
Proceedings Paper:
Derrick, J., North, S. and Simons, A.J.H. (2008) Z2SAL-building a model checker for Z. In: Borger, E., Butler, M., Bowen, J.P. and Boca, P., (eds.) Abstract State Machines, B and Z, Proceedings of. ABZ 2008, September 16-18, 2008, London, UK. Springer , 280 - 293. ISBN 978-3-540-87602-1

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@dcs.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 defficiency. 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 LaTeXsource 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 $\mathbb{Z}$ 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 $rented : PERSON \leftrightarrow TITLE$ 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 analyser 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:

Z	SAL translation
Built-in types such as $\mathbb{N}$ etc	Finite subranges of SAL equivalent types
Given sets	Enumerated finite type in SAL
Free types	Constructed type in SAL

So, in the example in Appendix A, the given type [*PERSON*] is translated to: `PERSON : TYPE = {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 `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 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.

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 *primed* and *unprimed* versions of variables. The primed `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 polylithic membership predicate over all of its elements. Set operations are defined in the following style:

```
union(setA : Set, setB : Set) : Set =
  LAMBDA (elem : T) : setA(elem) OR setB(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 `size?` function altogether from the standard, efficient *set*-context and provide this in a separately generated `countN`-context, parameterized over arbitrary 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 `size?` function from the *count*3-context:

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

```
relation{X, Y : TYPE; } : CONTEXT =
BEGIN
  XY : TYPE = [X, Y];
  YX : TYPE = [Y, X];
  Domain : TYPE = [X -> BOOLEAN];
  Range : TYPE = [Y -> BOOLEAN];
  Relation : TYPE = [XY -> 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?)) ...
```

The success of this partitioning approach motivated our splitting the complete definition of relations over three contexts, according to the number of types related. The standard context provides all operations on relations between two distinct base types. A separate context provides all operations on relations closed over a single type (such as identity, transitive closure); while a third context defines relational composition, relating three types. The translator exports these contexts only if they are needed.

In translating Z functions, we could either model them as sets of pairs, thereby easing the integration into the models for sets and relations, or use the SAL built-in function type. Timing experiments confirmed that using native SAL functions was far more efficient. However, SAL functions are *total*. To support the more commonly-occurring partial functions in Z we adopt a *totalizing* strategy, in which every type appearing in a function signature is extended with a *bottom* value. This is typically an extra symbolic value (such as `TITLE__B`) for basic types and an out-of-range value for numeric types. Partial Z functions are converted into total SAL functions, in which some domain or range values are *bottom*. At the same time, extra invariants are added to the translation of Z operation schemas to assert that input and output variables never take the *bottom* value. This approach was more scalable than defining two versions of each type, with and without *bottom*.

The *function*-context is parameterized over the domain and range element-types, but also includes value-parameters for the *bottom* element of each of these types. This allows operations on functions to recognize undefined cases. The context defines the function-type, but also pair-types for the corresponding relation and inverse relation, and supplies a `convert` operation to convert a SAL-function back into our preferred encoding for a relation as a set of pairs:

```
function {X, Y : TYPE; xb : X, yb : Y} : CONTEXT =
BEGIN
  XY : TYPE = [X, Y];
  YX : TYPE = [Y, X];
  Function : TYPE = [X -> Y];
  Relation : TYPE = [XY -> BOOLEAN];
  Inverse : TYPE = [YX -> BOOLEAN];
  Domain : TYPE = [X -> BOOLEAN];
...
  convert (fun : Function) : Relation =
    LAMBDA (pair : XY) : fun(pair.1) = pair.2
      AND pair.1 /= xb AND pair.2 /= yb;
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 (fun : Function, set : Domain) : Range =
    LAMBDA (y : Y) : EXISTS (x : X) :
      set(x) AND fun(x) = y AND y /= yb;
```

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 *bottom*. To simplify the definition of these and other operations on functions, a global constraint `fun(xb) = yb` 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:

```
  surjective? (f : Function) : BOOLEAN =
    FORALL (y : Y) : EXISTS (x : X) : f(x) = y;
```

and the extra conjunction is added to constrain *myFun* in the invariant:

```
  DEFINITION invariant__ = ...
    AND function{Dom, Ran, dom__b, ran__b}! surjective?(myFun)...
```

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 `insert` operation in the *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 `insert` operation in the *function*-context:

```
insert (fun : Function, pair : XY) : Function =
  LAMBDA (x : X) : IF x = pair.1
    THEN pair.2 ELSE fun(x) ENDIF;
```

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

5 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 [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.

Simulation: The SAL translation of the example can be simulated by running the `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 `INPUT` and `OUTPUT` variables (since we initialise the `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 `(step!)` function and the number of resulting states may be viewed:

Step	0	1	2	3	4	5
States	11664	221040	1752048	7918848	24593328	61568640

As well as displaying 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:

Step	Transition	Updates
0	Init	$members, rented, stockLevel = \varnothing$
1	AddTitle	$stockLevel(TITLE\_2) = 3$
2	AddMember	$PERSON\_2 \in members$
3	AddMember	$PERSON\_3 \in members$
4	RentVideo	(PERSON_3, TITLE_2) $\in rented$
5	Else	no change

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 nullop 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:

```
th1 : THEOREM State |- G(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 `G` for always. We run this through the model checker and this generates the smallest counterexample that proves the desired property:

Step	Transition	Updates
0	Init	$members, rented, stockLevel = \varnothing$
1	AddTitle	$stockLevel(TITLE\_2) = 3$
2	AddMember	$PERSON\_1 \in members$
3	RentVideo	$(PERSON\_1, TITLE\_2) \in rented$

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

1. C. Bolton. Using the Alloy Analyzer to Verify Data Refinement in Z. *Electronic Notes in Theoretical Computer Science*, 137(2):23–44, 2005.

2. Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Computers*, 35(8):677–691, 1986.
3. Randal E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Comput. Surv.*, 24(3):293–318, 1992.
4. E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.
5. L. de Moura, S. Owre, and N. Shankar. The SAL language manual. Technical Report SRI-CSL-01-02 (Rev.2), SRI International, 2003.
6. John Derrick, Siobhán North, and Tony Simons. Issues in implementing a model checker for z. In Zhiming Liu and Jifeng He, editors, *ICFEM*, volume 4260 of *Lecture Notes in Computer Science*, pages 678–696. Springer, 2006.
7. M. Leuschel and M. Butler. Automatic refinement checking for B. In K. Lau and R. Banach, editors, *International Conference on Formal Engineering Methods, ICFEM 2005*, volume 3785 of *LNCS*, pages 345–359. Springer-Verlag, 2005.
8. Tim Miller, Leo Freitas, Petra Malik, and Mark Utting. CZT Support for Z Extensions. In Judi Romijn, Graeme Smith, and Jaco Pol, editors, *Integrated Formal Methods, IFM 2005*, volume 3771 of *LNCS*, pages 227–245. Springer-Verlag, 2005.
9. Daniel Plagge and Michael Leuschel. Validating Z Specifications using the ProB Animator and Model Checker. *Integrated Formal Methods*, 4591:480–500, 2007.
10. G. Smith and L. Wildman. Model checking Z specifications using SAL. In H. Treharne, S. King, M. Henson, and S. Schneider, editors, *International Conference of Z and B Users*, volume 3455 of *LNCS*, pages 87–105. Springer-Verlag, 2005.

Appendix A

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

$[PERSON, TITLE]$

State

$$members : \mathbb{P}\, PERSON$$
$$rented : PERSON \leftrightarrow TITLE$$
$$stockLevel : TITLE \pfun \mathbb{N}$$
$$\mathrm{dom}\, rented \subseteq members$$
$$\mathrm{ran}\, rented \subseteq \mathrm{dom}\, stockLevel$$

Init

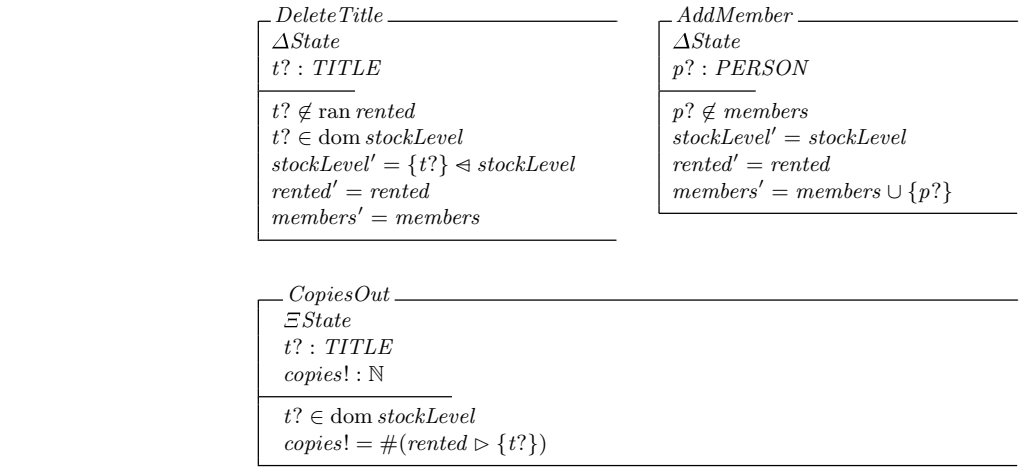
$$State'$$
$$members' = \varnothing$$
$$stockLevel' = \varnothing$$

RentVideo

$$\Delta State$$
$$p? : PERSON$$
$$t? : TITLE$$
$$p? \in members$$
$$t? \in \mathrm{dom}\, stockLevel$$
$$stockLevel(t?) > \#(rented \rhd \{t?\})$$
$$(p?, t?) \notin rented$$
$$rented' = rented \cup \{(p?, t?)\}$$
$$stockLevel' = stockLevel$$
$$members' = members$$

AddTitle

$$\Delta State$$
$$t? : TITLE$$
$$level? : \mathbb{N}$$
$$stockLevel' = stockLevel \oplus \{(t?, level?)\}$$
$$rented' = rented$$
$$members' = members$$



Appendix B

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

```
example : CONTEXT = BEGIN

PERSON : TYPE = {PERSON__1, PERSON__2, PERSON__3};
TITLE : TYPE = {TITLE__1, TITLE__2, TITLE__3, TITLE__B};
PERSON__X__TITLE : TYPE = [PERSON, TITLE];
NAT : TYPE = [0..4];

PERSON__X__TITLE__counter : CONTEXT = count12 {PERSON__X__TITLE;
  (PERSON__1, TITLE__1), (PERSON__1, TITLE__2), (PERSON__1, TITLE__3),
  (PERSON__1, TITLE__B), (PERSON__2, TITLE__1), (PERSON__2, TITLE__2),
  (PERSON__2, TITLE__3), (PERSON__2, TITLE__B), (PERSON__3, TITLE__1),
  (PERSON__3, TITLE__2), (PERSON__3, TITLE__3), (PERSON__3, TITLE__B)};

State : MODULE =
  BEGIN
    LOCAL members : set {PERSON;} ! Set
    LOCAL rented : set {PERSON__X__TITLE;} ! Set
    LOCAL stockLevel : [ TITLE -> NAT ]
    INPUT RentVideo__p? : PERSON
    INPUT RentVideo__t? : TITLE
    INPUT AddTitle__t? : TITLE
    INPUT AddTitle__level? : NAT
    INPUT DeleteTitle__t? : TITLE
    INPUT AddMember__p? : PERSON
    INPUT CopiesOut__t? : TITLE
```

```
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
    AddTitle__level? /= 4  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;} ! rangeRestrict(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__TITLE;} ! 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__TITLE;} ! 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__TITLE__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__TITLE;} ! Set | TRUE};
          stockLevel' IN { x : [ TITLE -> NAT ] | TRUE};
          CopiesOut__copies_' IN { x : NAT | TRUE}
      []
      ELSE -->
    ]
  END;

END
```