Translating Alloy and Extensions to Classical B

Sebastian Krings[a], Michael Leuschel[a], Joshua Schmidt[a], David Schneider[a], Marc Frappier[b]

[a] *Institut für Informatik, Universität Düsseldorf Universitätsstr. 1, 40225 Düsseldorf, Germany*
[b] *Université de Sherbrooke, Québec, Canada*

Abstract

In this article, we introduce a denotational 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, sequences and orderings. The translation is fully automated and our implementation can be used in ProB. We evaluate the usefulness by applying AtelierB and ProB to translated models, showing benefits for proof and constraint solving with integers and higher-order quantification.

1. Introduction

Both B [1] and Alloy [2] 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 defining 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 [2] is used to compute models by translating Alloy predicates to SAT using Kodkod [3]. The most prominent constraint solver for B, ProB [4, 5, 6], however mainly relies on constraint logic programming [7]. In particular, it uses the CLP(FD) library of SICStus Prolog [8] and extends it to support constraints over infinite domains [9]. Additionally, ProB allows using other backends, such as SMT solvers [10] or, again, Kodkod [11].

The different constraint solving techniques show different performance characteristics [12]. Certain predicates can be solved faster by using a particular

Email addresses: `sebastian.krings@hhu.de` (Sebastian Krings), `michael.leuschel@hhu.de` (Michael Leuschel), `joshua.schmidt@hhu.de` (Joshua Schmidt), `david.schneider@hhu.de` (David Schneider), `marc.frappier@usherbrooke.ca` (Marc Frappier)

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 cannot be handled efficiently by the Alloy Analyzer,
- it enables the application of the AtelierB provers [13] 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.

Our translation is integrated into ProB and is available at:

`https://www3.hhu.de/stups/prob`

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

`https://github.com/hhu-stups/alloy2b-doc`

This article is the extended version of our original ABZ submission [14]. For this article we extend our former work [14] in different aspects:

- The informal, more intuitive description of the translation from Alloy to B has been replaced by a formal description.
- We revised the translation of operations on orderings.
- The translation supports more Alloy constructs than the initial one. In particular, we added support for sequence operations, additional constraints on relations defined in `util/relation` and completed the translation of the join operator as well as all multiplicity annotations.
- We describe the tooling used in our translator in greater detail.
- Several special and edge cases are discussed more thoroughly.
- The empirical evaluation has been extended with more examples.

2. Introduction to Alloy and B

In the following, we will give brief introductions to Alloy and B, discussing their approach to modeling and their specific features. Afterwards, we will point out the main differences between both languages.

2.1. Primer on Alloy

The Alloy notation is based on first-order logic with n-ary finite relations as the only type of terms. Sets are represented as unary relations. Basic sets and relations are defined using signatures, a construct similar to classes in object-oriented programming languages, which supports inheritance.

An Alloy specification consists of a set of signatures, noted **sig**, which basically define sets and relations, and a set of constraints, noted **fact**, that are first-order formulae which condition the values of sets and relations. A model can also contain assertions, which should hold when the facts are satisfied. The declaration `sig X {r : Y}` declares a signature (unary relation) X and a binary relation r which is a subset of the Cartesian product $X \times Y$. Alloy supports the usual operations on relations, like union, intersection, difference, join, transitive closure, domain and range restriction. Fields (relations) of a signature are accessed using a convenient object-like notation (*e.g.*, $x.r = y$, with $x \in X$, $y \in Y$, and "." denotes the relational join operator). Alloy provides a universal type, noted **univ**, which is the union of all signatures. **Int** is the only predefined type; it is represented by the interval $-2^{n-1}..2^{n-1} - 1$, where n is the number of bits used to store **Int** values. UML-like cardinality constraints can be defined on relations. Functions and predicates can be declared.

Alloy specifications can be decomposed into modules. Predefined modules provide support for booleans, sequences, directed graphs and total orderings on signatures.

The Alloy tool provides an editor, a model finder/enumerator and a model viewer based on the dot package. Alloy uses SAT solvers to build models to verify the satisfiability of axioms (facts) defined in an Alloy specification and to find counterexamples for assertions which should follow from these axioms. Only finite models are explored; their size is determined by a scope specified for each signature. Alloy facts and signatures are translated into Boolean formulas using Kodkod [3].

2.2. Primer on B

The formal specification language B [1] is based on first-order-logic and set theory and follows the correct-by-construction approach. B has initially been developed for the specification and design of software systems. Specific properties can be proven mathematically using theorem provers, *e.g.*, using AtelierB [13], or be checked using a model checker such as ProB [4, 5, 6].

A formal model in B consists of a collection of machines, containing a high-level abstract specification which is successively refined and decomposed. The development in B is thus incremental, which increases the maintainability and eases the specification of complex models.

A machine consists of variable and type definitions as well as initial values. A state is defined by the current values of the machine variables. By defining machine operations, one is able to specify transitions between states, effectively computing successor states featuring all variables. A machine operation has a unique name and consists of a B substitution (aka statement) defining the machine state after its execution. An operation can have a precondition, allowing or prohibiting execution based on the current state. For instance, a valid machine operation `o` is defined by `o = PRE x>0 THEN x:=x+1 END` using the single assignment substitution of B. Several variables can be assigned either in parallel or in sequence.

To ensure a certain behavior, the user can define machine invariants, *i.e.*, safety properties that have to hold in every reachable state. Hence, the correctness of a formal model refers to the specified invariants. ProB further supports linear temporal logic and fairness constraints, which enables the specification and verification of liveness properties.

In addition to the types explicitly provided by the B language such as `INTEGER` or `BOOL`, one can provide user-defined sets. These sets can be defined by a finite enumeration of distinct elements (the set is then referred to as an enumerated set) or left open (called deferred sets). For instance, by defining a set $S = \{s1\}$ the element $s1$ is of type S and can be accessed by name within the machine. Deferred sets are assumed to be non-empty during proof and also finite for animation.

B is statically and strongly typed while ProB further executes runtime checks to ensure well-definedness. Type domains can be unbounded, possibly resulting in a model with an infinite state space. Further, B and ProB have support for higher-order quantification, in particular, sets and relations can be nested arbitrarily.

Code generators can be used to derive executable code from B models, targeting traditional programming languages such as C, C++ or Java.

2.3. Comparing Alloy and B

Although Alloy and B share common features, both languages have considerable differences. Most notably, Alloy and B have a different understanding of states. In B, state changes are encoded by means of operation executions, leading to successor states featuring all variables. At its heart, Alloy only has a single constant state, there is no concept of an operation. Alloy, however, allows defining orderings, allowing one to reason about sequences of states. In contrast to B, a predicate can then access any state in the sequence, not just the current state and its immediate successors.

Further, B is strongly typed, while Alloy only enforces the arities of an operation's arguments to match and provides the universal type, *i.e.*, the union of all signatures. On the one hand, strong typing is less error-prone than weak typing and enables a wide range of code analysis techniques to be applied. For instance, ProB throws a well-definedness error if a sequence operation is called on an improper sequence, whereas this is permitted in Alloy (but might not always be desired). On the other hand, strong typing possibly hinders a concise and idiomatic specification of software systems, especially in the context of object-oriented programming languages.

In B, tuples are encoded as nested pairs. Thus, several encodings of tuples exist and the modeler has to know which one is being used. For example, a triple can be represented as either $(x \mapsto (y \mapsto z))$ or $((x \mapsto y) \mapsto z)$. In Alloy, tuples

Listing 1: Own Grandpa (Alloy - Signatures)

```
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   }
```

are flat. This makes the join operator of Alloy powerful and enables expressing certain constructs more concisely than is possible in B.

Alloy follows the small scope hypothesis, which states that most bugs can be found by testing a program within a small scope [15], and bounds every domain. Hence, the Alloy Analyzer is not able to generally prove properties for a model but show the absence of counterexamples within a restricted scope. In contrast, domains can be unbounded in B. Besides using AtelierB's theorem provers, symbolic model checking techniques of ProB can be used to verify properties on infinite state spaces [33].

B and ProB have support for higher-order quantification, in particular, sets and relations can be nested arbitrarily. Higher-order specifications are also expressible in Alloy but cannot be handled by the Alloy Analyzer (an error is thrown). Alloy* [16] is an extension of Alloy which is able to do so.

3. Translation Example

In the following section, we will introduce our translation on a simple Alloy model taken from Chapter 4 in "Software Abstractions: Logic, Language, and Analysis" [2]. 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:

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 defined in the `PROPERTIES` clause. This includes typing predicates for the constants.
4. `DEFINITIONS`, aka B macros, to ease translating certain Alloy concepts.
5. B Operations for Alloy assertion checks.

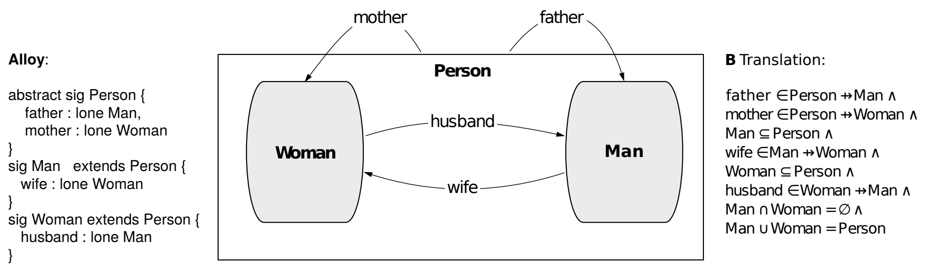
In particular, our translation does not use variables, invariants or assertions.

3.1. 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.

Listing 2: Own Grandpa (B - Signatures)

```
1  MACHINE SelfGrandpas
2  SETS
3      Person
4  CONSTANTS
5      Man, Woman, father, mother, wife, husband
6  PROPERTIES
7      father ∈ Person ⇸ Man ∧
8      mother ∈ Person ⇸ Woman ∧
9      Man ⊆ Person ∧
10     wife ∈ Man ⇸ Woman ∧
11     Woman ⊆ Person ∧
12     husband ∈ Woman ⇸ Man ∧
13     Man ∩ Woman = ∅ ∧
14     Man ∪ Woman = Person
15 END
```



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
```

Figure 1: Signatures and Fields in the Own Grandpa Model

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. 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. Deferred sets in B can have any size, just like signatures in Alloy. In Section 4.14 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 signatures `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

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
```

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.

3.2. 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 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 relational composition of the two relations, which is available in B by using the `;` operator. Other interpretations of the dot join operator will be discussed later. The keyword **no** 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 their own ancestor, introduces a quantified local variable `p`. Again, **no** enforces emptiness of the set. We translate the fact into a negated existential quantification, which is able to introduce the variable. Observe, that quantification in Alloy is over singleton sets only. More generally, we translate the quantification **no** $p \, : \, S \;\; | \;\; P$ into $\neg\exists p.(\{p\} \subseteq S \wedge P)$.

The function definition `grandpas` and the predicate definition `ownGrandpa`, both with a parameter, are encoded as B definitions, permitting their reuse throughout the model. The predicate definition `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⁻¹ ∧
    wife ∩ (closure((mother ∪ father)) ; mother) = ∅ ∧
    husband ∩ (closure((mother ∪ father)) ; father) = ∅ ∧
    not(∃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.[1] 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 in order 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 `Person`s, the definition again uses a set comprehension.

4. Formal Description of the Translation

The original paper by Daniel Jackson [17] (notably Figure 2) as well as Appendix C in "Software Abstractions: Logic, Language and Analysis" [2] provide a semantics of the kernel of Alloy in terms of logical and set-theoretic operators. It introduces a function M to give the meaning of formulas (aka predicates) and a function E that gives the meaning of expressions. One of the rules defined by Jackson [2] gives the meaning of the + operator as the set-theoretic union of the meaning of its arguments:

- $E[\![p + q]\!]i \ \widehat{=}\ E[\![p]\!]i \cup E[\![q]\!]i$

The argument i is an environment where some identifiers can be given a value. The environment is used to deal with quantifiers and identifiers: quantifiers update the environment, while the function is applied when computing the meaning of an identifier r as follows: $E[\![r]\!]i = i(r)$.

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

Our translation rules are an alternate specification of this semantics, using the B operators and also using B quantification. Our translation rules are more comprehensive and sometimes more involved due to the following reasons:

- The B language has a more restrictive syntax concerning set comprehensions and always requires explicit quantification of all introduced identifiers, in contrast to the "flexible" mathematical notation employed by Jackson [2, 17].
- In Alloy tuples are flat and Cartesian product is associative; in B Cartesian product is **not** associative and tuples are represented as nested pairs.
- We have to provide the full translation for all operators. (Jackson [2, 17] presents the kernel semantics and only a few translation rules for the full language to the kernel language.)
- The translation by Jackson [2, 17] does not specify all aspects of encoding signatures, which we have to deal with in an automated translation.

4.1. Overview of the Semantic Functions

We provide four semantic functions, one for expressions, one for predicates, and two for declarations that introduce new quantified variables.

Each semantic function has an argument i which is an environment storing different information to be used during translation. In particular, the information is used for optimization as it allows using more specialized encodings in certain situation. For instance, the environment stores identifiers that are singleton sets. Thus, if $x \in i$, then x is translated as $\{x\}$, whereas identifiers not in i are translated as x. This information is relevant to obtain a more effective encoding into B, using scalar values instead of set values whenever possible. Note that, as we translate Alloy quantification to B quantification, there is no need to store values for the quantified variables as in the functions defined by Jackson [2, 17]. However, we also store information about identifiers which are known to be total functions in i. We are then able to translate specific Alloy constructs in a more idiomatic and, in terms of solving constraints, more efficient way. For instance, when using total functions, we can translate element access using B's function application instead of using the relational image operator.

For the sake of readability and brevity, our translation rules will only detail how identifiers representing singleton sets are tracked. Identifiers for total functions are tracked similarly.

In particular, we provide the following semantic functions:

1. $E[\![A]\!]i$ is the B encoding of the Alloy **expression** A given the environment i.
2. $M[\![A]\!]i$ is the B encoding of the Alloy **predicate** A given the environment i.
3. $D[\![A]\!]i$ is the B encoding of the Alloy **quantification declaration** A given the environment i. $I[\![A]\!]i$ is the updated set of identifiers after processing the declaration A.

4. $F[\![A]\!]i$ is the B encoding of the Alloy **signature field declaration** A given the environment i.

4.2. Example

Before presenting the rules in detail, we process a small sub-predicate from Section 3.2:

no wife & *(mother + father).mother

To translate this Alloy predicate to B we need the following semantic rules:

- $M[\![\textbf{no}\ p]\!]i \ \widehat{=}\ E[\![p]\!]i = \varnothing$
- $E[\![p\ \texttt{\&}\ q]\!]i \ \widehat{=}\ E[\![p]\!]i \cap E[\![q]\!]i$
- $E[\![p\ \texttt{+}\ q]\!]i \ \widehat{=}\ E[\![p]\!]i \cup E[\![q]\!]i$
- $E[\![x]\!]i \ \widehat{=}\ x$ for identifiers not occurring in i (*e.g.*, signature names and fields)
- $E[\![\texttt{*}p]\!]i \ \widehat{=}\ \text{closure}(E[\![p]\!]i)$
- $E[\![p.q]\!]i \ \widehat{=}\ (E[\![p]\!]i; E[\![q]\!]i)$ if both p and q are binary relations

Note that closure is B's transitive and reflexive closure operator on relations, while ";" is the relational composition operator.

Here is a step-by-step application of these rules to obtain the B translation, where $i = \varnothing$.

1. $M[\![\textbf{no}\ \texttt{wife \& *(mother + father).mother}]\!]i$
2. $E[\![\texttt{wife \& *(mother + father).mother}]\!]i = \varnothing$
3. $E[\![\texttt{wife}]\!]i \cap E[\![\texttt{*(mother + father).mother}]\!]i = \varnothing$
4. $\text{wife} \cap (E[\![\texttt{*(mother + father)}]\!]i\ ;\ E[\![\texttt{mother}]\!]i) = \varnothing$
5. $\text{wife} \cap (\text{closure}(E[\![\texttt{(mother + father)}]\!]i)\ ;\ \text{mother}) = \varnothing$
6. $\text{wife} \cap (\text{closure}(E[\![\texttt{mother}]\!]i \cup E[\![\texttt{father}]\!]i)\ ;\ \text{mother}) = \varnothing$
7. $\text{wife} \cap (\text{closure}(\text{mother} \cup \text{father})\ ;\ \text{mother}) = \varnothing$

4.3. Signature and Field Declarations

Since a signature declaration can be quite complex, let us start with the simplest 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. In B, a deferred set introduces a user-defined type as a finite and non-empty set.

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, with $s_1 \neq s_2$, 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 [2]. 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 stating the cardinality of signatures using multiplicities. The quantifiers **lone** (at most one) and **some** (at least one) are translated straightforwardly using cardinality constraints in the B machine's `PROPERTIES` section. In case of **one** (exactly one), we define a signature as a singleton enumerated set in B instead of a deferred set.

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 $f_{S,j}$, each mapping an element of S to S_j, we add a constant $f_{S,j}$ and state that $f_{S,j}$ is a relation between S and S_j by the B constraint $f_{S,j} \in S \leftrightarrow S_j$.

We have to ensure that the names of the constants $f_{S,j}$ are unique in terms of the current model since fields of different signatures can have the same name in Alloy. Otherwise, the resulting B machine might not be well-defined because of clashes between constants with the same name but different types.

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 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 $f_{S,j} \in S \to S_j$, stating that $f_{S,j}$ is a total function. The translation of the quantifier **lone** results in a partial function. In case of **set**, no additional property is needed, since it is the default of B. For the multiplicity **some**, we add the additional constraint $\mathrm{dom}(f_{S,j}) = S$, *i.e.*, the field $f_{S,j}$ is a total relation.

Furthermore, a signature's field can be defined as a sequence of atoms. We translate a sequence q as a partial function with a finite and coherent domain $0..\,\mathrm{card}(q) - 1$ and ensure that the maximum allowed length of sequences $m \in \mathbb{N}$ is not exceeded. As a signature can have several instances, the complete domain of a signature field that is a sequence is defined as the Cartesian product of the signature and the partial function. We will explain the translation of Alloy sequences to B more precisely in Section 5.5.

Taken together, the formal translation of field quantifications for an exemplary signature `S` to B machine properties is as follows:

- $F[\![\mathit{field} : \textbf{set}\ \mathit{Set}]\!]i \ \widehat{=}\ \mathit{field} \in E[\![S]\!]i \ \leftrightarrow\ E[\![\mathit{Set}]\!]i$

- $F[\![field : \mathbf{one}\ Set]\!]i \ \widehat{=}\ field \in E[\![S]\!]i \ \rightarrow\ E[\![Set]\!]i$
- $F[\![field : \mathbf{lone}\ Set]\!]i \ \widehat{=}\ field \in E[\![S]\!]i \ \nrightarrow\ E[\![Set]\!]i$
- $F[\![field : \mathbf{some}\ Set]\!]i \ \widehat{=}\ field \in E[\![S]\!]i \ \leftrightarrow\ E[\![Set]\!]i \wedge \mathrm{dom}(field) = E[\![S]\!]i$
- $F[\![field : \mathbf{seq}\ Set]\!]i \ \widehat{=}\ field \in (E[\![S]\!]i \ \times\ 0..(m-1)) \ \nrightarrow\ E[\![Set]\!]i \ \wedge$ $\forall s.(s \in E[\![S]\!]i \Rightarrow \mathrm{dom}(field)[\{s\}] = 0..\,\mathrm{card}(\mathrm{dom}(field)[\{s\}]) - 1)$
- $F[\![field :\ Set]\!]i \ \widehat{=}\ F[\![field : \mathbf{one}\ Set]\!]i$

Besides constraining the quantification of signature fields, one can use the keyword **disj** in combination with any multiplicity in order to define that distinct members of a signature yield distinct field values. For instance, the signature **sig** S $\{f :\ $ **disj** $e\}$ defines the signature field f to be disjoint for distinct members of S. This results in the additional constraint **all** $a, b : S \mid a\ != \ b$ **implies no** $a.f$ & $b.f$ [2]. We straightforwardly translate the universal quantification and add it to the translated B machine's `PROPERTIES` section.

Alloy allows providing 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 section.

4.4. Universe and Identity

Alloy features two special constants: **univ**, referring to the set of all instances of all signatures and **iden**, the identity relation over the universe. Neither is available in B. To translate **univ**, we could 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. Further, in case integers are used in an Alloy model, we would need to create a singleton set for each integer extending the set `UNIVERSE` although B and ProB have native support for real integers. In consequence, we only create parent types for specific signatures if necessary.

For instance, if two signatures $S1$ and $S2$ have no parent type except for **univ** and interact with each other using a union $S1 + S2$, we introduce an additional deferred set in the translated B machine and declare membership for both signatures. The two signatures are then defined as machine constants rather than deferred sets in B. For above example, this results to $S1 \cap S2 = \varnothing \wedge S1 \cup S2 = P$ being added to the B machine properties where P is the introduced parent type, *i.e.*, a deferred set in B.

To do so, we analyze an Alloy model prior to the translation and collect pairs of signatures that interact with each other and have no parent type except for **univ**. All collected pairs are merged to distinct sets of signature types where in each set at least two signatures interact with each other. When translating an Alloy model, we define a parent type for each set of signatures. In case all signatures of a model interact with each other, our translation introduces one parent type for all signatures. Despite containing integers, this type is equal to Alloy's universal type given a specific model.

As we use the integer type of B to translate Alloy integers, which cannot be a subset of a deferred set, we cannot translate interactions between signatures and integers straightforwardly. To do so, we would need to create a singleton set for each integer extending the set `UNIVERSE` in B. To our knowledge, a binary interaction between a signature type and integers is not needed in any reasonable Alloy model. However, it is allowed by the Alloy Analyzer's typechecker.

For binary operations, the translation of **univ** 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 **iden** to id(`UNIVERSE`). However, as we want to avoid the universal type as much as possible, we again chose a more specialized translation. That is, instead of translating into the identity over the universe, we rely on the Alloy Analyzer's typechecking information and translate into a more restricted identity relation if possible without changing the semantics. For instance, the Alloy expression **iden** & r, where T is a signature and $r \in T \leftrightarrow T$, can be translated as $\mathrm{id}(T) \cap r$.

However, we do not allow the keywords **univ** or **iden** for specific operators which we cannot translate to an equivalent B expression without introducing the universal type in B. We instead throw an error and do not translate the Alloy model to B. In particular, these operators are equality, inequality and the subset relation **in**. For instance, consider an Alloy model which defines a signature **sig** T { r : **set** T } . We translate the signature as described in Section 4.3, *i.e.*, we introduce a deferred set in B, define the field as a machine constant, and add $E[\![r]\!]i \in E[\![T]\!]i \leftrightarrow E[\![T]\!]i$ to the machine properties. The Alloy model further defines a basic signature R without any field as well as a global fact $T.r = T$, *i.e.*, the signature field r contains all possible elements. When checking the assertion **iden** *in* $(\tilde{}r).r$, the Alloy Analyzer finds a counterexample as the identity relation of the signature R is not part of the dot join's result. If following our restricted translation of the keyword **iden** to B, we would translate the assertion into a more restricted expression $\mathrm{id}(E[\![T]\!]i) \subseteq ((E[\![r]\!]i)^{-1}; E[\![r]\!]i)$ which does not provide a counterexample in B, *i.e.*, the translation would be semantically non-equivalent. An equivalent behavior in B could be achieved by introducing the universal type and using id(`UNIVERSE`) rather than $\mathrm{id}(E[\![T]\!]i)$. Yet, we decided to not support the translation of said expressions containing one of the keywords **univ** and **iden**.

4.5. Connectives and Simple Predicates

Let us first look at how to translate Alloy's logical connectives. This part is very straightforward, as they have matching counterparts in B.

- $M[\![p$ **and** $q]\!]i \;\widehat{=}\; M[\![p]\!]i \wedge M[\![q]\!]i$
- $M[\![p$ **or** $q]\!]i \;\widehat{=}\; M[\![p]\!]i \vee M[\![q]\!]i$

- $M[\![p \textbf{ implies } q]\!]i \;\widehat{=}\; M[\![p]\!]i \Rightarrow M[\![q]\!]i$
- $M[\![p \textbf{ equiv } q]\!]i \;\widehat{=}\; M[\![p]\!]i \Leftrightarrow M[\![q]\!]i$
- $M[\![\textbf{not } p]\!]i \;\widehat{=}\; \neg M[\![p]\!]i$

Similarly, equality and inequality in Alloy and B are identical:

- $M[\![p \texttt{ = } q]\!]i \;\widehat{=}\; E[\![p]\!]i = E[\![q]\!]i$
- $M[\![p \texttt{ != } q]\!]i \;\widehat{=}\; E[\![p]\!]i \neq E[\![q]\!]i$

The following unary expressions can be used to constrain a set's cardinality:

- $M[\![\textbf{no } p]\!]i \;\widehat{=}\; E[\![p]\!]i = \varnothing$
- $M[\![\textbf{one } p]\!]i \;\widehat{=}\; \text{card}(E[\![p]\!]i) = 1$
- $M[\![\textbf{some } p]\!]i \;\widehat{=}\; \text{card}(E[\![p]\!]i) > 0$
- $M[\![\textbf{lone } p]\!]i \;\widehat{=}\; \text{card}(E[\![p]\!]i) \leq 1$

Now, to translate the **in** predicate, it is important to understand that Alloy only operates on set values. This means, that it is translated using the B $\subseteq$ predicate, and not the $\in$ predicate.

- $M[\![p \textbf{ in } q]\!]i \;\widehat{=}\; E[\![p]\!]i \subseteq E[\![q]\!]i$

4.6. Simple Expressions

First, we need to translate simple identifiers not occurring in the environment i, *e.g.*, signature names and fields. In general, identifiers are simply kept as they are. In the presence of modules and namespaces, identifiers have the module names prefixed to avoid ambiguity. This is already handled by the Alloy Analyzer's parser and typechecker we use as a frontend as we will outline in Section 6 where we give technical details on the implementation.

- $E[\![x]\!]i \;\widehat{=}\; x$ for identifiers x not occurring in i

In the following, we present simple translation rules, where one Alloy operator gets translated to a B operator or constant:

- $E[\![\textbf{none}]\!]i \;\widehat{=}\; \varnothing$
- $E[\![p + q]\!]i \;\widehat{=}\; E[\![p]\!]i \cup E[\![q]\!]i$
- $E[\![p \texttt{ \& } q]\!]i \;\widehat{=}\; E[\![p]\!]i \cap E[\![q]\!]i$
- $E[\![p \texttt{ - } q]\!]i \;\widehat{=}\; E[\![p]\!]i \setminus E[\![q]\!]i$
- $E[\![\tilde{}p]\!]i \;\widehat{=}\; (E[\![p]\!]i)^{-1}$
- $E[\![\hat{}p]\!]i \;\widehat{=}\; \text{closure1}(E[\![p]\!]i)$

- $E[\![*p]\!]i \ \widehat{=}\ \text{closure}(E[\![p]\!]i)$
- $E[\![p ++ q]\!]i \ \widehat{=}\ E[\![p]\!]i \mathbin{\lhd\!\!\!+} E[\![q]\!]i$

Note that the Alloy operators ~, ^ and * are only allowed for binary relations. Hence, we can translate them to the B counterparts. However, other Alloy operators also work for relations of higher arity. As such we can encounter not just ordered pairs but tuples, whose translation we discuss in the next subsection.

As discussed in the introductory example in Section 3, 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. In our current translation we use the let expression provided by ProB. Of course, a model can then not be processed by other tools like AtelierB anymore. To do so, ProB provides a pretty-printer which rewrites B abstract syntax trees to native B as, for instance, understood by AtelierB.

4.7. Representing Tuples

In Alloy tuples are flat and Cartesian product is associative. In B Cartesian product is **not** associative and tuples are represented as nested pairs. Hence, in B $(e_1 \mapsto e_2) \mapsto e_3$ is a different value and has a different type than $e_1 \mapsto (e_2 \mapsto e_3)$. Which encoding should we use for an Alloy triple (e_1, e_2, e_3) within a ternary relation r? Both have their advantages and drawbacks, concerning the use of the B operators such as domain, range, relational image or function application. The B language also provides a comma notation for pairs, whose associativity corresponds to the first alternative:

- $(e_1, e_2, e_3) = (e_1 \mapsto e_2) \mapsto e_3$

We have finally chosen to use the first alternative, as it allows us to write set comprehensions of the form $\{x_1, \ldots, x_n \mid P\}$ to generate n-ary relations of the right type.

Another alternative would have been to allow all variations in our translation, keeping track in the type system which associativity has been generated. It is not clear whether this is worthwhile and it definitely makes the translation rules much more complex.

4.8. Cartesian Product

Due to the difference in the treatment of tuples, the Cartesian product in Alloy can also behave differently than in B. When the second argument is a unary relation, we can reuse the B Cartesian product, otherwise we need to compute it using a set comprehension.

For example, the following B Cartesian product between a ternary and a unary relation works correctly:

- $\{(1 \mapsto 2) \mapsto 3\} \times \{4\} = \{((1 \mapsto 2) \mapsto 3) \mapsto 4\}$

However, for a ternary and binary relation it does not work, as the pairs in the result are incorrectly nested:

- $\{(1 \mapsto 2) \mapsto 3\} \times \{(4 \mapsto 5)\} = \{(((1 \mapsto 2) \mapsto 3) \mapsto (4 \mapsto 5))\}$

Using a set comprehension we can compute the correct result:

- $\{t, q1, q2 \mid t \in \{(1 \mapsto 2) \mapsto 3\} \wedge (q1, q2) \in \{(4 \mapsto 5)\}\}$ gives us a correctly encoded tuple: $\{(((1 \mapsto 2) \mapsto 3) \mapsto 4) \mapsto 5\}$

This leads to the two following rules:

- $E[\![p \texttt{ -> } q]\!]i \mathrel{\widehat{=}} E[\![p]\!]i \times E[\![q]\!]i$ if q is a unary relation
- $E[\![p \texttt{ -> } q]\!]i \mathrel{\widehat{=}} \{t, q_1, \ldots, q_n \mid t \in E[\![p]\!]i \wedge (q_1, \ldots, q_n) \in E[\![q]\!]i\}$ if q is an n-ary relation with $n > 1$.

4.9. Domain and Range Restriction

The domain restriction can be translated as follows. For binary relations q we can reuse the corresponding B operator $\lhd$, otherwise we need to compute the result using a set comprehension:

- $E[\![p \texttt{ <: } q]\!]i \mathrel{\widehat{=}} E[\![p]\!]i \lhd E[\![q]\!]i$ if q is a binary relation
- $E[\![p \texttt{ <: } q]\!]i \mathrel{\widehat{=}} \{q_1, \ldots, q_n \mid q_1 \in E[\![p]\!]i \wedge (q_1, \ldots, q_n) \in E[\![q]\!]i\}$ if q is an n-ary relation with $n > 2$.

For range restriction the corresponding B operator $\rhd$ works for all arities, since q must be a unary relation in Alloy.

- $E[\![p \texttt{ :> } q]\!]i \mathrel{\widehat{=}} E[\![p]\!]i \rhd E[\![q]\!]i$

Yet, there are some special cases for domain and range restriction with **iden** or **univ**. We thus encode the restriction of the identity relation to the domain p in B as the binary relation that relates every element of p to itself, while a domain restriction on **univ** returns the domain itself. For the identity relation, the assertion **all** p : **univ** | p <: **iden** = **iden** :> p holds (analogous for **univ**). This shows that both expressions are equivalent, which we represent using the symbol $\equiv$, resulting in the following translation to B:

- $E[\![p <: \textbf{iden}]\!]i \equiv E[\![\textbf{iden} :> p]\!]i \mathrel{\widehat{=}} \lambda x.(x \in E[\![p]\!]i \mid x)$
- $E[\![p <: \textbf{univ}]\!]i \equiv E[\![\textbf{univ} :> p]\!]i \mathrel{\widehat{=}} E[\![p]\!]i$

4.10. Join

The dot join $p.q$, one of the most important operators in Alloy, is also the most difficult to translate. We have quite a lot of special cases below, trying to use existing B operators if possible. First are three special cases, where one of the arguments is the Alloy universe. These operations correspond to computing the domain and range of the Alloy relations:

- $E[\![p.\textbf{univ}]\!]i \;\widehat{=}\; \mathrm{dom}(E[\![p]\!]i)$ where p is an n-ary relation, $n \geq 2$
- $E[\![\textbf{univ}.q]\!]i \;\widehat{=}\; \mathrm{ran}(E[\![q]\!]i)$ if q is a binary relation
- $E[\![\textbf{univ}.q]\!]i \;\widehat{=}\; \{q_2, \ldots, q_k \mid \exists j.((j, q_2, \ldots, q_k) \in E[\![q]\!]i)\}$ if q is an n-ary relation, $n > 2$

Another typical pattern in Alloy is to use the join operator for relational image or function application. The next three translation rules capture these patterns:

- $E[\![p.q]\!]i \;\widehat{=}\; \{E[\![q]\!]i\ (pv)\}$ if $E[\![p]\!]i = \{pv\}$, p is a unary relation, q is an n-ary relation with n ≥ 2, and q is known to be a total function in pv.
- $E[\![p.q]\!]i \;\widehat{=}\; E[\![q]\!]i\ [\ E[\![p]\!]i\ ]$ if p is a unary relation, q is a binary relation
- $E[\![p.q]\!]i \;\widehat{=}\; (E[\![p]\!]i)^{-1}[E[\![q]\!]i]$ if q is a unary relation, p is a binary relation

The translation rules above are optional, since the next three translation rules of the join operator also cover the same cases, but lead to a less idiomatic B translation. The first one uses B's relational composition operator:

- $E[\![p.q]\!]i \;\widehat{=}\; (E[\![p]\!]i\ ;\ E[\![q]\!]i)$ if both p and q are binary relations
- $E[\![p.q]\!]i \;\widehat{=}\; \{q_2, \ldots, q_k \mid \exists j.(j \in E[\![p]\!]i \wedge (j, q_2, \ldots, q_k) \in E[\![q]\!]i\}$ if p is unary relation
- $E[\![p.q]\!]i \;\widehat{=}\; \{t, q_2, \ldots, q_k \mid \exists j.((t, j) \in E[\![p]\!]i \wedge (j, q_2, \ldots, q_k) \in E[\![q]\!]i\}$ in all other cases (*i.e.*, p is an n-ary relation with $n > 1$ and q is a k-ary relation with $k > 1$).

One can observe that the join operator of Alloy is very elegant and flexible; together with flat tuples, it provides an expressive construct. One could think about extending the B relational composition operator to work more flexibly (*i.e.*, also with sets and n-ary relations). This would also make the translation to B easier.

Alloy further provides the box join operator $y[x]$ which is just syntactic sugar for the dot join operation $x.y$ though.

4.11. Quantifications, Set Comprehensions and Identifiers

Quantifications in Alloy can introduce a finite set of identifiers using the **:** operator for unary sets S:

- $D[\![x : \textbf{one } S]\!]i \;\widehat{=}\; \{x\} \subseteq E[\![S]\!]i$
- $D[\![x : \textbf{set } S]\!]i \;\widehat{=}\; x \subseteq E[\![S]\!]i$
- $D[\![x : \textbf{some } S]\!]i \;\widehat{=}\; x \subseteq E[\![S]\!]i \wedge x \neq \varnothing$
- $D[\![x : \textbf{lone } S]\!]i \;\widehat{=}\; x \subseteq E[\![S]\!]i \wedge \mathrm{card}(x) \leq 1$
- $D[\![x : S]\!]i \equiv D[\![x : \textbf{one } S]\!]i$ if the arity of S is 1
- $D[\![x : S]\!]i \equiv M[\![x \textbf{ in } S]\!]i$ if the arity of S is greater than 1
- $D[\![x_1, \ldots, x_k : S]\!]i \equiv D[\![x_1 : S]\!]i \wedge \ldots \wedge D[\![x_k : S]\!]i$ for $k > 1$
- $D[\![\textbf{disj } x_1, \ldots, x_k : S]\!]i \equiv D[\![x_1 : S]\!]i \wedge \ldots \wedge D[\![x_k : S]\!]i \wedge$ $\mathrm{card}(\{x_1, \ldots, x_k\}) = k$ for $k > 1$

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. However, we have to ensure that unique names are used for the translation of local identifiers to avoid clashes between identifier names for nested scopes.

We apply a separate function to compute updates to the environment by identifier declaration, which is defined as follows:

- $I[\![x : \textbf{one } S]\!]i \;\widehat{=}\; i \cup \{x\}$
- $I[\![x : m\ S]\!]i \;\widehat{=}\; i - \{x\}$ for $m \neq \textbf{one}$.
- $I[\![x : S]\!]i \;\widehat{=}\; i \cup \{x\}$ if arity of S is 1, $i - \{x\}$ otherwise

Given an environment i, we translate identifiers to B either as a singleton set or as a raw identifier:

- $E[\![x]\!]i \;\widehat{=}\; \{x\}$ if $x \in i$ (*i.e.*, the environment i states that x is a singleton set identifier (*e.g.*, quantified variables with multiplicity one)
- $E[\![x]\!]i \;\widehat{=}\; x$ for $x \notin i$ (*i.e.*, x is a signature name or field)

In the following, x are the left-hand side arguments of the declaration *Decl* and $i' = I[\![Decl]\!]i$ holds:

- $M[\![\textbf{some } Decl \mid P]\!]i \;\widehat{=}\; \exists x.(D[\![Decl]\!]i \wedge M[\![P]\!]i')$
- $M[\![\textbf{all } Decl \mid P]\!]i \;\widehat{=}\; \forall x.(D[\![Decl]\!]i \Rightarrow M[\![P]\!]i')$
- $M[\![\textbf{no } Decl \mid P]\!]i \;\widehat{=}\; \neg\exists x.(D[\![Decl]\!]i \wedge M[\![P]\!]i')$
- $M[\![\textbf{one } Decl \mid P]\!]i \;\widehat{=}\; \mathrm{card}(\{x \mid D[\![Decl]\!]i \wedge M[\![P]\!]i'\}) = 1$
- $M[\![\textbf{lone } Decl \mid P]\!]i \;\widehat{=}\; \mathrm{card}(\{x \mid D[\![Decl]\!]i \wedge M[\![P]\!]i'\}) \leq 1$

4.12. Conditionals

Alloy provides a conditional statement, which can either be treated as a predicate or as an expression. We therefore provide the following two translation rules, one of which uses the if-then-else expression of ProB:

- $M[\![p \Rightarrow q \text{ ELSE } r]\!]i \;\widehat{=}\; (M[\![p]\!]i \Rightarrow M[\![q]\!]i) \wedge (\neg M[\![p]\!]i \Rightarrow M[\![r]\!]i)$
- $E[\![p \Rightarrow q \text{ ELSE } r]\!]i \;\widehat{=}\; \text{IF } M[\![p]\!]i \text{ THEN } E[\![q]\!]i \text{ ELSE } E[\![r]\!]i \text{ END}$

Again, ProB is able to rewrite if-then-else expressions to be compatible to native B as, *e.g.*, understood by AtelierB.

4.13. Fact, Function & Predicate Declaration

Alloy's **fact** declaration has an optional name and contains any number of predicates, which pose additional constraints to be added to the model. We translate the expressions as described above. The results are conjoined and added to the `PROPERTIES` section of the B machine.

Alloy allows declaring 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.[2]

Functions will be listed in the `DEFINITIONS` section of the B machine if the machine contains at least one invocation. Each function is translated into a single B definition with matching parameters, consisting of a set comprehension wrapping the actual expression in the body to account for the expected return type. For instance, the function declaration **fun** $f\ [s : S] : S\ \{\ body\ \}$ is translated into the B definition $f(s) == \{x \mid D[\![s : S]\!]i \wedge x \in E[\![body]\!]i\}$, where $i' = I[\![s : S]\!]i$ as in Section 4.11. 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 resulting in a B predicate. For instance, the predicate declaration **pred** $p\ [s : S]\ \{\ body\ \}$ is translated into the B definition $p(s) == D[\![s : S]\!]i \wedge M[\![body]\!]i'$.

Again, we have to ensure that unique names are used for the translation of local identifiers in order to avoid clashes between identifier names in B for nested scopes.

Note that ProB inlines definitions to the positions where they are used, which basically is a text replacement. When loading a B machine in ProB, the machine's `DEFINITIONS` section is thus not present anymore.

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

4.14. 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. For instance, **assert** a { } is a named but empty assertion that is always true.

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 by searching for a counterexample.

We introduce an `operation` to the B machine for each **run** 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. In case of a **check** command, we proceed analogously but negate the command's instructions within the precondition in order to search for a counterexample. That is, the operation is enabled if a counterexample exists.

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, translates to an operation $\text{run} \;=\; \text{PRE card}(E[\![S]\!]i) \leq 3 \land M[\![p]\!]i \text{ THEN skip END}$ in B. In case of a **check** command, the only difference in the translation is to use $\neg M[\![p]\!]i$.

The Alloy keywords **Int** and **seq** can be used to specify the bitwidth used to represent integers and the maximum allowed length of sequences.

To execute an Alloy command with ProB one can, *e.g.*, use constraint-based checking as will be explained in Section 6. In short, constraint-based checking searches for a variable state that satisfies the precondition of an operation considering the machine's properties.

Given that the proof assistants for classical B do not require scopes we also support translating commands without scopes, which can be set via a user preference in ProB. While the resulting B machine does not mimic the Alloy model's behavior exactly, it allows for a more general proof, following the approach we will show in Section 8.3.

4.15. Multiplicity Annotations

Alloy supports the multiplicity annotations **some**, **one**, **lone** and **set**. If no multiplicity is given, the default multiplicity **set** is used.

When translating multiplicity annotations, the semantics are no longer denotational. The predicate $M[\![x \textit{ in } A \texttt{ -> } \textbf{one } B]\!]i$ cannot be encoded as $E[\![x]\!]i \subseteq E[\![A \texttt{ -> } \textbf{one } B]\!]i$, because the property of being a total function is not a closed subset. Hence, we translate the multiplicity annotations as follows:

- $M\llbracket x \textit{ in } A \texttt{ -> } B\rrbracket i \mathrel{\widehat{=}} E\llbracket x\rrbracket i \in E\llbracket A\rrbracket i \leftrightarrow E\llbracket B\rrbracket i$
- $M\llbracket x \textit{ in } A \texttt{ -> } \mathbf{some}\ B\rrbracket i \mathrel{\widehat{=}} E\llbracket x\rrbracket i \in E\llbracket A\rrbracket i \leftrightarrow E\llbracket B\rrbracket i \wedge \mathrm{dom}(E\llbracket x\rrbracket i) = E\llbracket A\rrbracket i$
- $M\llbracket x \textit{ in } A \texttt{ -> } \mathbf{one}\ B\rrbracket i \mathrel{\widehat{=}} E\llbracket x\rrbracket i \in E\llbracket A\rrbracket i \rightarrow E\llbracket B\rrbracket i$
- $M\llbracket x \textit{ in } A \texttt{ -> } \mathbf{lone}\ B\rrbracket i \mathrel{\widehat{=}} E\llbracket x\rrbracket i \in E\llbracket A\rrbracket i ⇸ E\llbracket B\rrbracket i$
- $M\llbracket x \textit{ in } A\ \mathbf{some} \texttt{ -> } B\rrbracket i \mathrel{\widehat{=}} E\llbracket x\rrbracket i \in E\llbracket A\rrbracket i \leftrightarrow E\llbracket B\rrbracket i \wedge \mathrm{ran}(E\llbracket x\rrbracket i) = E\llbracket B\rrbracket i$
- $M\llbracket x \textit{ in } \mathbf{some}\ A \texttt{ -> } \mathbf{some}\ B\rrbracket i \mathrel{\widehat{=}} E\llbracket x\rrbracket i \in E\llbracket A\rrbracket i \leftrightarrow E\llbracket B\rrbracket i \wedge \mathrm{dom}(E\llbracket x\rrbracket i) = E\llbracket A\rrbracket i \wedge \mathrm{ran}(E\llbracket x\rrbracket i) = E\llbracket B\rrbracket i$
- $M\llbracket x \textit{ in } \mathbf{some}\ A \texttt{ -> } \mathbf{lone}\ B\rrbracket i \mathrel{\widehat{=}} E\llbracket x\rrbracket i \in E\llbracket A\rrbracket i ⤀ E\llbracket B\rrbracket i$
- $M\llbracket x \textit{ in } \mathbf{some}\ A \texttt{ -> } \mathbf{one}\ B\rrbracket i \mathrel{\widehat{=}} E\llbracket x\rrbracket i \in E\llbracket A\rrbracket i \twoheadrightarrow E\llbracket B\rrbracket i$
- $M\llbracket x \textit{ in } \mathbf{lone}\ A \texttt{ -> } B\rrbracket i \mathrel{\widehat{=}} (E\llbracket x\rrbracket i)^{-1} \in E\llbracket B\rrbracket i ⇸ E\llbracket A\rrbracket i$
- $M\llbracket x \textit{ in } \mathbf{lone}\ A \texttt{ -> } \mathbf{some}\ B\rrbracket i \mathrel{\widehat{=}} (E\llbracket x\rrbracket i)^{-1} \in E\llbracket B\rrbracket i ⤀ E\llbracket A\rrbracket i$
- $M\llbracket x \textit{ in } \mathbf{lone}\ A \texttt{ -> } \mathbf{lone}\ B\rrbracket i \mathrel{\widehat{=}} E\llbracket x\rrbracket i \in E\llbracket A\rrbracket i ⤔ E\llbracket B\rrbracket i$
- $M\llbracket x \textit{ in } \mathbf{lone}\ A \texttt{ -> } \mathbf{one}\ B\rrbracket i \mathrel{\widehat{=}} E\llbracket x\rrbracket i \in E\llbracket A\rrbracket i \rightarrowtail E\llbracket B\rrbracket i$
- $M\llbracket x \textit{ in } \mathbf{one}\ A \texttt{ -> } B\rrbracket i \mathrel{\widehat{=}} (E\llbracket x\rrbracket i)^{-1} \in E\llbracket B\rrbracket i \rightarrow E\llbracket A\rrbracket i$
- $M\llbracket x \textit{ in } \mathbf{one}\ A \texttt{ -> } \mathbf{some}\ B\rrbracket i \mathrel{\widehat{=}} (E\llbracket x\rrbracket i)^{-1} \in E\llbracket B\rrbracket i \twoheadrightarrow E\llbracket A\rrbracket i$
- $M\llbracket x \textit{ in } \mathbf{one}\ A \texttt{ -> } \mathbf{lone}\ B\rrbracket i \mathrel{\widehat{=}} E\llbracket x\rrbracket i \in E\llbracket A\rrbracket i ⤗ E\llbracket B\rrbracket i$
- $M\llbracket x \textit{ in } \mathbf{one}\ A \texttt{ -> } \mathbf{one}\ B\rrbracket i \mathrel{\widehat{=}} E\llbracket x\rrbracket i \in E\llbracket A\rrbracket i ⤖ E\llbracket B\rrbracket i$

Note that B does not provide operators for each multiplicity annotation that are equivalent to Alloy's definition, *e.g.*, there is no operator to directly define a total relation in B. We thus translate the corresponding multiplicity annotations to B as relations and add additional constraints.

While ProB supports the translated predicates as typing predicates, AtelierB does not support directly typing the inverse as done, for instance, in $M\llbracket x \textit{ in } \mathbf{lone}\ A \texttt{ -> } B\rrbracket i \mathrel{\widehat{=}} (E\llbracket x\rrbracket i)^{-1} \in E\llbracket B\rrbracket i ⇸ E\llbracket A\rrbracket i$. Instead, one has to type the relation itself and add any restrictions on the inverse as additional constraints, *e.g.*, $M\llbracket x \textit{ in } \mathbf{lone}\ A \texttt{ -> } B\rrbracket i \mathrel{\widehat{=}} E\llbracket x\rrbracket i \in E\llbracket A\rrbracket i \leftrightarrow E\llbracket B\rrbracket i \wedge (E\llbracket x\rrbracket i)^{-1} \in E\llbracket B\rrbracket i ⇸ E\llbracket A\rrbracket i$.

4.16. Post-Processing Optimization Rules

As our translation has to be generalized and applicable to all possible Alloy constructs, some translations might not be ideal for the ProB constraint solver, especially when using singleton sets. For instance, our translation might define an identifier to be an element of a specific set like $x \in S$. Yet, if this set is a singleton set $S = \{y\}$, the membership relation can be replaced by a simple equality $x = y$.

In order to improve performance, ProB provides many rules to improve the representation of abstract syntax trees prior to solving constraints, which is referred to as an abstract syntax tree cleanup. In the following, we present the additional rewriting rules that arose during the implementation of the translation from Alloy to B mostly caused by the use of singleton sets:

- $\{x\} = \{y\} \rightsquigarrow x = y$
- $\{x\} \neq \{y\} \rightsquigarrow x \neq y$
- $x \in \{y\} \rightsquigarrow x = y$
- $x \notin \{y\} \rightsquigarrow x \neq y$
- $\{x\} \subseteq \{y\} \rightsquigarrow x = y$
- $\{x\} \cap \{y\} = \varnothing \rightsquigarrow x \neq y$
- $\{x\} * \{y\} \rightsquigarrow \{x \mapsto y\}$

5. Translation of Alloy Extensions

Alloy provides several language extensions, for instance, supporting integers or additional constraints for certain types. As the modules are specified in Alloy, we could directly translate them as well. Yet, we aim to provide an idiomatic and more efficient translation to B for each module. The translation of boolean operations such as `Nand` is trivial and implemented using B's logical operators. In the following, we present the translation of the extensions we currently support besides boolean operations.

5.1. Integers and Natural Numbers

An integer expression in Alloy is a set, just like any other expression. In order to deal with operators that expects a scalar value, Alloy first evaluates the sum of the elements of a set of integers before applying an operator. For instance, given `s = {x : Int | x=1 or x=2}`, the first operand in the alloy expression `minus[s,4]` evaluates to 3, so the result is -1. Similarly, the predicate `3 > s` returns false. Empty sets are evaluated to 0.

Since Alloy encodes constraints to SAT, each Alloy command defines a bitwidth n used to store integers, *i.e.*, the range $-2^{n-1}..2^{n-1}-1$ with one bit being used to represent the sign. Consequently, integer overflows might occur and the Alloy Analyzer may return a model which is invalid outside the given scope. For example, a model might satisfy `plus[5,3] = -8` with a bitwidth of 4. It is also possible to divide by zero. An option of the Alloy analyzer can be set to exclude models that entail an overflow or division by 0. However, this slows down the analysis process. Thus, when a modeler presumes that a model does not overflow, this option is usually set to off for efficiency reasons, but there is the risk that an overflow goes undetected. Since B has native support for full integers, overflows do not occur.

Although well-defined, the semantics of integers in Alloy is somewhat unnatural. For instance, accepting non-singleton sets as arguments of integer operations is error prone and might not always be desired. As overflows are a stumbling block in the use of integers in Alloy, we do not want to replicate this behavior by introducing a bitwidth in B. However, we provide a ProB preference to translate into bounded integers without overflows using ProB's settings for `MININT` and `MAXINT`. Further, we do not want to allow dividing by zero but throw a well-definedness error instead.

For the translation of integer operations, we use an additional semantic function $E_{int}[\![.]\!]i$ that transforms a set of integers into a scalar expression. It uses the Σ operator of B which returns the sum of the elements of a set of integers or 0 if the set is empty. For instance, $\Sigma(z).(z \in 1..4 \mid z)$ returns 10.

As mentioned above, we believe that accepting non-singleton sets as arguments of integer operations is error prone. We thus decided to provide a ProB preference which enables a strict translation of integers, *i.e.*, only accept singleton sets where integers are expected. This preference is set by default. To do so, we use the definite description operator, noted MU, which is defined as follows:

$$MU(x) = (\{TRUE\} \times x)(TRUE)$$

For instance, $MU(\{1\})$ returns 1. Since the operator uses B's function application, the MU operator has the well-definedness condition that its argument x is a singleton set. Otherwise, the function application would be undefined for empty sets or ambiguous for non-singleton sets. MU does not exist in the B notation, but is supported by ProB.

Let Nr be an integer constant. We have the following four rules for the semantic function $E_{int}[\![.]\!]i$:

- $E_{int}[\![Nr]\!]i \ \widehat{=}\ E[\![Nr]\!]i$ for integer constants Nr
- $E_{int}[\![p]\!]i \ \widehat{=}\ Nr$ if $E[\![p]\!]i = \{Nr\}$
- $E_{int}[\![p]\!]i \ \widehat{=}\ MU(E[\![p]\!]i)$ if ProB preference for a strict integer translation is set
- $E_{int}[\![p]\!]i \ \widehat{=}\ \Sigma(z).(z \in E[\![p]\!]i \mid z)$ otherwise

Note that the first two rules of $E_{int}[\![.]\!]i$ are in principle redundant, they "only" improve the performance of the translation (avoiding unnecessary Σ or MU constructs).

Using this definition the integer operations are translated as follows:

- $E[\![Nr]\!]i = \{Nr\}$ for integer constants Nr
- $E[\![\#p]\!]i \ \widehat{=}\ \{\text{card}(E[\![p]\!]i)\}$
- $E[\![min[p]]\!]i \ \widehat{=}\ \{\text{min}(E[\![p]\!]i)\}$
- $E[\![max[p]]\!]i \ \widehat{=}\ \{\text{max}(E[\![p]\!]i)\}$

- $E[\![plus[p,q]]\!]i \ \widehat{=}\ \{E_{int}[\![p]\!]i + E_{int}[\![q]\!]i\}$
- $E[\![mul[p,q]]\!]i \ \widehat{=}\ \{E_{int}[\![p]\!]i * E_{int}[\![q]\!]i\}$
- $E[\![minus[p,q]]\!]i \ \widehat{=}\ \{E_{int}[\![p]\!]i - E_{int}[\![q]\!]i\}$
- $E[\![div[p,q]]\!]i \ \widehat{=}\ \{E_{int}[\![p]\!]i / E_{int}[\![q]\!]i\}$
- $E[\![rem[p,q]]\!]i \ \widehat{=}\ \{E_{int}[\![p]\!]i \ mod\ E_{int}[\![q]\!]i\}$
  (currently only works for positive numbers)
- $E[\![negate[p]]\!]i \ \widehat{=}\ \{-E_{int}[\![p]\!]i\}$
- $M[\![eq[p,q]]\!]i \ \widehat{=}\ E_{int}[\![p]\!]i = E_{int}[\![q]\!]i$
- $M[\![gt[p,q]]\!]i \ \widehat{=}\ E_{int}[\![p]\!]i > E_{int}[\![q]\!]i$
- $M[\![lt[p,q]]\!]i \ \widehat{=}\ E_{int}[\![p]\!]i < E_{int}[\![q]\!]i$
- $M[\![gte[p,q]]\!]i \ \widehat{=}\ E_{int}[\![p]\!]i \geq E_{int}[\![q]\!]i$
- $M[\![lte[p,q]]\!]i \ \widehat{=}\ E_{int}[\![p]\!]i \leq E_{int}[\![q]\!]i$
- $M[\![zero[p]]\!]i \ \widehat{=}\ E_{int}[\![p]\!]i = 0$
- $M[\![pos[p]]\!]i \ \widehat{=}\ E_{int}[\![p]\!]i > 0$
- $M[\![neg[p]]\!]i \ \widehat{=}\ E_{int}[\![p]\!]i < 0$
- $M[\![nonpos[p]]\!]i \ \widehat{=}\ E_{int}[\![p]\!]i \leq 0$
- $M[\![nonneg[p]]\!]i \ \widehat{=}\ E_{int}[\![p]\!]i \geq 0$
- $E[\![signum[p]]\!]i \ \widehat{=}$ IF $E_{int}[\![p]\!]i < 0$ THEN -1
  ELSE IF $E_{int}[\![p]\!]i > 0$ THEN 1 ELSE 0 END END
- $E[\![next]\!]i \ \widehat{=}$ succ
- $E[\![prev]\!]i \ \widehat{=}$ pred
- $E[\![next[p]]\!]i \ \widehat{=}\ E[\![p.next]\!]i$
- $E[\![prev[p]]\!]i \ \widehat{=}\ E[\![p.prev]\!]i$
- $E[\![nexts[p]]\!]i \ \widehat{=}\ \{x \mid x \in \mathbb{Z} \wedge x > E_{int}[\![p]\!]i\}$
- $E[\![prevs[p]]\!]i \ \widehat{=}\ \{x \mid x \in \mathbb{Z} \wedge x < E_{int}[\![p]\!]i\}$
- $E[\![larger[p,q]]\!]i \ \widehat{=}\ \{\max(\{E_{int}[\![p]\!]i, E_{int}[\![q]\!]i\})\}$
- $E[\![smaller[p,q]]\!]i \ \widehat{=}\ \{\min(\{E_{int}[\![p]\!]i, E_{int}[\![q]\!]i\})\}$
- $E[\![min]\!]i \ \widehat{=}$ `MININT`
- $E[\![max]\!]i \ \widehat{=}$ `MAXINT`

Note that `MININT` and `MAXINT` are user preferences of ProB. Further, we do not consider Alloy's bitwidth for the translation of `nexts` and `prevs` but return unbounded sets of integers.

The definitions of division and modulo differ slightly between Alloy and B. B uses a floored division [**?**]. More precisely, the definition of division in B [1] is $n/m = \min(\{x|x \in \mathbb{Z} \land n < m * \mathrm{succ}(x)\})$. Furthermore, in B, $x \mod y$ is only defined if x is non-negative and y is positive.

In contrast, the definition used by Alloy permits both cases. Alloy's division rounds towards zero in general, but permits a number of special cases. According to comments in the Alloy utility module **util/integer**, there are three exceptions to the "round to zero" definition of a/b. First, if $a = 0$, the division returns zero. Second, if $a < 0 \land b = 0$, the division returns 1; if $a > 0 \land b = 0$ it returns -1. Last, if a is the smallest negative integer and $b = -1$, the division returns a. The different definitions of division and modulo [**?**] can easily be expressed in B by rewriting them to B's floored division [**?**]. However, as mentioned above, we do not want to completely reproduce the behavior of Alloy regarding integers.

The translation of operations on natural numbers as defined in the Alloy utility module **util/natural**[3] is analogous to integers but considering the condition to be a positive integer. ProB does not implement any special operations for natural numbers.

5.2. Relations

Alloy provides a module for common operations and constraints on binary relations. We translate the relational operations as follows where r is a binary relation with domain d and codomain c:

- $E[\![dom[r]]\!]i \ \widehat{=}\ \mathrm{dom}(E[\![r]\!]i)$
- $E[\![ran[r]]\!]i \ \widehat{=}\ \mathrm{ran}(E[\![r]\!]i)$
- $M[\![total[r,d]]\!]i \ \widehat{=}\ \forall x.(x \in E[\![d]\!]i \Rightarrow E[\![r]\!]i[\{x\}] \neq \varnothing)$
- $M[\![functional[r,d]]\!]i \ \widehat{=}\ \forall x.(x \in E[\![d]\!]i \Rightarrow \mathrm{card}(E[\![r]\!]i[\{x\}]) \leq 1)$
- $M[\![function[r,d]]\!]i \ \widehat{=}\ \forall x.(x \in E[\![d]\!]i \Rightarrow \mathrm{card}(E[\![r]\!]i[\{x\}]) = 1)$
- $M[\![injective[r,c]]\!]i \ \widehat{=}\ \forall y.(y \in E[\![c]\!]i \Rightarrow \mathrm{card}((E[\![r]\!]i)^{-1}[\{y\}]) \leq 1)$
- $M[\![surjective[r,c]]\!]i \ \widehat{=}\ \forall y.(y \in E[\![c]\!]i \Rightarrow (E[\![r]\!]i)^{-1}[\{y\}] \neq \varnothing)$
- $M[\![bijective[r,c]]\!]i \ \widehat{=}\ \forall y.(y \in E[\![c]\!]i \Rightarrow \mathrm{card}((E[\![r]\!]i)^{-1}[\{y\}]) = 1)$
- $M[\![bijection[r,d,c]]\!]i \ \widehat{=}\ M[\![function[r,d]]\!]i \land M[\![bijective[r,c]]\!]i$
- $M[\![reflexive[r,s]]\!]i \ \widehat{=}\ \mathrm{id}(s) \subseteq E[\![r]\!]i$

[3]See `http://alloytools.org/quickguide/util.html`

- $M[\![\mathit{irreflexive}[r]]\!]i \ \widehat{=}\ \mathrm{id}(\mathrm{dom}(E[\![r]\!]i)) \cap E[\![r]\!]i = \varnothing$
- $M[\![\mathit{symmetric}[r]]\!]i \ \widehat{=}\ (E[\![r]\!]i)^{-1} \subseteq E[\![r]\!]i$
- $M[\![\mathit{antisymmetric}[r]]\!]i \ \widehat{=}\ ((E[\![r]\!]i)^{-1} \cap E[\![r]\!]i) \subseteq \mathrm{id}(\mathrm{dom}(E[\![r]\!]i))$
- $M[\![\mathit{transitive}[r]]\!]i \ \widehat{=}\ (E[\![r]\!]i \ ; \ E[\![r]\!]i) \subseteq E[\![r]\!]i$
- $M[\![\mathit{acyclic}[r,s]]\!]i \ \widehat{=}\ \forall x.(x \in E[\![s]\!]i \Rightarrow x \mapsto x \notin \mathrm{closure1}(E[\![r]\!]i))$
- $M[\![\mathit{complete}[r,s]]\!]i \ \widehat{=}\ \forall (x,y).(x \in E[\![s]\!]i \wedge y \in E[\![s]\!]i \wedge$
  $x \neq y \Rightarrow x \mapsto y \in (E[\![r]\!]i \cup (E[\![r]\!]i)^{-1}))$
- $M[\![\mathit{preorder}[r,s]]\!]i \ \widehat{=}\ M[\![\mathit{reflexive}[r,s]]\!]i \wedge M[\![\mathit{transitive}[r]]\!]i$
- $M[\![\mathit{equivalence}[r,s]]\!]i \ \widehat{=}\ M[\![\mathit{preorder}[r,s]]\!]i \wedge M[\![\mathit{symmetric}[r]]\!]i$
- $M[\![\mathit{partialOrder}[r,s]]\!]i \ \widehat{=}\ M[\![\mathit{preorder}[r,s]]\!]i \wedge M[\![\mathit{antisymmetric}[r]]\!]i$
- $M[\![\mathit{totalOrder}[r,s]]\!]i \ \widehat{=}\ M[\![\mathit{partialOrder}[r,s]]\!]i \wedge M[\![\mathit{complete}[r,s]]\!]i$

5.3. 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. Yet, Alloy allows declaring a total order on signature elements by defining a signature to be ordered, and offers several operations for element access on ordered signatures. For instance, for an ordered Signature S_o, $S_o/\mathit{nexts}(s)$ returns the set of all successors of $s \in S_o$.

Initially, we translated ordered signatures to B sequences. Sequences are ordered sets of pairs whose domains are finite and coherent sets $1..n$, where $n \in \mathbb{N}$ 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 4.3. An ordered signature S_o can then be represented by a sequence of type S_o, *i.e.*, a set of pairs of integer and S_o. B directly offers most of the operations on ordered signatures while others 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 switched to 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 $s..(s+k-1)$, $s \in \mathbb{N}$, in B. The offset s is used to take into account that ordered signatures can interact, *e.g.*, when computing the union. We thus 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 as well 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.

We expect the input values of operations on orderings to be singleton integer sets or empty sets. When using integer intervals, the operations *first* and *last* can be translated using `min` and `max` wrapped in a singleton set. For an ordered signature S_o, we define $S_o/next$ and $S_o/prev$ using the successor and predecessor relations of B. The operations $S_o/nexts[e]$ and $S_o/prevs[e]$ are translated using set comprehensions.

We noticed that the relational operations on orderings like $S_o/lt[e1, e2]$ always return true if the left-hand side is an empty set. If the left-hand side is non-empty and the right-hand side is an empty set on the other hand, the relational operators always return false. In the remaining cases, the relational operators behave as expected from real integers.

In particular, we translate the operations on an ordered signature S_o in Alloy as follows:

- $E[\![S_o]\!]i \ \widehat{=}\ m..n$, where m is the lower and n the upper bound of the corresponding integer domain
- $E[\![S_o/\mathit{first}]\!]i \ \widehat{=}\ \text{IF } E[\![S_o]\!]i \neq \varnothing\ THEN\ \{\min(E[\![S_o]\!]i)\} \text{ ELSE } \varnothing \text{ END}$
- $E[\![S_o/\mathit{last}]\!]i \ \widehat{=}\ IF\ E[\![S_o]\!]i \neq \varnothing \text{ THEN } \{\max(E[\![S_o]\!]i)\} \text{ ELSE } \varnothing \text{ END}$
- $E[\![S_o/\mathit{min}[es]]\!]i \ \widehat{=}\ \text{IF } E[\![S_o]\!]i \neq \varnothing \text{ THEN } \{\min(E[\![es]\!]i)\} \text{ ELSE } \varnothing \text{ END}$
- $E[\![S_o/\mathit{max}[es]]\!]i \ \widehat{=}\ \text{IF } E[\![S_o]\!]i \neq \varnothing \text{ THEN } \{\max(E[\![es]\!]i)\} \text{ ELSE } \varnothing \text{ END}$
- $E[\![S_o/\mathit{next}[e]]\!]i \ \widehat{=}\ \text{IF } \text{succ}[E[\![e]\!]i] \subseteq E[\![S_o]\!]i \text{ THEN } \text{succ}[E[\![e]\!]i]$ $\text{ELSE } \varnothing \text{ END}$
- $E[\![S_o/\mathit{nexts}[e]]\!]i \ \widehat{=}\ \{x \mid x \in E[\![S_o]\!]i \wedge \min(\{x\} \cup E[\![e]\!]i) \neq x\}$
- $E[\![S_o/\mathit{prev}[e]]\!]i \ \widehat{=}\ \text{IF } \text{pred}[E[\![e]\!]i] \subseteq E[\![S_o]\!]i \text{ THEN } \text{pred}[E[\![e]\!]i]$ $\text{ELSE } \varnothing \text{ END}$
- $E[\![S_o/\mathit{prevs}[e]]\!]i \ \widehat{=}\ \{x \mid x \in E[\![S_o]\!]i \wedge \max(\{x\} \cup E[\![e]\!]i) \neq x\}$
- $E[\![S_o/\mathit{larger}[e1, e2]]\!]i \ \widehat{=}\ \text{IF } E[\![e1]\!]i \cup E[\![e2]\!]i \neq \varnothing \text{ THEN}$ $\{\max(E[\![e1]\!]i \cup E[\![e2]\!]i)\} \text{ ELSE } \varnothing \text{ END}$
- $E[\![S_o/\mathit{smaller}[e1, e2]]\!]i \ \widehat{=}\ \text{IF } E[\![e1]\!]i \cup E[\![e2]\!]i \neq \varnothing \text{ THEN}$ $\{\min(E[\![e1]\!]i \cup E[\![e2]\!]i)\} \text{ ELSE } \varnothing \text{ END}$
- $M[\![S_o/\mathit{lt}[e1, e2]]\!]i \ \widehat{=}\ (E[\![e1]\!]i = \varnothing) \vee (E[\![e1]\!]i \neq \varnothing \wedge E[\![e2]\!]i \neq \varnothing \wedge$ $E[\![e1]\!]i \neq E[\![e2]\!]i \wedge \{\min(E[\![e1]\!]i \cup E[\![e2]\!]i)\} = E[\![e1]\!]i))$
- $M[\![S_o/\mathit{lte}[e1, e2]]\!]i \ \widehat{=}\ (E[\![e1]\!]i = \varnothing) \vee (E[\![e1]\!]i \neq \varnothing \wedge E[\![e2]\!]i \neq \varnothing \wedge$ $\{\min(E[\![e1]\!]i \cup E[\![e2]\!]i)\} = E[\![e1]\!]i))$
- $M[\![S_o/\mathit{gt}[e1, e2]]\!]i \ \widehat{=}\ (E[\![e1]\!]i = \varnothing) \vee (E[\![e1]\!]i \neq \varnothing \wedge E[\![e2]\!]i \neq \varnothing \wedge$ $E[\![e1]\!]i \neq E[\![e2]\!]i \wedge \{\max(E[\![e1]\!]i \cup E[\![e2]\!]i)\} = E[\![e1]\!]i))$
- $M[\![S_o/\mathit{gte}[e1, e2]]\!]i \ \widehat{=}\ (E[\![e1]\!]i = \varnothing) \vee (E[\![e1]\!]i \neq \varnothing \wedge E[\![e2]\!]i \neq \varnothing \wedge$ $\{\max(E[\![e1]\!]i \cup E[\![e2]\!]i)\} = E[\![e1]\!]i))$

5.4. Enumerations

In Alloy, an enumeration can be used to define a number of distinct singleton signatures with a common ordered base signature. Enumerations are syntactical sugar, not providing new functionalities but enabling a less verbose specification. For instance, consider the following enumeration S:

```
enum S { S1,S2 }
```

The same behavior can be achieved by defining an abstract ordered signature S and two singleton signatures $S1$ and $S2$ which extend S:

```
open util/ordering[S]
abstract sig S {}
one sig S1, S2 extends S {}
```

In our current translation to B, we do not consider that enumerations are ordered. We rather translate enumerations based on the declared signatures, *i.e.*, we introduce a deferred set S and two constants $S1$ and $S2$ which are singleton subsets of S. Furthermore, we set the extending signatures to be distinct. For instance, for the signature $S1$, this is done using the additional constraint $S1 \subseteq S \wedge \mathrm{card}(S1) = 1 \wedge S1 \cap S2 = \varnothing$.

Not translating enumerations as ordered signatures allows ProB to use advanced optimization techniques such as symmetry reduction. In case a model relies on enumerations being ordered, we could of course treat enumerations as ordered signatures and translate as described in Section 5.3. In the future, our translator should check automatically if an Alloy models makes use of the ordering of enumerations and translate accordingly.

5.5. Sequences

In B, sequences are defined as partial functions with finite and coherent domains $1..n$, where $n \in \mathbb{N}$ is the size of the sequence. Sequences are therefore defined as sets of pairs and might be nested arbitrarily.

In Alloy, the field of a signature, the parameters of a quantification and the arguments of a function can be defined as a sequence of atoms using the **seq** keyword. In contrast to B, the elements of a sequence are enumerated from 0 to $n - 1$.

In consequence, we cannot straightforwardly translate sequences from Alloy to B. If using B's internal representation of sequences, we would need to increase each integer value from Alloy accessing a sequence by one, and decrease each integer value by one which is computed by a sequence operation or used within one. Moreover, there is no counterpart to most of Alloy's sequence operations in B, so we would have to manually implement most of the operations anyway. We thus decided to retain the domain of a sequence defined by Alloy. For the translation of a sequence, we define a partial function with domain $0..n - 1$ as described in Section 4.3, where $n \in \mathbb{N}$ is the sequence's cardinality, and use manually implemented operations on sequences without resorting to B's sequence operations.

Alloy allows to modify a set of sequences using a set of elements rather than a single element at a time. For instance, consider the signature **sig** `T {s:` **seq** `Int}` which defines a field s as a sequence of integers. Assuming that T has a scope of exactly two, *i.e.*, `T = {T$0, T$1}` in the Alloy Evaluator, a call to `T.s.insert[0,{1}+{2}]` is satisfiable. For instance, this results in inserting 1 at position 0 in `T$0.s` and inserting 2 at position 0 in `T$1.s`. As this behavior can also be achieved by accessing each field s of T's instances independently and cannot be described efficiently by a single expression in B, we decided to only allow operations on sequences using single elements.

Therefore, we use an additional semantic function $E_{one}[\![.]\!]i$ that transforms a singleton set into a scalar expression. If the input is not a singleton set, a well-definedness error is thrown by ProB. To do so, we use the definite description operator, noted MU, described in Section 5.1. We then define the semantic function as follows:

- $E_{one}[\![x]\!]i \ \widehat{=}\ y$ if $E[\![x]\!]i = \{y\}$
- $E_{one}[\![x]\!]i \ \widehat{=}\ MU(E[\![x]\!]i)$

It is very similar to the semantic function $E_{int}[\![.]\!]i$, just replacing Σ by the MU operator. Again, the first rule is in principle redundant, but improves the performance of the translation (avoiding the introduction of MU if possible).

By default, an Alloy model defines a maximum allowed length of sequences due to the SAT encoding. The maximum size can be changed within the scope of a **run** or **check** command by using the **seq** keyword.

Let m be the maximum allowed length of sequences of a specific **run** or **check** command, and c_s be the cardinality of the set s in B, *i.e.*, $c_s = \text{card}(E[\![s]\!]i)$. The sequence operations provided by Alloy are translated as follows:

- $E[\![s.\mathit{first}]\!]i \ \widehat{=}\ E[\![s]\!]i[\{0\}]$
- $E[\![s.\mathit{last}]\!]i \ \widehat{=}\ E[\![s]\!]i[\{c_s - 1\}]$
- $E[\![s.\mathit{rest}]\!]i \ \widehat{=}$ IF $E[\![s]\!]i = \varnothing$ THEN $\varnothing$
  ELSE $\lambda z.(z \in 0..(c_s - 2) \mid E[\![s]\!]i(z+1))$ END
- $E[\![s.\mathit{elems}]\!]i \ \widehat{=}\ \text{ran}(E[\![s]\!]i)$
- $E[\![s.\mathit{butlast}]\!]i \ \widehat{=}\ E[\![s]\!]i[\{c_s - 2\}]$
- $M[\![s.\mathit{isEmpty}]\!]i \ \widehat{=}\ E[\![s]\!]i = \varnothing$
- $M[\![s.\mathit{hasDups}]\!]i \ \widehat{=}\ c_s \neq \text{card}(\text{ran}(E[\![s]\!]i))$
- $E[\![s.\mathit{inds}]\!]i \ \widehat{=}\ 0..(c_s - 1)$
- $E[\![s.\mathit{lastIdx}]\!]i \ \widehat{=}\ \{c_s - 1\} \cap 0..(c_s - 1)$
- $E[\![s.\mathit{afterLastIdx}]\!]i \ \widehat{=}\ \{c_s\} \cap 0..(m - 1)$

- $E[\![s.idxOf[x]]\!]i \mathrel{\widehat{=}}$ IF $(E[\![s]\!]i)^{-1}[\{E_{one}[\![x]\!]i\}] \neq \varnothing$ THEN
  $\{\min((E[\![s]\!]i)^{-1}[\{E_{one}[\![x]\!]i\}])\}$
  ELSE $\varnothing$ END
- $E[\![s.lastIdxOf[x]]\!]i \mathrel{\widehat{=}}$ IF $(E[\![s]\!]i)^{-1}[\{E_{one}[\![x]\!]i\}] \neq \varnothing$ THEN
  $\{\max((E[\![s]\!]i)^{-1}[\{E_{one}[\![x]\!]i\}])\}$
  ELSE $\varnothing$ END
- $E[\![s.indsOf[x]]\!]i \mathrel{\widehat{=}} (E[\![s]\!]i)^{-1}[\{E_{one}[\![x]\!]i\}]$
- $E[\![s.append[t]]\!]i \mathrel{\widehat{=}} (0..(m-1)) \lhd (E[\![s]\!]i \cup$
  $\lambda z.(z \in c_s..(c_s + c_t - 1) \mid E[\![t]\!]i(z - c_s)))$
- $E[\![s.add[x]]\!]i \mathrel{\widehat{=}}$ IF $c_s < m$ THEN $E[\![s]\!]i \cup \{c_s \mapsto E_{one}[\![x]\!]i\}$
  ELSE $E[\![s]\!]i$ END
- $E[\![s.delete[j]]\!]i \mathrel{\widehat{=}}$ IF $(0 \leq E_{int}[\![j]\!]i)$ THEN
  $(0..(E_{int}[\![j]\!]i - 1) \lhd E[\![s]\!]i) \cup \lambda z.(z \in E_{int}[\![j]\!]i..(c_s - 2) \mid E[\![s]\!]i(z + 1))$
  ELSE $E[\![s]\!]i$ END
- $E[\![s.setAt[j, x]]\!]i \mathrel{\widehat{=}}$ IF $(E_{int}[\![j]\!]i \geq 0 \wedge E_{int}[\![j]\!]i \leq c_s)$ THEN
  $E[\![s]\!]i \Lleftarrow \{E_{int}[\![j]\!]i \mapsto E_{one}[\![x]\!]i\}$ ELSE $E[\![s]\!]i$ END
- $E[\![s.insert[j, x]]\!]i \mathrel{\widehat{=}} 0..(m-1) \lhd ((0..(E_{int}[\![j]\!]i - 1) \lhd E[\![s]\!]i) \cup \{E_{int}[\![j]\!]i \mapsto$
  $E_{one}[\![x]\!]i\} \cup \lambda z.(z \in (E_{int}[\![j]\!]i+1)..c_s \wedge (z-1) \in \mathrm{dom}(E[\![s]\!]i) \mid E[\![s]\!]i(z-1)))$
- $E[\![s.subseq[from, to]]\!]i \mathrel{\widehat{=}}$
  IF $E_{int}[\![from]\!]i \geq 0 \;\wedge\; E_{int}[\![from]\!]i \leq E_{int}[\![to]\!]i \;\wedge\; E_{int}[\![to]\!]i < c_s$ THEN
  $\lambda z.(z \in 0..(E_{int}[\![to]\!]i - E_{int}[\![from]\!]i) \mid E[\![s]\!]i(z + E_{int}[\![from]\!]i))$
  ELSE $\varnothing$ END

Note that in several translations of sequence operations, *e.g.*, in the translation of `s.append[t]`, we could use a set comprehension instead of a lambda expression. However, since lambda expressions constitute total functions, they improve performance when solving constraints.

As can be seen in the translations, the maximum allowed length of sequences influences the behavior of several operations. For instance, the result of appending two sequences is truncated if it exceeds the scope of sequences. As described in Section 4.14, we usually translate all commands into the same B machine. In case at least two commands define a different maximum allowed length of sequences, our translation would possibly behave differently than the Alloy model does as we can only consider a single scope at a time when translating operations on sequences.

We thus analyze an Alloy model prior to the translation to determine if it uses sequences within differently scoped commands, *i.e.*, commands that do not define a common maximum allowed length of sequences. If so, we only translate a single command at a time which can be selected by the user within ProB's graphical user interface. Otherwise, all commands are translated into the same B machine.

Note that the Alloy Analyzer does not enforce that sequence operations are called with well-defined sequences[4]. In contrast, our translation to B provides static type safety which improves error detection.

Further, we noticed that the Alloy Analyzer behaves inconsistently regarding the evaluation of preconditions. The operations `delete`, `setAt` and `insert` are unsatisfiable if their precondition is false while the other operations always succeed, *e.g.*, returning the input sequence if an operation's precondition is false. To achieve a consistent behavior regarding the use of preconditions, we decided to not fail for the mentioned operations if their precondition is false but return the input sequence.

However, we could also translate operations to be unsatisfiable if their precondition is false. As a B expression cannot fail without throwing an error, *e.g.*, a well-definedness error, we would need to pass a flag in the environment i to inform the preceding predicate to fail.

6. Tooling

In the following section, we will first give an overview over the tooling used to automate the translation from Alloy to B. Additionally, we will give insight into the Prolog implementation in Section 6.2 and further implementation details.

6.1. Overview

As shown in Figure 2, our automatic translation relies on two software tools. The first component is a small application (around 500 lines of code, not counting tests) written in Kotlin and running on the JVM. Its purpose is to use the original parser and typechecker of the Alloy Analyzer to parse Alloy files and pretty print the resulting abstract syntax tree into a Prolog representation that can be loaded by ProB's core. During this first translation, some changes in representation are done in order to make all information available to the Alloy Analyzer available to ProB as well, *i.e.*, we extend the abstract syntax tree with additional information.

Moreover, we generalize the types provided by the Alloy parser to their top-level signatures. For instance, let S be a top-level signature, and S_1, S_2 are both signatures that extend S. The type of the expression $S_1 + S_2$ provided by the Alloy Analyzer's parser is a set (a relation with arity 1) of the two types S_1 and S_2. For our translation to B, it is more intuitive and necessary to use the most general type except for **univ** if present. In the given example, the most general type is a set of type S. We are then able to easily set up typing constraints for each Alloy construct during the translation to B. Otherwise, we would need to generalize types on demand in Prolog. However, two signatures might not have a parent type except for **univ**. Since we want to avoid the universe type in B as described in Section 4.4, we define a parent type for each of such signature collections as a deferred set in B.

[4]See `http://alloytools.org/quickguide/seq.html`

Afterwards, the Alloy abstract syntax tree is read by ProB and translated into ProB's internal representation of a B machine, following the translation rules discussed in Section 4. The result is an untyped B abstract syntax tree, that is fed into the regular B typechecker. Once typed, it can be used inside the model checker or animator as well as in the constraint solver. Furthermore, all backends available to ProB consume the same internal representation, *i.e.*, the resulting typed B abstract syntax tree can be fed to them as well. For instance, a constraint solver which uses the Alloy Analyzer's Kodkod API [3] to translate B to SAT is available [11]. Furthermore, an integration with the SMT solver Z3 [18] can be used to solve constraints [10], or a combination of the CLP(FD) and SMT backends where both solvers share constraints [19].

As described in Section 4.14, **run** and **check** commands are translated to B machine operations. To execute an Alloy command 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 the `cbc_sequence` command of ProB, which will send the operation's precondition and the machine's properties to ProB's constraint solver. In the latter case the machine's properties are considered as we translate several constraints in this machine section, for instance, constraints on the fields of signatures.

In order to use the generated B machine inside other B tools such as AtelierB, ProB can export the internal representation to a regular B machine file. Further, we provide a ProB preference to translate a single command into the B machine's `ASSERTIONS` section rather than creating a machine operation as described in Section 4.14. As the assertion of a **check** command is negated to search for a counterexample, we remove the negation when adding the constraint to the B machine assertions. This enables the generation of proof obligations in AtelierB. The command to be translated can be selected by the user within ProB's graphical user interface.

6.2. Prolog Encoding of Translation Rules

As stated above, we use a small Kotlin library to extract the AST generated by the Alloy Analyzer's parser and typechecker. The resulting file is then read by ProB's Prolog core.

The mathematical rules featured in our translation can quite often be translated to Prolog clauses straightforwardly. In particular, the implementation usually consists of single translation rules being implemented by a single corresponding Prolog clause. This leads to an implementation that is close to the formal specification. In consequence, the implementation is comprehensible and can easily be reviewed, extended and adapted. Take for example the rule for the Alloy plus operator:

$$E[\![p + q]\!]i \;\widehat{=}\; E[\![p]\!]i \cup E[\![q]\!]i$$

Listing 5 shows parts of the Prolog code translating the set union from Alloy to B, where `translate_e_p` is the Prolog name of the function $E[\![.]\!]i$.

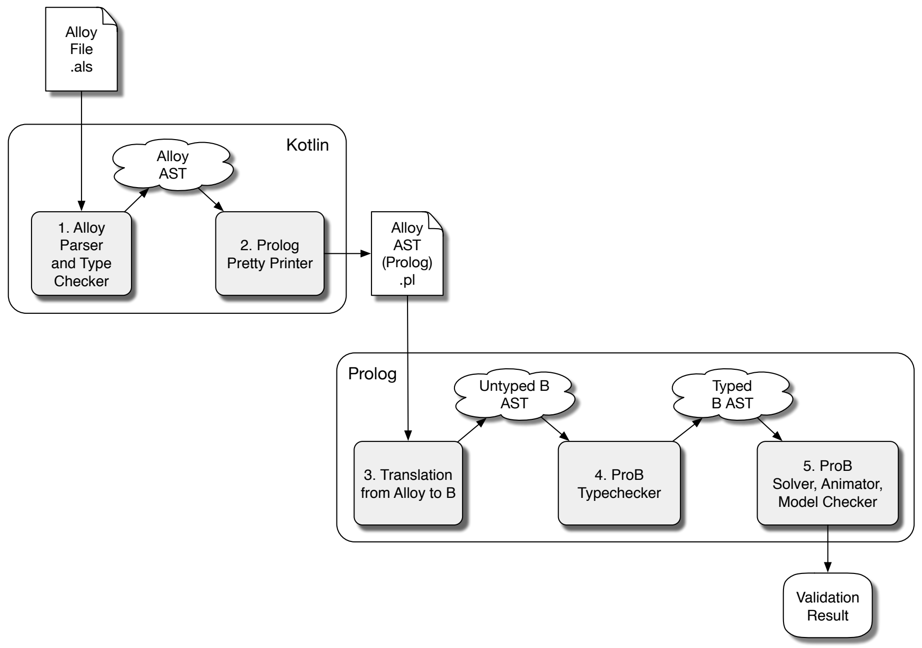


Figure 2: Phases of the Translation from Alloy to B

The code is somewhat more generic and factors several rules into one, namely all binary operators that can be translated directly to B operators. Additional operators that can be directly translated are given by Prolog facts further defining `alloy_to_b_binary_operator`. You can also see that the code translates Alloy's position information to B (for error messages). The keen observer will note that the environment i is not present; it is currently encoded using assert/retract (*i.e.*, as Prolog global variables).

7. Empirical Evaluation

To validate the correctness of our translation we have applied it to a variety of mathematical laws (see Fig. 6) and have checked that ProB does not find counterexamples to those laws on the translated B machines.

Furthermore, we have translated several Alloy models to B. In the following, we will give a brief empirical evaluation of selected models, comparing the performance of the Alloy Analyzer and ProB.

Benchmarks were run on an Intel Core i7-8750H CPU (2.2GHz) and 16GB of RAM. We use the median time of five independent checks. The runtime of the Alloy Analyzer includes generating the conjunctive normal form and uses the SAT4J backend. For the ProB constraint solver we purely use the CLP(FD) backend with a linear enumeration order and without extensions like KodKod [11] or Z3 [10]. Of course, despite constraint solving itself, processing

Listing 5: Parts of the Prolog Code that Translates Alloy's Set Union to B

```
translate_binary_e_p(Binary,TBinary) :-
  Binary =.. [Op,Arg1,Arg2,_Type,POS],
  alloy_to_b_binary_operator(Op,BOp),
  translate_e_p(Arg1,TArg1),
  translate_e_p(Arg2,TArg2),
  translate_pos(POS,BPOS),
  TBinary =.. [BOp,BPOS,TArg1,TArg2].

alloy_to_b_binary_operator(plus,union).
  ...
```

Listing 6: Checking of Mathematical Laws

```
abstract sig setX { }
one sig V {
  SS: set setX,
  TT: set setX,
  VV: set setX,
  Empty: set setX
}
fact EmptySet {  no V.Empty }
assert SetLaws {
  V.SS + V.SS = V.SS
  no V.SS - V.SS
  V.SS = V.SS & V.SS
  V.SS - V.Empty = V.SS
  V.Empty =  V.SS -  V.SS
  V.Empty -  V.SS =  V.Empty
  V.SS +  V.TT =  V.TT +  V.SS
  ...
}
check SetLaws for 5 setX, 7 int
```

an Alloy model using the Alloy Analyzer's parser, pretty printing the model to Prolog, transforming types as described in Section 6, and translating the model to B needs some time. However, this is not a bottleneck for performance. In the following, we thus assume each model to be loaded in the Alloy Analyzer and ProB, *i.e.*, we only measure the impact of our translation on finding solutions to a model's constraints.

Since the Alloy Analyzer translates models to SAT, we assume it to be efficient for mostly relational models. However, SAT encoding is often inefficient for integers, *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 weaker performance compared to the Alloy Analyzer. To explore both extremes, we chose different models for performance comparison.

An exact opposite to our translation has been presented by Plagge and

Leuschel [11], which uses the Alloy Analyzer's Kodkod API [3] to translate B to SAT. We further solve the translated models using ProB with its Kodkod backend in order to investigate if our translation from Alloy to B is needlessly complicated. If this is not the case, we expect the runtime to be only slightly larger than the one of the Alloy Analyzer. Note that in recent work [**?**] we have shown that an integration of the Alloy and ProB backends can be very useful for complex constraint satisfaction problems.

We start with translating 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 cannot be left alone with a chicken. The model is interesting for our performance evaluation as it uses an ordered signature for states. The Alloy Analyzer finds a solution in 21 ms (6 ms only SAT solving). Although the translated model is valid, ProB fails to find a solution in less than 5 minutes.

The B machine 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 about 7 s. After rewriting the model in idiomatic B style by hand, ProB can solve it in about 80 ms. However, this translation is a manual optimization using background knowledge and cannot simply be generalized. Using the Kodkod backend of ProB does not improve performance significantly. This indicates that for this specific model our translation is not performant which is most likely caused by the translation of ordered signatures.

Besides the river crossing puzzle, we translated a model of the n queens problem as it makes use of integer arithmetic. 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.

We evaluated the n queens model for $n \in 4..20$ using ProB and the Alloy Analyzer with the MiniSat and SAT4J backend. As a comparison, we also measured the time that the Alloy Analyzer needs to generate the conjunctive normal form. The evaluation in Figure 3 shows that ProB is the fastest solver for the chosen model. ProB's runtime for solving the constraints ranges from 5 - 1328 ms. The time needed for generating the conjunctive normal form is similar to the time ProB needs for solving the constraints and ranges from 18 - 1028 ms. The solving time of the Alloy Analyzer gets worse when increasing the bit-width for $n > 8$ and $n > 16$. In Figure 3 we can see that the runtime of each solver is increasing non-linearly, especially when using the Alloy Analyzer with the SAT4J backend. On the one hand, this might be caused by inaccuracies in our measurements. On the other hand, the constraints might be easier to solve for specific configurations. ProB's runtime for $n = 20$ is an outlier and the cause of this performance drop needs to be investigated more thoroughly. As a comparison, ProB can solve the translated model for $n = 21$ in about 100 ms. Further, when using a randomized enumeration order, ProB can solve the translated model for $n = 20$ in about 80 ms. Note that an idiomatic B version of the n queens puzzle for $n = 20$ can be solved in around 20 ms by ProB. Altogether, it can be seen that integers are a bottleneck for performance when

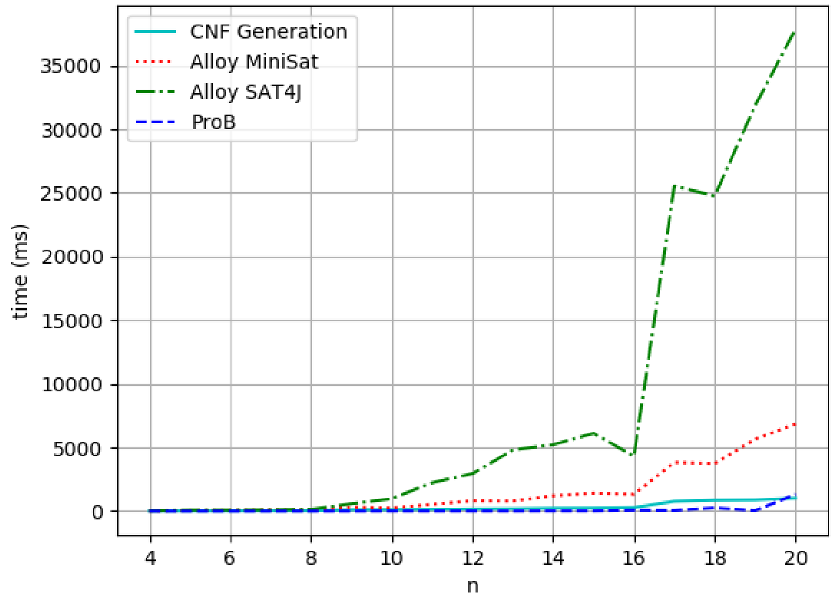


Figure 3: Finding a Single Solution for the n Queens Puzzle with Varying n

encoding constraints to pure SAT problems.

As a rather simple benchmark, we translated a model of the knights and knaves puzzle. The puzzle defines two types of humans, which either always tell the truth (knights) or always lie (knaves). The goal is to determine the type of several persons from a set of statements each made by one person. The model of the puzzle that we used contains three individual settings with statements made by two or three persons. The model just uses joins, set unions as well as one existential quantification. The Alloy Analyzer finds a solution for the model in 10 ms (6 ms only SAT solving) while ProB needs 5 ms. Solving the translated model with ProB and its Kodkod backend needs about 150 ms, which is most likely caused by the additional overhead of translating B to Kodkod.

Furthermore, we translated a model of the so called jobs puzzle [20], which defines eight distinct jobs and four persons whose names imply their gender. The goal is to allocate two different jobs to each person and establish the relationships between male and female persons considering a set of constraints. For instance, a constraint states that the husband of the chef is the telephone operator. Besides the common join operations, the model uses a predicate from the extension **util/relation**, defines a field to be quantified by **some**, and uses three quantifications as well as two cardinality constraints. The Alloy Analyzer finds a solution in 23 ms (9 ms only SAT solving). The ProB constraint solver is not able to find a solution within several minutes. As our translation from Alloy to B has to be generalized, some translations considering certain arities are currently not ideal for the ProB constraint solver. To counter this, we intend to provide additional rules to rewrite B abstract syntax trees prior to solving constraints as described in Section 4.16 and improve the constraint solver in general. When using the Kodkod backend of ProB, the translated model can be solved in about 50 ms. This shows that the translated B model is not needlessly complicated but contains specific constructs that cannot be handled efficiently by ProB's CLP(FD) backend. As a comparison, an idiomatic B version of the Jobs puzzle [**?**] can be solved by ProB's CLP(FD) backend in about 150 ms.

To obtain further benchmarks, we translated a model of the Zebra puzzle (also called Einstein's puzzle). The goal is to find a person owning a specific pet for given constraints describing the preferences and houses of a group of five persons. There is only one solution. The model defines one ordered signature, five unordered signatures, and uses fifteen existential quantifications. The Alloy Analyzer finds the solution in 12 ms (5 ms only SAT solving). ProB on the other hand currently needs 748 ms to find the solution. Again, some constraints are not ideal for ProB and require improvements to the post-processing described in Section 4.16 or the constraint solver itself. In this case, using the Kodkod backend of ProB neither improves nor worsens performance. Note that ProB can solve the original Z version of the Zebra puzzle in about 100 ms.

Lastly, we translated a model of the towers of Hanoi puzzle, with three stakes and several discs with different sizes. The model we use defines three ordered signatures, several joins and nested quantifiers. The Alloy Analyzer finds a solution in about 5.2 s while ProB is currently not able to find a solution within several minutes. Using the Kodkod backend of ProB does not improve performance significantly. In this case, our translation of orderings as presented in Section 5.3 is inefficient for constraint solving. The Alloy model defines a signature field as a relation between three ordered signatures: **sig** *State* { *on* : *Disc* -> **one** *Stake* }. In our current translation to B, this results in a possibly large set leading to a bad performance. To counter this, we want to investigate the causes of performance loss more thoroughly and improve our translation wherever possible. Moreover, we want to investigate a translation into a (symbolic or explicit) model checking rather than a constraint satisfaction problem. That is, in case access on ordered elements is linear, we can encode orderings as B machine states using machine variables and operations on orderings as state transitions using machine operations. Doing so, the ProB model checker can be used to find solutions for a model which uses ordered signatures. Note that the ProB model checker can solve manually specified B versions of the river crossing puzzle in about 100 ms and the towers of Hanoi puzzle in about 250 ms.

In summary, we have translated several Alloy models of well-known logic puzzles to classical B. As pointed out, our translation is not optimal for models using relational operators or ordered signatures regarding ProB's performance in solving constraints. Yet, ProB outperforms the Alloy Analyzer for models using integers by several orders of magnitude. Table 1 summarizes the comparison of the Alloy Analyzer's and ProB's runtimes in solving the presented models. We used a maximum solver timeout of five minutes. Further, we present

Table 1: Performance Evaluation

Model	Runtime in ms	
	Alloy	ProB
River Crossing Puzzle	21	$> 300\,000$
4 Queens Puzzle	26	5
8 Queens Puzzle	91	8
12 Queens Puzzle	820	16
16 Queens Puzzle	1334	78
20 Queens Puzzle	6850	1328
Knights and Knaves Puzzle	10	5
Jobs Puzzle	23	$> 300\,000$
Zebra Puzzle	12	748
Towers of Hanoi Puzzle	5201	$> 300\,000$

the results using the translation as is without any manual optimization of the generated B code and without using the Kodkod or Z3 backend of ProB. The presented times of the Alloy Analyzer for solving the n queens puzzle are the ones using the Minisat backend.

8. Improvements Over Existing Alloy Tools

Even though our translation cannot always compete with the Alloy Analyzer as we have demonstrated in Section 7, it provides several interesting improvements and applications.

8.1. Integers

Mathematically speaking, the integers in Alloy are unsound when overflow detection is turned off. In contrast, ProB has multi-precision integers without overflows[5]. According to Milicevic and Jackson [21] the 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 translating a model to B would be to use an SMT-based backend for Alloy [22, 23, 24].

For example, for the model shown in Listing 7 Alloy 4.2 finds a counterexample, while ProB correctly determines that no counterexample exists. If overflows are permitted (the default), the Alloy Analyzer finds a counterexample for the first formula. If overflows are forbidden, no counterexample is detected by the Alloy Analyzer for the first formula, but then a counterexample is found for the second one. With higher integer ranges the translation fails.

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

Listing 7: Alloy Model Demonstrating the Unsoundness of Integers

```
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
}
// for 8 int Translation capacity exceeded
check Bug for 3 setX, 7 int
```

Listing 8: Exemplary Alloy Model Using Higher-Order Quantification

```
open util/integer
abstract sig setX { }
one sig V {
  SS:   setX -> setX,
  TT:   setX -> setX
}
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
```

8.2. Higher-Order Quantification

The universal quantification shown in Listing 8, using the same signatures as in Listing 7, causes an error. The Alloy Analyzer states that analysis cannot be performed since it requires higher-order quantification that could not be skolemized. ProB, on the other hand, can check the validity of this assertion. An extension of Alloy called Alloy* [16] might be able to handle this example. In the future, we would like to investigate translating Alloy* models to B.

8.3. Proof

Finally, our translation to B also makes it possible to apply existing provers for the language, such as AtelierB [13], to translated Alloy models. One could thus try to develop a proof assistant for Alloy, similar to the work pursued by Ulbrich et al. [25] via a translation to the first-order logic supported by Key.

In the example shown in Listing 9, 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, removing one element from `src` and adding it to `dst`, preserves the invariant `src+dst=Object`, *i.e.*, that the union of `src` and `dst` covers exactly Object.

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

Listing 9: Exemplary Alloy Model to Prove an Assertion in AtelierB

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

a check operation encoding the predicate `add_preserves_inv` with universal quantification. Listing 10 shows the B machine we have input into AtelierB.

It was obtained by pretty-printing from ProB. For the translation from Alloy to B, we enabled the preference to translate a model without scopes described in the end of section Section 4.14 as well as the preference to translate a single command into the B machine assertions described in the end of Section 6.1 (so that AtelierB generates the desired proof obligation).

9. Related and Future Work

Translations to Alloy have been pursued from B [26, 27] and also Z [28]. Rather than translation directly to Alloy, a translation from B to Kodkod has been introduced and implemented inside ProB [11].

Other formal languages have previously been translated to B as well, *e.g.*, Z [29] and TLA^+ [30]. A comparison between $TLA^+$and Alloy has been presented by Macedo and Cunha [31].

The original paper by Jackson [17] (notably Figure 2) 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. Future work could be a formal proof of the equality of the different semantics given for Alloy.

Another, albeit less thorough approach, would be to implement a combined solver that runs the Alloy Analyzer and ProB in conjunction and thus verifies the results using a double chain.

While we strive for full support of the Alloy language, we currently do not provide custom implementations for all available utility modules. In particular, we are missing implementations for the translation of common operations on graphs and ternary relations. We currently just translate these modules using our tool as they are defined in Alloy. Of course, the resulting translation might not be as efficient as providing custom implementations.

Listing 10: Translated Alloy Model to Prove an Assertion in AtelierB

```
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
```

Furthermore, we intend to translate Alloy* [16] and Electrum [32] (which is a temporal extension of the Alloy modeling language) to B. As B and ProB have support for higher-order quantification and linear temporal logic, translation should be straightforward.

While our translation of orderings, as presented in Section 5.3, allows translating arbitrary Alloy models, the resulting B machine is often suboptimal for ProB's solving kernel as shown in Section 7. To improve performance, we want to investigate alternative translations of orderings. For instance, we could impose an order on the elements of a signature S by defining a bijective function $S \rightarrowtail\!\!\!\!\twoheadrightarrow 0..(\mathrm{card}(S)-1)$ allocating unique indices to the elements. Further, we want to investigate a translation into a (symbolic or explicit) model checking rather than a constraint satisfaction 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 [33] to find solutions.

Near and Jackson [34] presented an imperative extension of Alloy, *i.e.*, making a step towards B and its operations. Similarly, Frias et al. [35, 36] extended Alloy with actions. Cunha [37] presented an approach using bounded model checking for temporal properties in Alloy. It would be interesting to extend our translation and produce idiomatic B machines with B operations from such Alloy models.

As soon as our translation relies more on operations, we want to investigate translating into a set of models linked by refinement rather than translating an Alloy model into a single B machine. However, since we currently do not impose any restrictions on the Alloy model to be translated, it remains to be seen to what extent automatic refinement techniques such as the one used in BART [38] or the one introduced by Iliasov et al. [39] can be used efficiently.

10. Discussions and Conclusions

In summary, we have presented an automatic translation of Alloy to B, which provides an alternative semantics definition of Alloy, and enables proof and constraint solving tools of B to be applied to Alloy specifications. We have shown empirically that for certain constraints, the B language tools in general and ProB in particular are superior to the Alloy Analyzer and its SAT backend. For other constraints however, the Alloy Analyzer outperforms ProB. As expected, different backends exhibit different strengths and weaknesses. Using our translation, we make ProB's backends available to Alloy users, enabling them to experiment with technologies other than the ones employed by the Alloy Analyzer.

The formal definition of the translation revealed both shortcomings and elegant features of Alloy and B. One aspect where B is awkward is the treatment of tuples: many encodings exist and the modeler has to know which one is being used. Associative tuples with flexible join and projection operations (similar to database operations) would be a very useful addition to B.

The object-oriented notation of Alloy makes specifications more modular and easier to read than classical B and is closer to a UML-like model that most conventional designers are familiar with. In B, one can use records or use B's machine decomposition statements like `INCLUDE`, but the syntax is not as handy as Alloy's.

Alloy allows expressing certain constructs in a much more concise fashion, showing that B sometimes is not as expressive as desired. However, the same applies for Alloy as well. Multiplicity annotations in Alloy are inspired from conceptual modelling notations, but their mathematical representation relies on well-known classes of functions that the B notation natively supports concisely. We have also shown that B can be much more concise and expressive especially when dealing with integers.

Alloy is not tailored for transition system analysis; system behavior is analyzed using bounded traces. ProB offers sophisticated tools for analyzing the transition graph of a system; it supports invariant and deadlock checks, LTL[e] and CTL, fairness constraints, reachability analysis, and model-based testing.

In general, a comparison and translation like the one presented in this article should inspire the evolution of both languages. We hope that our translation can serve as a vehicle of communication between the Alloy and B communities.

[1] J.-R. Abrial, The B-book: Assigning Programs to Meanings, Cambridge University Press, New York, NY, USA, 1996.

[2] D. Jackson, Software Abstractions: Logic, Language and Analysis, MIT Press, 2006.

[3] E. Torlak, D. Jackson, Kodkod: A Relational Model Finder, in: Proceedings TACAS, Vol. 4424 of LNCS, Springer, 2007, pp. 632–647.

[4] M. Leuschel, J. Bendisposto, I. Dobrikov, S. Krings, D. Plagge, From Animation to Data Validation: The ProB Constraint Solver 10 Years On, in: J.-L. Boulanger (Ed.), Formal Methods Applied to Complex Systems: Implementation of the B Method, Wiley ISTE, Hoboken, NJ, 2014, Ch. 14, pp. 427–446.

[5] M. Leuschel, M. Butler, ProB: An automated analysis toolset for the B method, Int. J. Softw. Tools Technol. Transf. 10 (2) (2008) 185–203.

[6] M. Leuschel, M. Butler, ProB: A model checker for B, in: Proceedings FME, Vol. 2805 of LNCS, Springer, 2003, pp. 855–874.

[7] J. Jaffar, S. Michaylov, Methodology and Implementation of a CLP System, in: Proceedings ICLP, MIT Press, 1987, pp. 196–218.

[8] M. Carlsson, G. Ottosson, B. Carlson, An Open-Ended Finite Domain Constraint Solver, in: Proceedings PLILP, Vol. 1292 of LNCS, Springer, 1997, pp. 191–206.

[9] S. Krings, M. Leuschel, Constraint Logic Programming over Infinite Domains with an Application to Proof, in: Proceedings WLP, Vol. 234 of EPTCS, Electronic Proceedings in Theoretical Computer Science, 2016.

[10] S. Krings, M. Leuschel, SMT Solvers for Validation of B and Event-B Models, in: Proceedings iFM, Vol. 9681 of LNCS, Springer, 2016.

[11] D. Plagge, M. Leuschel, Validating B, Z and TLA$^+$ using ProB and Kodkod, in: Proceedings FM, Vol. 7436 of LNCS, Springer, 2012, pp. 372–386.

[12] A. Sülflow, U. Kühne, R. Wille, D. Große, R. Drechsler, Evaluation of SAT-like Proof Techniques for Formal Verification of Word-Level Circuits, in: Proceedings IEEE WRTLT, IEEE Computer Society Press, Beijing, China, 2007.

[13] ClearSy, Atelier B, User and Reference Manuals, Aix-en-Provence, France, available at `http://www.atelierb.eu/` (2009).

[14] S. Krings, J. Schmidt, C. Brings, M. Frappier, M. Leuschel, A Translation from Alloy to B, in: Proceedings ABZ, Vol. 10817 of LNCS, Springer, 2018, pp. 71–86.

[15] D. Jackson, C. A. Damon, Elements of Style: Analyzing a Software Design Feature with a Counterexample Detector, in: Proceedings ISSTA, ISSTA '96, ACM, New York, NY, USA, 1996, pp. 239–249.

[16] S. Krings, M. Leuschel, Proof Assisted Bounded and Unbounded Symbolic Model Checking of Software and System Models, Sci. Comput. Program.

[17] A. Milicevic, J. P. Near, E. Kang, D. Jackson, Alloy*: A General-Purpose Higher-Order Relational Constraint Solver, Formal Methods in System Design.

[18] D. Jackson, Alloy: A Lightweight Object Modelling Notation, ACM Transactions on Software Engineering and Methodology 11 (2002) 256–290.

[19] D. E. Knuth, The Art of Computer Programming, Volume 1: Fundamental Algorithms, Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1997.

[20] R. T. Boute, The Euclidean Definition of the Functions Div and Mod, ACM Trans. Program. Lang. Syst. 14 (2) (1992) 127–144.

[21] S. Krings, Towards Infinite-State Symbolic Model Checking for B and Event-B, Ph.D. thesis, Heinrich-Heine-University, Düsseldorf, Germany (2017).

[22] L. de Moura, N. Bjørner, Z3: An Efficient SMT Solver, in: Proceedings TACAS, Vol. 4963 of LNCS, Springer, 2008, pp. 337–340.

[23] S. Krings, M. Leuschel, SMT Solvers for Validation of B and Event-B Models, in: Proceedings iFM, Vol. 9681 of LNCS, Springer, 2016, pp. 361–375.

[24] S. Krings, M. Leuschel, P. Körner, S. Hallerstede, M. Hasanagic, Three Is a Crowd: SAT, SMT and CLP on a Chessboard, in: Proceedings PADL 2018, Vol. 10702 of LNCS, Springer, 2018, pp. 63–79.

[25] L. Wos, R. Overbeck, E. Lusk, J. Boyle, Automated Reasoning: Introduction and Applications, Prentice Hall, Old Tappan, 1984.

[26] M. Leuschel, D. Schneider, Towards B as a High-Level Constraint Modelling Language, in: Y. Ait Ameur, K.-D. Schewe (Eds.), Abstract State Machines, Alloy, B, TLA, VDM, and Z, Vol. 8477 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2014, pp. 101–116.

[27] A. Milicevic, D. Jackson, Preventing Arithmetic Overflows in Alloy, Sci. Comput. Program. 94 (2014) 203–216.

[28] E. Torlak, M. Taghdiri, G. Dennis, J. P. Near, Applications and Extensions of Alloy: Past, Present and Future, Mathematical Structures in Computer Science 23 (4) (2013) 915–933.

[29] A. A. E. Ghazi, M. Taghdiri, Analyzing Alloy Formulas using an SMT Solver: A Case Study, CoRR abs/1505.00672.

[30] B. Meng, A. Reynolds, C. Tinelli, C. W. Barrett, Relational Constraint Solving in SMT, in: Proceedings CADE, Vol. 10395 of LNCS, Springer, 2017, pp. 148–165.

[31] M. Ulbrich, U. Geilmann, A. A. E. Ghazi, M. Taghdiri, A Proof Assistant for Alloy Specifications, in: Proceedings TACAS, Vol. 7214 of LNCS, 2012, pp. 422–436.

[32] L. Mikhailov, M. J. Butler, An Approach to Combining B and Alloy, in: Proceedings ZB, Vol. 2272 of LNCS, Springer, 2002, pp. 140–161.

[33] P. J. Matos, J. Marques-Silva, Model Checking Event-B by Encoding into Alloy, in: Proceedings ABZ, Vol. 5238 of LNCS, Springer, 2008, p. 346.

[34] P. Malik, L. Groves, C. Lenihan, Translating Z to Alloy, in: Proceedings ABZ, Vol. 5977 of LNCS, Springer, 2010, pp. 377–390.

[35] D. Plagge, M. Leuschel, Validating Z Specifications using the ProB Animator and Model Checker, in: Proceedings iFM, Vol. 4591 of LNCS, Springer, 2007, pp. 480–500.

[36] D. Hansen, M. Leuschel, Translating TLA+ to B for validation with ProB, in: Proceedings iFM, Vol. 7321 of LNCS, Springer, 2012, pp. 24–38.

[37] N. Macedo, A. Cunha, Alloy Meets TLA+: An Exploratory Study, CoRR abs/1603.03599.

[38] N. Macedo, J. Brunel, D. Chemouil, A. Cunha, D. Kuperberg, Lightweight Specification and Analysis of Dynamic Systems with Rich Configurations, in: Proceedings FSE, FSE 2016, ACM, New York, NY, USA, 2016, pp. 373–383.

[39] J. P. Near, D. Jackson, An Imperative Extension to Alloy, in: Proceedings ABZ, Vol. 5977 of LNCS, 2010, pp. 118–131.

[40] M. F. Frias, J. P. Galeotti, C. L. Pombo, N. Aguirre, DynAlloy: Upgrading Alloy with Actions, in: Proceedings ICSE, 2005, pp. 442–451.

[41] M. F. Frias, C. L. Pombo, J. P. Galeotti, N. Aguirre, Efficient Analysis of DynAlloy Specifications, ACM Trans. Softw. Eng. Methodol. 17 (1) (2007) 4:1–4:34.

[42] A. Cunha, Bounded Model Checking of Temporal Formulas with Alloy, in: Proceedings ABZ, Vol. 8477 of LNCS, 2014, pp. 303–308.

[43] A. Requet, BART: A Tool for Automatic Refinement, in: E. Börger, M. Butler, J. P. Bowen, P. Boca (Eds.), Abstract State Machines, B and Z, Springer Berlin Heidelberg, Berlin, Heidelberg, 2008, pp. 345–345.

[44] A. Iliasov, E. Troubitsyna, L. Laibinis, A. Romanovsky, Patterns for Refinement Automation, in: F. S. de Boer, M. M. Bonsangue, S. Hallerstede, M. Leuschel (Eds.), Formal Methods for Components and Objects, Springer Berlin Heidelberg, Berlin, Heidelberg, 2010, pp. 70–88.