A Translation from Alloy to B

Sebastian Krings[1], Joshua Schmidt[1], Carola Brings, Marc Frappier[2], and Michael Leuschel[1]

[1] Institut für Informatik, Universität Düsseldorf
Universitätsstr. 1, D-40225 Düsseldorf
`{krings,leuschel}@cs.uni-duesseldorf.de`
[2] Université de Sherbrooke, Québec, Canada
`marc.frappier@usherbrooke.ca`

Abstract. In this paper, we introduce a translation of the specification language Alloy to classical B. Our translation closely follows the Alloy grammar, each construct is translated into a semantically equivalent component of the B language. In addition to basic Alloy constructs, our approach supports integers and orderings. The translation is fully automated by the tool "Alloy2B". We evaluate the usefulness by applying AtelierB and ProB to the translated models, and show benefits for proof and solving with integers and higher-order quantification.

1 Introduction

Both B [1] and Alloy [10] are specification languages based on first-order logic. The languages share several features, such as native support for integers, sets and relations as well as user-defined types. However, there are also considerable differences. For instance, one of B's key concepts is to encode state changes by means of transitions, effectively computing successor states featuring all variables. In contrast, Alloy allows to define orderings over certain types.

Another difference between Alloy and B is tool support, especially when it comes to available backends for constraint solving. For Alloy, the Alloy Analyzer [10] is used to compute models by translating Alloy predicates to SAT using Kodkod [30]. The most prominent constraint solver for B, ProB [16,18,17], however mainly relies on constraint logic programming [11]. In particular, it uses the CLP(FD) library of SICStus Prolog [2] and extends it to support constraints over infinite domains [12]. Additionally, ProB allows to use other backends, such as SMT solvers[13] or, again, Kodkod [28].

The different constraint solving techniques show different performance characteristics [29]. Certain predicates can be solved faster by using a particular backend or combination of backends; others cannot be handled by a particular solving technique at all. We thus suppose that a translation from Alloy models to B models serves different purposes:

- It provides Alloy users access to a set of new backends, and might enable constraint solving for Alloy models that can not be handled efficiently by the Alloy Analyzer,

Listing 1. Own Grandpa (Alloy)

```
module SelfGrandpas
abstract sig Person {
    father : lone Man,
    mother : lone Woman
}
sig Man   extends Person {   wife : lone Woman    }
sig Woman extends Person {   husband : lone Man   }
```

Listing 2. Own Grandpa (B - Signatures)

```
MACHINE SelfGrandpas
SETS
    Person
CONSTANTS
    Man, Woman, father, mother, wife, husband
PROPERTIES
    father : Person +-> Man &
    mother : Person +-> Woman &
    Man <: Person &
    wife : Man +-> Woman &
    Woman <: Person &
    husband : Woman +-> Man &
    Man /\ Woman = {} &
    Man \/ Woman = Person
END
```

- it enables the application of the Atelier-B provers [3] to Alloy models,
- it enables the usage of ProB as a second toolchain to validate the results of the Alloy Analyzer,
- it provides new test cases and benchmarks to the B community and should aid in improving ProB,
- it helps communication between the Alloy and B communities.

Details about installing and using our translation can be found at:

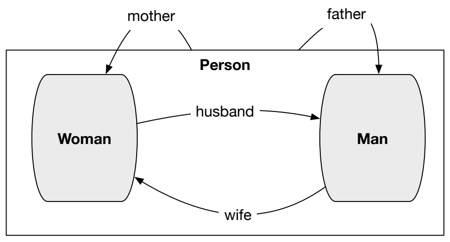
`https://www3.hhu.de/stups/prob/index.php/Alloy`

2 Translation Example

In the following section, we will introduce our translation on a simple Alloy model taken from [10]. The model is given in Listing 1 and Listing 3, the translation is given in Listing 2 and Listing 4. Our translation will only use the following concepts of a B machine:

Alloy:

```
abstract sig Person {
    father : lone Man,
    mother : lone Woman
}
sig Man   extends Person {
   wife : lone Woman
}
sig Woman extends Person {
   husband : lone Man
}
```



B Translation:

```
father ∈ Person ⇸ Man ∧
mother ∈ Person ⇸ Woman ∧
Man ⊆ Person ∧
wife ∈ Man ⇸ Woman ∧
Woman ⊆ Person ∧
husband ∈ Woman ⇸ Man ∧
Man ∩ Woman = ∅ ∧
Man ∪ Woman = Person
```

Fig. 1. Signatures and Fields in the Own Grandpa Model

1. Deferred sets, introducing new types for Alloy signatures in the SETS clause
2. Constants, introduced in the CONSTANTS clause,
3. Predicates about the constants and deferred sets in the PROPERTIES clause,
4. DEFINITIONS, aka B macros, to ease translating certain Alloy concepts,
5. B Operations for Alloy assertion checks.

In particular, our translation does not use variables, invariants or assertions.

2.1 Translating Signatures

We first concentrate on the translation of Alloy's signatures and fields in Listing 1 to B types. An overview of the signatures and fields can be found in Figure 1.

In order to translate the Alloy module `SelfGrandpas`, we create a B machine with the same name. Inside, the basic signature `Person`, defined in line 2 of the Alloy model, is represented as a user-given set in line 3 of the B machine in Listing 2.[3] Deferred sets in B can have any size, just like signatures in Alloy. (In Section 3.3 we show how a limit on the size of the signature is translated.)

The signature features two fields, `father` and `mother`, each representing a relation of members of `Person` to members of `Man` and `Woman`. The keyword `lone` states that the relation is in fact a partial function, i.e., a 1-to-at-most-1 mapping. This can be encoded into B using a partial function, as created by the `+->` operator in lines 7 and 8 of Listing 2.

The extending `Man` and `Woman` are subsets of `Person`. As user-given sets in B are distinct, we introduce constants `Man` and `Woman` and assert the subset property in lines 9 and 11 of Listing 2. As above, the fields `wife` and `husband` are translated into partial functions in lines 10 and 12.

Since `Person` was declared `abstract`, two additional properties have to hold for the sub-signatures: each element of `Person` has to be in one of the sub-signatures and the two sub-signatures have to be disjoint. This partitioning of `Person` is encoded in B's set theory in lines 13 and 14 of Listing 2.

[3] For the sake of readability, the example translation uses the same identifiers as the Alloy module. Of course, one has to ensure the translation is valid, e.g., identifiers do not collide with B's keywords.

Listing 3. Own Grandpa (Alloy - Facts and Predicates)

```
...
fact Terminology {   wife = ~husband   }
fact SocialConvention {
    no wife & *(mother + father).mother
    no husband & *(mother + father).father
}
fact Biology {
    no p : Person | p in p.^(mother + father)
}
fun grandpas[p : Person] : set Person {
    let parent = mother + father + father.wife + mother.husband
    | p.parent.parent & Man
}
pred ownGrandpa[m : Man] {
    m in grandpas[m]
}
run ownGrandpa for 4 Person
```

2.2 Translating Facts and Predicates

Alloy facts are added to the B machine's PROPERTIES clause. For example, the Alloy fact `Terminology` of Listing 3, stating that `wife` is the inverse of `husband`, can be encoded in B using the relational inverse, see line 11 of Listing 4.

The first fact in `SocialConvention` states that your wife cannot be your mother or the mother of any of your ancestors. The second fact asserts the same property for husband and father. Both can be translated directly as far as set union, intersection and closure computation are concerned. The dot join in this case is interpreted as the composition of the two relations, which is available in B as using the `;` operator. Other interpretations of the dot join operator will be discussed later. The `no` keyword enforcing the emptiness of a set is translated to equalities to the empty set in lines 12 and 13.

The Alloy fact `Biology`, stating that nobody can be its own ancestors, introduces a quantified local variable `p`. We translate the fact into a set comprehension, which again is able to introduce the variable. Again, `no` enforces emptiness of the set comprehension. Observe, that quantification in Alloy is over singleton sets only. More generally, we translate the quantification `no p : S | P` into $\{p|\{p\} \subseteq S \wedge P\} = \varnothing$.

The function definition `grandpas` and the predicate definition `ownGrandpa`, both with a parameter, are encoded as B definitions to allow their reuse throughout the model. `ownGrandpa` only includes the application of `grandpas` as well as a membership check and can thus be translated directly.

Listing 4. Own Grandpa (B - Facts and Predicates)

```
MACHINE SelfGrandpas
...
DEFINITIONS
    parent == mother \/ father \/
              (father ; wife) \/ (mother ; husband);
    ownGrandpa(m) == {m} <: Man  &  ({m} <: grandpas(m));
    grandpas(p) == {tmp | {p} <: Person &
                          tmp : (parent[parent[{p}]] /\ Man)}
PROPERTIES
    ...
    wife = husband~ &
    wife /\ (closure((mother \/ father)) ; mother) = {} &
    husband /\ (closure((mother \/ father)) ; father) = {} &
    {p | {p} <: Person &
         {p} <: closure1((mother \/ father))[{p}]} = {} &
    card(Person) <= 4
OPERATIONS
    run_ownGrandpa = PRE #(m).(ownGrandpa(m)) THEN skip END
END
```

Translating `grandpas` however is not straightforward, as it includes a let expression, which is not available in B.[4] As an alternative to inlining, we again create a definition named `parent` in order to hold the value of the newly introduced variable. Note that this changes the scope in which the variable resides and might make renaming necessary to avoid conflicts. Furthermore, observe that there are no free variables in the definition of `parent`. Otherwise, those would be passed to the B definition as parameters. As `grandpas` returns a set of `Persons`, the definition again uses a set comprehension.

3 Translating Alloy to B

In this section, we will outline how to translate the components of an Alloy model into semantically equivalent B components. Each Alloy module is translated into a corresponding B machine. As of now, our translation supports all basic Alloy constructs, i.e., everything that is not inside a library. In particular, this includes integers and the corresponding operations. Regarding libraries, we support orderings, because they are closely related to state transitions in B. Further libraries will be considered in the future.

[4] Let expressions are available in an extended version of B understood by ProB.

3.1 Signature Declarations

Since a signature declaration can be quite complex, let us start with the most simple one, omitting everything optional, i.e., we only add a named signature to the model. A signature has the properties of a set, containing atoms of the signature's type. For the translation to B, we will create a new deferred set for each signature.

Additionally, a signature can extend another signature by making use of either the `in` or the `extends` keyword. In this case, we set up a subset of an already existing set, i.e., for each sub-signature s extending base signature s_b we define a constant s and add $s \subseteq s_b$ to the `PROPERTIES` clause.

For the `extends` keyword, we ensure that extending signatures are pairwise disjoint by adding $s_1 \cap s_2 = \varnothing$ for each combination of extending signatures s_1, s_2 to the B machine's `PROPERTIES` clause.

Next, base signatures can be declared as `abstract`: Abstract signatures are used for the sole purpose of being extended by other signatures. They do not contain elements which are not also elements of other sets [10]. In B, this property can be modeled by adding the following constraint to the `PROPERTIES` section:

$$s_b = \bigcup_{s \text{ extends } s_b} s.$$

Alloy allows to state the cardinality of signatures by using one of the quantifiers `no` (empty), `lone` (at most one), `one` (exactly one), `some` (at least one) or `set` (any number). The quantifiers can be translated straightforwardly using cardinality constraints as well as existential and universal quantification.

An Alloy signature may contain a list of fields, i.e., relations defined over the signature's elements. Since B natively supports relations, the translation is straightforward - for a signature s with fields f_i, each mapping an element of s to s_i, we add a constant f_i and state that f_i is a relation between s and s_i by the B constraint $f_i \in s \leftrightarrow s_i$.

It is also possible to make use of quantifiers when declaring field variables: In this way we can decide on the number of elements that are mapped to. The default quantification for relations in Alloy is a 1-to-1 mapping (Alloy quantifier `one`) while in B it is an 1-to-n mapping (Alloy quantifier `set`). Therefore, if no quantifier is given in the Alloy model, the translation to B has to be adapted, i.e., we add the constraint $f_i \in s \rightarrow s_i$, stating that f is a total function.

The translation of the remaining quantifiers is analogous, e.g., the quantifier `lone` results in a partial function. In case of `set`, no additional property is needed, since it is the default of B. Alloy allows to provide additional constraints on signature elements together with the signature definition. However, aside of syntactical sugar, they do not differ from regular constraints stated via `fact` declarations and are thus not considered further in this article.

3.2 Fact, Function & Predicate Declaration

Alloy's `fact` declaration has an optional name and contains any number of expressions, which pose additional constraints to be added to the model. The translation of these expressions will be discussed in Section 3.4. The results are conjoined and added to the `PROPERTIES` section of our B model.

Alloy allows to declare functions and predicates for later reuse. As usual, a function declaration takes a name, a (possibly empty) list of parameters and a body containing the actual computation. Parameters are scoped and can only be referred to by the function itself. Furthermore, they are typed as subsets of an Alloy signature and can again be quantified to constrain the set sizes.[5]

Functions will be listed in the `DEFINITIONS` section of the B model, if the model contains at least one invocation. Each function is translated into a single definition with matching parameters, consisting of a set comprehension wrapping the actual body to account for the expected return type, e.g., the function declaration `fun f [p : S] : S { body }` is translated into the B definition $f(p) == \{x|p \in S \wedge \text{body}\}$. We include the translation of parameter types conjoined with the translated body.

Syntax and functionality of the predicate definition is slightly different. For the predicate to evaluate to true or false instead of computing a value, we omit the set comprehension.

3.3 Assertion Declaration and Run & Check Commands

In Alloy, assertions can be stated using the `assert` declaration. An assertion does not immediately enforce further constraints. Rather it can later be verified or falsified in a given variable scope, using the `run` and `check` commands. To do so, assertions are named and contain any number of predicates to be checked. The predicates are translated and added to the `DEFINITIONS` clause of the model once they are used inside a `run` or `check` command.

The `run` command instructs the Alloy Analyzer to search for variable states that satisfy the model's constraints. It can either refer to a named predicate introduced by one of the declarations above or include an explicit Alloy predicate. The `check` command is used to check an assertion.

We introduce an `operation` to the B machine for each `run` and `check` command having the translated instructions of the command as its precondition. The operation's substitution is a skip, i.e., we only test if the operation can be executed, without any effect on the model. If the translated model satisfies the predicate to be checked, its specific operation is enabled.

Together with the predicate to be checked, both `run` and `check` include a scope, used to control the search space. By default, the scope defines an upper bound for the cardinality of a signature. The size can be set to a fixed value by using the keyword `exactly`. We define the translated scope in the precondition of the corresponding operation. For instance, the command `run p for 3 S`, for a predicate p and an unordered signature S, results to $card(S) \leq 3$ in B.

To run the Alloy check with ProB one can either use model checking, i.e., try all possible ways to instantiate the constants of the B translation and examine whether the operation is covered, or use constraint-based checking, e.g., using

[5] Quantifiers are used for typing but do not enforce restrictions on possible models.

the `cbc_sequence` command of ProB, which will send the operation's guard and the properties to ProB's constraint solver.

3.4 Expressions

Numbers, Identifiers and Blocks The most basic expressions in Alloy are numbers, identifiers and blocks. Renaming aside, numbers and identifiers can simply be copied to the B machine.

Integer arithmetic is available both in Alloy and in B. Operators have direct counterparts and no involved translation is needed. However, the Alloy Analyzer's approach of translating to SAT is limited: bit width has to be restricted and overflows might occur. We discuss several implications in Section 6.1.

Two kinds of blocks can be used for grouping and to manage precedences: `(` *expr* `)` and `{` *expr** `}`, where the list of expressions in the second case is connected conjunctively. Both can directly be translated.

Operations Aside of basic expressions mentioned above, Alloy expressions can be operations on expressions. As usual, the Alloy grammar distinguishes between unary, binary and comparison operators. For the sake of brevity, we will only discuss operators, that have no direct correspondence in B.

One such special case is the implication operation: *ifExpr* (`implies` | `=>`) *thenExpr* `else` *elseExpr*. B does not include a native if-then-else. However, we can achieve the same behavior using two implications: $ifExpr \Rightarrow thenExpr \wedge \neg ifExpr \Rightarrow elseExpr$.

The translation of the join operation is the most challenging one. Since all variables are tuples - either unary or binary ones - this operator can be used on any two variables (with the exception of two unary variables, which would always result in an empty set). Joining a field variable (binary tuples) with a signature variable (unary tuples), returns a set of unary tuples. Joining a field variable with another field variable, returns a set of binary tuples.

Unfortunately, there is no universal operator to achieve this behavior in B. Thus, we have a different translation for each of the three possibilities:

- Join of unary tuple e_u with binary tuple e_b is translated as the relational image $e_b[e_u]$,
- Join of binary tuple e_b with unary tuple e_u is translated as the relational image of the inverse binary tuple $(e_b \sim)[e_u]$,
- Join of two binary tuples b_1, b_2 is translated as the sequential substitution $(b_1; b_2)$.

In order to select the correct translation for the join operation, we compute the arity of the expressions involved.

Universe and Identity Alloy features two special constants: `univ`, referring to the set of all instances of all signatures and `ident`, the identity relation over the universe. Both are unavailable in B. To translate `univ`, we create a top-level set `UNIVERSE` and ensure that all base signatures implicitly extend it. This negatively impacts ProB's solving capabilities: without distinct sets for different signatures, techniques such as symmetry reduction cannot be applied as efficiently. ProB's kernel becomes unable to reason on types and thus has to perform more involved case distinctions. In consequence, we only create the universal type if necessary. Translation can be avoided in several typical use cases, e.g., left and right joins with the universe can be translated into domain and range computation.

Using `UNIVERSE`, we could translate `ident` to `id(UNIVERSE)`. However, as we want to avoid the universal type as much as possible, we again chose a more specialized translation wherever possible, i.e., instead of translating into the identity over the universe, we rely on Alloy's type checking information and translate into a more restricted identity relation.

Let, Quantified Expressions and Set Comprehensions As discussed in the introductory example in Section 2, classical B does not feature a let expression. This can either be resolved by using a definition as done in the example, inlining or by using the extended version of B understood by ProB.

Alloy features two types of quantified expressions, one that constrains the cardinality of sets and one that allows to introduce quantified variables. The first one uses the quantifiers introduced in Section 3.1 followed by a set expression, e.g., `no Number & Letter`, and is translated accordingly. In the second case, quantified variables are introduced and translated to B using set comprehensions as well as universal and existential quantification.

Both Alloy and B feature set comprehensions, consisting of local identifiers and a constraining predicate. Translation is straightforward, as only the predicate has to be translated according to the rules given above.

4 Translating Orderings

Alloy data types are universally based on relations. For instance, sets are unary relations while scalars are singleton sets. Signatures are not ordered by default. However, Alloy allows to declare a total order on signature elements.

Alloy offers the operations *first*, *last*, *next*, *prev*, $nexts(s)$ and $prevs(s)$ for element access on ordered signatures. For an ordered Signature s_o, $s_o/nexts(s)$ returns the set of all successors of $s \in s_o$.

Initially, we translated ordered signatures to B sequences. Sequences are ordered sets of couples whose domains are finite and enumerated from $1..n$, where n is the number of elements. Usually, we translate an Alloy signature to a deferred set in B having the same name as described in Section 3.1. The ordered signature can then be represented by a sequence of type s_o, i.e., a set of couples of integer and s_o. B directly offers the operations *first*, *last*, *next*, *prev* for sequences while $nexts(s)$ and $prevs(s)$ can be implemented using set comprehensions.

However, ProB's performance on predicates involving sequences can be lacking when compared to (sets of) integers. In consequence, we tried a different translation: The scope of a signature is defined within the run or check statement of an Alloy model. Assuming the ordered signature s_o has size $k \in \mathbb{N}$, we translate it to an interval $1, \ldots, k$ in B.

However, we have to consider that ordered signatures can interact, e.g., when computing the union. In consequence, we ensure that ordered signatures are distinct by translating them into disjoint intervals.

Besides that, ordered signatures might interact with unordered ones in Alloy. We then have to define the unordered signature as a set of integer too to avoid type errors in B. To do so, we check an Alloy model for interactions between ordered and unordered signatures prior to the translation.

Using integer intervals, we can define the operations provided by Alloy orderings using set comprehensions. For *first* and *last* we memorize the bounds of each defined set in B which are constant values. We then define $s_o/next$ and $s_o/nexts(s)$ (and *prev* and *prevs*(s) analogously) for a signature s_o as

$$next(s) == \{x | x = s + 1 \land x \in s_o\} \qquad nexts(s) == \{x | x > s \land x \in s_o\}.$$

5 Empirical Evaluation

To validate the correctness of our translation we have applied it to a variety of mathematical laws and have checked that ProB does not find counter examples to those laws on the translated B models. In this section we will give a brief empirical evaluation, comparing the Alloy Analyzer and ProB applied to Alloy models. Since the Alloy Analyzer translates models to SAT, we assume it to be efficient for mostly relational models. However, for integers SAT encoding is often inefficient, e.g., one has to encode arithmetic using binary adders. ProB on the other hand has native support for integers, hopefully leading to better performance for arithmetic calculations. In contrast, relations often cause a combinatorial explosion, which results in a weaker performance compared to the Alloy Analyzer.

To explore both extremes, we chose two different models: First, we translate an Alloy model of the river crossing puzzle, a type of transport puzzle with the goal to carry several objects from one river bank to another. There are constraints defining which objects are safe to be left alone, e.g., a fox can not be left alone with a chicken. The model uses an ordered signature for states.

Second, we translate a model of the n queens problem. Here, the goal is to place n queens on a $n*n$ chess board without two queens threatening each other. The chess board is represented as tuples of row and column, encoded as integers.

Benchmarks were run on an Intel Core I7-7700HQ CPU (2.8GHz) and 32GB of RAM. We use the median time of five independent checks where the runtime of the Alloy analyzer includes generating the conjunctive normal form.

For the river crossing puzzle, the Alloy analyzer finds a solution in $10ms$. The translated model is valid, yet ProB fails to find a solution in < 5 minutes.

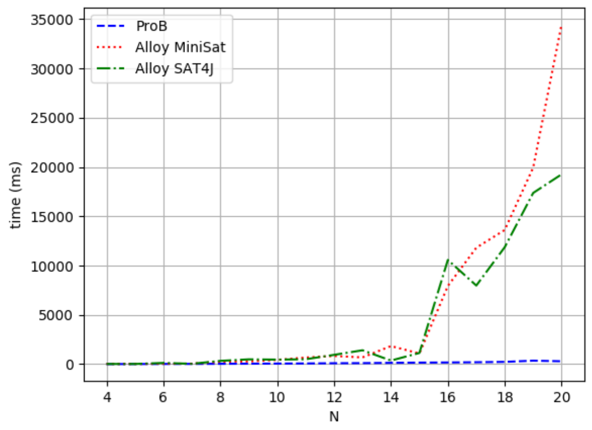


Fig. 2. Find a Single Solution for the n Queens Puzzle with varying N

The B model defines three relations, two of which have an ordered signature for a domain. Using a total function instead of a relation improves performance: ProB now finds a solution in $\sim 7s$. After rewriting the model in idiomatic B style by hand, ProB can solve it in about $80ms$. However, this translation is a manual optimization using background knowledge and cannot simply be generalized. An exact opposite to our translation is [28], which uses the Alloy Analyzer's Kodkod API [30] to translate B to SAT. When we use this backend within ProB, the unmodified Alloy translation is solved in about 0.3 seconds. Note that in recent work [15], we have shown that an integration of the Alloy and ProB backend can be very useful for complex constraint satisfaction problems.

We evaluated the n queens model for $n \in 4..20$ using ProB and the Alloy analyzer with the MiniSat and SAT4J backend. The evaluation in Figure 2 shows that ProB is the fastest solver for the chosen model. The runtime of the Alloy analyzer gets worse when increasing the bit-width for $n \geq 8$ and $n \geq 16$.

6 Improvements over existing Alloy Tools

Even though our translation cannot always compete with the Alloy Analyzer as we have demonstrated in Section 5, it provides several interesting improvements and applications.

6.1 Integers

Mathematically speaking, integers in Alloy are unsound due to overflows. In contrast, ProB has multi-precision integers without overflows[6]. Using [24] the

[6] CLP(FD) overflows are caught and handled by custom implementation.

Alloy Analyzer can detect models with overflows, but to our knowledge cannot detect where an overflow has prevented a model being found. For this purpose, an alternative to translation is to use an SMT-based backend, e.g., [31,7] or [22].

For example, for the following model Alloy 4.2 finds a counter example, while ProB correctly determines that no counter example exists. If overflows are permitted (the default), the Alloy Analyzer finds a counter example for the first formula. If overflows are forbidden, no counter example is detected by the Alloy Analyzer for the first formula, but then a counter example is found for the second one. With higher integer ranges the translation fails.

```
open util/integer
abstract sig setX { }
one sig V {
  SS:   setX -> setX
}
assert Bug {
 #(V.SS)>1 implies #(V.SS->V.SS) >3
 #(V.SS->V.SS)=0 iff no V.SS
}
check Bug for 3 setX, 7 int // for 8 int Translation capacity exceeded
```

6.2 Higher-Order Quantification

The universal quantification below, using the same signatures as in Section 6.1 above, causes an error[7], while ProB can check the validity of this assertion. An extension of Alloy called Alloy* [25] might be able to handle this example. In future, we would like to investigate translating Alloy* models to B.

```
assert HO {
   V.SS + V.SS = V.SS
   all xx : V.SS | (xx in V.TT implies xx in V.SS & V.TT)
}
check HO for 3 setX
```

6.3 Proof

Finally, our translation to B also makes it possible to apply its provers, such as [3]. One could thus try and develop a proof assistant for Alloy, similar to the work pursued in [32] by a translation to Key.

In the example below, we can prove the assertion using AtelierB's prover for any scope, by applying it to the translated B machine. We check that the move predicate preserves the invariant `src+dst=Object`.

```
sig Object {}
sig Vars {
```

[7] Analysis cannot be performed since it requires higher-order quantification that could not be skolemized.

```
    src,dst : Object
}
pred move (v, v': Vars, n: Object) {
    v.src+v.dst = Object
    n in v.src
    v'.src = v.src - n
    v'.dst = v.dst + n
}
assert add_preserves_inv {
    all v, v': Vars, n: Object |
         move [v,v',n] implies  v'.src+v'.dst = Object
}
check add_preserves_inv for 3
```

Note that our translation does not (yet) generate an idiomatic B encoding, with `move` as a B operation and `src+dst=Object` as an invariant: it generates a check operation encoding the predicate `add_preserves_inv` with universal quantification. Below we show the B machine we have input into AtelierB. It was obtained by pretty-printing from ProB, and putting the negated guard of the `add_preserves_inv` into an assertion (so that AtelierB generates the desired proof obligation).

```
MACHINE alloytranslation
SETS /* deferred */
  Object_; Vars_
CONCRETE_CONSTANTS
  src_Vars, dst_Vars
PROPERTIES
    src_Vars : Vars_ --> Object_
  & dst_Vars : Vars_ --> Object_
ASSERTIONS
  !(v_,v__,n_).(v_ : Vars_ & v__ : Vars_ & n_ : Object_
   =>
   (src_Vars[{v_}] \/ dst_Vars[{v_}] = Object_ &
    v_ |-> n_ : src_Vars &
    src_Vars[{v__}] = src_Vars[{v_}] - {n_} &
    dst_Vars[{v__}] = dst_Vars[{v_}] \/ {n_}
    =>
    src_Vars[{v__}] \/ dst_Vars[{v__}] = Object_)
   )
END
```

7 Related Work, Future Work and Conclusions

Translations to Alloy directly have been pursued from B [23,21] and also Z [20]. Other formal languages have previously been translated to B as well [27,8]. A comparison between TLA+ and Alloy can be found in [19].

The original paper [9] (notably figure 2 in [9]) provides a semantics of the kernel of Alloy in terms of logical and set-theoretic operators. Our translation rules can be seen as an alternate specification of this semantics, using the B operators and also using B quantification.

While our translation of orderings, as given in Section 4, allows to translate arbitrary Alloy models, the resulting B machine is often suboptimal for ProB's solving kernel. To improve performance, we want to investigate a translation into a (bounded or explicit) model checking rather than a constraint problem. In particular, we intend to translate predicates over states and their successors into B operations. While this is not possible in general, e.g., in the presence of predicates relating more than two states, it would allow us to use symbolic model checking algorithms [14] to find solutions. [26] presents an imperative extension of Alloy, i.e, making a step towards B and its operations. In a similar fashion, [5,6] extended Alloy with actions or bounded model checking [4]. It would be interesting to extend our translation and produce idiomatic B machines with B operations from such Alloy models.

In summary, we have presented an automatic translation of Alloy to B, which provides an alternative semantic definition of Alloy, enables proof and constraint solving tools of B to be applied, and can serve as a vehicle of communication between the Alloy and B community.

References

1. J.-R. Abrial. *The B-book: Assigning Programs to Meanings.* Cambridge University Press, New York, NY, USA, 1996.
2. M. Carlsson, G. Ottosson, and B. Carlson. An open-ended finite domain constraint solver. In *Proceedings PLILP*, volume 1292 of *LNCS*, pages 191–206. Springer, 1997.
3. ClearSy. *Atelier B, User and Reference Manuals.* Aix-en-Provence, France, 2009. Available at `http://www.atelierb.eu/`.
4. A. Cunha. Bounded Model Checking of Temporal Formulas with Alloy. In *Proceedings ABZ*, volume 8477 of *LNCS*, pages 303–308, 2014.
5. M. F. Frias, J. P. Galeotti, C. L. Pombo, and N. Aguirre. DynAlloy: upgrading alloy with actions. In *Proceedings ICSE*, pages 442–451, 2005.
6. M. F. Frias, C. L. Pombo, J. P. Galeotti, and N. Aguirre. Efficient Analysis of DynAlloy Specifications. *ACM Trans. Softw. Eng. Methodol.*, 17(1):4:1–4:34, 2007.
7. A. A. E. Ghazi and M. Taghdiri. Analyzing Alloy Formulas using an SMT Solver: A Case Study. *CoRR*, abs/1505.00672, 2015.
8. D. Hansen and M. Leuschel. Translating TLA+ to B for validation with ProB. In *Proceedings iFM*, volume 7321 of *LNCS*, pages 24–38. Springer, 2012.
9. D. Jackson. Alloy: A Lightweight Object Modelling Notation. *ACM Transactions on Software Engineering and Methodology*, 11:256–290, 2002.
10. D. Jackson. *Software Abstractions: Logic, Language and Analysis.* MIT Press, 2006.
11. J. Jaffar and S. Michaylov. Methodology and Implementation of a CLP System. In *Proceedings ICLP*, pages 196–218. MIT Press, 1987.
12. S. Krings and M. Leuschel. Constraint Logic Programming over Infinite Domains with an Application to Proof. In *Proceedings WLP*, volume 234 of *EPTCS*. Electronic Proceedings in Theoretical Computer Science, 2016.
13. S. Krings and M. Leuschel. SMT Solvers for Validation of B and Event-B models. In *Proceedings iFM*, volume 9681 of *LNCS*. Springer, 2016.

14. S. Krings and M. Leuschel. Proof Assisted Bounded and Unbounded Symbolic Model Checking of Software and System Models. *Sci. Comput. Program.*, 2017.
15. S. Krings, M. Leuschel, P. Körner, S. Hallerstede, and M. Hasanagic. Three Is a Crowd: SAT, SMT and CLP on a Chessboard. In *Proceedings PADL 2018*, pages 63–79, 2018.
16. M. Leuschel, J. Bendisposto, I. Dobrikov, S. Krings, and D. Plagge. From Animation to Data Validation: The ProB Constraint Solver 10 Years On. In J.-L. Boulanger, editor, *Formal Methods Applied to Complex Systems: Implementation of the B Method*, chapter 14, pages 427–446. Wiley ISTE, Hoboken, NJ, 2014.
17. M. Leuschel and M. Butler. ProB: A model checker for B. In *Proceedings FME*, volume 2805 of *LNCS*, pages 855–874. Springer, 2003.
18. M. Leuschel and M. Butler. ProB: an automated analysis toolset for the B method. *Int. J. Softw. Tools Technol. Transf.*, 10(2):185–203, 2008.
19. N. Macedo and A. Cunha. Alloy meets TLA+: An exploratory study. *CoRR*, abs/1603.03599, 2016.
20. P. Malik, L. Groves, and C. Lenihan. Translating Z to Alloy. In *Proceedings ABZ*, volume 5977 of *LNCS*, pages 377–390, 2010.
21. P. J. Matos and J. Marques-Silva. Model Checking Event-B by Encoding into Alloy. In *Proceedings ABZ*, volume 5238 of *LNCS*, page 346, 2008.
22. B. Meng, A. Reynolds, C. Tinelli, and C. W. Barrett. Relational constraint solving in SMT. In *Proceedings CADE*, volume 10395 of *LNCS*, pages 148–165, 2017.
23. L. Mikhailov and M. J. Butler. An Approach to Combining B and Alloy. In *Proceedings ZB*, volume 2272 of *LNCS*, pages 140–161. Springer, 2002.
24. A. Milicevic and D. Jackson. Preventing arithmetic overflows in Alloy. *Sci. Comput. Program.*, 94:203–216, 2014.
25. A. Milicevic, J. P. Near, E. Kang, and D. Jackson. Alloy*: a general-purpose higher-order relational constraint solver. *Formal Methods in System Design*, Jan 2017.
26. J. P. Near and D. Jackson. An Imperative Extension to Alloy. In *Proceedings ABZ*, volume 5977 of *LNCS*, pages 118–131, 2010.
27. D. Plagge and M. Leuschel. Validating Z Specifications using the ProB Animator and Model Checker. In *Proceedings iFM*, volume 4591 of *LNCS*, pages 480–500. Springer, 2007.
28. D. Plagge and M. Leuschel. Validating B, Z and TLA$^+$ using ProB and Kodkod. In *Proceedings FM*, volume 7436 of *LNCS*, pages 372–386. Springer, 2012.
29. A. Sülflow, U. Kühne, R. Wille, D. Große, and R. Drechsler. Evaluation of SAT-like Proof Techniques for Formal Verification of Word-Level Circuits. In *Proceedings IEEE WRTLT*, Beijing, China, Oct. 2007. IEEE Computer Society Press.
30. E. Torlak and D. Jackson. Kodkod: A Relational Model Finder. In *Proceedings TACAS*, volume 4424 of *LNCS*, pages 632–647. Springer, 2007.
31. E. Torlak, M. Taghdiri, G. Dennis, and J. P. Near. Applications and extensions of Alloy: past, present and future. *Mathematical Structures in Computer Science*, 23(4):915–933, 2013.
32. M. Ulbrich, U. Geilmann, A. A. E. Ghazi, and M. Taghdiri. A Proof Assistant for Alloy Specifications. In *Proceedings TACAS*, volume 7214 of *LNCS*, pages 422–436, 2012.