Building a refinement checker for Z

John Derrick, Siobhán North and Anthony J.H. Simons
Department of Computing, University of Sheffield, Sheffield, S1 4DP, UK.
J.Derrick@dcs.shef.ac.uk

In previous work we have described how refinements can be checked using a temporal logic based model-checker, and how we have built a model-checker for Z by providing a translation of Z into the SAL input language. In this paper we draw these two strands of work together and discuss how we have implemented refinement checking in our Z2SAL toolset.

The net effect of this work is that the SAL toolset can be used to check refinements between Z specifications supplied as input files written in the LaTeXmark-up. Two examples are used to illustrate the approach and compare it with a manual translation and refinement check.
Keywords: Z, refinement, model-checking, SAL.

1 Introduction

In this paper we discuss the development of tool support for refinement checking in Z. In doing so we draw on two strands of work - one on providing a translation of Z into the input language of the SAL tool-suite, and the other on using model checking to verify refinements in state-based languages.

The SAL [18] tool-suite is used in both strands, and is designed to support 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 [4] including those for LTL and CTL.

Our work on the first strand has resulted in a translation tool which converts Z specifications to a SAL module, which groups together a number of definitions including types, constants and modules for describing a state transition system. The declarations in a state schema in Z are translated into local variables in a SAL module, and any state predicates become appropriate invariants over the module and its transitions.

A SAL specification defines its behaviour by specifying transitions, thus it is natural to translate each Z operation into one branch of a guarded choice in the transitions of the SAL module. 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. This methodology has been implemented in a tool-set, as described in [9, 8].

Our work on the second strand has derived a methodology for verifying a refinement using a model-checker by combining two specifications in a special way and verifying particular CTL properties for this combination. Specifically, [21, 22, 10] described how refinements in Z and other state-based languages could be verified by encoding downward and upward simulations as CTL theorems - the simulation conditions being the standard way to verify refinements in state-based languages such as Z, B etc.

The contribution we describe in this paper is to implement this methodology in our Z to SAL translation toolkit. This extension to the tool enables two Z specifications to be input in LaTeXformat, and for

J. Derrick , E.A. Boiten, S. Reeves (Eds.):
Refinement Workshop 2011.
EPTCS 55, 2011, pp. 37–52, doi:10.4204/EPTCS.55.3

a refinement check to be performed. Internally this is achieved by translating each specification from LaTeXto a single SAL specification upon which appropriate CTL theorems can be verified using the SAL CTL witness model-checker sal-wmc.

The purpose of this paper is to describe how this is done, using two examples as way of illustration. The structure of the paper is thus as follows. In Section 2 and Section 3 we provide background on refinement and the Z to SAL translation respectively. How specifications can be combined to enable a model checker to verify a refinement is described in Section 4, and this section also describes our implementation of this methodology. To illustrate the process we present a slightly more complicated example in Section 5 and we conclude in Section 6.

2 Refinement

Data refinement [5, 6] is a formal notion of development, based around the idea that a concrete specification can be substituted for an abstract one as long as its behaviour is consistent with that defined in the abstract specification.

Each language, method or notation has its own variants. In Z, refinement is defined so that the observable behaviour of a specification is preserved. This behaviour is in terms of the operations that are performed, and their input and output values. Values of the state variables are regarded as being internal, and thus refinement can be used to change the representation of the state of a system. Hence the term *data refinement*.

In a state-based setting such as provided by Z, data refinements are verified by defining a relation (called a *retrieve relation*) between the two specifications and verifying a set of *simulation conditions*. The retrieve relation shows how a state in one specification is represented in the other. For refinement to be complete, a relation, rather than simply a function, is required [6].

In general, there are two forms the simulation conditions take, depending on the interpretation given to an operation, specifically that given to the operation's guard or precondition [6]. The two interpretations are often called the *blocking* and *non-blocking* semantics. We consider the latter, i.e., the standard, approach in this paper.

For any interpretation, there are two simulation rules for refinement which are together complete, i.e., all possible refinements can be proved with a combination of the rules. The first rule, referred to as *downward* (or *forward*) *simulation* [6, 5], requires that

initialisation the initial states of the concrete specification are related to abstract initial states

applicability the concrete operations are enabled (at minimum) in states related to abstract states where the corresponding abstract operations are enabled, and

correctness the effect of each concrete operation is consistent with the requirements of the corresponding abstract operation.

We do not consider the alternative kind of simulation known as an *upward* simulation in this paper, although there is nothing to stop the the appropriate methodology being implemented in our tool suite.

Definition 1 *A Z specification with state schema CState, initial state schema CInit and operations* $COp_1 \ldots COp_n$ *is a downward simulation of a Z specification with state schema AState, initial state schema AInit and operations* $AOp_1 \ldots AOp_n$*, if there is a retrieve relation R such that the following hold for all* $i : 1..n$*.*

1. $\forall CState \bullet CInit \Rightarrow (\exists AState \bullet AInit \land R)$
2. $\forall AState;\ CState \bullet R \land pre\,AOp_i \Rightarrow pre\,COp_i$
3. $\forall AState;\ CState;\ CState' \bullet R \land pre\,AOp_i \land COp_i \Rightarrow (\exists AState' \bullet R' \land AOp_i)$

The use of a retrieve relation allows the state spaces of the abstract and concrete specifications to be different - the retrieve relation documents their relationship. The first condition ensures appropriate initial states are related, and the second that the concrete operations are defined whenever abstract ones are (modulo the retrieve relation). The third conditions ensures that the concrete operations have an effect that is consistent with the abstract, whilst also allowing non-determinism to be reduced.

As an example refinement, consider the following simple specification. It defines two operations that add and remove an input from a set s of some given type T.

$[T]$

$\mid max : \mathbb{N}$

$A \mathrel{\hat=} [s : \mathbb{P}\,T \mid \#s \leq max]$

$AInit \mathrel{\hat=} [A' \mid s' = \varnothing]$

$$\begin{array}{|l}
\multicolumn{1}{l}{AEnter} \\ \hline
\Delta A \\
p? : T \\ \hline
\#s < max \\
p? \notin s \\
s' = s \cup \{p?\} \\ \hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{ALeave} \\ \hline
\Delta A \\
p? : T \\ \hline
p? \in s \\
s' = s \setminus \{p?\} \\ \hline
\end{array}$$

A simple data refinement replaces the set s by an injective sequence l as follows (assuming the same T and max):

$C \mathrel{\hat=} [l : iseq\ T \mid \#l \leq max]$

$CInit \mathrel{\hat=} [C' \mid l' = \langle\rangle]$

$$\begin{array}{|l}
\multicolumn{1}{l}{CEnter} \\ \hline
\Delta C \\
p? : T \\ \hline
\#l < max \\
p? \notin \operatorname{ran} l \\
l' = l \frown \langle p? \rangle \\ \hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{CLeave} \\ \hline
\Delta C \\
p? : T \\ \hline
p? \in \operatorname{ran} l \\
l' = l \upharpoonright (T \setminus \{p?\}) \\ \hline
\end{array}$$

It is easy to see that the second specification is a downward simulation of the first, using as retrieve relation the following:

$R == [A;\ C \mid s = \operatorname{ran} l]$

Our task is to build a tool that can automatically check this kind of refinement.

3 Z2SAL

The original idea of translating Z into SAL specifications was due to Smith and Wildman [20], however, our implementation has increasingly diverged from the original idea as optimization issues have been tackled. In [9, 8] we have described the basics of our implementation, which provides a bespoke parser and generator, written in Java, to translate from the LaTeX encoding of Z into the SAL input language.

A Z specification written in the state-plus-operations style is translated into a SAL finite state automaton, following a template-driven strategy with a number of associated heuristics. The Z-style of specification is preserved in this strategy, including postconditions that mix primed and unprimed variables arbitrarily, possibly asserting posterior states in non-constructive ways. We also preserve the Z mathematical toolkit's approach to the modelling of relations, functions and sequences as sets of tuples, permitting interchangeable views of functions, sequences and relations as sets.

A specification in the SAL input language consists of a collection of separate input files, known as *contexts*, in which all the declarations are placed. At least one *context* must contain the definition of a *module*, an automaton to be simulated or checked. In our translation strategy, we use a master *context* for the main Z specification and refer to other *context* files, which define the behaviour of data types from the mathematical toolkit. The master *context* consists of a prelude, declaring types and constants, followed by the main declaration of a SAL *module*, defining the finite state automata, which implements the behaviour of the Z state and operation schemas. The states of the SAL translation are created by aggregating the variables from the Z state schema, and the transitions are created by turning the operation schemas into *guarded commands*, triggered by preconditions on input and local (state) variables, and asserting postconditions on local and output variables.

The implementation of this basic strategy is presented in [8], here we recap on its salient points on two examples. Consider the first specification above. Upon translation the specification becomes a context, here called *a*.

The *built-in* types of Z are translated into finite subranges in SAL, according to a scheme described in [8]. For example, $\mathbb{N}$ is translated to:

```
NAT : TYPE = [0..4];
```

The *basic types* of Z are converted into finite, enumerated sets in SAL, consisting of three symbolic ground elements by default (but sometimes with an extra *bottom* element to deal with partiality of functions etc.). For example, the given type T is translated to:

```
T : TYPE = {T__1, T__2, T__3};
```

Where the Z specification expresses predicates involving the cardinality of sets, the translator generates a bespoke counting-context for sets containing up to the maximum number of symbolic ground elements generated for the set, as described in [8]. For this example, a `count3` context is generated; the instantiation for counting up to three elements of type `T` is named:

```
T__counter : CONTEXT = count3 {T; T__1, T__2, T__3};
```

The bounding constant *max* is an uninterpreted constant in Z, which we translate in SAL as a local variable, which can in principle take any value in the `NAT` type's range. This leads to some simulation states where the limits of the system's behaviour are reached quickly (e.g. if $max = 0$), but other states in which all three elements may be added to the set `s`.

State and initialisation schemas. The state variables from the Z state schema are translated into the *local* variables of the SAL *module*, which together constitute the aggregate states of the automaton. The state predicate is treated as follows: we define a corresponding `DEFINITION` clause to represent the schema invariant. This is achieved by introducing an extra *local* boolean variable, called `invariant__`, and declaring a formula for this in the *definition* sub-clause.

The Z initialization schema is translated in a non-constructive style into a guarded command in the `INITIALIZATION` clause of the SAL module, with the invariant as part of the guard. Thus, for the above example, we get the following translation.

```
State : MODULE =
BEGIN
 LOCAL max : NAT
 LOCAL s : set {T;} ! Set
 INPUT p? : T
 LOCAL invariant__ : BOOLEAN
 DEFINITION
   invariant__ = (T__counter ! size?(s) <= max)
 INITIALIZATION [
     s = set {T;} ! empty AND
     invariant__
   -->
 ]
```

The challenge of the translation strategy is to deal efficiently with the large vocabulary of mathematical data types such as sets, products, relations, functions, sequences and bags. The translation tool has to represent these efficiently in SAL, whilst preserving the expressiveness and flexibility of the Z language.

The basic approach is to define one or more context files for each data type in the toolkit. For example, the set mathematical data type in Z is translated into a SAL context, which models the set as a boolean-valued membership predicate on elements (following Bryant's optimal encoding of sets for translation into BDDs, [2, 3]). All other set operations are based on this encoding:

```
set {T : TYPE; } : CONTEXT = BEGIN
Set : TYPE = [T -> BOOLEAN];
empty : Set = LAMBDA (elem : T) : FALSE;
...
contains? (set : Set, elem : T) : BOOLEAN =
 set(elem);
...
union(setA : Set, setB : Set) : Set =
 LAMBDA (elem : T) : setA(elem) OR setB(elem);
...
END
```

Similar contexts are defined for the function, relation and sequence data types. Whereas Z sets and relations are modelled as boolean maps, Z functions and sequences are modelled using SAL's total functions. We adopt a totalising strategy, introducing bottom elements for types that participate in the domain or range of functions, or range of sequences.

Translating the Z operation schemas. Each operation schema in Z contributes in two ways to the SAL translation. Firstly, an operation schema may optionally declare input, or output variables (or both), which are extracted and declared in the prelude of the *module* clause, as SAL *input* and *output* variables. Secondly, the predicate of each operation schema is converted into a *guarded command* in the *transition* sub-clause, the last sub-clause in the *module* clause.

The input and output variables are understood to exist in the local scope of each operation schema, which has consequences in the translation. The SAL translation eventually substitutes the suffix '`_`''`_`' for '`!`' in the output variables, since the latter is reserved.

The computation performed by each operation schema is expressed as a *guarded command* in the *transition* sub-clause. The name of the schema is used for the transition label, which aids readability. The *guarded command* has the general syntactic form: `label : guard --> assignments`.

The guards for each transition include the primed `invariant__'` as one of the conjuncts, which asserts the state predicate in the posterior state of every transition. This, combined with the assertion of the unprimed `invariant__` in the initial state, ensures that the state predicate holds universally.

Finally, a catch-all `ELSE` branch is added to the guarded commands, to ensure that the transition relation is total (for soundness of the model checking). In practice, this allows model-checking to complete, even if the simulation blocks at a given point. Admitting the `ELSE`-transition allows simulations to pass through states in which the `invariant__'` fails to hold. Normally, this does not matter, since we can also ensure that `LOCAL` state variables are not modified, whenever the `ELSE`-transition is taken.

However, a new soundness problem emerged when admitting *bottom* values, as part of a totalising strategy for partial types. Our previous practice was to assert that `INPUT` variables never took *bottom* values, as part of the invariant. However, a loophole was discovered that allowed the system to pass through states in which the invariant did not hold (due to selecting bottom values for inputs) and then recover in the following cycle, in which the invariant held once more, but undefined values had been accepted as inputs from the previous cycle. Ideally, we would have liked to rule out invalid inputs in the `ELSE`-transition, but the SAL tools do not permit this.

Instead, we now assert both the primed `invariant__'` and unprimed `invariant__` in the guard to each transition, so closing the loophole. In practice, simulations can still pass through states where the invariant fails to hold, but they are then forced to pass through `ELSE`-transitions repeatedly, until some valid input is selected. The new translation is once again sound, but simulations may have more latent cycles. Thus for the transition component of our example we have the following:

```
TRANSITION [
  AEnter :
      T__counter ! size?(s) < max AND
      NOT set {T;} ! contains?(s, p?) AND
      s' = set {T;} ! insert(s, p?) AND
      invariant__ AND
      invariant__'
    -->
      s' IN {x : set {T;} ! Set | TRUE}
  []
  ALeave :
      set {T;} ! contains?(s, p?) AND
      s' = set {T;} ! remove(s, p?) AND
      invariant__ AND
      invariant__'
    -->
      s' IN {x : set {T;} ! Set | TRUE}
  []
  ELSE -->  s' = s
```

```
]
```

A similar translation is produced for C, this time producing a SAL input file using contexts defined to model Z sequences; see Appendix A.

4 Model-checking a refinement

A series of approaches to model-checking a refinement is described in [21, 22, 10] by Smith and Derrick with varying degrees of sophistication. They all work by taking two specifications, A and C say, and building a combined system M which encodes the behaviour of both in such a way that it is possible to write CTL properties to check the various aspects that are needed for simulation conditions to hold. There are variations to this approach as follows.

1. Three different combinations are formed, M_{init}, M_{app}, M_{corr}, one for each of the three downward simulation conditions (and a similar methodology for upward simulations);
2. One combination is formed, M, encoding all three properties to be checked in one system.

These two approaches need the candidate retrieve relation to be passed to the tool, thus a final approach is

- Additionally have the model-checker search to find if such a retrieve relation exists.

For efficiency reasons (and here to aid readability) we describe our implementation of the first approach, again restricting ourselves for brevity to downward simulations. Thus in the approach we describe, which is an abbreviated discussion of [22], here three systems are formed and if all three checks are satisfied then the concrete system is indeed a downward simulation of the abstract system with the chosen retrieve relation.

To illustrate the approach, we use the example specified above, noting that although for readability we describe it as a combination of Z schemas, in our implementation the combination acts at the level of combining SAL modules. We will combine the two specifications into one system so that we can encode the simulation conditions on the combined system, thus the combined specification includes all the abstract and concrete variables. The methodology assumes the state variables of the abstract and concrete systems are disjoint (as in fact they are in our example), but if not, then renaming is applied first to achieve it.

Initialisation. The simulation condition on initial states requires that for each concrete initial state, we are able to find an abstract initial state related by the retrieve relation R. To encode this condition we initialise M_{init} so that the concrete part of the state is initialised. Hence in our example, the combined system's state and initialisation are as follows:

M_{init}
$s : \mathbb{P}\,T$
$l : iseq\ T$

$\#s \leq max$
$\#l \leq max$

$Init_{init}$
M'_{init}

$l' = \langle\,\rangle$

To check whether an abstract initial state exists that is related to any particuar concrete initial state, we use just one operation (normally called $InitA_{init}$) which changes the abstract part of the state to an initial value and leaves the concrete part unchanged. In our example this operation is then:

$$[\Delta M_{init} \mid s' = \varnothing \wedge l' = l]$$

For any non-trivial specification $InitA_{init}$ is total, thus we do not need the "catch-all" ELSE branch in the SAL model-checker which is needed for non-total systems as described above. Then, with a system with one operation the required initialisation condition holds if the operation can be performed such that the resulting abstract and concrete parts of the state are related by R. That is, we require that there exists a next state such that $s = \operatorname{ran} l$, i.e.:

$$\mathbf{EX}\ (s = \operatorname{ran} l)$$

Applicability. Applicability conditions in refinements check the consistency of the operations' preconditions. To encode this as a temporal formula we introduce a variable *ev* to the combined state to denote the name of the last operation that occurred, and, as in [22], we use a different font for the values of type *ev*. Since we will need an additional operation to ensure totality, the combined state for an applicability check in our example will be the following:

$$\begin{array}{|l}
\multicolumn{1}{l}{M_{app}} \\
\hline
s : \mathbb{P}\, T \\
l : iseq\ T \\
ev : \{\mathsf{AEnter}, \mathsf{CEnter}, \mathsf{ALeave}, \mathsf{CLeave}, \mathsf{Choose}\} \\
\hline
\#s \leq max \\
\#l \leq max \\
\hline
\end{array}$$

The applicability condition requires that if abstract and concrete states are related by the retrieve relation, then the concrete operation must be applicable whenever the abstract one was. For the sake of efficiency we initialise to states which are already related by the retrieve relation, that is, here of the form[1]:

$$Init_{app} = [M'_{app} \mid s' = \operatorname{ran} l']$$

Operations are then specified, one for each abstract or concrete operation, each shadowing the behaviour of the original operation, and only specifying the values of that operation (eg $AEnter_{app}$ defines values for variables that originate from the abstract specification). In addition, we introduce a *Choose* operation.

$$\begin{array}{|l}
\multicolumn{1}{l}{AEnter_{app}} \\
\hline
\Delta M_{app} \\
p? : T \\
\hline
\#s < max \\
p? \notin s \\
s' = s \cup \{p?\} \\
ev' = \mathsf{AEnter} \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\multicolumn{1}{l}{ALeave_{app}} \\
\hline
\Delta M_{app} \\
p? : T \\
\hline
p? \in s \\
s' = s \setminus \{p?\} \\
ev' = \mathsf{ALeave} \\
\hline
\end{array}$$

[1] The value of *ev* can be left underspecified.

$$\begin{array}{|l} \multicolumn{1}{l}{\underline{\mathit{CEnter}_{app}}} \\ \Delta M_{app} \\ p? : T \\ \hline \# l < max \\ p? \notin \operatorname{ran} l \\ l' = l \frown \langle p? \rangle \\ ev' = \mathsf{CEnter} \\ \hline \end{array}$$

$$\begin{array}{|l} \multicolumn{1}{l}{\underline{\mathit{CLeave}_{app}}} \\ \Delta M_{app} \\ p? : T \\ \hline p? \in \operatorname{ran} l \\ l' = l \upharpoonright (T \setminus \{p?\}) \\ ev' = \mathsf{CLeave} \\ \hline \end{array}$$

$$Choose_{app} \mathrel{\widehat=} [\Delta M_{app} \mid ev' = \mathsf{Choose}]$$

The applicability check can now be written in CTL as follows.

$$(\mathbf{EX}\ (ev = \mathsf{AEnter}) \Rightarrow \mathbf{EX}\ (ev = \mathsf{CEnter})) \wedge (\mathbf{EX}\ (ev = \mathsf{ALeave}) \Rightarrow \mathbf{EX}\ (ev = \mathsf{CLeave}))$$

Correctness. A similar methodology is applied to check the correctness condition, and here we use the same combined state and initialisation as used for applicability, as well as the same totalisation *Choose*:

$$M_{corr} \mathrel{\widehat=} M_{app}$$
$$Init_{corr} \mathrel{\widehat=} Init_{app}$$
$$Choose_{corr} \mathrel{\widehat=} Choose_{app}$$

The downward simulation correctness condition requires that any after-state of a concrete operation is related by the retrieve relation to an after-state of the abstract operation. To encode this correctly one needs to ensure that each operation in the combined state does not alter variables from the portion of state it is not representing. Thus we have operations of the form:

$$AOp_{corr} \mathrel{\widehat=} [AOp_{app} \mid l' = l]$$
$$COp_{corr} \mathrel{\widehat=} [COp_{app} \mid s' = s]$$

This allows us to perform the operations COp_{corr} and AOp_{corr} in sequence so that the abstract part of the final state reached is identical to that which could have been reached by performing only AOp_{corr}, and the concrete part is identical to that which could have been reached by performing only COp_{corr}. The correctness condition is then:

$$\begin{array}{l} \mathbf{EX}\ (ev = \mathsf{AEnter}) \Rightarrow \mathbf{AX}\ (ev = \mathsf{CEnter} \Rightarrow \mathbf{EX}\ (ev = \mathsf{AEnter} \wedge R)) \\ \quad \wedge \\ \mathbf{EX}\ (ev = \mathsf{ALeave}) \Rightarrow \mathbf{AX}\ (ev = \mathsf{CLeave} \Rightarrow \mathbf{EX}\ (ev = \mathsf{ALeave} \wedge R)) \end{array}$$

Implementation in SAL. The above is described in terms of combinations of Z specifications, although, of course, it is implemented in terms of combining SAL modules in our tool-suite.

The process of combining the two LaTeX Z specifications plus retrieve relation into a single SAL specification in order to check the downward simulation conditions was achieved using an extension to our Z to SAL parser. When translating a single Z specification to SAL our compiler first parses the Z, then transforms it into an internal SAL representation and finally the SAL file is generated. In extending the tool-set to combine two specifications in the manner described above the major modification was to the middle phase, the transformation from Z to SAL. Nevertheless the process of parsing two specifications sequentially required some modification for a number of issues.

For example, declarations in the abstract and concrete state schemas need to be checked to ensure that they contain distinct identifiers, but where types and constants occur in both specifications they have to be identical to cope with SAL's strict type checking. Neither of these problems caused much difficulty since, e.g., there already was a mechanism to ensure that a variable name used in two different Z operations did not lead to a conflict in the SAL translations (where all variables had the same scope). In our simple, single specification, translation this is achieved by prefixing the variable name by the name of its transition wherever an ambiguous name is detected and the same mechanism was used when producing a single combined specification. The only modification was that variables from axiomatic definitions were prefixed by the specification name rather than the transition name.

Treating types declared in two different specifications as the same was slightly more complicated as types from the abstract specification occurring in the concrete had to be identified. In our single translation types are canonical, for reasons explained in [8] and this had to be maintained in the combined translation without the parser rejecting a concrete specification which contains an apparently second declaration of a type which has been declared in the abstract specification. This problem also occurred with identical constants in both specifications.

Having parsed the two specifications, the retrieve relation is read in and parsed as a single Z operation with everything from both the abstract and concrete specifications in scope.

The process of transforming a single Z specification into SAL consists of fixing the finite ranges of all the types, eliminating redundant predicates, giving initial values to all the constants and identifying any named types that would have to be generated in SAL. In transforming two specifications into one SAL specification the finite ranges were fixed to the widest required by either specification but apart from that the process is essentially simple. The two sets of initial declarations were combined and the two lists of operation schemas in Z became a single list of transitions in SAL. The resulting structure is that of our internal representation of any SAL specification and a SAL text file could be generated from it in the standard way.

The result produced by our tool-kit of the two SAL modules for the correctness condition is given in Appendix B. It is then a trivial matter to check the required theorem on it.

5 A further example

A further example, which provides a comparative analysis with the manual approach to refinement checking, is given by the following (now standard) example.

The Marlowe box office allows customers to book tickets in advance using the *Book* operation – *mpool* is the set of tickets, and if a ticket is available ($mpool \neq \varnothing$) then one is allocated then and there. When the customer arrives, operation *Arrive* presents this ticket. *Ticket* is the set of all tickets, and a free type adds a possibly null ticket, and *tkt* models which tickets have been allocated.

$[Ticket]$ $\qquad\qquad$ $MTicket ::= null \mid ticket\langle\langle Ticket\rangle\rangle$

Marlowe

$mpool : \mathbb{P}\, Ticket$
$tkt : MTicket$

MInit

$Marlowe$

$tkt = null$

$$\begin{array}{|l}
\multicolumn{1}{l}{\underline{\mathit{MBook}}} \\
\Delta \mathit{Marlowe} \\
\hline
\mathit{tkt} = \mathit{null} \\
\mathit{mpool} \neq \varnothing \\
\mathit{tkt}' \neq \mathit{null} \\
\mathit{ticket}^{-1}(\mathit{tkt}') \in \mathit{mpool} \\
\mathit{mpool}' = \mathit{mpool} \setminus \{\mathit{ticket}^{-1}(\mathit{tkt}')\} \\
\hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\underline{\mathit{MArrive}}} \\
\Delta \mathit{Marlowe} \\
t! : \mathit{Ticket} \\
\hline
\mathit{tkt} \neq \mathit{null} \\
\mathit{tkt}' = \mathit{null} \\
t! = \mathit{ticket}^{-1}(\mathit{tkt}) \\
\mathit{mpool}' = \mathit{mpool} \\
\hline
\end{array}$$

In an alternative description - the Kurbel - customers still book tickets in advance. However, now if there is an available ticket then this is simply recorded by the operation *Book* provided the customer has not already booked. Only when the customer actually arrives at the box office, is the ticket allocated by *Arrive*. *kpool* is the pool of tickets and *bkd* denotes whether a ticket has been booked.

$$\mathit{Booked} ::= \mathit{yes} \mid \mathit{no} \qquad\qquad [\mathit{Ticket}]$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\underline{\mathit{Kurbel}}} \\
\mathit{kpool} : \mathbb{P}\,\mathit{Ticket} \\
\mathit{bkd} : \mathit{Booked} \\
\hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\underline{\mathit{KInit}}} \\
\mathit{Kurbel} \\
\hline
\mathit{bkd} = \mathit{no} \\
\hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\underline{\mathit{KBook}}} \\
\Delta \mathit{Kurbel} \\
\hline
\mathit{bkd} = \mathit{no} \\
\mathit{kpool} \neq \varnothing \\
\mathit{bkd}' = \mathit{yes} \\
\mathit{kpool}' = \mathit{kpool} \\
\hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\underline{\mathit{KArrive}}} \\
\Delta \mathit{Kurbel} \\
t! : \mathit{Ticket} \\
\hline
\mathit{bkd} = \mathit{yes} \\
\mathit{kpool} \neq \varnothing \\
\mathit{bkd}' = \mathit{no} \\
t! \in \mathit{kpool} \\
\mathit{kpool}' = \mathit{kpool} \setminus \{t!\} \\
\hline
\end{array}$$

The Marlowe specification is a downward simulation of the Kurbel (and in fact Kurbel is an upward simulation of Marlowe). The retrieve relation linking the two that one is tempted to write down is the following:

$$\begin{array}{|l}
\multicolumn{1}{l}{\underline{R}} \\
\mathit{Marlowe} \\
\mathit{Kurbel} \\
\hline
\mathit{bkd} = \mathit{no} \Rightarrow \mathit{tkt} = \mathit{null} \wedge \mathit{kpool} = \mathit{mpool} \\
\mathit{bkd} = \mathit{yes} \Rightarrow \mathit{tkt} \neq \mathit{null} \wedge \mathit{kpool} = (\mathit{mpool} \cup \{\mathit{ticket}^{-1}(\mathit{tkt})\}) \\
\hline
\end{array}$$

In [22] a hand translation of these specifications into SAL was performed, followed by a merging into a single SAL specification - also performed by hand. A natural question to ask therefore is to what extent our automatic translation and combination is comparable with the manual process. The above candidate retrieve relation was used in the manual process, which revealed a failure to pass the necessary refinement conditions - both specification and retrieve relation needing adjustment before the Marlowe was shown to be a valid downward simulation of the Kurbel.

It is interesting to note that the results of the automatic translation were broadly comparable to the manual one, and in fact due to our optimizations show slight reduction in state space size (see table below). The automatic combination essentially identical to the manual. The latter is to be expected - the combination is essentially simple once the specifications have been converted into SAL.

Step	Manual	Auto
0	1344	840
1	3360	6072
2	8544	6072
3	8544	6072
4	8544	6072

6 Conclusion

This work contributes on one hand to the strand of work on providing tool support for Z, and on the other hand to automatic refinement checking.

Recent work on providing tool support for Z includes the CZT (Community Z Tools) project [16], our own work [9], as well as related work such as ProZ [19], which adapts the ProB [15] tool for the Z notation.

Work on automatic refinement checking includes that of Bolton who has used Alloy to verify data refinements in Z [1]. There have also been a number of encoding of subsets of Z-based languages in the CSP model checker FDR [11, 17, 14], which checks that refinement holds between two specifications by comparing the failures/divergences semantics of the specifications; and simulation-based refinement can be encoded as a failures/divergences check [7, 13, 12].

Clearly there is much to be done in terms of further work here, not least some performance characterisations of when such an approach produces feasible state spaces.

Acknowledgements: This work was done as part of collaborative work with Graeme Smith and Luke Wildman of the University of Queensland. Tim Miller also gave valuable advice on the current CZT tools.

References

[1] C. Bolton (2005): *Using the Alloy Analyzer to Verify Data Refinement in Z*. *Electronic Notes in Theoretical Computer Science* 137(2), pp. 23–44, doi:10.1016/j.entcs.2005.04.023.

[2] Randal E. Bryant (1986): *Graph-Based Algorithms for Boolean Function Manipulation*. *IEEE Trans. Computers* 35(8), pp. 677–691.

[3] Randal E. Bryant (1992): *Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams*. *ACM Comput. Surv.* 24(3), pp. 293–318.

[4] E. Clarke, O. Grumberg & D. Peled (2000): *Model Checking*. MIT Press.

[5] W.-P. de Roever & K. Engelhardt (1998): *Data Refinement: Model-Oriented Proof Methods and their Comparison*. CUP, doi:10.1017/CBO9780511663079.

[6] J. Derrick & E. Boiten (2001): *Refinement in Z and Object-Z, Foundations and Advanced Applications*. Springer-Verlag, doi:10.1007/978-1-4471-0257-1.

[7] J. Derrick & E.A. Boiten (2003): *Relational Concurrent Refinement*. *Formal Aspects of Computing* 15(2-3), pp. 182–214, doi:10.1007/s00165-003-0007-4.

[8] J. Derrick, S. North & A.J. H. Simons (2008): *Z2SAL - Building a Model Checker for Z*. In E. Börger, M.J. Butler, J.P. Bowen & P. Boca, editors: *ABZ*, *Lecture Notes in Computer Science* 5238, Springer, pp. 280–293, doi:10.1007/978-3-540-87603-8.

[9] J. Derrick, S. North & T. Simons (2006): *Issues in Implementing a Model Checker for Z*. In Z. Liu & J. He, editors: *ICFEM*, *Lecture Notes in Computer Science* 4260, Springer, pp. 678–696, doi:10.1007/11901433-37.

[10] John Derrick & Graeme Smith (2008): *Using Model Checking to Automatically Find Retrieve Relations*. *Electr. Notes Theor. Comput. Sci.* 201, pp. 155–175, doi:10.1016/j.entcs.2008.02.019.

[11] C. Fischer & H. Wehrheim (1999): *Model-checking CSP-OZ specifications with FDR*. In K. Araki, A. Galloway & K. Taguchi, editors: *International Conference on Integrated Formal Methods (IFM'99)*, Springer-Verlag, pp. 315–334.

[12] J. He (1989): *Process refinement*. In J. McDermid, editor: *The Theory and Practice of Refinement*, Butterworths.

[13] M. Josephs (1988): *A state-based approach to communicating processes*. *Distributed Computing* 3, pp. 9–18.

[14] G. Kassel & G. Smith (2001): *Model Checking Object-Z Classes: Some Experiments with FDR*. In: *Asia-Pacific Software Engineering Conference (APSEC 2001)*, IEEE Computer Society Press.

[15] M. Leuschel & M. Butler (2005): *Automatic Refinement Checking for B*. In K. Lau & R. Banach, editors: *International Conference on Formal Engineering Methods, ICFEM 2005*, *LNCS* 3785, Springer-Verlag, pp. 345–359.

[16] Tim Miller, Leo Freitas, Petra Malik & Mark Utting (2005): *CZT Support for Z Extensions*. In Judi Romijn, Graeme Smith & Jaco Pol, editors: *Integrated Formal Methods, IFM 2005*, *LNCS* 3771, Springer-Verlag, pp. 227–245.

[17] A. Mota & A. Sampaio (2001): *Model-checking CSP-Z: strategy, tool support and industrial application*. *Science of Computer Programming* 40, pp. 59–96.

[18] L. de Moura, S. Owre & N. Shankar (2003): *The SAL language manual*. Technical Report SRI-CSL-01-02 (Rev.2), SRI International.

[19] Daniel Plagge & Michael Leuschel (2007): *Validating Z Specifications using the ProB Animator and Model Checker*. *LNCS* 4591, pp. 480–500.

[20] G. Smith & L. Wildman (2005): *Model checking Z specifications using SAL*. In H. Treharne, S. King, M. Henson & S. Schneider, editors: *International Conference of Z and B Users*, *LNCS* 3455, Springer-Verlag, pp. 87–105.

[21] Graeme Smith & John Derrick (2005): *Model Checking Downward Simulations*. *Electr. Notes Theor. Comput. Sci.* 137(2), pp. 205–224, doi:10.1016/j.entcs.2005.04.032.

[22] Graeme Smith & John Derrick (2006): *Verifying data refinements using a model checker*. *Formal Asp. Comput.* 18(3), pp. 264–287, doi:10.1007/s00165-006-0002-7.

Appendix A

Here is the SAL translation of the concrete specification from Section 2

```
c : CONTEXT = BEGIN
NAT : TYPE = [0..4];
T : TYPE = {T__1, T__2, T__3, T__B};
```

```
State : MODULE =
BEGIN
 LOCAL max : NAT
 LOCAL l : sequence {T; T__B, 3} ! Sequence
 INPUT p? : T
 LOCAL invariant__ : BOOLEAN
 DEFINITION
   invariant__ = (sequence {T; T__B, 3} ! injective?(l) AND
     sequence {T; T__B, 3} ! valid?(l) AND
     p? /= T__B AND
     sequence {T; T__B, 3} ! size?(l) <= max)
 INITIALIZATION [
     l = sequence {T; T__B, 3} ! empty AND invariant__
   -->
 ]
 TRANSITION [
   CEnter :
       sequence {T; T__B, 3} ! size?(l) < max AND
       NOT set {T;} ! contains?(sequence {T; T__B, 3} ! range(l), p?) AND
       l' = sequence {T; T__B, 3} ! append(l, p?) AND
       invariant__ AND
       invariant__'
     -->
       l' IN {x : sequence {T; T__B, 3} ! Sequence | TRUE}
   []
   CLeave :
       set {T;} ! contains?(sequence {T; T__B, 3} ! range(l), p?) AND
       l' = sequence {T; T__B, 3} ! remove(l,p?) AND
       invariant__ AND
       invariant__'
     -->
       l' IN {x : sequence {T; T__B, 3} ! Sequence | TRUE}
   []
   ELSE -->    l' = l
 ]
END;
END
```

Appendix B

The result of automatically combining the two SAL modules from Z specifications given in Section 2:

```
r2corr : CONTEXT = BEGIN

NAT : TYPE = [0..5];
T : TYPE = {T__1, T__2, T__3, T__B};
EVENT__ : TYPE = {AEnter, ALeave, CEnter, CLeave, Choose__};
T__counter : CONTEXT = count4 {T; T__1, T__2, T__3, T__B};

State : MODULE =
BEGIN
 LOCAL max : NAT
```

```
LOCAL max : NAT
LOCAL s : set {T;} ! Set
INPUT p? : T
LOCAL l : sequence {T; T__B, 3} ! Sequence
LOCAL ev__ : EVENT__
LOCAL invariant__ : BOOLEAN
DEFINITION
  invariant__ =
    (T__counter ! size?(s) <= max AND
    sequence {T; T__B, 3} ! injective?(l) AND
    p? /= T__B AND
    sequence {T; T__B, 3} ! valid?(l) AND
    sequence {T; T__B, 3} ! size?(l) <= max)
INITIALIZATION [
    (s = sequence {T; T__B, 3} ! range(l))
  -->
]
TRANSITION [
  AEnter :
      T__counter ! size?(s) < max AND
      NOT set {T;} ! contains?(s, p?) AND
      s' = set {T;} ! insert(s, p?) AND
      ev__' = AEnter AND
      invariant__ AND
      invariant__'
    -->
      s' IN {x : set {T;} ! Set | TRUE};
      l' IN {x : sequence {T; T__B, 3} ! Sequence | TRUE};
      ev__' IN {x : EVENT__ | TRUE}
  []
  ALeave :
      set {T;} ! contains?(s, p?) AND
      s' = set {T;} ! remove(s, p?) AND
      ev__' = ALeave AND
      invariant__ AND
      invariant__'
    -->
      s' IN {x : set {T;} ! Set | TRUE};
      l' IN {x : sequence {T; T__B, 3} ! Sequence | TRUE};
      ev__' IN {x : EVENT__ | TRUE}
  []
  CEnter :
      sequence {T; T__B, 3} ! size?(l) < max AND
      NOT set {T;} ! contains?(sequence {T; T__B, 3} ! range(l), p?) AND
      l' = sequence {T; T__B, 3} ! append(l, p?) AND
      ev__' = CEnter AND
      invariant__ AND
      invariant__'
    -->
      s' IN {x : set {T;} ! Set | TRUE};
      l' IN {x : sequence {T; T__B, 3} ! Sequence | TRUE};
      ev__' IN {x : EVENT__ | TRUE}
```

```
    []
    CLeave :
        set {T;} ! contains?(sequence {T; T__B, 3} ! range(l), p?) AND
        l' = sequence {T; T__B, 3} ! remove(l,p?) AND
        ev__' = CLeave AND
        invariant__ AND
        invariant__'
      -->
        s' IN {x : set {T;} ! Set | TRUE};
        l' IN {x : sequence {T; T__B, 3} ! Sequence | TRUE};
        ev__' IN {x : EVENT__ | TRUE}
    []
    Choose__ :
        ev__' = Choose__ AND
        invariant__ AND
        invariant__'
      -->
        s' IN {x : set {T;} ! Set | TRUE};
        l' IN {x : sequence {T; T__B, 3} ! Sequence | TRUE};
        ev__' IN {x : EVENT__ | TRUE}
 ]
END;
END
```