts- und Landesbibliothek der Heinrich-Heine-Universität Düsseldorf, 2017.

[Hic08] R. Hickey. The Clojure programming language. In *Proceedings DLS*. ACM, 2008.

[HL12] Dominik Hansen and Michael Leuschel. Translating TLA+ to B for validation with ProB. In *Proceedings IFM*, volume 7321 of *LNCS*, pages 24–38. Springer, 2012.

[Jac98] Daniel Jackson. An intermediate design language and its analysis. *SIGSOFT Software Engineering Notes*, 23(6):121–130, November 1998.

[Jac00] Daniel Jackson. Automating first-order relational logic. *ACM SIGSOFT Software Engineering Notes*, 25(6):130–139, 2000.

[Jac02] Daniel Jackson. Alloy: A lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology*, 11(2):256–290, apr 2002.

[KBD+20] Philipp Körner, Jens Bendisposto, Jannik Dunkelau, Sebastian Krings, and Michael Leuschel. Integrating formal specifications into applications: the ProB Java API. *Formal Methods in System Design*, 2020.

[LB03] Michael Leuschel and Michael Butler. Prob. In *Proceedings FME*, volume 2805 of *LNCS*, pages 855–874. Springer, 2003.

[LB08] Michael Leuschel and Michael Butler. ProB: An automated analysis toolset for the B method. *International Journal on Software Tools for Technology Transfer*, 10(2):185–203, March 2008.

[Pel93] Doron Peled. All from one, one for all: on model checking using representatives. In *Proceedings CAV 1993*, volume 697 of *LNCS*, pages 409–423. Springer, 1993.

[Req08] Antoine Requet. BART: A tool for automatic refinement. In Egon Börger, Michael Butler, Jonathan P. Bowen, and Paul Boca, editors, *Abstract State Machines, B and Z*, pages 345–345. Springer, 2008.

[TJ06] Emina Torlak and Daniel Jackson. The design of a relational engine. Technical Report MIT-CSAIL-TR-2006-068, MIT, September 2006.

[TJ07] Emina Torlak and Daniel Jackson. Kodkod: A relational model finder. In *Proceedings TACAS*, volume 4424 of *LNCS*, pages 632–647. Springer, 2007.

[Val98] Antti Valmari. *The state explosion problem*, pages 429–528. Springer, 1998.