

Model checking Z specifications using SAL

Graeme Smith and Luke Wildman

School of Information Technology and Electrical Engineering
The University of Queensland 4072, Australia
smith@itee.uq.edu.au luke@itee.uq.edu.au

Abstract. The Symbolic Analysis Laboratory (SAL) is a suite of tools for analysis of state transition systems. Tools supported include a simulator and four temporal logic model checkers. The common input language to these tools was originally developed with translation from other languages, both programming and specification languages, in mind. It is, therefore, a rich language supporting a range of type definitions and expressions. In this paper, we investigate the translation of Z specifications into the SAL language as a means of providing model checking support for Z. This is facilitated by a library of SAL definitions encoding the Z mathematical toolkit.

Keywords: Z, model checking, SAL, tool support

1 Introduction

The Symbolic Analysis Laboratory (SAL) [1] is aimed at allowing different verification tools, such as various types of model checkers and theorem provers, to be combined. The input language shared by the tools was originally proposed as a target for translation of a variety of specification and programming languages and provides a broad range of features to support them. In this paper, we describe the translation of Z [17] to SAL for the purpose of taking advantage of the variety of verification tools that SAL supports, as well as those it will support in the future.

Currently SAL provides a suite of four model checkers and a simulator. The SAL simulator is a fully customisable environment for manipulating state transition systems and their traces, and it is a significant advance on Z animators such as Possum [7]. The model checkers are also fully customisable. There are two symbolic model checkers for checking linear time temporal logic (LTL) and branching time temporal logic (CTL) properties. There are also two *bounded* LTL model checkers, one of which supports infinite types in the model's state space. Bounded model checking involves a state space search to a given depth, rather than of the entire state space. This is a way of finding counter-examples in large models (proving a property, however, still involves a complete state space search). SAL also supports a technique called *k-induction* [12] which allows the bounded model checkers to be called iteratively to search further into the state space when a counter-example is not yet found.

As discussed by Smith and Winter [16], applying temporal logic model checking to Z specifications is not possible unless either all operations are *totalised*, i.e., have a true precondition, or a *blocking* semantics of operations is adopted, i.e., where operations are unable to occur outside their calculated preconditions. The standard non-blocking semantics of Z allows operations to occur outside their preconditions resulting in arbitrary state changes. This makes it impossible to prove any interesting temporal properties (over states) when operations are not totalised. In this paper, we adopt a blocking semantics of Z (which also allows a non-blocking interpretation to be taken with totalised operations).

Note that with a non-blocking semantics even properties proved to hold when no operation is applied outside its precondition are not necessarily true for refinements (where preconditions may be weakened). Hence, they would be of limited use. Under a blocking semantics, most temporal logic properties written in LTL are preserved by refinement, and all are preserved under simple restrictions on the retrieve relation [3]. Similar results can be shown to hold for CTL.

There are also significant challenges in model checking Z . As Z is based on set theory, a specification may involve infinite sets, partial functions, recursive types, predicative types, etc.; all of these things are difficult to compile into a representation suitable for model checking. In addition, a Z specification may involve a large state space; the larger the state space, the more computation effort required to perform model checking. Such challenges have meant that Z specifications have had to be rewritten in order to model check them. A Z specification writer may not, however, want to give up the expressiveness of Z (which aids in understandability, separation of concerns, etc.) for a less expressive model-checkable language.

SAL may help to solve this dilemma. Its compilation routines, based on the ICS decision procedure library [4], enable expressive mathematics to be used. In addition, the module mechanism of SAL is quite similar to the concept of a Z state transition system. In this paper, we explore the similarities between Z and SAL by defining a translation which enables (most) Z specifications to be accepted by the model checkers.

The paper is structured as follows. In Section 2, we give an overview of the SAL language. In Section 3, we show how *minimal* Z specifications, those not involving definitions from the Z mathematical toolkit nor advanced use of schemas, can be translated into SAL. In Section 4, we describe how the Z mathematical toolkit may be encoded as a library of SAL definitions and, hence, how our translation scheme may be extended. We then look at uses of schemas as predicates and types, and the schema calculus in Section 5. In Section 6, we close with a discussion of related work, an evaluation of the results of our translation, and our plans for further development.

2 The SAL Language

The common input language to the SAL tools was originally developed with translation from other languages, both programming and specification languages,

in mind. It is a rich language supporting a range of type definitions and expressions. The former include basic types such as integers and naturals, tuple (or Cartesian product) types, total functions, enumerated types, recursive datatypes, array and record types, and subtypes of any other type defined in terms of set comprehensions. Expressions in the language may involve lambda abstractions and set comprehensions. Predicates (i.e., Boolean-valued expressions) may involve universal and existential quantification.

A complete syntax of the type and expression sublanguages of SAL can be found in the SAL language manual [2]. Syntax relevant to the work in this paper will be introduced as it is required. Below we describe the main structuring mechanisms of the SAL language: *contexts* and *modules*.

2.1 Contexts

A SAL context groups together a number of definitions and properties. The definitions include types, constants and modules for describing state transition systems. The properties, expressed in LTL or CTL, refer to modules and are intended to be checked by the model checking tools. Contexts may have both type and value parameters. Type parameters are treated as uninterpreted types, and value parameters as uninterpreted constants. For example, a context with type parameters X and Y and value parameter N of type natural number is defined as follows.

```

mycontext{ $X, Y : TYPE; N : NATURAL$ } : CONTEXT =
  BEGIN
    ...
  END

```

where the ... represents the elided definitions and properties.

Contexts may refer to definitions in other contexts. In this way a SAL specification can be structured across several contexts, and contexts can be used to create libraries of commonly used definitions. The context above, for example, may be instantiated within another context. We could instantiate it with the type *NATURAL* for X and Y and the value 3 for N as follows.

```

mc : CONTEXT = mycontext{NATURAL, NATURAL; 3}

```

Then, given a definition *def* in *mycontext*, we can refer to *def* by *mc!def*.

Alternatively, we could have referred to *def* without the declaration of *mc* via *mycontext*{*NATURAL, NATURAL*; 3}*!def*.

2.2 Modules

Modules appear in contexts and are used to describe state transition systems. A module comprises the declaration of a set of variables representing the state of

the module, and optional sections describing state invariants, module initialisation and possible transitions. An example of a module can be found at the end of Section 3.

The variables may be input variables, output variables, global variables or local variables. Input variables are not under the control of the module and are nondeterministically assigned a value in their type before each transition. All other variables are controlled by the module. Global variables may be controlled by more than one module within the same context.

There are also means of composing modules. These, however, are not discussed in this paper.

3 Basic Translation Approach

In this section, we show how *minimal* Z specifications can be encoded in the SAL language. By minimal, we mean specifications which do not make use of definitions from the Z mathematical toolkit nor the advanced use of schemas (including the use of schemas as predicates and types, and the operators of the schema calculus). The encoding of the former will be detailed in Section 4 and the latter in Section 5.

Our translation scheme between Z and SAL requires the Z specification to be syntactically correct and type correct. Hence, we assume our input comes from a Z parser and type checker. To simplify the translation, we assume the parser performs some syntactic simplifications. Specifically, schema references in the declaration part of quantified expressions and schemas need to be expanded, and predicates involving the quantifier \exists_1 need to be rewritten in terms of the quantifiers \exists and \forall . Also, set comprehensions of the form $\{d \mid p \bullet e\}$ are converted to the simpler $\{d \mid p\}$ form. For example, $\{n : \mathbb{N} \mid n < 10 \bullet 2 * n\}$ is converted to $\{m : \mathbb{N} \mid \exists n : \mathbb{N} \bullet n < 10 \wedge m = 2 * n\}$. Similarly, μ -expressions of the form $(\mu d \mid p \bullet e)$ are converted to the simpler $(\mu d \mid p)$ form.

3.1 Types and Constants

SAL supports the basic types *NATURAL* of natural numbers, and *INTEGER* of integers¹. These types however can only be used with one of the four model checkers (the infinite-bounded model checker). In general, the model checkers only work with finite types. Therefore, subranges of *NATURAL* and *INTEGER* need to be used. For example, we could define the type *NAT* as the natural numbers between 0 and 100 as follows.

$$NAT : TYPE = [0..100]$$

The actual subrange required to perform effective checks will depend on the particular specification. Setting of the maximum value for the natural numbers (and

¹ It also supports the types *REAL* of real numbers, and *BOOLEAN* of Boolean values which are useful for certain extensions of Z.

minimum value for the integers) is therefore a task for the user. The value would manifest itself as a parameter to the automatic translation process between Z and SAL.

Given sets are used in Z when we want to abstract away from the actual values of a type. For example, the type *NAME* representing people's names can be specified as

$$[NAME]$$

In SAL, we need to give values for all types. Hence, such a type is represented by an enumerated type as follows

$$NAME : TYPE = \{NAME1, NAME2, NAME3\};$$

To make this translation automatic requires a default value for the cardinality of given sets. The user should, however, be able to override this default value in order to ensure that enough values are available to perform effective checks.

Free types in Z enable the definition of enumerated types whose values are either constants or *constructors*. The latter construct values of the free type from other values. For example, the type of a process identifier may be *null* when no identifier has been allocated, and a natural number, otherwise.

$$PID ::= null \mid id\langle\mathbb{N}\rangle$$

SAL supports a similar type definition facility. The above is represented in SAL as

$$\begin{aligned} PID : TYPE = DATATYPE \\ & \quad null, \\ & \quad id(nat : NAT) \\ END; \end{aligned}$$

The label *nat* above is necessary and provides access to the number allocated to a process, i.e., if *p* is a non-null process identifier then *nat(p)* returns the number. Hence, an expression in Z which returns the number allocated to a process, i.e., $id^{\sim}(p)$, is translated as *nat(p)*.

In Z, the values used in a constructor may also be of the free type being defined. This allows recursive type definitions. Although this is also supported by SAL, it will result in infinite types and hence disallow the use of most of the model checking tools.

Abbreviation definitions introduce types via expressions. For example, the type of all prime numbers could be specified as follows.

$$Primes == \{n : \mathbb{N} \mid \forall m : \mathbb{N} \bullet n \bmod m = 0 \Rightarrow m = n \vee m = 1\}$$

This is represented in SAL as a subtype of the type NAT defined above.

$$Primes : TYPE = \{n : NAT \mid FORALL (m : NAT) : \\ n \text{ MOD } m = 0 \Rightarrow m = n \text{ OR } m = 1\};$$

Note the close mapping between the Z and SAL predicates. Of the primitive constructs used in Z predicates and expressions only μ (used for selecting a unique value satisfying a predicate) and \mathbb{P} (used for denoting an arbitrary subset of a set) are not directly supported in SAL. We will delay the discussion of \mathbb{P} until Section 4. With μ -expressions, assuming our specification is correct Z, the predicate of the expression will guarantee a unique value. Hence, there is no need for SAL to check this. A predicate $x = (\mu n : \mathbb{N} \mid n * 1 = 3)$ can be represented in SAL by $x * 1 = 3$. When the predicate is not a simple equality, for example, $(\mu n : \mathbb{N} \mid n * 1 = 3) < y$ then we introduce an existentially quantified variable to denote the unique value defined by the μ -expression. That is, the above is represented in SAL by $EXISTS (n : NAT) : n * 1 = 3 \text{ AND } n < y$.

Constants are defined in Z using axiomatic definitions. For example,

$$\left| \begin{array}{l} n : \mathbb{N} \\ p : \mathbb{N} \times \mathbb{N} \end{array} \right. \\ \hline \left| \begin{array}{l} n < 10 \\ first(p) = n \end{array} \right.$$

In SAL, we declare them using set comprehensions in order to capture the predicate of the axiomatic definition.

$$n : \{n : NAT \mid n < 10\}; \\ p : \{p : [NAT, NAT] \mid p.1 = n\};$$

Z generic constants can also be translated by making the generic type a parameter of the SAL context in which the constant is declared. Since we may require more than one instantiation of a generic constant, it should be declared in its own context and its definition imported and instantiated as required.

3.2 Schemas

Schemas may be used in Z to define types, and as abbreviations for recurring predicates. We will discuss these uses of Z schemas in Section 5. The main use of schemas in Z is in the definition of state transition systems. Typically, a specification has a single state schema, an initial state schema and a number of operation schemas. Such a specification can be represented by a SAL module.

The variables of the state schema become local variables of the module, and inputs and outputs of the operations become input and output variables of the module respectively. The output variables need to be renamed since ! is not

allowed as part of a variable name in SAL. We choose to translate an output $x!$ in Z to an output variable $x_!$ in SAL.

The key to representing the schema predicates is a powerful *guarded command* facility supported by SAL for defining the initialisation and transition sections of a module. Guarded commands are of the form

$$\begin{array}{l} \textit{Guard} \text{ -- } > \textit{Assignment} ; \\ \quad \quad \quad \vdots \\ \quad \quad \quad \textit{Assignment} \end{array}$$

A guard is an arbitrary predicate which may refer to values of primed (post-state) variables. The assignments they guard may be nondeterministic, i.e., each variable is assigned a value from a set of potential values. For example, a variable x can be assigned a value from the set of natural numbers less than 10 as follows.

$$x \text{ IN } \{n : \textit{NAT} \mid n < 10\}$$

Any variables which are not assigned a value remain unchanged.

To support such guarded commands, SAL checks the guard after the assignments are made and, if it is false, undoes the assignments and makes new assignments when possible. The guarded command is enabled only when a set of assignments satisfying the guard can be found.

A Z predicate referring to a variable $x : \mathbb{N}$

$$x' = 3 * x$$

can hence be represented in SAL as the guarded command

$$x' = 3 * x \text{ -- } > x' \text{ IN } \{x : \textit{NAT} \mid \textit{TRUE}\}$$

Note that the guard can be any predicate and hence if the above predicate were instead

$$x = x' * 3$$

it could be represented in SAL as

$$x = x' * 3 \text{ -- } > x' \text{ IN } \{x : \textit{NAT} \mid \textit{TRUE}\}$$

This approach allows the flexibility required to directly translate Z predicates into SAL. Some care must be taken with quantified variables, however, as discussed after the example below.

Guarded commands may be used in the initialisation and transition sections of a SAL module. The initialisation section of a SAL module may comprise a single guarded command. The transition section may comprise a choice between several guarded commands separated by the syntax []. These guarded commands may be labelled to aid the understanding of counter-examples generated by the SAL model checkers. Normally, a counter-example is given as a sequence of

states; labels in the counter-example indicate which branch of the transition fired between consecutive states.

Guarded commands may not be used in a state invariant section of a SAL module. This is not a problem, however, given the syntactic preprocessing we have assumed. Specifically, schema references used in the declarations of schemas are expanded before the translation to SAL. Hence, in the Z specification which we translate to SAL, the state schema's predicate will be included in the initial state schema, and in both unprimed and primed form in each operation schema. Hence, it will be true initially and after each operation as required.

In the translation from Z to SAL, the initial state schema's predicate becomes the guard of the initialisation section of the module. The operations' predicates become the guards of the transition section. In each guarded command representing initialisation or an operation, all state variables are assigned values non-deterministically from their types. For operations with outputs, the corresponding output variables are additionally assigned values nondeterministically from their types. For such operations, we need to refer to the output variable in primed form in SAL.

Finally, SAL's model checkers may produce unsound results when a module being checked can deadlock. It is therefore necessary to ensure that the transition relation is total. This can be ensured by a final guarded command in the transition section of the form

ELSE -- >

This guard will evaluate to true only when all other guards evaluate to false. There are no assignments, so the state will remain unchanged.

As an example, consider the following Z specification.

$\frac{\textit{State}}{x : \mathbb{N}}}{x < 10}$	$\frac{\textit{Init}}{\textit{State}}}{x = 0}$
$\frac{\textit{Increment}}{\Delta\textit{State}}}{\begin{array}{l} x! : \mathbb{N} \\ x' = x + 1 \\ x! = x' \end{array}}$	$\frac{\textit{Choose}}{\Delta\textit{State}}}{\begin{array}{l} x? : \mathbb{N} \\ \exists x : \mathbb{N} \bullet x < x? \wedge x' = x \end{array}}$

It's SAL translation is

```

state : MODULE =
  BEGIN
    INPUT x? : NAT
    LOCAL x : NAT
    OUTPUT x_ : NAT
    INITIALIZATION [x = 0 AND x < 10
      -- > x IN {n : NAT | TRUE}]
    TRANSITION [ increment : x' = x + 1 AND x_ = x' AND
      x < 10 AND x' < 10
      -- > x' IN {n : NAT | TRUE};
      x_ IN {n : NAT | TRUE}
    [] choose : (EXISTS (x0 : NAT) :
      x0 < x? AND x' = x0) AND
      x < 10 AND x' < 10
      -- > x' IN {n : NAT | TRUE}
    [] ELSE -- > ]
  END

```

Note that the quantified variable in the transition corresponding to *Choose* is renamed to a fresh variable $x0$. This is because SAL regards the prime symbol as an operator on a state variable rather than a decoration on a name. If we left the quantified variable as x , it would not be possible to refer to x' in its scope since x would no longer refer to the state variable, but instead to the quantified variable. For similar reasons, it is not possible to translate a predicate $\forall x' : NAT \bullet \dots$ directly; x' is not a name but an expression involving the prime operator. We would translate this predicate to $\forall x1 : NAT \bullet \dots$ where $x1$ is a fresh variable.

4 Z Mathematical Toolkit

The Z mathematical toolkit [17] is a library of definitions of types and operators used commonly in Z. It includes operators for sets and natural numbers, and types and operators for relations, functions, sequences and bags. The encoding of these definitions in SAL are, for the most part, relatively straightforward. Complications arise for those definitions given for Z in terms of the μ operator since this is not supported by SAL. The affected definitions are those for function application and set size, $\#$. Complications also arise from our use of a finite subrange of natural numbers. The affected definitions are those for sequence concatenation and bag union.

4.1 Sets

As mentioned in Section 3, SAL does not support an operator equivalent to the \mathbb{P} operator of Z. However, the standard SAL installation comes with a context *set* which defines a type which is an arbitrary subset of another type given as a type parameter. A part of this context is shown below.

```

set{ T : TYPE; } : CONTEXT =
BEGIN
  Set : TYPE = [ T - > BOOLEAN ];
  ⋮
END

```

The type *Set* defined in *set* maps all elements in the parameter type *T* to a Boolean value ($- >$ is the total function symbol). The variables which are mapped to *TRUE* are regarded as being in the set whereas those mapped to *FALSE* are not. Given this representation of sets, the basic set operators are defined in context *set*. For example, set membership is defined as

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

which is a short-hand for writing

```
contains? : [[Set, T] - > BOOLEAN] = LAMBDA ( s : Set, e : T ) : s(e);
```

Similarly, set union is defined as

```
union( s1 : Set, s2 : Set ) : Set = LAMBDA ( e : T ) : s1(e) OR s2(e);
```

We have extended *set* with additional set operators that are used in Z. For example, we defined subset, \subseteq , as

```
subset?( s1 : Set, s2 : Set ) : BOOLEAN =
  FORALL ( e : T ) : contains?( s1, e ) => contains?( s2, e );
```

The other set operators can be defined similarly except for the size operator, $\#$, which is defined using the μ operator in Spivey [17] as follows.

```
#S = ( μ n : ℕ | ( ∃ f : 1 .. n ↦ S • ran f = S ) )
```

In SAL, we encode it as a function which takes a set and a natural number as arguments and returns *TRUE* when the number is the size of the set.

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

The first conjunct ensures that the total function *f* is an injection, and the second that its range is equal to the set *s*.

A Z predicate of the form $n = \#s$ could, therefore, be translated to *size?*(*s*, *n*). However, in the general case where we do not have a simple equality, an existentially quantified variable will need to be introduced. For example, $\#s < 5$ is translated to *EXISTS* (*n* : NAT) : *size?*(*s*, *n*) AND *n* < 5.

4.2 Relations

Relations are modelled, as they are in Z, as sets of tuples. This allows set operators to be used directly on relations. We define a context *relation* which has two type parameters X and Y corresponding to the domain and range types respectively.

```

relation{X, Y : TYPE; } : CONTEXT =
BEGIN
  xset : CONTEXT = set{X; };
  yset : CONTEXT = set{Y; };
  tset : CONTEXT = set{[X, Y]; };

  dom(r : tset!Set) : xset!Set =
    {x : X | EXISTS (t : [X, Y]) : tset!contains?(r, t) AND t.1 = x};
  ran(r : tset!Set) : yset!Set =
    {y : Y | EXISTS (t : [X, Y]) : tset!contains?(r, t) AND t.2 = y};
  image(r : tset!Set, xs : xset!Set) : yset!Set =
    {y : Y | EXISTS (t : [X, Y]) : tset!contains?(r, t) AND
      xset!contains?(xs, t.1) AND t.2 = y};
  :
END

```

This context instantiates the *set* context three times: with type X , type Y and the tuple type $[X, Y]$. Hence, it can refer to operators on sets of these types. This is needed in the definitions of the operators for domain (*dom*), range (*ran*) and relational image (*image*) shown. Other operators, for example, relational composition, domain and range restriction and subtraction, and relational inverse, can be defined similarly.

4.3 Functions

Although total functions are supported as a primitive in SAL, partial functions are not. To encode partial functions, we have simply followed the approach of Z, i.e., partial functions are appropriately restricted relations. This approach has the advantage that relational operators defined in the *relation* context can be used directly on partial functions, as can set operators defined in the context *set*.

Aside. An alternative approach would be to encode partial functions as SAL total functions. This would require introducing an ‘undefined’ element to all partial function range types (which could be done using the SAL datatype facility), and additional definitions of the set and relational operators. These definitions would be complicated by the fact that their argument and result types could be a combination of sets and SAL total functions. Investigating the comparative efficiency of such an approach is an area of future work.

```

function{X, Y : TYPE; } : CONTEXT =
BEGIN
  tset : CONTEXT = set{[X, Y]; };
  rel : CONTEXT = relation{X, Y; };

  pfun : TYPE = {f : tset!Set |
    FORALL (x : X, y1, y2 : Y) :
      tset!contains?(f, (x, y1)) AND
      tset!contains?(f, (x, y2)) =>
        y1 = y2};
  tfun : TYPE = {f : pfun | rel!dom(f) = {x : X | TRUE}};
  ⋮
END

```

The context *function* introduces a type *pfun* of partial functions. A type for total functions *tfun* is also shown. Similar types can be given for the other kinds of functions (injections, surjections and bijections) in Z. The functional override operator can be defined as in Spivey [17].

Function application is defined using the μ operator in Spivey [17] as follows.

$$f(x) = (\mu y : Y \mid (x, y) \in f)$$

In SAL, we encode it (within context *function*) as a function which takes a (partial) function *f* and a domain value *x* and range value *y* of *f* as arguments and returns *TRUE* when $f(x) = y$.

$$apply?(f : tset!Set, x : X, y : Y) : BOOLEAN = tset!contains?(f, (x, y));$$

Its use is similar to that of the *size?* function defined for sets. A Z predicate of the form $y = f(x)$ could be translated to *apply?*(*f*, *x*, *y*). However, in the general case where we do not have a simple equality, an existentially quantified variable will need to be introduced. For example, $f(x) < 5$ is translated to *EXISTS* (*y* : NAT) : *apply?*(*f*, *x*, *y*) AND *y* < 5.

Note that *apply?* takes a tuple set, rather than a partial function, as its first argument. This is for reasons of efficiency. Since we assume that the Z specification is correct Z, function application will only be used on variables which are functions. Therefore, it is not necessary for SAL to check the type of such a variable, i.e., check that it is a partial function, each time function application occurs.

4.4 Sequences and Bags

Sequences and bags are specified, as in Z, as appropriately restricted partial functions. In this section, we will focus on sequences; bags can be defined in SAL in a similar manner.

As well as a parameter *X* for the range type, the context *sequence* requires a variable parameter *N* of type *NATURAL* to represent the maximum number of

elements in a sequence. This number would be equal to the maximum natural number (given as a parameter to the translation process). It is used to ensure the type *seq* of sequences is finite; specifically, sequences can have at most N elements.

```

sequence{X : TYPE; N : NATURAL} : CONTEXT =
BEGIN
  nat : TYPE = [0..N];
  nat1 : TYPE = [1..N];
  nset : CONTEXT = set{nat1; };
  tset : CONTEXT = set{[nat1, X]; };
  rel : CONTEXT = relation{nat1, X; };
  fun : CONTEXT = function{nat1, X; };

  seq : TYPE = {s : fun!pfun | EXISTS (n : nat) :
                                     rel!dom(s) = {x : nat1 | x <= n}};

  cat(s1 : tset!Set, s2 : tset!Set) : tset!Set =
    LET s : tset!Set =
      {t : [nat1, X] | EXISTS (n : nat) :
                            rel!dom(s1) = {x : nat1 | x <= n} AND
                            (EXISTS (t2 : [nat1, X]) :
                              tset!contains?(s2, t2) AND
                              (t2.1 + n < N => t.1 = t2.1 + n) AND
                              (t2.1 + n >= N => t.1 = N) AND
                              t.2 = t2.2)}
    IN tset!union(s1, s);
  :
END

```

The context defines two types *nat* and *nat1* which represent the natural numbers and strictly positive natural numbers, respectively, up to N . The latter set is used for the domain of the sequence type *seq*. The predicate in the set comprehension defining this type ensures that the actual domain of any sequence is a contiguous set of numbers up to a maximum value less than or equal to N .

The definition of concatenation is shown above. Again we have used tuple sets as the types of the arguments for reasons of efficiency. The definition uses a *LET ... IN* clause to introduce a set *s* which corresponds to the second argument of the operator (*s2* above) with each of its domain values increased by the length of the first argument (*s1* above), except when the new value would exceed N , in which case the domain value is set to N .

The result of concatenation is the union of *s1* and *s*. Note that this tuple set will not be a sequence when the combined length of *s1* and *s2* exceeds N . In this case, there will be more than one tuple with N as its domain value.

In the translation of a *Z* specification, however, the result of sequence concatenation will be restricted to also be a sequence, i.e., of type *seq*. Hence, trying

to concatenate a pair of sequences which together have more than N elements will not be possible; the result would not be of type *seq*. That is, the guard in SAL will not be able to be satisfied by any assignment of values and the branch of the transition in which the concatenation occurs will not fire. This is what we should expect since otherwise the domain of a sequence could extend beyond our subrange of natural numbers.

Other sequence operators such as *head*, *tail*, *front* and *last* are readily defined as in Spivey [17].

4.5 Extending the Basic Translation Approach

To translate Z specifications using the Z mathematical toolkit, we simply import the definitions required from the appropriate contexts defined above. When a state variable is declared to be a relation, function, sequence or bag, it is declared to be a tuple set in the corresponding SAL. For example, a Z declaration $s : \text{seq } T$ is translated to $s : \text{set}\{[1..N], T\}; \text{!Set}$, where N is the maximum natural number. To ensure that s is indeed a sequence, we also make sure its assignment initially and after each operation is restricted to sequences. That is, the assignments are of the form $s \text{ IN } \{s : \text{sequence}\{T; N\}; \text{!seq} \mid \text{TRUE}\}$.

This approach is more efficient than declaring the SAL variable to be of the more complex type. When the latter is done, the tools spend a significantly larger amount of time converting the specified state transition system into the internal Binary Decision Diagram (BDD) representation on which analysis is performed.

5 Schema Calculus

Z allows the use of schemas to define other schemas via their use as predicates and types, via the operators of the schema calculus and via schema decoration and renaming. Most of this is peculiar to Z, and hence it is not directly supportable in SAL. Our translation scheme is extended to support these uses of schemas as follows.

A schema occurring as a predicate is translated to the SAL translation of its predicate. A schema occurring as a type is translated to a subtype of a SAL record type whose indices correspond to the names of the schema's variables. For example, given the following Z schema

S
$x, y : \mathbb{N}$
$x \leq y$

the predicate S is translated to $x \leq y$. The variable declaration $a : S$ is translated to the more general declaration $a : [\# x : \text{NAT}, y : \text{NAT} \#]$ and the value of a restricted by assignments (occurring in the initialisation and transitions of its module) of the form $a \text{ IN } \{s : [\# x : \text{NAT}, y : \text{NAT} \#] \mid s.x \leq s.y\}$. The notation $[\# \dots \#]$ denotes a SAL record type.

The Z notation θS constructs a schema binding of schema type S with values taken from common-named variables in the current scope. It can be translated to the SAL record literal ($\# x := x, y := y \#$), i.e., the record whose x index maps to the value of x in the current scope, and whose y index maps to the value of y in the current scope.

All expressions involving the operators of the schema calculus can be flattened to a single equivalent schema [17]. Our approach is to perform this flattening as part of the translation process. For example, given S above and

$$\frac{T}{\begin{array}{|l} y, z : \mathbb{N} \\ \hline z = y \end{array}}$$

we flatten the schema expression $S \Rightarrow T$ to

$$\frac{S \Rightarrow T}{\begin{array}{|l} x, y, z : \mathbb{N} \\ \hline x \leq y \Rightarrow y = z \end{array}}$$

and translate this resulting schema as before. Some schema calculus operators, for example the precondition operator, pre , involve quantification over state variables. The flattening process needs to be defined such that fresh variables are used instead of the state variables as described at the end of Section 3.

Similarly, a schema involving schema decoration or renaming is translated to a simple equivalent schema and then translated as before.

6 Discussion

Although Z has been encoded in a number of theorem provers including PVS [18], Isabelle/HOL [10] and EVES [13], there have been no full encodings in a model checking tool. The Alloy approach of Jackson [8] brings “Z-style specification the kind of automation offered by model checkers.” However, the Alloy language, although quite elegant and expressive in its own right, is significantly different to Z. Furthermore, the Alloy analyser is not a temporal logic model checker; it is predominantly used for checking system invariants.

The closest work to ours is that of encoding the Z extensions, CSP-OZ, CSP-Z and Object-Z, in the CSP model checker, FDR [5, 11, 9]. The input language to FDR is quite expressive allowing a straightforward translation of most Z predicates. Although FDR lacks direct support for some constructs supported by SAL, such as quantification, it includes direct support for others, such as sequences and sequence concatenation, which SAL does not.

The main difference between the two approaches is the way the model checkers are used. FDR is not a temporal logic model checker. Rather, it checks that a refinement relation holds between two models. Therefore, to check a property

holds for a given model M , we need to state the property itself as a model: one that is refined by M . This can be difficult for a novice user.

FDR does, however, support checks on state transition systems with a non-blocking semantics, as shown by Fischer and Wehrheim [5], as well as those with a blocking semantics, as shown by Kassel and Smith [9]. We regard the FDR approaches and ours to be complementary.

As with the FDR approaches, our approach has certain limitations. Firstly, all types must be finite (unless the infinite bounded model checker is used). This rules out the use of the types \mathbb{N} and \mathbb{Z} , given sets and recursive type definitions in \mathcal{Z} specifications. Furthermore, even finite types need to be small. As well as causing the state-explosion problem, large types can result in the conversion process from the input language to the analysable BDD representation becoming a bottleneck.

As an indication of the time required for model checking, the table below presents some figures for an Alternating Bit Protocol (ABP) specification adapted from the Object- \mathcal{Z} specification in Smith [14]. The translated SAL specification has 7 state variables, including 3 which are sequences (2 of these are sequences of tuples), and 8 operations. The property proved was an LTL encoding of the obvious one for this protocol: when there is no indefinite loss of messages on the message channel, all transmitted messages will eventually be received. The property was checked on a PC with a 3GHz Intel Pentium 4 processor and 512MB of RAM.

Max. natural number (N)	Verification time (seconds)	Total time (seconds)
2	0.09	3.34
4	0.18	8.39
6	1.66	92.68

For N greater than 6, the model checker did not return after 30 minutes due to the limits of the available memory being reached and the need for extensive swapping. This is due to the large BDD representation to cater for the sequences, especially those whose elements are tuples. Other case studies have allowed larger subranges for the natural numbers. Changing the tuple sequences in the ABP case study to sequences of a simple enumerated type, for example, allows us to model check with N equal to 13; the check takes around 13 minutes. However, the complexity of the ABP case study is typical of that arising in \mathcal{Z} specifications.

There are a number of ways to improve on these results. Firstly, our encoding of \mathcal{Z} in SAL is only one possible encoding. Some improvement in the above results could be made by a more efficient encoding. We have already discovered it is more efficient to declare variables by their base types (sets of tuples for functions, sequences and bags) and constrain them when assignments are made. In the ABP case study, this resulted in a five-fold decrease in total model checking time.

Secondly, the model checking tools support many optimisation features (including control of BDD variable ordering) which we have not utilised. Thirdly, they also support a variable abstraction facility which can be used to effectively

ignore variables not influencing a property we wish to prove. In addition to this, future versions of SAL are expected to support predicate abstraction [6].

Our future work will investigate all of the above options, as well as developing our own abstraction techniques. The latter will be based on recent work on data abstraction [19] and predicate abstraction [16] in the Z context.

Regarding other future work, an obvious next step is to automate our translation scheme and pretty print the SAL output to make it more familiar to Z users. These tasks should not prove too onerous given that both Z and SAL have XML representations, and the SAL tools have explicit support for pretty printing. Our approach could also be adapted to variants of Z. We are particularly interested in additionally developing support for Object-Z[15]. As well as abstraction, decomposition of specifications based on the modular structure of Object-Z may be beneficial for this work [20].

7 Conclusion

We have presented an automatable approach for translating between Z specifications and the input language to the SAL tool suite. This makes it possible to simulate and model check Z specifications. The latter includes both LTL and CTL model checking and bounded model checking with infinite types. Such temporal logic model checking requires adopting a blocking semantics for Z which is equivalent to the usual non-blocking semantics when operations are totalised. Our future work will focus on extending the limits on the size and complexity of types in Z specifications that can be supported by the approach.

Acknowledgements

Thanks to John Derrick and Kirsten Winter for discussions on aspects of this work. Luke Wildman was supported by an Australian Research Council Discovery grant, DP0343877: *Practical Tools and Techniques for the Testing of Concurrent Software Components*.

References

1. L. de Moura, S. Owre, H. Rueß, J. Rushby, N. Shankar, M. Sorea, and A. Tiwari. SAL 2. In R. Alur and D. Peled, editors, *Computer-Aided Verification (CAV 2004)*, volume 3114 of *LNCS*, pages 496–500. Springer-Verlag, 2004.
2. L. de Moura, S. Owre, and N. Shankar. The SAL language manual. Technical Report SRI-CSL-01-02 (Rev. 2), SRI International, 2003.
3. J. Derrick and G. Smith. Linear temporal logic and Z refinement. In C. Rattray, S. Maharaj, and C. Shankland, editors, *Algebraic Methodology and Software Technology (AMAST 2004)*, volume 3116 of *LNCS*, pages 117–131. Springer-Verlag, 2004.
4. J.-C. Filliâtre, S. Owre, H. Rueß, and N. Shankar. ICS: integrated canonizer and solver. In G. Berry, H. Comon, and A. Finkel, editors, *Computer-Aided Verification (CAV 2001)*, volume 2102 of *LNCS*, pages 246–249. Springer-Verlag, 2001.

5. C. Fischer and H. Wehrheim. Model-checking CSP-OZ specifications with FDR. In K. Araki, A. Galloway, and K. Taguchi, editors, *Proceedings of the 1st International Conference on Integrated Formal Methods (IFM '99)*, pages 315–334. Springer-Verlag, 1999.
6. S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *Int. Conf. on Computer Aided Verification (CAV '97)*, volume 1254 of *LNCS*, pages 72–83. Springer-Verlag, 1997.
7. D. Hazel, P. Strooper, and O. Traynor. Possum: An animator for the SUM specification language. In W. Wong and K. Leung, editors, *Asia Pacific Software Engineering Conference (APSEC '97)*, pages 42–51. IEEE Computer Society, 1997.
8. D. Jackson. Alloy: A lightweight modelling language. Technical Report 797, MIT Laboratory for Computer Science, 2000.
9. G. Kassel and G. Smith. Model checking Object-Z classes: Some experiments with FDR. In *Asia-Pacific Software Engineering Conference (APSEC 2001)*, pages 445–452. IEEE Computer Society Press, 2001.
10. Kolyang, T. Santen, and B. Wolff. A structure preserving encoding of Z in Isabelle/HOL. In J. von Wright, J. Grundy, and J. Harrison, editors, *Theorem Proving in Higher Order Logics (TPHOLs '96)*, volume 1125 of *LNCS*, pages 283–298. Springer-Verlag, 1996.
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. Rueß and L. de Moura. Bounded model checking and induction: From refutation to verification. In W. Hunt and F. Somenzi, editors, *Computer-Aided Verification (CAV 2003)*, volume 2725 of *LNCS*, pages 14–26. Springer-Verlag, 2003.
13. M. Saaltink. The Z-Eves system. In J. Bowen, M. Hinchey, and D. Till, editors, *International Conference of Z User (ZUM '97)*, volume 1212 of *LNCS*, pages 72–85. Springer-Verlag, 1997.
14. G. Smith. *An Object-Oriented Approach to Formal Specification*. PhD thesis, University of Queensland, 1992. <http://www.itee.uq.edu.au/~smith/papers/thesis.pdf>.
15. G. Smith. *The Object-Z Specification Language*. Advances in Formal Methods. Kluwer Academic Publishers, 2000.
16. G. Smith and K. Winter. Proving temporal properties of Z specifications using abstraction. In D. Bert, J.P. Bowen, S. King, and M. Waldén, editors, *3rd International Conference of Z and B Users (ZB 2003)*, volume 2651 of *LNCS*, pages 260–279. Springer-Verlag, 2003.
17. J.M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall, 2nd edition, 1992. <http://spivey.oriel.ox.ac.uk/~mike/zrm/>.
18. D. Stringer-Calvert, S. Stepney, and I. Wand. Using PVS to prove a Z refinement: A case study. In J. Fitzgerald, C. Jones, and P. Lucas, editors, *Formal Methods Europe (FME '97)*, volume 1313 of *LNCS*, pages 573–588. Springer-Verlag, 1997.
19. H. Wehrheim. Data abstraction for CSP-OZ. In J. Woodcock and J. Wing, editors, *World Congress on Formal Methods (FM '99)*, volume 1709 of *LNCS*. Springer-Verlag, 1999.
20. K. Winter and G. Smith. Compositional verification for Object-Z. In D. Bert, J.P. Bowen, S. King, and M. Waldén, editors, *3rd International Conference of Z and B Users (ZB 2003)*, volume 2651 of *LNCS*, pages 280–299. Springer-Verlag, 2003.