This is a repository copy of \textit{Z2SAL-building a model checker for Z}.

White Rose Research Online URL for this paper:  
http://eprints.whiterose.ac.uk/78798/

\textbf{Proceedings Paper:}  
Borger, E., Butler, M., Bowen, J.P. and Boca, P., (eds.) Abstract State Machines, B and Z,  
ISBN 978-3-540-87602-1

\textbf{Reuse}  
Unless indicated otherwise, fulltext items are protected by copyright with all rights reserved. The copyright  
exception in section 29 of the Copyright, Designs and Patents Act 1988 allows the making of a single copy  
solely for the purpose of non-commercial research or private study within the limits of fair dealing. The  
publisher or other rights-holder may allow further reproduction and re-use of this version - refer to the White  
Rose Research Online record for this item. Where records identify the publisher as the copyright holder,  
users can verify any specific terms of use on the publisher's website.

\textbf{Takedown}  
If you consider content in White Rose Research Online to be in breach of UK law, please notify us by  
emailing eprints@whiterose.ac.uk including the URL of the record and the reason for the withdrawal request.
This is an author produced version of a paper published in Abstract State Machines, B and Z, Proceedings of.

White Rose Research Online URL for this paper: http://eprints.whiterose.ac.uk/78798

Published paper

http://dx.doi.org/10.1007/978-3-540-87603-8_22
Z2SAL - building a model checker for Z

John Derrick, Siobhán North and Anthony J. H. Simons

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

Abstract. In this paper we discuss our progress towards building a model-checker for Z. The approach we take in our Z2SAL project involves implementing a translation from Z into the SAL input language, upon which the SAL toolset can be applied. The toolset includes a number of model-checkers together with a simulator. In this paper we discuss our progress towards implementing as complete as a translation as possible, the limitations we have reached and the optimizations we have made. We illustrate with a small example.

Keywords: Z, model-checking, SAL.

1 Introduction

Z has, for some time, lagged behind other specification languages in its provision of tools. There are a number of reasons for this, although most are connected with the language itself and its semantics: its expressivity has made it more difficult to build tractable tools. However, recently a number of projects have begun to tackle this deficiency. These include the CZT (Community Z Tools) project [8], our own work [6], as well as related work such as ProZ [9], which adapts the ProB [7] tool for the Z notation, and that of Bolton who has used Alloy to verify data refinements in Z [1].

Our concern is that of providing a model-checking [4] tool via translation of Z specifications into the input language of an appropriate toolset. Here we choose the SAL [5] tool-suite, 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 including those for LTL and CTL.

The original idea of translating Z into SAL specifications was due to Smith and Wildman [10]. In [6] we described the basics of our implementation, which essentially is a Java based compiler of a subset of Z into SAL. Here we discuss the implementation in broader scope, describing how different parts of the Z mathematical toolkit are translated.

The aim of [10] 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. Given this theoretical basis (the translation in [10] was not optimized for implementation) the actual implementation has preserved this general approach but has increasingly diverged as optimization issues have been tackled.

The general scheme of the translation is that a Z specification is translated 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.

As would be expected the work to be done in such a translation is in translating the mathematical toolkit; yet a naive translation quickly produces SAL input which is infeasible to simulate or model-check. Thus our work has optimized the translation as much as possible by using appropriate combinations of the inbuilt SAL types. Much of our discussion below concerns these issues.

The structure of the paper is as follows. The basic architecture of our implementation is described in Section 2. Section 3 gives an overview of the approach to translation, and more specifics about translating the mathematical toolkit is given in Section 4. Finally, Section 5 discusses the use of the tool.

2 The Z2SAL architecture

The Z2SAL tool is currently implemented in Java directly (rather than using the CZT components) in order to rapidly prototype and evaluate a number of ideas. At present it works by scanning a Z LaTeX source file in a single pass into something which is basically a list of schema class instances with associated classes representing the named constants, types and variables and one expression structure to represent the restrictions derived from the constraints in the axiomatic definitions. The use of a single scan is feasible because we restrict the Z we accept to definition before use of all identifiers (which in practice is not a significant limitation).

Having parsed the Z a certain amount of optimisation is performed with a view to producing efficient SAL. Given we are producing output for use with a model-checker, the first step is to turn any aspect that might be unbounded into a finite size. In order to keep the state space to a minimum the size of any basic types we use is also restricted as far as possible. Thus if Z is used in Z a type called INT will be declared in the SAL output which ranges between one less than the smallest constant used in the Z to one more than the largest. Given types from the Z specification have to be assigned explicit values in the SAL
output, and by default they are set to consist of a type with three elements but this can be varied by supplying the parser with a different value as a parameter.

The combination of small fixed ranges for basic types and giving all constants a value allows us to optimize expressions derived from both the constraints of the schemas and the axiomatic definitions. The latter are optimized first. Z expressions are initially parsed into a conventional tree but this is immediately transformed into a list of subtrees which represents the series of conjoined predicates. This is both a natural way to represent the predicate in a Z schema and a convenient structure to modify when combining predicates derived from different sources, something we have to do in various places starting with the predicates of all the axiomatic definitions.

This combined predicate is scanned both to eliminate redundant predicates and to restrict all the variables constrained by any axiomatic definitions to as small a range as possible. In an earlier release (see [6]) we dealt with these variables by giving them arbitrary values and treating them as constants, however, it turned out that the properties of the resulting SAL were too dependent on the translator’s choice. Thus here we restrict their type as far as possible using the conditions imposed by the axiomatic definitions constraints. Sometimes the type restriction process allows a variable to be restricted to a single value, and at that point it is transformed into a constant, removing the need for a SAL predicate, and this in turn can lead to further optimization. If, during this process a Z predicate proves to be unsatisfiable, the translator terminates under the assumption that the Z specification is erroneous. Any predicates remaining, ones that cannot be converted into restrictions of type, are then added to the state schema predicates and the variables are added to those of the state schema.

Next, all the variables are scanned to identify any types that must be constructed in SAL. For example (see Appendix A) to express rented : PERSON <- TITLE in SAL we have to create a symbolic name for a SAL product type PERSON_ X_ TITLE : TYPE = [PERSON, TITLE] for use in the declaration rented: set(PERSON_ X_ TITLE)!Set since the product type cannot be used directly in the instantiation of the set context. So, unless the type is named elsewhere, we have to generate an artificial name for it, which is declared early in the SAL output. Finally the schema predicates are scanned, to optimize them and also to identify any extra declarations we need, including those required by a count context (see below), which is used to express set cardinality.

Having cleaned up the Z as far as possible, in the manner just described, the SAL is generated. First the named types and constants are generated. The order of these is insignificant in SAL but, from a human point of view, it is useful to have them in the same order as they appear in the original Z. To this end a list of identifiers in order of first use is kept by the lexical analyzer and can be used to order the initial declarations in the SAL correctly. After this the types and counters generated by the translator are generated, the state invariant and finally the operation schemas, transformed into transitions, are exported. Here again we maintain the same order for readability.
3 Overview of the translation strategy

A specification in the SAL input language consists of a number of context files, in which all the declarations are placed. In our translation, we use a master CONTEXT for the main Z specification and refer to other context files, which define the behaviour of the mathematical toolkit. The master context includes declarations of the basic types and constants; and declares the finite state automaton, known as a MODULE, which represents the Z state schema as a collection of local state variables and the Z operation schemas as transitions of the automaton, acting on the local, input and output variables.

Types: We adopt the following scheme for the translation of Z types:

<table>
<thead>
<tr>
<th>Z</th>
<th>SAL translation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Built-in types such as (\mathbb{N}) etc</td>
<td>Finite subranges of SAL equivalent types</td>
</tr>
<tr>
<td>Given sets</td>
<td>Enumerated finite type in SAL</td>
</tr>
<tr>
<td>Free types</td>
<td>Constructed type in SAL</td>
</tr>
</tbody>
</table>

So, in the example in Appendix A, the given type \([\textit{PERSON}]\) is translated to: \texttt{PERSON : \textit{TYPE} = \{\textit{PERSON__1, PERSON__2, PERSON__3}\}}; SAL constructed types may be recursive, but some implementations of the SAL tools cannot process recursive definitions because they expand all recursive constructions infinitely as the definitions are compiled to BDDs. This problem may be fixed in future releases of the SAL toolset. We assume for the moment that the input does not contain recursively-defined data types.

Constants and axiomatic definitions: The most direct way to translate constants and axiomatic definitions is to declare them as a SAL local variable, within the module clause. However, this multiplies the state space of the system, but with many of the states being over-constrained. We thus attempt to identify suitable exact values for constants in the translation, which can often be done by looking at predicates which involve the constant elsewhere in the specification.

State and initialisation schemas: The Z state schema is converted into LOCAL variable declarations within the MODULE clause, with a corresponding DEFINITION clause to represent the schema invariant. This defines a local abbreviation \texttt{invariant__} for a boolean equation expressing constraints on the values of local variables; and could also include further constraints resulting from axiomatic definitions. The Z initialisation schema is translated in a non-constructive style into a guarded command in the INITIALIZE clause of the SAL module, with the invariant as part of the guard.

Operation Schemas: The translation of Z operation schemas into SAL consists of three stages. First, all input and output variables are extracted and converted into their cognate forms in SAL. In addition, each operation schema is converted into a single transition, such that the Z schema predicate becomes the guard for the guarded command, expressing the relationship between the \textit{primed} and \textit{unprimed} versions of variables. The primed \texttt{invariant__’} is added to the guard, to indicate that the invariant must hold after firing each transition. 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 Z, input and output variables are locally scoped to each operation schema, but exist in the same scope in the SAL MODULE clause, therefore we prefix input and outputs by the name of the Z schema from which they originate (additionally, outputs use an underscore decoration rather than a !).

Thus the example in Appendix A is translated into the following SAL fragment (we have elided parts of the translation - the complete output is in Appendix B). In particular, the assignment of updated values occurs before the --> in the transitions in this non-constructive style of encoding.

example : CONTEXT = BEGIN
PERSON : TYPE = (PERSON__1, PERSON__2, PERSON__3);
NAT : TYPE = [0..4];
...
State : MODULE =
BEGIN
LOCAL members : set (PERSON;) ! Set
LOCAL rented : set (PERSON__X__TITLE;) ! Set
LOCAL stockLevel : [ TITLE -> NAT ]
INPUT RentVideo_p? : PERSON
INPUT AddTitle_t? : TITLE
...
OUTPUT CopiesOut_copies_ : NAT
LOCAL invariant_ : BOOLEAN
DEFINITION
invariant_ = ...
INITIALIZATION []
members = set (PERSON;) ! empty AND
stockLevel = function(TITLE, NAT; TITLE__B, 4) !empty AND invariant_ -->
]
TRANSITION [
RentVideo :
...
--> members' IN { x : set (PERSON;) ! Set | TRUE};
rented' IN { x : set (PERSON__X__TITLE;) ! Set | TRUE};
stockLevel' IN { x : [ TITLE -> NAT ] | TRUE}
[]
AddTitle :
...
ELSE -->
END

4 The mathematical toolkit

The heart of the translation deals with the Z mathematical toolkit, which provides a rich vocabulary of mathematical data types, including sets, products, relations, functions, sequences and bags. The challenge is to represent these types, and the operations that act upon them, efficiently in SAL, whilst still preserving
the expressiveness of Z. The basic approach is to define one or more context files
for each data type in the toolkit. For example, the set-context implements a set
as a function from elements to BOOLEAN. This is a standard encoding for sets,
optimized for symbolic model checkers that use BDDs as the core representation
[2, 3]. A set is not a single, monolithic entity, but rather a polythetic membership
predicate over all of its elements. Set operations are defined in the following
style:

\[
\text{union}(\text{setA} : \text{Set}, \text{setB} : \text{Set}) : \text{Set} = \\
\text{LAMBDA} (\text{elem} : T) : \text{setA}(\text{elem}) \text{ OR setB}(\text{elem});
\]

However, the encoding causes problems when calculating the cardinality of
sets. In [10] cardinality is defined as the search for a relation between sets and
natural numbers. However, we found that this was inefficient when implemented
[6]. We also tried two other encodings for counted sets, which relied on brute-
force enumeration of elements. Since then, we have removed the \text{size?} function
altogether from the standard, efficient set-context and provide this in a separa-
tely generated \text{countN-context}, parameterized over arbitrary \text{N} elements. This
separation of concerns provides brute-force counting only when the specification
actually requires this. For example, three possible elements may be counted by
the \text{size?} function from the \text{count3-context}:

\[
\text{count3}(T : \text{TYPE}; e1, e2, e3 : T) : \text{CONTEXT} = \\
\text{BEGIN} \\
\text{Set} : \text{TYPE} = [T \to \text{BOOLEAN}]; \\
\text{size?} (\text{set} : \text{Set}) : \text{NATURAL} = \\
\text{IF set}(e1) \text{ THEN } 1 \text{ ELSE } 0 \text{ ENDIF } + \\
\text{IF set}(e2) \text{ THEN } 1 \text{ ELSE } 0 \text{ ENDIF } + \\
\text{IF set}(e3) \text{ THEN } 1 \text{ ELSE } 0 \text{ ENDIF}; \\
\text{END}
\]

A similar strategy was adopted for encoding relations. The initial idea was
to define relations as sets of pairs. We found that the SAL parser rejected type-
instantiation with anything other than a simple symbolic type name, which
initially limited the use of constructed product-types, but we later adopted the
work-around of defining symbolic aliases for each product-type.

The standard relation-context is parameterized over the domain and range
element-types, and internally defines two pair-types, two set-types for the domain
and range, and two set-types for the relation and inverse relation, followed by
operations on relations:

\[
\text{relation}(X, Y : \text{TYPE}; ) : \text{CONTEXT} = \\
\text{BEGIN} \\
\text{XY} : \text{TYPE} = [X, Y]; \\
\text{YX} : \text{TYPE} = [Y, X]; \\
\text{Domain} : \text{TYPE} = [X \to \text{BOOLEAN}]; \\
\text{Range} : \text{TYPE} = [Y \to \text{BOOLEAN}]; \\
\text{Relation} : \text{TYPE} = [XY \to \text{BOOLEAN}];
\]
Inverse : TYPE = [YX -> BOOLEAN];

... domain (rel : Relation) : Domain =
    LAMBDA (x : X) : EXISTS (y : Y) :
        LET pair : XY = (x, y) IN rel(pair);
END

This translation makes maximally-efficient use of the direct encoding of sets as boolean functions, for example, using internal variables (introduced by LET) to facilitate the toolset’s manipulation of symbolic structures.

There was the option of re-implementing all of the set operations again in the relation-context; however, for efficiency we provide definitions only for the additional operations upon relations. In our example, specific relation operations (such as domain) are selected from this relation-context, whereas standard set-operations (such as contains?) are selected from a set-context, treating the same relation as a set of pairs:

... relation {PERSON, TITLE;} ! domain(rented) ...
... set {PERSON_X__TITLE;} ! contains?(rented,
    (RentVideo__p?, RentVideo__t?)) ...
function \( \{ X, Y : \text{TYPE}; \) \( x_b : X, y_b : Y \} : \text{CONTEXT} = \)
BEGIN
\( \)\( XY : \text{TYPE} = \{ X, Y \}; \)
\( \)\( YX : \text{TYPE} = \{ Y, X \}; \)
\( \)Function : \( \text{TYPE} = \{ X -> Y \}; \)
Relation : \( \text{TYPE} = \{ XY -> \text{BOOLEAN} \}; \)
Inverse : \( \text{TYPE} = \{ YX -> \text{BOOLEAN} \}; \)
Domain : \( \text{TYPE} = \{ X -> \text{BOOLEAN} \}; \)
\( \ldots \)
convert \( \text{fun} : \text{Function} \) : Relation =
\( \)LAMBDA (pair : XY) : \( \text{fun}(\text{pair}.1) = \text{pair}.2 \)
\( \text{AND} \) pair.1 /= \( x_b \) \( \text{AND} \) pair.2 /= \( y_b \);
END

Since this encoding is quite different from the relation encoding, we re-implemented some of the standard operations on relations, to optimize these for functions, eg:

image \( \text{fun} : \text{Function}, \text{set} : \text{Domain} \) : Range =
\( \)LAMBDA \( y : Y \) : \( \exists X \) : \( x : X \) :
\( \)\( \text{set}(x) \) \( \text{AND} \) \( \text{fun}(x) = y \) \( \text{AND} \) \( y /= y_b \);

Here, maximal use is made of native function application, which is extremely efficient in SAL, while including extra side-constraints to rule out mappings that would include \textit{bottom}. To simplify the definition of these and other operations on functions, a global constraint \( \text{fun}(x_b) = y_b \) is asserted in the main context.

\( Z \) distinguishes many function types for plain, injective, surjective and bijective functions (in total and partial combinations). The strategy in SAL is not to create additional function types, which would either require duplication of all function operations, or would prevent treating e.g. an injective function just as a plain function. Instead, the \( Z \) definitions of each function type are converted into predicates, that are added to the system invariant. For example, a (partial) surjective function is constrained by the predicate:

\( \text{surjective?} \) \( f : \text{Function} \) : \( \text{BOOLEAN} = \)
\( \forall \) \( y : Y \) : \( \exists x : X \) : \( f(x) = y \);

and the extra conjunction is added to constrain \textit{myFun} in the invariant:

\( \text{DEFINITION} \text{Invariant} = \ldots \)

\( \text{AND} \) function Dom, Ran, \( \text{dom}_b, \text{ran}_b \)\! surjective?(myFun)\ldots

The predicates must be coded in such a way that they are not violated if the functions are in fact empty, as is typical at the start of a simulation.

Finally, the SAL contexts encoding \( Z \) sets, functions and relations also include a number of optimized operations for dealing with common \( Z \) cases. For example, the insertion of single elements into sets is expressed in \( Z \) as the union of a set with a constructed singleton set. In SAL, this can be achieved much more simply with an extra \textit{insert} operation in the \textit{set}-context:
insert (set : Set, new : T) : Set =
  LAMBDA (elem : T) : elem = new OR set(elem);

Likewise, the Z style of replacing maplets in functions using the override operator
is handled much more efficiently by providing an extra \texttt{insert} operation in the

\texttt{function}-context:

\begin{verbatim}
insert (fun : Function, pair : XY) : Function =
  LAMBDA (x : X) : IF x = pair.1
  THEN pair.2 ELSE fun(x) ENDIF;
\end{verbatim}

These special cases are identified in our parser. Similar special operations are
supplied for empty and universal sets, singleton sets and empty functions.

5 \hspace{1em} \textbf{Use of the tool}

The SAL toolset uses a command line interface, as does our translator. The
translator accepts the \LaTeX\ markup as defined in the Z standard, and the
translator output is a plain SAL file. We have work in progress to port the
translator to accept the ZML markup for Z, using the ASTs constructed by the
parser produced by the CZT project \cite{8}. We also have work in progress to build
a GUI interface to the command-line tools and interpret the results, which are
rather dense at the moment.

\textbf{Simulation:} The SAL translation of the example can be simulated by running
the \texttt{sal-sim} tool and loading the SAL file. The compilation process takes
about 6.7 seconds for this example on a standard desktop. Our example creates
11664 initial states, most of which are due to assigning all possible values to the
\texttt{INPUT} and \texttt{OUTPUT} variables (since we initialise the \texttt{LOCAL} variables to fixed
values). While it is necessary to represent all possible input conditions for the
first simulation step, we could reduce the number of initial states by constraining
the unused values of output variables. The simulation is triggered by repeated
calls to the (\texttt{step1}) function and the number of resulting states may be viewed:

\begin{center}
\begin{tabular}{|c|c|c|c|c|c|}
\hline
Step & 0 & 1 & 2 & 3 & 4 & 5 \\
\hline
States & 11664 & 221040 & 1752048 & 7918848 & 24593528 & 61568640 \\
\hline
\end{tabular}
\end{center}

As well as displaying \texttt{n} of the states found at each step, it is possible to see
an arbitrary trace through the system, by a command which selects a random
trace. For example, after five steps, a trace is returned that shows how the system
performed the following:

\begin{center}
\begin{tabular}{|c|c|}
\hline
Step & Transition & Updates \\
\hline
0 & Init & \texttt{members, rented, stockLevel = }0 \\
1 & AddTitle & \texttt{stockLevel(TITLE\_2) = 3} \\
2 & AddMember & \texttt{PERSON\_2 \in members} \\
3 & AddMember & \texttt{PERSON\_3 \in members} \\
4 & RentVideo & \texttt{(PERSON\_3, TITLE\_2) \in rented} \\
5 & Else & no change \\
\hline
\end{tabular}
\end{center}
From this, it can be seen that the system acquired some videos and members and rented a title to one of the members. The final step selected the default ELSE-transition, a nullhop that is always possible, in case a simulation deadlocks.

**Model Checking**: The SAL toolkit has several simple and bounded model-checkers that support both LTL and CTL temporal logics. At the moment, we add theorems by hand to the end of the translated SAL file. Eventually, we expect to add an extension to Z to express theorems in temporal logic.

Suppose that we want to show that videos eventually get rented to members of the video club. In SAL, we propose the negation of this as a theorem:

\[
\text{th1 : THEOREM } \neg\diamond\text{set \{PERSON\_X\_TITLE\};!empty?(rented)};
\]

This says that "the State module allows us to derive that the relation rented is always empty," using the LTL operator \(\neg\) for always. We run this through the model checker and this generates the smallest counterexample that proves the desired property:

<table>
<thead>
<tr>
<th>Step</th>
<th>Transition</th>
<th>Updates</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Init</td>
<td>members, rented, stockLevel = ∅</td>
</tr>
<tr>
<td>1</td>
<td>AddTitle</td>
<td>stockLevel(TITLE_2) = 3</td>
</tr>
<tr>
<td>2</td>
<td>AddMember</td>
<td>PERSON_1 ∈ members</td>
</tr>
<tr>
<td>3</td>
<td>RentVideo</td>
<td>(PERSON_1, TITLE_2) ∈ rented</td>
</tr>
</tbody>
</table>

For our example, the time taken is again about 6-7 seconds, however, the majority of time is taken up compiling the example, and the execution time to find the counterexample was 0.17 seconds. Proper evaluation and scalability is left for future work.

6 Conclusion

In conclusion, we have achieved a fairly efficient translation of Z into SAL, demonstrating the benefits of encodings that are close to SAL's internal BDD structures and giving heuristics for reducing the initial state-space. New results reported in this paper include the translation of schema invariants and the optimized datatypes for the Z mathematical toolkit. We have also identified some problems in handling constructed and recursive types in SAL. Future work will include translating the rest of the mathematical toolkit.

**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.

References


**Appendix A**

The following defines a video shop and the process of renting videos etc.

\[
\begin{array}{ll}
\text{State} & \\
\text{members} : \mathbb{P} \text{PERSON} & \\
\text{rented} : \text{PERSON} \rightarrow \text{TITLE} & \\
\text{stockLevel} : \text{TITLE} \rightarrow \mathbb{N} & \\
\text{dom rented} \subseteq \text{members} & \\
\text{ran rented} \subseteq \text{dom stockLevel} & \\
\end{array}
\]

\[
\begin{array}{ll}
\text{Init} & \\
\text{State'} & \\
\text{members'} = \emptyset & \\
\text{stockLevel'} = \emptyset & \\
\end{array}
\]

\[
\begin{array}{ll}
\text{RentVideo} & \\
\Delta \text{State} & \\
p? : \text{PERSON} & \\
t? : \text{TITLE} & \\
p? \in \text{members} & \\
t? \in \text{dom stockLevel} & \\
\text{stockLevel}(t?) > \#(\text{rented} \uplus \{t?\}) & \\
(p?, t?) \notin \text{rented} & \\
\text{rented'} = \text{rented} \uplus \{(p?, t?)\} & \\
\text{stockLevel'} = \text{stockLevel} & \\
\text{members'} = \text{members} & \\
\end{array}
\]

\[
\begin{array}{ll}
\text{AddTitle} & \\
\Delta \text{State} & \\
t? : \text{TITLE} & \\
\text{level}? : \mathbb{N} & \\
\text{stockLevel'} = \text{stockLevel} \uplus \{\langle t?, \text{level}? \rangle\} & \\
\text{rented'} = \text{rented} & \\
\text{members'} = \text{members} & \\
\end{array}
\]
\[
\begin{align*}
\text{DeleteTitle} & \quad \Delta \text{State} \\
\text{p?} : \text{TITLE} & \\
\text{?} \notin \text{run rented} & \\
\text{?} \in \text{dom stockLevel} & \\
\text{stockLevel}' = \{\text{?}\} \cup \text{stockLevel} & \\
\text{rented}' = \text{rented} & \\
\text{members}' = \text{members} & \\
\end{align*}
\]

\[
\begin{align*}
\text{AddMember} & \quad \Delta \text{State} \\
\text{p?} : \text{PERSON} & \\
\text{?} \notin \text{members} & \\
\text{stockLevel}' = \text{stockLevel} & \\
\text{rented}' = \text{rented} & \\
\text{members}' = \text{members} \cup \{\text{p?}\} & \\
\end{align*}
\]

\[
\begin{align*}
\text{CopiesOut} & \quad \Xi \text{State} \\
\text{p?} : \text{TITLE} & \\
\text{copies}! : \text{N} & \\
\text{?} \in \text{dom stockLevel} & \\
\text{copies}! = \#(\text{rented} \ni \{\text{?}\}) & \\
\end{align*}
\]

Appendix B

This following is the SAL output from the translation of the above.

element : \text{CONTEXT} = \text{BEGIN}

\begin{align*}
\text{PERSON} : \text{TYPE} = \{\text{PERSON\_1, PERSON\_2, PERSON\_3}\}; \\
\text{TITLE} : \text{TYPE} = \{\text{TITLE\_1, TITLE\_2, TITLE\_3, TITLE\_B}\}; \\
\text{PERSON\_X\_TITLE} : \text{TYPE} = \{\text{PERSON, TITLE}\}; \\
\text{NAT} : \text{TYPE} = \{0..4\}; \\
\text{PERSON\_X\_TITLE\_counter} : \text{CONTEXT} = \text{count12 (PERSON\_X\_TITLE;} \\
(\text{PERSON\_1, TITLE\_1}), (\text{PERSON\_1, TITLE\_2}), (\text{PERSON\_1, TITLE\_3}), \\
(\text{PERSON\_1, TITLE\_B}), (\text{PERSON\_2, TITLE\_1}), (\text{PERSON\_2, TITLE\_2}), \\
(\text{PERSON\_2, TITLE\_3}), (\text{PERSON\_3, TITLE\_1}), (\text{PERSON\_3, TITLE\_B}), (\text{PERSON\_3, TITLE\_3}), (\text{PERSON\_3, TITLE\_B}); \\
\end{align*}

\text{State} : \text{MODULE} = \text{BEGIN}

\begin{align*}
\text{LOCAL members} : \text{set} \{\text{PERSON};\} & \text{! Set} \\
\text{LOCAL rented} : \text{set} \{\text{PERSON\_X\_TITLE};\} & \text{! Set} \\
\text{LOCAL stockLevel} : \{\text{TITLE} \rightarrow \text{NAT}\} & \\
\text{INPUT RentVideo\_p?: PERSON} \\
\text{INPUT RentVideo\_t?: TITLE} \\
\text{INPUT AddTitle\_t?: TITLE} \\
\text{INPUT AddTitle\_level?: NAT} \\
\text{INPUT DeleteTitle\_t?: TITLE} \\
\text{INPUT AddMember\_p?: PERSON} \\
\text{INPUT CopiesOut\_t?: TITLE} \\
\end{align*}
OUTPUT CopiesOut.copies : NAT
LOCAL invariant__ : BOOLEAN

DEFINITION

invariant__ = (set {PERSON;} ! subset?(relation {PERSON, TITLE;} ! domain(rented), members) AND
set {TITLE;} ! subset?(relation {PERSON, TITLE;} ! range(rented),
function {TITLE, NAT; TITLE__.B, 4} ! domain(stockLevel)) AND
stockLevel (TITLE__.B) = 4 AND
RentVideo_t? /= TITLE__.B AND
AddTitle_t? /= TITLE__.B AND
DeleteTitle_t? /= TITLE__.B AND
CopiesOut_t? /= TITLE__.B AND
CopiesOut.copies /= 4)

INITIALIZATION [members = set {PERSON;} ! empty AND
stockLevel = function {TITLE, NAT; TITLE__.B, 4} ! empty AND invariant__
--> ]

TRANSITION [
RentVideo :
set {PERSON;} ! contains?(members, RentVideo.p?) AND
set {TITLE;} ! contains?({function {TITLE, NAT; TITLE__.B, 4} ! domain(stockLevel), RentVideo.t?}) AND
stockLevel (RentVideo_t?) > PERSON___X__TITLE__.counter ! size?(relation {PERSON, TITLE;} ! range(rented, set {TITLE;} ! singleton(RentVideo_t?))) AND
NOT set {PERSON___X__TITLE;} ! contains?(rented, (RentVideo.p?, RentVideo.t?)) AND
rented' = set {PERSON___X__TITLE;} ! insert(rented, (RentVideo.p?, RentVideo.t?)) AND
stockLevel' = stockLevel AND
members' = members AND
invariant__'
--> members' IN { x : set {PERSON;} ! Set | TRUE};
rented' IN { x : set {PERSON___X__TITLE;} ! Set | TRUE};
stockLevel' IN { x : [ TITLE -> NAT ] | TRUE}
[]
AddTitle :
stockLevel' = function {TITLE, NAT; TITLE__.B, 4} ! insert(stockLevel,
(AddTitle_t?, AddTitle_level?)}) AND
rented' = rented AND
members' = members AND
invariant__'
--> members' IN { x : set {PERSON;} ! Set | TRUE};
rented' IN { x : set {PERSON___X__TITLE;} ! Set | TRUE};
stockLevel' IN { x : [ TITLE -> NAT ] | TRUE}
[]
DeleteTitle:
NOT set {TITLE;} ! contains?(relation {PERSON, TITLE;} !
range(rented), DeleteTitle_t?) AND
set {TITLE;} ! contains?((function (TITLE, NAT; TITLE_B, 4) !
domain(stockLevel), DeleteTitle_t?) AND
stockLevel' = function (TITLE, NAT; TITLE_B, 4) ! domainSubtract(set
{TITLE;} ! singleton(DeleteTitle_t?), stockLevel) AND
rented' = rented AND
members' = members AND
invariant'_

--> members' IN { x : set {PERSON;} ! Set | TRUE};
rented' IN { x : set {PERSON_X_T TLE;} ! Set | TRUE};
stockLevel' IN { x : [ TITLE -> NAT ] | TRUE}
[]

AddMember:
NOT set {PERSON;} ! contains?(members, AddMember_p?) AND
stockLevel = stockLevel' AND
rented' = rented AND
members' = set {PERSON;} ! insert(members, AddMember_p?) AND
invariant'_

--> members' IN { x : set {PERSON;} ! Set | TRUE};
rented' IN { x : set {PERSON_X_T TLE;} ! Set | TRUE};
stockLevel' IN { x : [ TITLE -> NAT ] | TRUE}
[]

CopiesOut:

members = members' AND
rented = rented' AND
stockLevel = stockLevel' AND
set {TITLE;} ! contains?((function (TITLE, NAT; TITLE_B, 4) !
domain(stockLevel), CopiesOut_t?) AND
CopiesOut_copies_ = PERSON_X_T TLE_counter ! size?(relation
{PERSON, TITLE;} ! rangeRestrict(rented, set {TITLE;} !
singleton(CopiesOut_t?))) AND
invariant'_

--> members' IN { x : set {PERSON;} ! Set | TRUE};
rented' IN { x : set {PERSON_X_T TLE;} ! Set | TRUE};
stockLevel' IN { x : [ TITLE -> NAT ] | TRUE};
CopiesOut_copies'_ IN { x : NAT | TRUE}
[]
ELSE -->
]
END;

END