Issues in implementing a model checker for Z

John Derrick, Siobhán North and Tony Simons

Department of Computing, University of Sheffield, Sheffield, S1 4DP, UK
J.Derrick@dcs.shef.ac.uk

Abstract. In this paper we discuss some issues in implementing a model checker for the Z specification language. In particular, the language design of Z and its semantics, raises some challenges for efficient model checking, and we discuss some of these issues here. Our approach to model checking Z specifications involves implementing a translation from Z into the SAL input language, upon which the SAL toolset can be applied. In this paper we discuss issues in the implementation of this translation algorithm and illustrate them by looking at how the mathematical toolkit is encoded in SAL and the resultant efficiency of the model checking tools.
Keywords: Z, model-checking, SAL.

1 Introduction

Computing is a tool-based activity, and this applies to design and specification as much as to programming. Furthermore, all design and development methods which have ultimately gained acceptance have been supported, if not based on, toolsets for integral parts of their activity, and this is true for both formal as well as informal methods.

For example, it is inconceivable to imagine UML without tool support, and similarly the use of, say, B or SDL depend on their associated toolsets. Indeed, many notations have been designed with tool support in mind, resulting in efficient type-checkers, simulators, proof assistants etc. for these languages.

However, this is not the case for the specification language Z [14, 1], where the notation and needs of abstraction have been the driver behind the language and its development rather than tool support. The language itself has been very successful and the challenge now is to develop usable tool support for it. The CZT (Community Z Tools) project (see http://czt.sourceforge.net/ or [10]) aims to tackle some of these issues, and is building a range of tools around a common exchange format. In this paper we discuss some issues in implementing a model checker for Z which is being developed by the Universities of Queensland, Australia, and Sheffield, England. In particular, the language design of Z and its

semantics raises some challenges for efficient model checking and we illustrate some of these issues here.

Model checking [4], which aims to determine whether a specified system satisfies a given property, works by exhaustively checking the state space of a specification to determine whether or not the property holds. Model checkers will also provide a counter-example when the property does not hold, thus giving some insight into the failure of the property on the current specification. There has been a considerable amount of success in applying model checking to real large-scale systems, and the technique is now applied routinely in some industrial sectors.

Originally model checking technology was only feasible for small, finite state spaces, and this meant that their application was restricted to notations suited to modelling systems where the complexity lay in the control structure, rather than the data, e.g., hardware systems and communication protocols. However, these factors have become less of an issue due to the maturity of the technology. For example, it is now feasible to model check systems involving very large state-spaces (e.g., systems with 10^{20} states), and the restriction to specification notations with control rather than data has now gone. In addition, automatic techniques for property-preserving abstraction [7, 12, 3] and bounded model checking allow systems with infinite state spaces to be checked. Furthermore, powerful automatic decision procedures allow model-checker languages to support high-level specification constructs such as lambda expressions, set comprehensions and universal and existential quantifiers [5].

Instead of implementing a model-checker from scratch we have been investigating using an intermediate format into which we translate a Z specification. In particular, we have been using the SAL input format as our intermediate format. SAL [5] is a tool-suite for the analysis and verification of systems specified as state-transition systems. Its aim is to allow different verification tools to be combined, all working on an input language designed as a format into which programming and specification languages can be translated. The input language provides a range of features to support this aim, such as guarded commands, modules, definitions etc., and can, in fact, be used as a specification language in its own right. The tool-suite currently comprises a simulator and four model checkers including those for LTL and CTL.

The basis of the translation algorithm of Z into SAL was defined by Smith and Wildman in [13], and in this paper we discuss the implementation of these ideas. The advantage of using SAL is that many aspects of Z, such as structuring via schemas, use of primes for after-state etc., can have a similar representation in SAL. The focus of the translation thus comes down to how the mathematical toolkit (i.e., sets, relations, sequences etc) is encoded. We illustrate this point by discussing the approach, and problems, of representing sets in SAL.

The structure of the paper is as follows. In Section 2 we introduce our running example. The basic approach to translation is discussed in Section 3 which briefly

explains how a Z specification is translated into a SAL module. Subsequent sections discuss the implementation of types (Section 4) and axiomatic definitions (Section 5) before we focus on the issues surrounding the implementations of sets in Section 6. Section 7 discusses the use of the tool, and Sections 8 and 9 provide a discussion and some conclusions respectively.

2 Example

A Z specification defines a number of components, including types, constants, abbreviations and schemas. The schemas define the state space of the system under consideration, its initial configuration and the operations which define the transitions of the system.

For example, the following defines the process of joining an organisation which has a set of *members* and a set of people *waiting* to join.

$$[NAME]$$

$$Report ::= yes \mid no$$

$$total : \mathbb{N}_1$$
$$total = 4096$$

$$capacity : \mathbb{N}_1$$
$$1 < capacity \leq 4096$$

$$\#NAME > capacity$$

State
$$member, waiting : \mathbb{P}\, NAME$$
$$member \cap waiting = \varnothing$$
$$\#member \leq 4096$$
$$\#waiting \leq total$$

Init
$$State'$$
$$member' = \varnothing$$
$$waiting' = \varnothing$$

Join

$\Delta State$
$n? : NAME$

$n? \in waiting \land \#waiting < capacity$
$member' = member \cup \{n?\}$
$waiting' = waiting \setminus \{n?\}$

JoinQ

$\Delta State$
$n? : NAME$

$n? \notin (waiting \cup member)$
$\#waiting < total$
$waiting' = waiting \cup \{n?\}$
$member' = member$

Remove

$\Delta State$
$n? : NAME$

$n? \in member$
$waiting' = waiting$
$member' = member \setminus \{n?\}$

Query

$\Xi State$
$n? : NAME$
$ans! : Report$

$n? \in member \Rightarrow ans! = yes$
$n? \notin member \Rightarrow ans! = no$

3 Basic translation

Our approach to model checking Z involves implementing a translation from Z into the SAL input language upon which the SAL toolset can be applied.

The translation scheme is based upon that presented in [13], whose aim was to preserve the Z-style of specification including predicates where primed and unprimed variables are mixed, and the approach of the Z mathematical toolkit to the modelling of relations, functions etc., as sets of tuples. No claim is made

about how optimised the translation is; the aim of [13] was simply to show how much of the Z style could be preserved within SAL. Here we consider some of the implementation issues.

A Z specification is translated to a SAL module, which groups together a number of definitions which include types, constants and modules for describing the state transition system. In general a SAL module will have the following form (where elided parts are written ...):

```
State : MODULE =
  BEGIN
   INPUT ...
   LOCAL ...
   OUTPUT ...
   INITIALIZATION [ ... ]
   TRANSITION [
                 ....
              ]
  END
```

3.1 State and initialisation schemas

As seen in the example above, in a states plus operations style schemas form one of the basic building blocks of a Z specification. Different schemas take on different roles, and the translation into SAL needs to take account of the intended role of each schema in the specification. In our current implementation, we assume that there is a single state and initialisation schema, and that the first schema in the Z input is the state schema, and that the second is the initialisation schema. All other schemas are taken as operation schemas.

In general, schema references are allowed within a schema, and are used in initialisation and operation schemas, e.g., $\Xi State$ in the operation *Query* above. However, the schema references can be expanded, and then there is no need for a predicate in the state schema, since it will be included in the initial and operation schemas, and for the latter in both primed and unprimed form. Our tool does this expansion of schema references in the declaration part of any schema automatically. Once this has been done, the (single) state schema contains only a list of declarations. These declarations will be translated to local variables of the SAL module.

In translating the initialisation schema we note that all schema references to the state will have been expanded out, and the translation of the state schema will have produced local variables. Currently we do not allow new declarations (e.g., of inputs) in the initialisation schema, it thus remains to translate the predicate of the initialisation which becomes a guard of the initialisation section of the

module. The guard is followed by a list of assignments, one for each declaration in the state schema. We allow both styles of specification where the initialisation can contain either primed or unprimed components in the Z specification, but are unprimed in the resultant SAL output. In the translation these assignments allow any value of the appropriate type to be assigned.

Thus the state and initialisation schemas in our example above produce the following SAL fragment.

```
State : MODULE =
  BEGIN
    INPUT ...
    LOCAL member : set{NAME;}!Set
    LOCAL waiting : set{NAME;}!Set
    OUTPUT ...
    INITIALIZATION [
        member = set{NAME;}!empty_set  AND
        waiting = set{NAME;}!empty_set
      -->
    ]
```

3.2 Operation schemas

For the operation schemas, all schema references to the state will have been expanded out. The translation of the state schema will have produced local variables as above. It thus remains to translate the predicate (which will include the predicate of the expanded state schema reference) and the input and output of the schema.

Variables of the state schema (e.g., *member*, *waiting* but not the primed versions) have become local variables of the module. Inputs and outputs of the operations are translated to input and output variables of the module, respectively. Output variables need to be renamed since ! is not allowed as part of the variable name in SAL. We choose to translate an output variable *output*1! in Z to `output_` in SAL prefixed by the schema name and two underscores to avoid ambiguity. This is possible because the translator ensures unique names are used by restricting the acceptable Z input to having names involving a single consecutive _ character so we can safely use double underlining for system generated names.

Each operation is translated into one branch of a guarded choice in the transitions of the SAL module. We choose to label each choice by the name of the operation in the Z specification, although, strictly speaking this is optional for SAL.

In addition, it is necessary to ensure that the transition relation is total (for soundness of the model checking). This is achieved in the translation by a final

guarded command which is an `else` branch to provide a catch-all, and will evaluate to true only when all other guards evaluate to false.

Each choice in the transition (e.g., `Join : ...`) consists of a guarded assignment as in the initialisation. The predicate in the operation schema becomes a guard of the particular choice. The guard is followed by a list of assignments, one for each output and primed declaration in the operation schema. In the translation these assignments allow any compatible value to be assigned.

For example, the translation of our example will result in input and output declarations and a transition as follows:

```
INPUT Join__n? : NAME
INPUT JoinQ__n? : NAME
INPUT Remove__n? : NAME
INPUT Query__n? : NAME
OUTPUT Query__ans_ : Report

TRANSITION [
  Join : ...
     ...
       -->
          member' IN { x : set{NAME;}!Set | TRUE};
          waiting' IN { x : set{NAME;}!Set | TRUE}
[]
  JoinQ : ...
     ...
       member' = member
       -->
          member' IN { x : set{NAME;}!Set | TRUE};
          waiting' IN { x : set{NAME;}!Set | TRUE}
[]
            ...
[]
  Query : ...
    ...
       member' = member AND
       waiting = waiting'
       -->
          member' IN { x : set{NAME;}!Set | TRUE};
          waiting' IN { x : set{NAME;}!Set | TRUE};
          Query__ans_' IN { x : Report | TRUE}
[]
  ELSE -->
  ]
```

where we have elided parts of the translation we have not yet defined. In particular, the assigment of after-state values occurs before the `-->` in the transitions in this style of encoding.

Any local declarations, i.e., those not arising from a state schema but declared locally in the operationare translated with the `Op__` prefix as in the inputs and outputs (where *Op* is the name of the schema).

4 Implementing types

The above fragment already includes translations of some types defined in the specification, and Z includes a limited number of built in types. In particular, arithmos is defined which provides 'a supply of values to be used in specifying number systems'. In practice it is assumed that $\mathbb{N}$ and $\mathbb{Z}$ are available.

In contrast SAL supports the basic types NATURAL of natural numbers, and INTEGER of integers. These types can only be used with some of the SAL model-checkers, thus we will translate them into finite subranges. We translate the Z types $\mathbb{Z}$ and $\mathbb{N}$ into bounded SAL types `INT` and `NAT` respectively. We also translate the nonzero Z type $\mathbb{N}_1$ into the SAL type `NZNAT`. SAL definitions for these bounded types are included if the Z specification requires them.

The default finite subrange for `NAT` contains 4 elements in our implementation. Thus if $\mathbb{N}$ occurs in the LaTeX input, then `NAT: TYPE = [0..3]` will be generated at the start of the SAL specification. Likewise, the `NZNAT` type has a default subrange of 3 elements starting at 1 and the `INT` type has a default subrange of 5 elements starting at -1. This is to ensure that every type has at least three elements, while preserving the Z inclusion relationships: $\mathbb{N}_1 \subset \mathbb{N} \subset \mathbb{Z}$. However, the user can supply different bounds as parameters to the translator if desired, and a larger subrange will automatically be used if the specification contains a constant which is out of range. Our example uses the type $\mathbb{N}_1$ and a constant value 4096 occurs of this type. The translator therefore defines `NZNAT: TYPE = [1..4097]` automatically.

A given type, as in $[NAME]$ above, represents a user defined unstructured type. Although SAL supports a range of types, in general, the model checkers work with finite types. Thus we need to provide a finite enumeration of the given set *NAME*, and we translate it to:

```
NAME: TYPE = {NAME__1,NAME__2,NAME__3};
```

The number of elements in a given set enumeration is whatever the user has supplied as a parameter as the upper bound - or 3 by default.

Free types in Z define types whose values are either constants or constructors. The latter construct values of the free type from other values (see [13] for how

these free types are dealt with). In the example above, we translate *Report* directly as

```
Report: TYPE = DATATYPE
  yes,
  no
END;
```

5 Axiomatic descriptions

After the type declarations, a Z specification continues with the declaration of uninterpreted constants, which may or may not be constrained. A basic constant declaration has the form $capacity : \mathbb{N}_1$ and a constrained constant is declared using an axiomatic definition, given in standard schema format:

$$\begin{array}{|l} capacity : \mathbb{N}_1 \\ \hline capacity \leq 4096 \end{array}$$

In the SAL language definition [5] it is clear that the intention is to support uninterpreted constants, eventually. The obvious translation of an uninterpreted constant would be `capacity : NZNAT;`, and the translation of a constrained constant would rely on SAL's definition by set comprehension:

```
capacity : { x : NZNAT | x <= 4096 };
```

However, the current SAL toolset does not support uninterpreted constants, which are rejected by the semantic analyser. This means that the translation from Z into SAL has the choice of initialising all such uninterpreted constants with suitable sentinel values, or treating them like SAL local variables. The tradeoff is that the precise sentinel values to choose may be difficult to find; whereas SAL variables range over many values, causing a state explosion in the checking tools.

The SAL variable translation treats all uninterpreted constants as LOCAL variables. If they are constrained, this translation is identical to the translation of state schemas (see Section 3). Because of the state explosion this approach may cause, the preferred translation is to choose suitable precise values for constants. The heuristic chosen is to initialise constants by default to some value within the range of the type concerned. For example, where `NZNAT: TYPE = [1..2]` it would make sense to define:

```
capacity : NZNAT = 2;
```

for both of the above cases (constrained and unconstrained). The main concern is to find a suitable value which satisfies all the predicates in which the constants appear, otherwise the specification would, as a whole, be false. The assignment of values to constants is relatively simple for predicates which consist of an identifier compared to a literal. These straightforward predicates are used to determine the set of possible values each constant could be given and an initial value is chosen from these at random. If the predicates are unsatisfiable, our translation tool displays an error message and halts.

Predicates which compare constants with each other, such as $x < y$, or, worse still, $(u + v) \leq (y - z)$ are dealt with when the limits on each constant have been determined from the simple predicates. Currently, our tool uses a naive algorithm which cycles through the remaining possible initial values until a combination is found that satisfies all of the predicates. After 20 iterations the translator gives up with a message that the initial constraints cannot be resolved. Clearly, a more sophisticated constraint solving approach might be used; however the need to solve large systems of constraints rarely arises in typical Z specifications.

In our particular example we have the following pair of constraints:

$$\begin{array}{|l} \mathit{capacity} : \mathbb{N}_1 \\ \hline 1 < \mathit{capacity} \leq 4096 \end{array}$$

$$\begin{array}{|l} \#\mathit{NAME} > \mathit{capacity} \end{array}$$

In the SAL translation, a value for *capacity* is chosen (at random) which satisfies all the constraints, e.g., one possible value it will be instantiated to is:

```
capacity : NZNAT = 2;
```

This simplification is possible because the tool knows the size of the given type NAME and so $\#NAME$ is replaced by a constant 3. The largest *capacity* that is smaller than 3 is 2. The predicate $\mathit{capacity} \leq 4096$ is redundant and is eliminated by the tool.

6 Implementing sets

A basic translation scheme for sets is given in the *set.sal* context, provided with the SAL distribution, and this represents a set as a function from elements to Booleans. All of the set operations can be expressed in a logically succinct way, for example the set membership function *contains*? simply applies the set to the element. In [13] the *set.sal* context was extended to include a means of determining the cardinality of nonempty sets. We have adapted this encoding to work with all sets as follows:

```
set{T : TYPE; } : CONTEXT =
BEGIN

  Set : TYPE = [T -> BOOLEAN];

  empty_set : Set = LAMBDA (e : T) : FALSE;

  full_set : Set = LAMBDA (e : T) : TRUE;

  insert (aset : Set, e : T) : Set =
    LAMBDA (e1 : T) : e = e1 OR aset(e1);

  remove (aset : Set, e : T) : Set =
    LAMBDA (e1 : T) : e /= e1 AND aset(e1);

  contains? (aset : Set, e : T) : BOOLEAN =
    aset(e);

  empty? (aset : Set) : BOOLEAN =
    (FORALL (e : T) : aset(e) = FALSE);

  union(aset1 : Set, aset2 : Set) : Set =
    LAMBDA (e : T) : aset1(e) OR aset2(e);

  intersection(aset1 : Set, aset2 : Set) : Set =
    LAMBDA (e : T) : aset1(e) AND aset2(e);

  difference(aset1 : Set, aset2 : Set) : Set =
    LAMBDA (e : T) : aset1(e) AND NOT aset2(e);

  size?(aset:Set, n:NATURAL) : BOOLEAN =
    (n = 0 AND empty? (aset)) OR
    (n > 0 AND
    (EXISTS (f:[[1..n] -> T]) :
      (FORALL (x1,x2:[1..n]) : f(x1)=f(x2) => x1=x2) AND
      (FORALL (y:T) : aset(y) <=> (EXISTS (x:[1..n]) : f(x) =y))));
END
```

This allows a succinct translation of declarations and predicates involving sets. A declaration $member : \mathbb{P}\, NAME$ is translated to `member : set{NAME;}!Set`. Predicates involving set operators are also translated in the obvious way. For example, $member = \varnothing$ becomes

```
member = set{NAME;}!empty_set
```

Similarly, $n? \in waiting$ in the operation *Join* becomes

```
set{NAME;}!contains?(waiting, Join__n?)
```

and $member' = member \cup \{n?\}$ becomes

```
member' = set{NAME;}!insert(member, Join__n?)
```

Set cardinality (the most troublesome operator) cannot be expressed directly as a function returning the element count in SAL, since nowhere does SAL store a representation of the set as a whole, but only as a distributed collection of function-valued variables. Instead, the *size* function computes the relation between sets and natural numbers, returning true when a set is of a given size. Z predicates involving the cardinality of sets, such as $\#waiting < total$, can be translated to the following existentially quantified SAL predicate:

```
EXISTS(n: NAT) : set{NAME;}!size?(waiting,n) AND n < total.
```

Although this implementation of the translation algorithm is correct, a number of issues arose, some of which were to do with representation, others with efficiency, and we deal with each in turn.

6.1 Literals

The translation of literal sets causes particular problems as they can only occur in type declarations in SAL, but can be used in variable declarations or predicates in Z. The translation process addresses the problem of set literals in declarations by introducing a named type. Thus a state variable

$s : \mathbb{P}\{1, 2, 3\}$

becomes

```
LOCAL s : set{Set__1__2__3;}!Set
```

and `Set__1__2__3: TYPE = { x : NAT | x < 4 };` appears before the state module. Any other use of the same or an equivalent (e.g., $\{3, 2, 1\}$) set literal in a declaration will be translated to the type name. Literal sets in predicates can be dealt with more simply: $x \in \{1, 2\}$ becomes `(x=1 OR x=2)`.

6.2 Re-implementing set cardinality

It was soon discovered that Z specifications that made reference to the cardinality of sets generated SAL translations which did not execute in any sensible amount of time. Simulations did not terminate in half a day, whilst some model checks terminated, depending on how the checked LTL theorem further constrained the state space search. We therefore experimented with alternative set encodings that might have a more efficient implementation of `size?`.

Attempt I - A recursive definition of sets According to the SAL language manual it should be possible to define inductive data types, similar to the illustrated definition of lists, which have recursive operations. In this case, `size?` could be provided as an efficient recursive function on sets. Thus one would define:

```
cset{T : TYPE; } : CONTEXT =
BEGIN
  Set : TYPE = DATATYPE
   add(elem : T, rest : Set),
   empty
       END;

  insert (set : Set, e : T) : Set =
    IF empty?(set)
      THEN add(e, set)
    ELSIF e = elem(set)
      THEN set
    ELSE
    add (elem(set), insert(rest(set), e))
  ENDIF;
...

  size?(set : Set) : NATURAL =
    IF empty?(set)
        THEN 0
    ELSE
        1 + size?(rest(set))
    ENDIF;
END
```

In this, `empty` and `add` are the primitive type constructors and `elem` and `rest` are the implicitly defined deconstructors that break apart a set. Set operations like `insert` are defined recursively by always `add`ing a new element to an empty set, otherwise deconstructing the head `elem` to see if this is equal to the new element, returning the set unchanged if so, otherwise inserting the new element into the `rest` of the set and `add`ing the deconstructed head back onto this. The `size?` function recursively counts the number of `add`s wrapping the `empty?` set.

Unfortunately, it was discovered afterwards that the current release of the SAL toolset does not yet support simulation or model checking with inductively defined datatypes. Even a simple recursive definition of `size?` fails to load into the simulator, because SAL attempts to expand all possible recursive trees and runs out of memory. A future release of the toolset is planned to handle recursively defined functions and inductive types.

Attempt II - countable finite sets After experimenting with other encodings, a workable SAL translation was found for encoding counted finite sets. This is a brute-force encoding that is specific to the maximum expected set cardinality. Various set contexts `setN` were designed for different `N`, corresponding to the maximum expected cardinality. This is reasonable in SAL, since every scalar type must have a known lower and upper bound. Our translation fixes the range of scalar types for small `N`. The following excerpt is from the `set5` context, which holds a maximum of five elements.

```
set5{T : TYPE; e1, e2, e3, e4, e5 : T} : CONTEXT =
BEGIN

%% A countable set over a domain of 5 elements.  The context
%% parameters are: the element type T, and an exhaustive
%% enumeration of all the elements e1..e5 of the domain.

  Set : TYPE = [T -> BOOLEAN];

  empty_set : Set =
    LAMBDA (e : T) : FALSE;

  full_set : Set =
    LAMBDA (e : T) : TRUE;

  size? (set : Set) : NATURAL =
    IF set(e1) THEN 1 ELSE 0 ENDIF +
    IF set(e2) THEN 1 ELSE 0 ENDIF +
    IF set(e3) THEN 1 ELSE 0 ENDIF +
    IF set(e4) THEN 1 ELSE 0 ENDIF +
    IF set(e5) THEN 1 ELSE 0 ENDIF;

  ... %% the rest as per the set.sal context
END
```

The main difference between this and the standard `set.sal` context is that the context accepts value-parameters for all possible elements of the set, as well as the usual element type-parameter. This allows a brute-force encoding of the `size?` function, which tests for the presence of each element in turn. This encoding executes very efficiently, since it builds a shallow symbolic execution tree, in contrast with a recursively-defined function. The rest of the context is defined exactly as per the original `set.sal` context, using the encoding of sets as Boolean-valued functions over its elements, since this is the optimal encoding for translation to BDDs in the SAL tools.

A number of contexts may be pre-generated, for different `N`. We have also successfully generated different `setN.sal` on demand, to cater for unknown ranges. The only changes are the number of value parameters required and the number of subexpressions in the `size?` function. To use these bounded contexts, it is preferred to instantiate all parameters once in a new named context, in the following style:

```
PersonSet : CONTEXT =
  set3{PERSON; PERSON__1, PERSON__1, PERSON__3};
...
LOCAL set : PersonSet!Set
INITIALIZE set = PersonSet!empty_set
```

Then, all types and operations are accessed from the new named context. This makes the rest of the generated code easier to read than if the contexts were instantiated at every point of usage.

Attempt III - Direct enumeration Our current approach to problems in SAL with cardinality are to limit the state space explosion possibilities in SAL during the translation process. In general the structure of a predicate in Z is much the same as its translation in SAL, but not where `size?` is involved. When the cardinality of a set is tested for equality (or inequality) the test can be transformed fairly simply because the SAL `size?` function is designed to test for a particular cardinality. Thus $\#waiting = capacity$ becomes `set{NAME;}!size?(waiting, capacity)`.

Our initial approach to translating comparisons like $\#waiting < 3$ was to use an existential quantifier in the translated expression. However, since the standard translation of `size?` already used nested quantification, this merely exacerbated the state space explosion. Our current solution is to exploit the translator's knowledge of the maximum cardinality of the sets we are using to produce an expression which does not involve an existential quantifier. So if the maximum cardinality of *waiting* was 3 the translation of $\#waiting < 3$ is `NOT set{NAME;}!size?(waiting, 3)`, whereas if the the set *waiting* could have up to 5 elements the translation is

```
(set{NAME;}!size?(waiting, 0) OR set{NAME;}!size?(waiting, 1) OR
              set{NAME;}!size?(waiting, 2))
```

Where the comparison is with a variable the expression is slightly more complex. So $\#waiting < capacity$, where the maximum cardinality of the sets is 3 and the variable's upper bound is 3, becomes

```
(((0<capacity) AND set{NAME;}!size?(waiting,0) ) OR ((1<capacity)
```

```
        AND set{NAME;}!size?(waiting, 1) ) OR ( (2<capacity) AND
        set{NAME;}!size?(waiting, 2) ) OR ( (3<capacity) AND
        set{NAME;}!size?(waiting, 3) ))
```

This assumes that SAL does a lazy evaluation of the expression but experimental results seem to confirm this. Evaluating $\#member > \#waiting$ is possible by this technique too although the resulting expression does get rather long:

```
((set{NAME;}!size?(member,0) AND NOT set{NAME;}!size?(waiting,0))
    OR
 (set{NAME;}!size?(member, 1) AND (set{NAME;}!size?(waiting, 2)
          OR set{NAME;}!size?(waiting, 3)) )
    OR
 (set{NAME;}!size?(member, 2) AND (set{NAME;}!size?(waiting, 3))))
```

7 Using the translation tool

Currently we use a command line interface. As input format we use the LaTeX markup as given in the Z standard, and the output a plain SAL file. Since XML markups exist for both Z and SAL, these might eventually be the ultimate exchange format.

The components of the SAL toolset can now be applied to the output. For example, we can simulate the specification or use one of the model checkers on it. As we alluded to above, experiments revealed substantial efficiency problems with the naive version of `size?`.

Labelling the three implementations as Original (the first in Section 6), Canonical (Attempt III) and Alternate (Attempt II) we can compare the efficiency of the different size implementations.

For example, if we run the SAL simulator on the above example using the Canonical set representation, we find that it takes 2 seconds to create the initial state(s) of the system. Then invoking

```
(display-curr-states)
```

reports that 162 possible initial states were generated.

```
(display-curr-trace)
```

gives a single trace (one of the sets of initialisations leading to one of the initial states):

```
sal > (display-curr-trace)
Step 0:
--- Input Variables (assignments) ---
(= Join__n? NAME__3);
(= JoinQ__n? NAME__3);
(= Remove__n? NAME__3);
(= Query__n? NAME__3);
--- System Variables (assignments) ---
(= (member NAME__1) false);
(= (member NAME__2) false);
(= (member NAME__3) false);
(= (waiting NAME__1) false);
(= (waiting NAME__2) false);
(= (waiting NAME__3) false);
(= Query__ans_ yes);
```

The next stage is to try to step through the simulation.

```
(step!)
```

advances by a single step.

```
(display-curr-states)
```

reports that 648 states were created in this first step. This is consistent with attempting 4 transitions for each of the 162 states in the initialisation (4 * 162 = 648). The simulation can then be continued by exploring subsequent traces, working our way through the specification.

The performance of the Alternate representation was similar, however, the Original representation failed to create an initial state of the system in the simulator in over 12 hours. Experiments with the model checker produced similar results. We formalised the following three theorems:

th1 : the size of the *member* set can never reach 3 (expected false)
th2 : the size of the *waiting* set can never reach 3 (expected false)
th3 : the combined sizes of both sets is always ≤ 3 (expected true)

For the Canonical example, the theorems are expressed:

```
th1 : THEOREM State |- NOT F(set{NAME;}!size?(member', 3));
th2 : THEOREM State |- NOT F(set{NAME;}!size?(waiting', 3));
th3 : THEOREM State |- G( (FORALL(x,y:NZNAT) :
                (set{NAME;}!size?(member', x) AND
                  set{NAME;}!size?(waiting', y)) => x+y <= 3));
```

For the Alternate example, the same theorems were supplied slightly differently, reflecting the simpler `size?` function:

```
th1 : THEOREM State |- NOT F(FSet!size?(member') = 3);
th2 : THEOREM State |- NOT F(FSet!size?(waiting') = 3);
th3 : THEOREM State |- G( (FSet!size?(member) +
                                    FSet!size?(waiting)) <= 3);
```

The following table gives the approximate timings for the simulation and proof or refutation of the theorems. The entries marked "$>$ 12 hours" mean that, for example, the Canonical representation took over 12 hours for the third theorem without terminating.

	Original	Canonical	Alternate
Simulation	$>$ 12 hours	2 sec	1.5 sec
th1	$>$ 12 hours	4 sec	3 sec
th2	$>$ 12 hours	4 sec	3 sec
th3	$>$ 12 hours	$>$ 12 hours	3 sec

Where the model checking was feasible, for theorems $th1$ and $th2$, both examples discovered the path of $3 * JoinQ$ to fill the waiting set, and the path of $3 * JoinQ + 3 * Join$ to fill the member set.

As can be seen, the original encoding does not model check in a feasible time. The Canonical encoding is feasible for some properties (and some specifications), and we have yet to find an example where the Alternate encoding is infeasible.

8 Discussion

Although, it is feasible in general to model check specifications with very large state spaces, model checking Z specifications poses some serious challenges. The inclusion of non-trivial data types is well known to cause state-space explosions, and even the use of small sets in the above example pushed the model checker to its limits in its memory and time capabilities in some of the encodings. The restriction to small set sizes and the fact that generic parameters (e.g., *capacity* in the example above) have to be instantiated mean that we are really using model checking to explore the specification rather than to perform the complete verification of a property. The use of data abstraction to alleviate this problem is a topic for further investigation.

However, in addition to the inclusion of data, the approach to the semantics in Z adds to the overhead. Specifically, everything in Z is modelled in the semantics as a set, thus relations are sets of pairs, functions are relations with a constraint, sequences are partial functions with a constraint, and so on. If this approach is

preserved in the translation to SAL, as we have done so far, then this results in a computational overhead for the model checker which is likely to become prohibitive for any non-trivial specification.

The alternative would be to code the data structures in the mathematical toolkit directly. For example, SAL has total relations as one of the built in data-types, but does not have partial relations, however, the latter could be encoded as total functions with an undefined element to represent elements outside the precondition (i.e., one would model a partial relation as its *totalisation*). This is likely to give a better efficiency than the existing approach but there are complications in the translation. For example, in Z it is acceptable to write (for a function $f : \mathbb{N} \pfun \mathbb{N}$):

$$f' = f \cup \{(1, 3)\}$$

This is translated easily at present, but if functions are coded directly then a version of set union needs to be defined on the function space, which also has to be able to mix total and partial functions freely.

9 Conclusions

In this paper, we have discussed our current approach to building a model checker for Z specifications. This is work in progress, and there is much to be done. For example, we need to explore the use of constraint solvers in resolving the instantiation of constants introduced by axiomatic definitions. Similarly we need to investigate alternative representations for the mathematical toolkit which are not just their set-based expansion, and to compare the efficiency of such an approach. Finally, the current prototype is stand-alone, and a re-engineered version would involve integration with the CZT platform of tools.

Of course, this is not the first attempt to provide model checking facilities for Z. For example, there have been a number of encoding of subsets of Z-based languages in FDR [6, 11, 8]. However, FDR is not a temporal logic model checker, but rather is designed to check whether a refinement relation holds between two specifications. Additionally, FDR was developed for a process algebra, rather than a state-based notation, thus encoding a language such as Z is non-trivial, and to date there is no full encoding of Z in FDR.

Other relevant work directly concerned with state-based languages includes that on the ProB model checker [9] which provides model checking capabilities for B. Bolton has recently experimented with using Alloy to verify data refinements in Z [2]. The Alloy Analyzer is a SAT-based verification tool that automatically determines whether a model exists for a specification within given set bounds for the basic types. Bolton translates a Z specification into Alloy by encoding its relational semantics in Alloy and using the latter to see if a simulation can

be found. However, the encoding of the relational semantics is not automatic in contrast to the implemented Z to SAL translation discussed above.

Acknowledgements: This work was done as part of collaborative work with the University of Queensland, and in particular, Graeme Smith and Luke Wildman. Tim Miller also gave valuable advice on the current CZT tools. Thanks is also due to the financial support of the EPSRC via the *RefineNet* grant.

References

1. ISO/IEC 13568:2002. Information technology—Z formal specification notation—syntax, type system and semantics. International Standard.
2. C. Bolton. Using the Alloy Analyzer to Verify Data Refinement in Z. *Electronic Notes in Theoretical Computer Science*, 137(2):23–44, 2005.
3. E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In E.A. Emerson and A.P. Sistla, editors, *International Conference on Computer Aided Verification (CAV'00)*, volume 1855 of *LNCS*, pages 154–169. Springer-Verlag, 2000.
4. E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.
5. L. de Moura, S. Owre, and N. Shankar. The SAL language manual. Technical Report SRI-CSL-01-02 (Rev.2), SRI International, 2003.
6. C. Fischer and H. Wehrheim. Model-checking CSP-OZ specifications with FDR. In K. Araki, A. Galloway, and K. Taguchi, editors, *International Conference on Integrated Formal Methods (IFM'99)*, pages 315–334. Springer-Verlag, 1999.
7. S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *International Conference on Computer Aided Verification (CAV'97)*, volume 1254 of *LNCS*, pages 72–83. Springer-Verlag, 1997.
8. G. Kassel and G. Smith. Model checking Object-Z classes: Some experiments with FDR. In *Asia-Pacific Software Engineering Conference (APSEC 2001)*. IEEE Computer Society Press, 2001.
9. M. Leuschel and M. Butler. Automatic refinement checking for B. In K. Lau and R. Banach, editors, *International Conference on Formal Engineering Methods, ICFEM 2005*, volume 3785 of *LNCS*, pages 345–359. Springer-Verlag, 2005.
10. Tim Miller, Leo Freitas, Petra Malik, and Mark Utting. CZT Support for Z Extensions. In Judi Romijn, Graeme Smith, and Jaco Pol, editors, *Integrated Formal Methods, IFM 2005*, volume 3771 of *LNCS*, pages 227–245. Springer-Verlag, 2005.
11. A. Mota and A. Sampaio. Model-checking CSP-Z: strategy, tool support and industrial application. *Science of Computer Programming*, 40:59–96, 2001.
12. H. Saïdi and N. Shankar. Abstract and model check while you prove. In N. Halbwachs and D. Peled, editors, *International Conference on Computer Aided Verification (CAV'99)*, volume 1633 of *LNCS*, pages 443–453. Springer-Verlag, 1999.
13. G. Smith and L. Wildman. Model checking Z specifications using SAL. In H. Treharne, S. King, M. Henson, and S. Schneider, editors, *International Conference of Z and B Users (ZB 2005)*, volume 3455 of *LNCS*, pages 87–105. Springer-Verlag, 2005.
14. J.M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall, 2nd edition, 1992.