Available online at www.sciencedirect.com
SCIENCE DIRECT®
Electronic Notes in Theoretical Computer Science
ELSEVIER Electronic Notes in Theoretical Computer Science 137 (2005) 23–44
www.elsevier.com/locate/entcs

Using the Alloy Analyzer to Verify Data Refinement in Z

Christie Bolton[1,2]

Department of Computer Science
University of Warwick
Coventry CV4 7AL, UK

Abstract

In the development of critical systems, standards dictate that it is necessary to first design, construct and formally analyse abstract models of the system. Developers must then verify that the final implementation is consistent with these more abstract specifications.
Z is an example of a state-based specification language. It has been shown to be effective in a variety of cases—indeed it was developed as part of a joint collaboration between Oxford University's PRG and IBM Hursley for the specification of the CICS system. However, Z's main weakness is that it does not have the necessary tool support: whilst there are associated type checkers, there is no tool for automatically verifying refinement in Z.
The contribution of this paper is to show how data refinement in Z can be automatically verified using the Alloy Analyzer. The soundness and joint completeness of the simulation rules for Z have already been established: here we translate them to Alloy. We then show how data types expressed in Z can also be translated to Alloy, before presenting the assertions necessary for the Alloy Analyzer to identify the retrieve relation and hence verify refinement. We present a simple example in which the Alloy Analyzer successfully identifies the retrieve relation between two data types thereby verifying simulation and hence refinement. We conclude the paper with a discussion of the suitability of the Alloy Analyzer for such a task.

Keywords: Refinement, simulation, automatic verification, Alloy, Z.

[1] This research was funded in part by the University of Warwick and in part by the UK Ministry of Defence through QinetiQ. Many thanks are due to Gavin Lowe, Irfan Zakiuddin and Daniel Jackson for their helpful comments and encouragement.
[2] Email: christie@dcs.warwick.ac.uk

doi:10.1016/j.entcs.2005.04.023

1 Introduction

In the development of critical systems, standards dictate that it is necessary to design, construct and formally analyse abstract models of the system [10]. Developers must then verify that the final implementation is consistent with (or satisfies the properties captured by) these more abstract specifications. Refinement is a technique that is used for verifying such a consistency. The precise notion of refinement and the means for determining whether or not one model is refined by another depends on the choice of specification language and semantic model.

Z [14] is an example of a state-based specification language. It has a fully formal semantics and relies heavily on mathematical constructs such as set theory, logic and relational calculus. In addition, it employs a construct called the *schema* for structuring the mathematics. There are two standard semantic models associated with the Z notation [5]: the *blocking* (behavioural) interpretation, in which operations cannot be called outside their preconditions, and the *non-blocking* (contract) interpretation, in which operations can be called outside their preconditions but no guarantees are made about subsequent behaviour. In this paper we consider also a third, the *stable failures* model, which corresponds to histories refinement in Object-Z [12].

Z has been shown to be effective in a variety of cases—indeed it was developed as part of a collaboration between Oxford University's PRG and IBM Hursley for the specification of the CICS system. However, Z's main weakness is that it lacks the necessary tool support: whilst there are associated type checkers [13,15,9], there is no tool for automatically verifying refinement in Z.

The main contribution of this paper is to show how data refinement in Z can be automatically verified using the Alloy Analyzer [8], a SAT-based verification tool. Moreover, in the process of this verification the Alloy Analyzer identifies the associated retrieve relation: typically the most difficult part of verifying refinement is not the application of the simulation rules but the identification of the correct retrieve relation. A further contribution of this paper is to define a notion of refinement within the Alloy language [8].

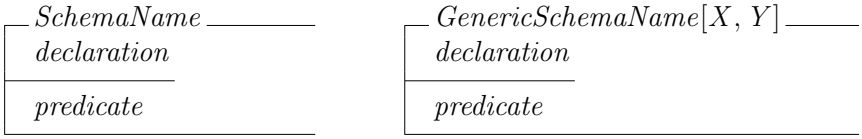
We translate the three sets of simulation rules—previously shown to be both sound and jointly complete with respect to their respective refinement orderings—to Alloy, and show the natural correspondence between data types in Z and in Alloy. We present the Alloy commands necessary for identifying the retrieve relation and verifying refinement, and we discuss the appropriate scope for each type. Finally we use a simple example to illustrate the application of our techniques.

We begin the paper by showing how abstract data types are modelled

in Z. We then identify the refinement orderings with which we will be concerned throughout the rest of the paper, together with their associated sets of simulation rules . In Sections 3 and 4 we show how these data types and simulation rules can be expressed equivalently in Alloy. We conclude Section 4 by defining the assertions necessary for enabling the Alloy Analyzer to identify the retrieve relation between two data types and hence verify simulation and refinement. We illustrate these techniques in Section 5 through a simple example. We conclude the paper with a discussion of the suitability of the Alloy Analyzer for such a task.

2 The Z Notation

The Z notation is a state-based specification language that relies heavily on mathematical constructs such as set theory, logic and relational calculus. In addition, it employs a construct called the *schema* for structuring the mathematics when modelling systems. The schema incorporates a declaration of variables and a predicate constraining those variables.



Generic schemas are used to define the same structure over a variety of types.

2.1 Abstract data types

An abstract data type comprises a notion of state with a collection of named operations on the state space and a non-empty set of possible initial states. For data types with an implicit or explicit notion of communication, separate initialisation and finalisation operations might also be identified.

There are a variety of ways of capturing data types in Z [5]. Since the purpose of this paper is to show how data types expressed in Z might be automatically analysed using the Alloy Analyzer, we adopt the schema representation since this is closest to our Alloy representation. For further clarity, we consider only *simple* data types, those in which operations neither give outputs nor receive inputs.

A simple abstract data type has three components: a set of private internal states *state* of type *State*; a non-empty set *init* of possible initial states taken from *state*; and a function *ops* mapping the name of each operation onto the relation on the state space describing the effect of the operation. It can be defined generically as follows:

$$
\begin{array}{|l}
\hline
ADT[State, Name] \\
state : \mathbb{P}\, State \\
init : \mathbb{P}\, State \\
ops : Name \pfun (State \rel State) \\
\hline
\emptyset \subset init \ \land \ init \subseteq state \ \land \\
ops \ \in \ Name \pfun (state \rel state) \\
\hline
\end{array}
$$

For any name n of an operation on the data type, that is for any $n \in \operatorname{dom} ops$, the relation $ops\, n$ defines the effect of n on the state space.

Note that when modelling an actual abstract data type rather than the specification of a data type as given above, the type *State* may be introduced as a basic type or as a compound type captured by another schema. Furthermore, the operations may also be expressed as schemas.

2.2 Data refinement and simulation

Intuitively we understand that a model is a data refinement [4] of its specification if the behaviour of the concrete model somehow conforms to the behaviour of its more abstract specification: we may replace the abstract specification with the concrete model. The measure of conformity depends on the choice of semantic model.

There are two standard semantic models associated with the Z notation: the *blocking* (behavioural) interpretation in which operations cannot be called outside their preconditions, and the *non-blocking* (contract) interpretation in which operations can be called outside their preconditions but no guarantees are made about subsequent behaviour.

In this paper we will consider also a third semantic model, the *stable failures* semantic model for Z: in [3] it was shown that the simulation rules for Object-Z [12], an object-oriented extension to Z, were unsound with respect to the histories semantic model of Object-Z, and revised rules were proposed in [2]. These revised rules are sound and jointly complete with respect to both histories refinement within Object-Z and stable failures refinement [11] within Communicating Sequential Processes (CSP) [6] and record the availability of *combinations* of operations not just individual operations. Provided this is the appropriate level of granularity and that the developer states clearly that this is the framework within which they are working, there is no reason why the stable failures model should not be adopted for Z.

Within each of these models, we say that data type A is refined by data type C—a second model of the same system with the same set of operations—if the effect of every sequence of operations on C is a possible effect of the same sequence of operations on A. Alternatively, refinement may be verified inductively using sets of *simulation rules* [7] relating the concrete and abstract models: if we can show *operation by operation* that the behaviour of A simulates the behaviour of C, then refinement follows.

Where the data types have different state spaces we must find a *retrieve relation* explaining how the state of one of the data types can be retrieved from the state of the other. If some of the non-determinism of the more abstract of the two data types has been resolved then we look to establish a *forwards* simulation whereas if some of the non-determinism has been postponed then we look to establish a *backwards* simulation.

2.3 Simulation rules

In this section we present the forwards and backwards simulation rules corresponding to the three semantic models under consideration:

- Data refinement with the non-blocking interpretation [16];
- Data refinement with the blocking interpretation [1];
- Stable failures refinement [2].

To be consistent with the histories semantic model of Object-Z, our stable failures model adopts a blocking interpretation. For each of these models, the associated forwards and backwards simulation rules have been shown to be sound and jointly complete with respect to the associated refinement ordering.

Simulation corresponding to data refinement with the non-blocking interpretation

Here we present the forwards and backwards simulation rules corresponding to data refinement within the non-blocking context as defined in [16].

Assuming that the primed components are associated with the more concrete model and that the unprimed components are associated with the more abstract model—a convention that we will adhere to throughout the rest of Section 2—the relational representation of the forwards simulation rules for

verifying data refinement with the non-blocking context are as follows

$$init' \subseteq r(\!| \ init \ |\!) \tag{F_n 1}$$

$$\forall\, n : \mathrm{dom}\ ops \bullet r(\!| \ \mathrm{dom}\,(ops\ n) \ |\!) \subseteq \mathrm{dom}\,(ops'\ n) \tag{F_n 2}$$

$$\forall\, n : \mathrm{dom}\ ops \bullet \mathrm{dom}\,(ops\ n) \lhd r \fatsemi (ops'\ n) \subseteq (ops\ n) \fatsemi r \tag{F_n 3}$$

where $X \lhd R$ denotes relation R domain restricted to set X, $R(\!| X |\!)$ denotes the relational image of X under R and where $R \fatsemi S$ denotes the sequential composition of relations R and S: see [14].

The initialisation rule (F_n 1) states that for every initial concrete state there is a matching initial abstract state. The applicability rule (F_n 2) insists that the concrete operation is defined whenever the abstract operation would be. Finally, the correctness rule (F_n 3) insists that *whenever an abstract operation is applicable*, the corresponding concrete operation produces only concrete states for which an abstract equivalent is reachable via the abstract operation.

If a retrieve relation r such that the above rules hold can be found then the more abstract data type forward simulates—and hence is data refined by—the more concrete data type within the non-blocking context.

Conversely, assuming once more that the primed components are associated with the more concrete data type and that the unprimed components are associated with the more abstract data type, if we can find a retrieve relation s such that the following rules hold then the more abstract data type backward simulates—and hence is refined by—the more concrete data type within the non-blocking context of data refinement.

$$s(\!| \ init' \ |\!) \subseteq init \tag{B_n 1}$$

$$\forall\, n : \mathrm{dom}\ ops \bullet state' \setminus \mathrm{dom}\ ops'\ n \subseteq \mathrm{dom}(s \rhd (\mathrm{dom}\ ops\ n)) \tag{B_n 2}$$

$$\forall\, n : \mathrm{dom}\ ops \bullet \mathrm{dom}(s \rhd (\mathrm{dom}\ ops\ n)) \lhd (ops'\ n) \fatsemi s \subseteq s \fatsemi (ops\ n) \tag{B_n 3}$$

$$state' \subseteq \mathrm{dom}\ s \tag{B_n 4}$$

The symbols $\lhd$ and $\rhd$ respectively denote domain subtraction, and range subtraction: see [14]. Rule (B_n 2), the complement constraint, is a point-free way of expressing the condition that any concrete state for which every corresponding abstract state is in the domain of an abstract operation must lie within the domain of the corresponding concrete operation: equivalently,

$$\forall\, n : \mathrm{dom}\ ops \ \bullet\ \forall\, c : state' \bullet s(\!| \ \{c\} \ |\!) \subseteq \mathrm{dom}(ops\ n) \Rightarrow c \in dom\,(ops'\ n).$$

Simulation corresponding to data refinement within the blocking interpretation

The forwards simulation rules for verifying data refinement within the blocking, or behavioural, context as defined in [1] are very similar to those for verifying refinement in the non-blocking context. Indeed, they differ only in the third rule, the correctness rule. This is strengthened to requiring correctness of the concrete operation from all states not just those corresponding to states for which the abstract operation is defined. This strengthening is captured by the removal of the domain restriction in rule (F_b 3).

$$init' \subseteq r(\!| init |\!) \tag{F_b 1}$$

$$\forall n : \mathrm{dom}\ ops \bullet r(\!| \mathrm{dom}\,(ops\ n) |\!) \subseteq \mathrm{dom}\,(ops'\ n) \tag{F_b 2}$$

$$\forall n : \mathrm{dom}\ ops \bullet r \fatsemi ops'\ n \subseteq ops\ n \fatsemi r \tag{F_b 3}$$

Like the forwards simulation rules, the backwards simulation rules for verifying data refinement within the blocking, or behavioural, context are very similar to those for verifying refinement in the non-blocking context. Again they differ only in the correctness rule which, as with the forwards simulation rules, is strengthened. The correctness condition for backwards simulation within the blocking context concerns all concrete states not only those that correspond only to abstract states that lie within the domain of the operation. This strengthening is captured by the removal of the domain subtraction in rule (B_b 3).

$$s(\!| init' |\!) \subseteq init \tag{B_b 1}$$

$$\forall n : \mathrm{dom}\ ops \bullet state' \setminus \mathrm{dom}\ ops'\ n \subseteq \mathrm{dom}(s \rhd (\mathrm{dom}\ ops\ n)) \tag{B_b 2}$$

$$\forall n : \mathrm{dom}\ ops \bullet (ops'\ n) \fatsemi s \subseteq s \fatsemi (ops\ n) \tag{B_b 3}$$

$$state' \subseteq \mathrm{dom}\ s \tag{B_b 4}$$

Simulation rules for stable failures refinement

The forwards simulation rules for verifying stable failures refinement, as defined in [2], are identical to the corresponding rules for data refinement with the blocking interpretation: that is, rules F_{sf} 1, F_{sf} 2 and F_{sf} 3 are respectively

equivalent to rules $F_b\,1$, $F_b\,2$ and $F_b\,3$.

$$init' \subseteq r(\!|\ init\ |\!) \tag{$F_{sf}\,1$}$$

$$\forall\, n : \mathrm{dom}\ ops \bullet r(\!|\ \mathrm{dom}\,(ops\ n)\ |\!) \subseteq \mathrm{dom}\,(ops'\ n) \tag{$F_{sf}\,2$}$$

$$\forall\, n : \mathrm{dom}\ ops \bullet r \fatsemi ops'\ n \subseteq ops\ n \fatsemi r \tag{$F_{sf}\,3$}$$

The backwards simulation rules concerning initialisation and correctness are also identical to the corresponding rules for data refinement within the blocking context: that is rules $B_{sf}\,1$ and $B_{sf}\,3$ are respectively equivalent to rules $B_b\,1$ and $B_b\,3$:

$$s(\!|\ init'\ |\!) \subseteq init \tag{$B_{sf}\,1$}$$

$$\forall\, X : \mathbb{P}(\mathrm{dom}\ ops) \bullet state' \setminus \bigcup\{n : X \bullet \mathrm{dom}\ ops'\ n\} \subseteq \mathrm{dom}(s \rhd \bigcup\{n : X \bullet \mathrm{dom}\ ops\ n\}) \tag{$B_{sf}\,2$}$$

$$\forall\, n : \mathrm{dom}\ ops \bullet (ops'\ n) \fatsemi s \subseteq s \fatsemi (ops\ n) \tag{$B_{sf}\,3$}$$

$$state' \subseteq \mathrm{dom}\ s \tag{$B_{sf}\,4$}$$

where the set $\bigcup\{n : X \bullet \mathrm{dom}\ ops\ n\}$ is the union of all the states in the domain of $ops\ n$ for each n in X, and similarly for $\bigcup\{n : X \bullet \mathrm{dom}\ ops'\ n\}$. The difference lies in applicability: whilst data refinement requires that if a concrete state lies outside the domain of an operation then there must be a corresponding abstract state that lies outside the domain of that operation, stable failures refinement requires the stricter condition that each concrete state must correspond to a *single* abstract state that lies outside the domains of all the operations that the concrete state lies outside the domain of.

3 Capturing data types in Alloy

Alloy [8] is a structural modelling language. It has many of the features of Z [14]; however, unlike Z, it is based on *first order* logic. Whilst this can restrict expressibility, it facilitates automatic analysis which can be performed by the associated constraint solver, the Alloy Analyzer.

Atoms and types may be introduced in Alloy using the keyword `sig`. We introduce below the types `AState`, `CState` and `Op` that respectively model the state spaces of the more abstract and more concrete data types and the set of names of all operations.

```
sig AState {}
```

```
sig CState {}
sig Op {}
```

None of these types have any attributes, although, as we will demonstrate later, we can extend them and define subtypes that do have attributes.

Since our more abstract and more concrete representations of data types may have different state spaces, it is convenient to introduce them as separate types. We introduce first the more abstract data type.

```
sig DataTypeA {
  state : set AState,           // a set of abstract states
  init : set state,             // a subset of state
  names : set Op,               // the names of the operations
  trans : names ->+ state -> state
                  // the relation on state for each operation
} {
  some init                     // init is a non-empty set
}
```

We see the close correspondence between this definition and the generic definition of a data type in Z in Section 2.1. One interesting point to note is that, unlike Z, the declarations can be self-referential: rather than declaring that `init` is of type "`set AState`" and subsequently, in the constraints, stating that it is a subset of state "`init in state`" we can simply include this in the declarations "`init : set state`".

For later convenience we introduce the attribute "`names`" not included explicitly in the Z description, the set of the names of all operations on the data type. This corresponds to "dom *ops*" in our Z description. The attribute "`trans`" is a ternary relation. More specifically it is a total relation (`->+`) from the set `names` to a relation [3] on the state space `(state -> state)`. The declaration "`some init`" states that the set `init` is non-empty.

The declaration for the more concrete data type, `DataTypeC`, is identical except that the state space contains elements from the concrete state space not the abstract state space: "`state : CState`".

```
sig DataTypeC {
  state : set CState,           // a set of concrete states
  init : set state,
  names : set Op,
  trans : names ->+ state -> state
} {
```

[3] Note that whilst → denotes a function in Z, `->` denotes a relation in Alloy.

```
    some init
}
```

Finally, we introduce pairs: these comprise an abstract data type, its corresponding concrete data type and a retrieve relation relating their state spaces, insisting that the same set of operations are defined over both data types.

Since we use the type `FwdsPair` when looking to establish a forward simulation, our retrieve relation (`retr : AState -> CState`) maps abstract states onto concrete states. The type `BkwdsPair` is identical to `FwdsPair` except that the retrieve relation maps concrete states onto abstract states.

```
sig FwdsPair {
  abstract :  DataTypeA,
  concrete :  DataTypeC,
  retr :  AState -> CState
} {
  abstract.names
     = concrete.names
}
```

```
sig BkwdsPair {
  abstract :  DataTypeA,
  concrete :  DataTypeC,
  retr :  CState -> AState
} {
  abstract.names
     = concrete.names
}
```

4 Using the Alloy Analyzer to verifying refinement and simulation

The Alloy Analyzer is a SAT-based verification tool that is used to determine automatically whether a model exists for a specified system given set bounds on the domains of each basic type within the model. The tool translates the system description to a SAT problem and an underlying SAT-solver checks whether these constraints can be satisfied.

As observed above, sets of simulation rules can be used to verify refinement, since one data type is refined by another precisely when the first simulates the second. In this section we define Alloy versions of the simulation rules before presenting the Alloy check for automatically verifying refinement. See the Appendix and [8] for necessary notation.

4.1 Functions capturing the simulation rules

Here we introduce Alloy versions of the forwards and backwards simulation rules corresponding to traditional data refinement within the non-blocking context, traditional data refinement within the blocking context and stable failures refinement within the blocking context as discussed in Section 2.3.

Forwards simulation corresponding to data refinement within the non-blocking interpretation

As observed in Section 2.3, some of the simulation rules for data refinement within the non-blocking context are equivalent to those for stable failures refinement or data refinement within the blocking context. Therefore, to permit re-use, we consider each rule separately.

First, recalling that `pair.abstract`, `pair.concrete` and `pair.retr` respectively denote the concrete model, the abstract model and the retrieve relation recorded by element `pair` of type `FwdsPair`, we translate Rule $F_n\,1$ concerning initialisation. This states that `pair.concrete.init`, the concrete initialisation, is contained in the relational image under the retrieve relation of the abstract initialisation, that is `(pair.abstract.init).(pair.retr)`.

```
fun RuleFn1 (pair : FwdsPair) {
  pair.concrete.init in (pair.abstract.init).(pair.retr)
}
```

Next we consider rule $F_n\,2$ concerning applicability. For brevity and clarity we use the Alloy variant of the "let ... within" clause to introduce abbreviations. Rule $F_n\,2$ states that for all operations on the data types, the domain of the concrete operation, or `(C.trans[n]).CState`, must contain the relational image under the retrieve relation of the domain of the abstract operation, or equivalently `((A.trans[n]).AState).R`.

```
fun RuleFn2 (pair : FwdsPair) {
  let A = pair.abstract, C = pair.concrete, R = pair.retr {
    all n : A.names |
      ((A.trans[n]).AState).R in (C.trans[n]).CState
  }
}
```

Finally we consider Rule $F_n\,3$ concerning correctness, once more simplifying matters by using the Alloy variant of the 'let ... within" clause.

```
fun RuleFn3 (pair : FwdsPair) {
  let A = pair.abstract, C = pair.concrete, R = pair.retr {
    all n : A.names {
      all a : (A.trans[n]).AState |
        (R.(C.trans[n]))[a] in ((A.trans[n]).R)[a]
    }
  }
}
```

This verifies correctness for each operation `n`. In particular it verifies that

for every abstract state `a` in the domain of abstract operation `n`, that is for all abstract states in the set `(A.trans[n]).AState`, the relational image of `a` under the sequential composition of the retrieve relation and concrete operation `n`, that is `(R.(C.trans[n]))[a]`, is a subset of the relational image of `a` under the sequential composition of the abstract operation `n` and the retrieve relation, equivalently `((A.trans[n]).R)[a]`.

Given these three definitions we can now introduce a function that defines forwards simulation corresponding to data refinement within the non-blocking context.

```
fun FwdsDataNonBlocking (pair : FwdsPair) {
  RuleFn1(pair) && RuleFn2(pair) && RuleFn3(pair)
}
```

Given argument `pair` of type `FwdsPair`, this function returns the value `true` precisely when Rules $F_n\,1$, $F_n\,2$ and $F_n\,3$ all hold, or equivalently when the more abstract model, `pair.abstract`, forwards simulates within the non-blocking context the less abstract model, `pair.concrete`.

Backwards simulation corresponding to data refinement within the non-blocking interpretation

As with forwards simulation, to permit re-use, we consider each rule separately. We consider first Rule $B_n\,1$ concerning initialisation. This states that the abstract initialisation `pair.abstract.init` contains the relational image under the retrieve relation of the concrete initialisation, or equivalently `(pair.concrete.init).(pair.retr)`.

```
fun RuleBn1 (pair : BkwdsPair) {
  (pair.concrete.init).(pair.retr) in pair.abstract.init
}
```

Next we translate $B_n\,2$, the somewhat complicated backwards simulation rule concerning applicability.

```
fun RuleBn2 (pair : BkwdsPair) {
  let A = pair.abstract, C = pair.concrete, S = pair.retr {
    all n : A.names |
      (C.state - (C.trans[n]).CState)
         in S.(AState - (A.trans[n]).AState)
  }
}
```

Observe first that for given operation `n`, set `AState - (A.trans[n]).AState` contains all abstract states outside the domain of operation `n`, and similarly

for `C.state - (C.trans[n]).CState`. This rule then states that for each operation `n`, all concrete states outside the domain of operation `n` correspond to an abstract state outside the domain of `n`.

Next we consider Rule $B_n\,3$, the backwards simulation rule concerning correctness and Rule $B_n\,4$ concerning applicability of the retrieve relation.

```
fun RuleBn3 (pair : BkwdsPair) {
  let A = pair.abstract, C = pair.concrete, S = pair.retr {
    all n : A.names {
      all c : CState - S.(AState - (A.trans[n]).AState) |
        ((C.trans[n]).S)[c] in (S.(A.trans[n]))[c]
    }
  }
}

fun RuleBn4 ( pair : BkwdsPair) {
  (pair.retr).AState = pair.concrete.state
}
```

Rule *Bn*3 states that for each operation `n` and for each concrete state `c` that corresponds to an abstract state outside the domain of `n` , that is each concrete state `c` in the set `S.(AState - (A.trans[n]).AState)`, any abstract state that lies in the relation image of `c` under the sequential composition of the concrete operation `n` and then the retrieve relation—that is any abstract state in the set `((C.trans[n]).S)[c]`—must also lie in the relation image of `c` under the sequential composition of the retrieve relation and the abstract operation, or equivalently the set `(S.(A.trans[n]))[c]`.

*RuleBn*4 is slightly stronger than necessary insisting that the domain of the retrieve relation contains precisely thoses states in the concrete domain rather than at least those states. The additional strength removes unnecessary non-determinism from the choice of retrieve relation.

Given these four definitions we can now introduce a function that defines forwards simulation of data refinement within the non-blocking context.

```
fun BkwdsDataNonBlocking (pair : BkwdsPair) {
  RuleBn1(pair) && RuleBn2(pair) &&
  RuleBn3(pair) && RuleBn4(pair)
}
```

Given argument `pair` of type `BkwdsPair`, this function will return the value `true` precisely when Rules $B_n\,1$, $B_n\,2$, $B_n\,3$ and $B_n\,4$ all hold, or equivalently when `pair.abstract`, the more abstract model, backwards simulates within the non-blocking context `pair.concrete`, the less abstract model.

Forwards simulation corresponding to traditional data refinement within the blocking interpretation

As we have already observed, the forwards simulation rules concerning initialisation and applicability corresponding to traditional data refinement within the blocking interpretation, that is Rules $F_b\,1$ and $F_b\,2$, are equivalent to the corresponding forwards simulation rules within the non-blocking interpretation. We can therefore re-use the above definitions.

We now consider Rule $F_b\,3$ concerning correctness. Once more we use the Alloy variant of the "let ... within" for brevity and clarity.

```
fun RuleFb3 (pair : FwdsPair) {
  let A = pair.abstract, C = pair.concrete, R = pair.retr {
    all n : A.names | R.(C.trans[n]) in (A.trans[n]).R
  }
}
```

This states that for any operation `n`, the relation corresponding to the sequential composition of the retrieve relation and the concrete variant of the operation, that is `R.(C.trans[n])` must be a subset of `(A.trans[n]).R`, the relation corresponding to the sequential composition of the abstract variant of the operation and the retrieve relation. Equivalently, for all operations `n` and for all abstract states `c`, if `a` is mapped onto `c` by applying the retrieve relation and then the concrete operation `n`, then it must also be mapped onto `c` by applying the abstract operation `n` and then the retrieve relation.

Recalling that Rules $F_b\,1$ and $F_b\,2$ are respectively equivalent to Rules $F_n\,1$ and $F_n\,2$, we define as follows the function that holds for given argument `pair` of type `FwdsPair` precisely when `pair.abstract`, the more abstract model, forwards simulates `pair.concrete`, the less abstract model, within the blocking context.

```
fun FwdsDataBlocking (pair : FwdsPair) {
  RuleFn1(pair) && RuleFn2(pair) && RuleFb3(pair)
}
```

Backwards simulation corresponding to traditional data refinement within the blocking interpretation

Once more only the rule concerning correctness differs from its corresponding rule within the non-blocking context. Rule $B_b\,3$ can be captured in Alloy as follows.

```
fun RuleBb3 (pair : BkwdsPair) {
  let A = pair.abstract, C = pair.concrete, S = pair.retr {
   all n : A.names | (C.trans[n]).S in S.(A.trans[n])
```

```
  }
}
```

This states that for any operation `n`, the relation corresponding to the sequential composition of the concrete variant of the operation and the retrieve relation, that is `(C.trans[n]).S`, must be a subset of `S.(A.trans[n])`, the relation corresponding to the sequential composition of the retrieve relation and the abstract variant of the operation.

Finally we introduce the function that defines backwards simulation of data refinement within the blocking context.

```
fun BkwdsDataBlocking (pair : BkwdsPair) {
  RuleBn1(pair) && RuleBn2(pair) &&
  RuleBb3(pair) && RuleBn4(pair)
}
```

Recalling that Rules $B_b\,1$, $B_b\,2$ and $B_b\,4$ are respectively equivalent to Rules $B_n\,1$, $B_n\,2$ and $B_n\,4$ this function returns `true` when the backwards simulation rules corresponding to data refinement within the blocking context all hold.

Simulation corresponding to stable failures refinement

Since the forwards simulation rules for stable failures refinement are identical to the corresponding rules for data refinement within the blocking context, the function that captures forwards simulation corresponding to stable failures refinement is as follows.

```
fun FwdsStableFailures (pair : FwdsPair) {
  RuleFn1(pair) && RuleFn2(pair) && RuleFb3(pair)
}
```

It is identical to the corresponding function for capturing forwards simulation corresponding to data refinement within the blocking context.

The backwards simulation rules corresponding to stable failures refinement differ from their counter-parts corresponding to data refinement within the blocking context only in the rule concerning applicability. Unlike Rule $B_b\,2$, Rule $B_{sf}\,2$ records the availability of *combinations* of operations and can be expressed in Alloy as follows.

```
fun RuleBsf2 (pair : BkwdsPair) {
  let A = pair.abstract, C = pair.concrete, S = pair.retr {
    all X : set (A.names) |
      (C.state - (C.trans[X]).CState)
         in S.(AState - (A.trans[X]).AState)
  }
```

```
}
```

Where Rule $B_b\,2$ considered individual operations `n`, this rule is applied to sets of operations `X`.

Finally, recalling that Rules $B_{sf}\,1$ and $B_{sf}\,4$ are equivalent to Rules $B_n\,1$ and $B_n\,4$, and that Rule $B_{sf}\,3$ is equivalent to Rule $B_b\,3$, we can introduce the function that defines backwards simulation corresponding to stable failures refinement.

```
fun BkwdsStableFailures (pair : BkwdsPair) {
  RuleBn1(pair) && RuleBsf2(pair) &&
  RuleBb3(pair) && RuleBn4(pair)
}
```

Given argument `pair` of type `BkwdsPair`, this function returns `true` precisely when Rules $B_{sf}\,1$, $B_{sf}\,2$, $B_{sf}\,3$ and $B_{sf}\,4$ all hold, or equivalently when `pair.abstract`, the more abstract model, backwards simulates `pair.concrete`, the less abstract model.

4.2 Automatic analysis

In this section we identify the assertions necessary for enabling the Alloy Analyzer to identify the retrieve relation between the state spaces of a pair of data types. The actual data types under consideration must be uniquely captured. As illustrated in Section 5, in order to do this, the developer will need to extend the types `Op`, `AState` and `CState` to respectively include the names of all operations on the data types, all the states in the abstract state space, and all the states in the concrete state space.

For a forwards simulation we use the type `SpecificFwdsPair` to uniquely capture the concrete and abstract models under consideration. For a backwards simulation we use the type `SpecificBkwdsPair`. [4]

```
sig SpecificFwdsPair {
  pair : FwdsPair
} {
// Predicates uniquely defining pair.abstract and pair.concrete.
}

sig SpecificBkwdsPair {
  pair : BkwdsPair
```

[4] The reason for using `SpecificFwdsPair` and `SpecificBkwdsPair` rather than extending the types `FwdsPair` and `BkwdsPair` is so that we can fix the scope of each. When the Alloy Analyzer allows the user to specify the number of each subtype to be considered this will become unnecessary.

```
} {
// Predicates uniquely defining pair.abstract and pair.concrete.
}
```

The functions for identifying forwards and backwards retrieve relations corresponding to data refinement within the blocking context are then as follows.

```
fun ShowRetrForFwdsDataBlocking (s : SpecificFwdsPair) {
  FwdsDataBlocking (s.pair)
}

fun ShowRetrForBkwdsDataBlocking (s : SpecificBkwdsPair) {
  BkwdsDataBlocking (s.pair)
}
```

The functions for verifying forwards and backwards simulation corresponding to stable failures refinement and to data refinement within the non-blocking context are analogous.

Finally, in order to execute these functions, we must include a `run` command and set the appropriate scope, or number of each type to be considered. The scope should be as small as possible whilst guaranteeing exploration of the entire system. A sensible strategy is to restrict the definitions of `AState` and `CState` so that they are respectively equivalent to the abstract and concrete state spaces of the data types under consideration.

When considering forwards and backwards simulations corresponding to data refinement within the blocking context the following commands and scopes should respectively be used

```
run ShowRetrForFwdsDataBlocking for 1
  but 0 BkwdsPair, 0 SpecificBkwdsPair, x Op, y AState, z CState

run ShowRetrForBkwdsDataBlocking for 1
  but 0 FwdsPair, 0 SpecificFwdsPair, x Op, y AState, z CState
```

where `x` is the number of operations on the data types, `y` is the size of the abstract state space and `z` is the size of the concrete state space. This means that for forwards simulation one `SpecificFwdsPair`, one `FwdsPair`, one `DataTypeA` and one `DataTypeC` would be considered. We can consider individual pairs but we must always consider the entire state space of each data type.

The commands and scopes for identifying the retrieve relations corresponding to stable failures refinement and data refinement within the non-blocking context are analogous.

When the appropriate function is executed, if a pair is found then the retrieve relation has been found and simulation and hence refinement have been verified.

5 Example

In this section we demonstrate the techniques discussed in the previous sections. We present a pair of simple data types in Z. We translate them to Alloy and use the Alloy Analyzer to automatically identify the retrieve relation relating their state space and hence to verify refinement. We adopt the semantic model corresponding to data refinement within the blocking context.

5.1 Z description

Let data types A and C be defined as follows. They each have two operations Op_1 and Op_2. Initially data type A offers a non-deterministic choice between these operations and data type C nondeterministically will either deadlock or offer a non-deterministic choice between Op_1 and Op_2. Both data types will deadlock after any operation occurs.

Given the following definitions,

$$N ::= Op_1 \mid Op_2 \qquad State_A ::= a_1 \mid a_2 \mid a_3 \qquad State_C ::= c_1 \mid c_2 \mid c_3 \mid c_4$$

data types A and C may be captured in Z in the following way (see [14]).

$$A \mathrel{\widehat{=}} [ADT[State_A, N] \mid state = \{a_1, a_2, a_3\} \wedge init = \{a_1, a_2\} \wedge ops = \{Op_1 \mapsto \{(a_1, a_3)\}, Op_2 \mapsto \{(a_2, a_3)\}\}]$$

$$C \mathrel{\widehat{=}} [ADT[State_C, N] \mid state = \{c_1, c_2, c_3, c_4\} \wedge init = \{c_1, c_2, c_4\} \wedge ops = \{Op_1 \mapsto \{(c_1, c_3)\}, Op_2 \mapsto \{(c_2, c_3)\}\}]$$

We see that data type A may initially non-deterministically either be in state a_1 in which case operation Op_1 is available, or state a_2 in which case operation Op_2 is available. If either operations occurs the data type will end up in state a_3 and no operation will be available.

Similarly data type C may initially non-deterministically either be in state c_1 in which case operation Op_1 is available, or state c_2 in which case operation Op_2 is available, or in state c_4 in which case neither operation is available. If either operations occurs the data type will end up in state c_3 and no operation will be available.

5.2 Alloy description

First we extend the type `Op`.

```
static part sig Op1, Op2 extends Op {}
```

This declares `Op1` and `Op2` to be elements of type `Op`. The keywords `static` and `part` state that there is only one element of each subtype and that together they partition `Op`

Similarly, we extend types `AState` and `CState`.

```
static part sig a1, a2, a3 extends AState {}
static part sig c1, c2, c3, c4 extends CState {}
```

Note that for more complex data types we could include attributes within these subtypes.

We are going to look for a backwards simulation, so we consider the type `SpecificBkwdsPair`. It uniquely captures data types A and C.

```
sig SpecificBkwdsPair {
  pair : BkwdsPair
} {
    let A = pair.abstract, C = pair.concrete {
      A.state = AState && C.state = CState
      A.init = a1 + a2 && C.init = c1 + c2 + c4
      A.names = Op            // Recall that A.names = C.names
      A.trans = (Op1 -> a1 -> a3) + (Op2 -> a2 -> a3)
      C.trans = (Op1 -> c1 -> c3) + (Op2 -> c2 -> c3)
    }
}
```

We see that this corresponds precisely to the Z description of these data types.

Running the following check

```
run ShowRetrForBkwdsDataBlocking for 1
  but 0 FwdsPair, 0 SpecificFwdsPair, 2 Op, 3 AState, 4 CState
```

the Alloy Analyzer immediately identifies a pair with the following backwards retrieve relation:

$$\{ (c_1, a_1), (c_1, a_2), (c_2, a_2), (c_3, a_3), (c_4, a_1), (c_4, a_2) \}.$$

This is indeed a correct retrieve relation. We have demonstrated, albeit in this simple case, how the Alloy Analyzer can identify a retrieve relation and hence verify that data type A simulates, and thus is data refined by, data type C within the blocking context.

6 Discussion

In this paper we have shown how the Alloy Analyzer can be used to identify retrieve relations and hence to verify simulation and refinement in Z. We have worked within the context of three semantic models: the first two traditionally associated with Z and the third corresponding to the histories semantic model for Object-Z.

Suitability of the Alloy Analyzer

The Alloy Analyzer was the natural choice of tool for automating the verification of refinement in Z because of the close relationship between the two languages, as evinced by the ease of translation of both data types and simulation rules from Z to Alloy.

Although state-space explosion can be a potential problem for model-checkers, the fact that we need consider only one concrete and one abstract model at a time indicates that the techniques presented here may equally be applied to large systems. Indeed, the author is currently working on an industrial-scale case study, with complex data structures as well as inputs and outputs to operations, and preliminary results are promising.

A novel application of the Alloy Analyzer

The Alloy Analyzer, like most model-checkers, is typically used to obtain a negative result such as identifying a counter-example. Thus our use of the tool to obtain a positive result—the verification of refinement—through identification of a retrieve relation is of particular interest.

A notion of refinement for Alloy

Although the Alloy Language has been strongly influenced by Z, and as such has the capability of modelling data types, it has no associated notion of refinement. This work fills that gap.

Summary

This is an important contribution for the following reasons:

- Z is a powerful modelling language, but one of its main drawbacks is lack of tool support. Here we address that issue.
- Typically, one of the hardest steps in the verification of refinement in Z is identifying potential candidates for the retrieve relation: our techniques mean that this step is no longer necessary.

- Automatic verification of correctness is often perceived to be more credible than hand-proofs: here we provide the sought after techniques for automatic verification of refinement within Z.
- Developers familiar with Alloy but not Z could omit the steps involving Z and define their data types directly in Alloy, before performing the verification of data refinement.

References

[1] C. Bolton. *On the Refinement of State-Based and Event-Based Models*. D.Phil., University of Oxford, 2002.

[2] C. Bolton and J. Davies. A Comparison of Refinement Orderings and their Associated Simulation Rules. In J. Derrick, E. Boiten, J. Woodcock, and J. von Wright, editors, *Proceedings of REFINE'02: The BCS Refinement Workshop*. Elsevier Science Publishers, 2002.

[3] C. Bolton and J. Davies. Refinement in Object-Z and CSP. In M. Butler, L. Petre, and K. Sere, editors, *Proceedings of Integrated Formal Methods (IFM '02)*, 2002.

[4] W.-P. de Roever and K. Engelhardt. *Data Refinement: Model-oriented Proof Methods and their Comparison*. Cambridge Tracts in Theoretical Computer Science, 1998.

[5] J. Derrick and E. Boiten. *Refinement in Z and Object-Z*. Springer, 2001.

[6] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[7] C. A. R. Hoare, J. He, and J. W. Sanders. Prespecification in Data Refinement. *Information Processing Letters*, 1987.

[8] D. Jackson, I. Shlyakhter, and M. Sridharan. A Micromodularity Mechanism. In *Proceedings of the ACM SIGSOFT Conference on Foundations of Software Engineering/European Software Engineering Conference (FSE/ESEC '01)*, 2001.

[9] I. Meisels and M. Saaltink. *The Z/EVES Reference Manual*, 1997.

[10] UK Ministry of Defence. Requirements for the Procurement of Safety Critical Software in Defence Equipment (00-55 / Issue 2), 1997.

[11] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall Series in Computer Science, 1998.

[12] G. Smith. *The Object-Z Specification Language*. Kluwer Academic Publishers, 2000.

[13] J. M. Spivey. *The fuzz Manual*, 1988.

[14] J. M. Spivey. *The Z Notation: a Reference Manual*. Prentice-Hall International, 1992.

[15] I. Toyn. *CADiZ web pages*, 2002. http://www-users.cs.york.ac.uk/~ian/cadiz/.

[16] J. C. P. Woodcock and J. Davies. *Using Z: Specification, Proof and Refinement*. Prentice Hall International Series in Computer Science, 1996.

A Alloy notation

In this section we present the Alloy notation necessary for Sections 3, 4 and 5. For further details see [8].

Relations play a fundamental role in the Alloy language, and the `.` operator on relations occurs in many guises.

- *Using `.` to obtain the value of an attribute*: Given an element `pair` of type `BkwdsPair`, the set `pair.concrete.init` contains the possible initial states of the more concrete model captured by `pair`.
- *Using `.` to obtain the relational image*: Given an element `pair` of type `BkwdsPair`, the set `(pair.abstract.init).(pair.retr)` is the relational image of the abstract initialisation `pair.abstract.init` under the retrieve relation `pair.retr`. [5]
- *Using `.` to obtain the domain or range of an operation*: Given an element `pair` of type `BkwdsPair`, the sets `(pair.concrete.trans[n]).CState` and `CState.(pair.concrete.trans[n])` are respectively the domain and the range of the concrete operation `n`.
- *Using `.` to denote sequential composition*: Given an element `pair` of type `BkwdsPair`, the relation `(pair.abstract.trans[n]).(pair.retr)` is the sequential composition of the abstract operation `n` and the retrieve relation.

The *union* of sets or individual atoms [6] is captured by the `+` operator: thus the set `a1 + a2` contains the elements `a1` and `a2`, whilst the tuples `(Op1,a1,a3)` and `(Op2,a2,a3)` are contained in the set

```
Op1 -> a1 -> a3 + Op2 -> a2 -> a3.
```

Similarly, *set difference* is represented by the `-` operator: given an element `pair` of type `BkwdsPair`, the set containing those elements of the concrete state space that lie outside the domain of concrete operation `n` is described as follows `pair.concrete.state - (pair.concrete.trans[n]).CState`.

Logical conjunction is represented by the `&&` operator and set containment is represented by the operator `in`.

[5] Note that relational image can also be expressed using square brackets. For instance, given `pair` of type `BkwdsPair`, the relation `pair.abstract.trans[n]` describes the abstract operation `n`.

[6] Recall that the Alloy language does not distinguish between sets and single elements