DOI 10.1007/s00165-009-0126-7

Formal Aspects of Computing

Formal Aspects of Computing

Z2SAL: a translation-based model checker for Z

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

Department of Computer Science, University of Sheffield, Regent Court, 211 Portobello, Sheffield, S1 4DP, UK.
E-mail: J.Derrick@dcs.shef.ac.uk

Abstract. Despite being widely known and accepted in industry, the Z formal specification language has not so far been well supported by automated verification tools, mostly because of the challenges in handling the abstraction of the language. In this paper we discuss a novel approach to building a model-checker for Z, which involves implementing a translation from Z into SAL, the input language for the Symbolic Analysis Laboratory, a toolset which includes a number of model-checkers and a simulator. The Z2SAL translation deals with a number of important issues, including: mapping unbounded, abstract specifications into bounded, finite models amenable to a BDD-based symbolic checker; converting a non-constructive and piecemeal style of functional specification into a deterministic, automaton-based style of specification; and supporting the rich set-based vocabulary of the Z mathematical toolkit. This paper discusses progress made towards implementing as complete and faithful a translation as possible, while highlighting certain assumptions, respecting certain limitations and making use of available optimisations. The translation is illustrated throughout with examples; and a complete working example is presented, together with performance data.

Keywords: Z, model-checking, SAL

1. Introduction

Despite being widely known and accepted by the software industry, the formal notation Z [Spi92] has for some time lagged behind other specification languages in the provision of tools for automatically verifying specifications, whether by simulation, model-checking or theorem proving. There are a number of reasons for this, although most are connected with the language itself and its semantics: the inherent expressivity of Z makes it harder to build tractable tools for it.

Historically, early tools, such as *fuZZ* [Spi00] and *CADiZ* [TM95], were closely linked to the typesetting languages, *troff* and LaTeX, used to write Z and focused on creating, formatting and type-checking Z specifications. Later versions of *CADiZ* and the *Z/Eves* composer and proof tool [Saa97, Saa99] were able to perform additional functions, such as domain checking (stricter than type checking, for partial functions), schema expansion, with redundant term elimination, and interactive theorem-proving using heuristics and proof tactics suggested by the user. *CADiZ* is under continuous development and has since evolved towards the ISO-Z standard.

Correspondence and offprint requests to: J. Derrick, E-mail: J.Derrick@dcs.shef.ac.uk

Published online: 02 October 2009

1.1. Community Z tools

More recently, a concerted effort has been made in the wider global Z user community to address the general tool deficiency. The Community Z Tools (CZT) project [MFMU05] is one leading example. This group is in the process of developing a set of open source tools for Z, based around the ZML markup language [DUT+03], an XML dialect developed specifically for Standard ISO-Z [135]. So far, there is a parser and a type-checker for Z, an AST package developed in Java for use in third-party modules, and a number of other proposed modules, including cross-language translators and model-checkers, based on the same parser and AST. The tools handle a number of input formats, including ZML and LaTeX. This work is foundational and will provide long term grass-roots support for Z. However, the progress towards finished provers and checkers is slow, partly due to the complexity of the Z standard and the use of automatic code generation technology to develop the parser and AST, whose API needs more user-friendly documentation, to encourage a wider take-up.

Elsewhere, others have sought a quicker route to developing model-checkers for Z by adapting existing tools that support automated checking. An example of this is the *ProZ* tool [PL07], which extends and adapts the earlier *ProB* tool [LB05] for Z. The B language [Abr96], though related to Z, is much closer to the state-transition formalism used by symbolic model-checkers. *ProB* and *ProZ* both use an underlying Prolog engine to simulate (or "animate") a specification, by populating variable terms with ground values, chosen from restricted ranges. Configurations of state variables form the states of the automaton, while each operation is styled as a transition from state to state, affecting the values of variables. The validity of models is checked by simulating forwards in time from the initial state, exploring multiple future states in parallel. Consistency is checked by verifying the state invariant in each state, or by detecting deadlocks (failure to instantiate all variable terms). A further facility exists for checking refinement relations between specifications.

A similar approach was taken by Bolton [Bol05], who used the Alloy SAT-solver based counter-example finder [Jac02] to verify data refinements in Z, after translating from Z into the Alloy input language. This is similar to our philosophy [DNS06] of translating Z into the input language for the SAL tool-suite [dMOS03], which uses a BDD-based symbolic model checker as its core engine. We consider that this strategy of translating into the input format of another proven toolset will result in better model-checking capabilities than building a bespoke model-checker for Z on top of the CZT toolkit. The SAL core engine is already quite sophisticated, transforming set-theoretic and mathematical formulae into boolean judgements on ordered variables, which are compiled to optimized binary decision trees (BDDs), before generating Büchi automata for each theorem expressed in temporal logic to explore the compacted state space. Developing a similar tool from scratch would require considerably more effort than building a cross-translator.

1.2. The symbolic analysis laboratory

Our choice of the symbolic analysis laboratory (SAL) tool-suite [dMOS03] as the target for the translation was motivated by a number of reasons. Firstly, there were already a number of different tools using the SAL input language. These included a simulator, a model-checker, a bounded model-checker and a counterexample finder, with other tools in the pipeline. Secondly, there was a sizeable international user-group engaged in developing and using the tools, offered gratis by SRI under an academic licence, with some support offered from the developers. Thirdly, the SAL input language [dMOS03] is purposely designed to be formalism-neutral, positioning itself somewhere between the highly restrictive machine-centric syntax required by Spin [Hol97] and SMV [CGL94], and the complete expressiveness offered by conventional programming languages. The SAL input language supports finite sets, tuples, subranges, arrays, records, total functions, concurrently executing and parameterised modules and (in principle) recursive definitions. The kinds of theorem that may be checked include first-order predicate terms and both LTL and CTL temporal logic expressions. Finally, the core SAL engine compiles all definitions into optimized binary decision trees (BDDs) and simulates models using Büchi automata, proven approaches for dealing with large state spaces efficiently.

The original idea of translating Z into SAL specifications was due to Smith and Wildman [SW05]. In [DNS06] we described the basics of our implementation, which is essentially a bespoke parser and generator, written in Java, which translates from the LaTeX encoding of Z into the SAL input language. The collection of Z schemas is translated into a SAL finite state automaton, following a template-driven strategy with a number of associated heuristics. Like [SW05], we aim to preserve the Z-style of specification, including postconditions that mix primed

and unprimed variables arbitrarily, possibly asserting posterior states in non-constructive ways, and preserving 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. Given this theoretical basis, our implementation has increasingly diverged from [SW05] as optimization issues have been tackled. In [DNS06] we highlighted problems with countable sets. In [DNS08] we described an improved treatment for countable sets, and new translations for relations and functions. In this extended paper, we also report on improvements in the treatment of Z function types, a translation for sequences and an improved translation for operation schema variables, which greatly reduces the state space.

1.3. Overview of the translation strategy

The structure of the paper is as follows. Section 2 describes the main challenges in converting an abstract specification written in Z into a grounded format more acceptable to a BDD-based model checker. Section 3 describes the main features of our Z parser and generator, which accepts several widely-known LaTeX markup formats for Z and performs some early optimisations, prior to generating the SAL output. Section 4 describes the basic template for translating Z types, constants, state and operation schemas into a SAL automaton. Sections 5 and 6 describe the additional SAL modular units that translate the set, relation, function and sequence datatypes from the Z mathematical toolkit. Section 7 describes a complete working example, showing the translation of a complete Z specification, and the results of simulating and model-checking temporal logic properties of the translation. The space and time performance of the SAL tool-suite on our example is also indicated, showing how improvements to the translation strategy have brought performance gains.

2. Challenges in translating from Z into SAL

In this section, we highlight some of the main challenges encountered when dealing with the high level of abstraction in Z. We also identify some fundamental structural obstacles caused by the mismatch between Z's partial function paradigm and the total, automaton-based paradigm of the SAL tool-suite. In particular, we highlight certain limitations of the BDD-based formalism, which particularly affect Z, but which are addressed later in our translation.

2.1. Bounding the infinite

When considering how to translate from a specification language like Z, which supports fully abstract (non-grounded, non-constructive) specification styles, into the concrete and grounded language of a model-checker, the first and most obvious challenge is how to deal with unbounded or infinite structures and uninterpreted symbols. All model checkers require types with finite, bounded ranges, so that the variable product space, though potentially large, should not result in a state-space explosion, eventually exhausting the host computer's available memory.

For example, Z supports the built-in numerical types $\mathbb{Z}$, $\mathbb{N}$ and $\mathbb{N}_1$, all of which have infinite ranges; and while SAL has the cognate unbounded types $INTEGER$, $NATURAL$ and $NZNATURAL$, these may only be used as the base types of finite subranges in actual specifications. Z also supports the declaration of arbitrary basic types, such as $[NAME, PHONE]$, which have the semantics of uninterpreted sets (sorts). It is presumed that an infinite number of objects may populate the $NAME$ or $PHONE$ sets, which are not further analyzed. Clearly, any translation must choose suitable finite enumerated ranges for each of these sets, without compromising the ability of a model-checker to explore unusual combinations of value assignments. Our approach to bounding ranges is discussed in Sect. 3.

Much of the flexibility of Z comes from the practice of declaring uninterpreted symbols, such as the constants used as range limits in predicates. In Z, this style is applauded, since it supports different concrete refinements, where particular values are chosen for the limits. In a model-checker, all constants must be grounded from the start; but can suitable values be chosen, without compromising the searching behaviour of the checker? Our approach to uninterpreted symbols is subtle, treating them either as constrained variables or constants (see Sect. 4).

2.2. Mis-matched formal paradigms

The second main challenge when translating from Z into SAL is the structural mismatch between the two specification models. Consider that a specification in Z is built up incrementally, as a piecemeal collection of state and operation schemas. The viewpoint in Z is local and functional, examining how each operation schema may act upon its own input and output variables, or upon the variables of one (or more) included state schemas. By contrast, a SAL specification is constructed as a monolithic finite state automaton, in which all input, output and local (state) variables are compiled into aggregate states and all operations are styled as guarded transitions from state configuration to state configuration. This aspect, while awkward, merely requires a re-ordering of all the information present in the Z specification; but this is not the only structural mismatch.

Practical Z specifications make widespread use of partial functions, both to express incomplete computations (in operation schemas) and, even more commonly, to express the associative data types, also known as maps (in state schemas), which are dynamic in size. In SAL, functions are always total, since a BDD-based formalism converts map-like structures into ordered sets of judgements over variables, which must cover all variable assignments. In other words, for every value of a function's domain, a mapping must exist to some range value. This requires a work-around if we wish to represent undefined mappings for certain values; yet the need to do this arises frequently: for example, a map initialised to the empty set is undefined for every domain value. We adopt a totalising approach to address this problem (see Sect. 6).

In much the same way that function-valued variables are converted into ordered sets of judgements from domain to range values, set-valued variables are represented using Bryant's encoding [Bry86, Bry92], as an ordered set of judgements from elements to true or false, denoting whether that element is present or absent from the set. While this supports a very efficient and compact treatment of set operations (converting the usual *union*, *intersection* and *difference* operations acting on sets into logical *or*, *and* and *not* operations acting on atomic propositions, which are then collapsed in the compiled BDDs), it has the unusual side-effect that a set cannot be treated as a monolithic whole in SAL, but only as the polylithic collection of judgements over its elements. This has a particular deleterious effect when seeking to compute the cardinality of a set (viz. count its elements), since there is no such countable object in SAL. We adopt a bespoke work-around to solve this problem (see Sect. 5).

2.3. Non-computable specifications

A final translation challenge is the tension between non-constructive styles of specification in Z, and the desire to express a computable update step after each state-modifying transition in SAL. The normal style in SAL would be to write a series of update assignments to primed variables, indicating the posterior variable states. However, a Z postcondition need not necessarily be written in such a way and sometimes should not. For example, the postcondition of the square root function: $y = sqrt(x)$ is most naturally expressed non-constructively in Z as: $y^2 = x$ which is perfectly laudable, since it states the relationship between input and output succinctly. However, this would not usually serve in SAL, which expects a constructive update step, such as the Newton-Raphson algorithm for computing square roots. Our translation adopts a work-around, which asserts the posterior existence of variables, but constrains their posterior values in the precondition. This effectively turns a deterministic update into a search for suitable posterior values, supporting non-constructive (and non-deterministic) schemas (see Sect. 4).

3. The bespoke Z parser and SAL generator

The current version of our Z2SAL translator is a bespoke Java implementation of a Z parser accepting LaTeX input, coupled to a generator producing SAL output. We deliberately chose to go down this route, rather than build a SAL generator on top of the CZT parser and AST toolkit [MFMU05] in the first instance. This offered

greater flexibility and a faster route to evaluating different translation prototypes, compared with the overhead of learning how to interpret the CZT parse tree (whose Java API was machine-generated from the Standard ISO-Z specification and not documented beyond the automated description of syntactic interfaces). Conceptual barriers to using CZT from the outset included that the parser has schema unfolding rules that normalise schemas to Z base types, making it harder to know when a variable denotes a function or a product; and that the CZT parse-tree uses multi-purpose abstract nodes, whose contents require some further interpretation according to the context. We were therefore able to avoid having to deal with this complexity, until we had finished prototyping different templates for the SAL translation. However, once our translation technology is stable and we have identified the best route to overall optimisation, we expect to provide a SAL generator for the CZT toolkit. Progress towards this goal was reported in [AWS08].

3.1. The Z analyser

The current Z analyser is a hand-written tokeniser and recursive descent parser, which scans a LaTeX source file in a single pass, extracting all LaTeX elements relating to a Z specification and ignoring any other surrounding text. Over time, we have expanded the input vocabulary of the tokeniser to accept different sets of LaTeX macros for Z, from Spivey's original `zed.sty` macro package, to the more recent `oz.sty` macro package developed for Object Z. These often have different conventions for expressing the same Z construct, for example, the main division in a schema between the variable declarations and the predicates may be typeset using: `\where`, or alternatively: `\ST` (a mnemonic for "such that"). The tokeniser signals any unrecognised tokens and the parser likewise signals any syntactically incorrect structures, using a comprehensive error-reporting strategy, easy to implement in a top-down, recursive-descent parser, which highlights the error context and lists what possible legal terms were expected instead. This has proven useful across all stages of development, both to trap incorrect Z, and also to identify legal alternative LaTeX formats and new Z structures to be added to the grammar. We are reasonably confident that the Z analyser is now quite robust, since it has been subjected to the vagaries of several different, and often inconsistent, styles of Z specification, drawn from a back-catalogue of old Z examples.

The parser builds a memory model of the Z specification, which is essentially a list of declared types, constants and schemas, in order of definition. Although the ISO-Z specification does not require definition before usage, we make that (in practice not significant) limitation here and thus we can use a single scan (i.e., we only accept a style of Z which defines all identifiers before usage). The parser analyses type declarations and constant declarations. A single expression structure is also constructed incrementally, to represent the restrictions derived from the constraints picked up from any axiomatic definitions. The parser identifies the first two schemas as the state schema (which is typically unnamed) and the initialisation schema (which conventionally has the standard name $Init$). All other schemas are assumed to be named operation schemas. Our translation strategy currently assumes a single state schema, which is typical in small-to-medium Z specifications. This may be easily adapted for large, multi-part specifications.

The parser is implemented in an object-oriented style, such that the memory model structures are all instances of corresponding Java classes that implement the relevant AST nodes. The top-level structure is a list of schema class instances, which refer to associated instances of classes representing the named constants, types and variables, and to lists of expression trees, representing the predicates.

3.2. Bounding the ranges of types

Once the Z input has been analysed, and the types of all expressions have been checked, the translator performs a series of optimisations, designed to map the unbounded, abstract types of Z into concrete types with small, finite ranges, as required for the input to a model-checker (see the earlier discussion in Sect. 2). Likewise, suitable upper, or lower bound values must be computed for any uninterpreted constant symbols which, from their context of usage, are identified as the limits in range constraints. Eventually, it becomes possible to optimise predicate expressions, by reasoning symbolically about limits. All of this depends on the ranges chosen for each type.

The choice of a suitable range for any given type is determined by three considerations. Firstly, every type should have a sufficient population to allow the specification model to be exercised properly. Jackson has suggested that Alloy finds most counterexamples if types have at least three instances [Jac02]. We therefore take this as the minimum population size. Secondly, the built-in rules governing Z types should be preserved, in which the following relation between numerical types must hold:

$$\mathbb{N}_1 \subset \mathbb{N} \subset \mathbb{Z}$$

A consequence of this is that if $\mathbb{N}_1$ has at least three positive integral values, then $\mathbb{N}$ must also include zero (and so has at least four values) and $\mathbb{Z}$ must also include -1 (so having at least 5 values). A third consideration is that the Z specification itself may indicate particular literal values, which then are used to determine upper or lower range limits. The default strategy is to expand the minimal range to one above the highest, and one below the lowest, literal constant. Where possible, the smallest numerical ranges are used. The symbolic basic types of Z, which are uninterpreted sets, are typically given three symbolic instances, where nothing to the contrary is defined in the Z (but sets may possibly acquire a fourth *bottom* element: see Sect. 6). These settings are all defaults; they may be varied by supplying the translator with different ranges as parameters.

Limiting the ranges of types could in principle make the translation unsound for some Z specifications. For example, if a sum term caused a variable to exceed its range, the solver would eliminate terms in this variable, eventually reporting no future states. This might affect an attempt to prove the absence of a property which was only satisfiable outside the represented range. When proving properties, we therefore never rely on failure as negation. Our preferred proof strategy is to propose counter-theorems and accept found counterexamples as valid evidence. In practice, seeking to prove some range-influenced property affects how the translator bounds the ranges, for example

$$\frac{count : \mathbb{N}}{count \leq 1024}$$

will result in the translator giving $count$ the range 0..1025, according to the strategy described above.

3.3. Grounding uninterpreted constants

Once the ranges of types have been set, the uninterpreted Z constant symbols are revisited, to seek suitable ground values for them. In an earlier release of the translator (see [DNS06]) arbitrary values were assigned to Z symbolic constants (within the range of their type), grounding them as SAL constants. This was also informed by a heuristic in which the highest or lowest value was chosen, if the constant occurred in the context of a limit in an inequality expression. However, it turned out that the properties of the resulting SAL model were too dependent on the translator's choice. Instead, the current optimiser reasons symbolically about the possible intervals over which a symbolic constant may range.

This is accomplished by collecting together all the constraints from the schemas and axiomatic definitions, in which a symbolic constant appears. An axiomatic definition in Z is simply a way of qualifying an uninterpreted constant by attaching a predicate, for example the constant max in:

$$\frac{max : \mathbb{N}}{max > 1}$$

The same constant may also appear in the state schema predicate (the invariant), or in operation schema predicates, where it is further constrained with respect to the ranges of input and output variables. Initially, each of these schemas is parsed into a conventional tree structure in memory, but this is immediately transformed by extracting the list of subtrees that represent all the conjoined predicate expressions. 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, which is an activity performed at various stages, starting with the predicates of the axiomatic definitions.

The combined predicate is scanned, both to eliminate any redundant predicate expressions, and to restrict the intervals attached to symbolic constants, according to the constraints supplied by the axiomatic definitions. Sometimes the interval is restricted to a single value, at which point the Z symbolic constant is converted into a grounded SAL constant. If this cannot be done, the Z symbolic constant is simply treated as another variable

in the SAL translation, and a SAL predicate must be generated from the axiomatic definition, to constrain the variable's range. Clearly, where symbolic constants can be grounded, this removes the need to generate an additional SAL predicate, which can lead to further optimisation. Later, all schema predicates are scanned again, to eliminate any redundant predicates resulting from earlier optimisations. If, during the interval restriction process, a Z predicate proves to be unsatisfiable, the translator terminates under the assumption that the Z specification is faulty.

3.4. Synthesising definitions and types

Another activity performed by the translator, before generating actual SAL output, is to identify any types, whose definitions must be imported from external code units (known as SAL *contexts*), or for which synthetic alias names must be constructed. This is achieved during the parser's initial pass over the Z source. We have developed standard SAL library *contexts* for each data type in the Z mathematical toolkit (see Sects. 5 and 6). The SAL tools construct particular type-instantiations of these parametric *contexts* as required, and notes that the external code units must be included, in a comment inserted at the end of the translated output. Since these are library *contexts*, they may be placed in a standard `include` directory, visible to the SAL tool-suite.

A particular work-around is required to mitigate a fault discovered in some SAL tool-suite implementations, which prevented certain constructed types from being passed as first-class types in the SAL type system. This chiefly affects tuple-types, especially when these are passed as parameters to declare the element-types of relations. (The same fault seems to affect SAL tuples in general, which are sometimes mistakenly interpreted as parameter lists, resulting in unexpected type failures). The work-around is fortunately simple: a symbolic alias name may be defined and used in place of the constructed type:

```
PERSON__X__TITLE : TYPE = [PERSON, TITLE];
```

and this is then processed successfully by all versions of the SAL tools. This type alias may be used as the type of maplets (pairs), or as the element-type of the relation containing the maplets.

Another work-around is required to support counting the elements of sets. For this, the translator may generate one or more bespoke versions of the element-counting *context* (see Sect. 5), tailored to sets of different maximum capacity. The code for each *context* is generated by algorithm, according to the capacity, with variable numbers of *context*-parameters and body statements. Any generated bespoke *contexts* must eventually be placed in the same directory as the master *context*, for the SAL tool-suite to find them.

3.5. The SAL generator

The main generation phase produces the SAL *context* for the automaton, whose states are formed by aggregating all the variables from the Z state schema, and whose transitions are generated from each operation schema. Declarations are output in the overall order of the prequel, consisting of types and constants, then the main SAL *module* defining the behaviour of the automaton. This has an internal ordered structure consisting of local, input and output variable declarations, formula definitions, state initialisation and then a description of each state transition (see Sect. 4 for a complete description of generation and examples of generated SAL output).

Within the main *context*, the order of generation must satisfy SAL's definition-before-usage criterion, for example, set and subrange types must be declared before any constructed types, and types before any constants and variables declared of these types. Within the *module*, all variable names must be declared before any formula, initialisation or transition in which they appear. Where the ordering in SAL is otherwise irrelevant, the generator seeks to preserve the same order of declaration as in the original Z. To this end, a list of identifiers, in order of first appearance, is kept by the lexical analyser; and this can be used to order type and constant declarations in the prequel. This policy is useful from a human point of view, since it improves the readability of the generated SAL, which may be inspected manually, to check for faithfulness to the original Z, before being validated by simulation or model-checking.

4. Exposition of the Z2SAL translation templates

A specification in the SAL input language may consist 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 prequel, declaring types and constants, followed by the main declaration of a SAL *module*, defining the finite state automaton, which reproduces the behaviour of the Z state and operation schemas. As described in Sect. 2, the states of the automaton are created by aggregating the variables from the Z state schema, and the transitions of the automaton 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.

4.1. Translating fundamental Z types

The *built-in* types of Z are translated into finite subranges in SAL, according to the flexible scheme described in Sect. 3. For example, the following SAL translations are typical for the Z types $\mathbb{N}_1$, $\mathbb{N}$ and $\mathbb{Z}$:

```
NZNAT : TYPE = [1..3];
NAT : TYPE = [0..3];
INT : TYPE = [-1..3];
```

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—see Sect.6). For example, the following translation is typical for a pair of Z basic types declared as $[PERSON, TITLE]$:

```
PERSON : TYPE = {PERSON__1, PERSON__2, PERSON__3};
{TITLE : TYPE = TITLE__1, TITLE__2, TITLE__3, TITLE__B};
```

Sometimes an additional sentinel value is inserted at the end of the range, to stand for the undefined *bottom* element. This is the case with $TITLE$ where the translator has determined that an extra undefined *bottom* element is needed, called `TITLE__B`, whereas $PERSON$ is never used in a context requiring a *bottom* element.

The *free types* of Z are converted into similar constructed data types in SAL. For example, the free type in Z declared as $REPORT ::= ok \mid error\langle\langle MESSAGE\rangle\rangle$ is translated into the following cognate data type in SAL:

```
REPORT : DATATYPE
 ok,
 error(message : MESSAGE)
END;
```

In principle, the syntax of the SAL input language allows constructed data types to be recursively-defined; however, many of the SAL tools do not yet handle recursive definitions well. This is because they expand all recursive constructions infinitely as the definitions are compiled into BDDs. This limitation may be fixed in future releases of the SAL tool-suite, but for the moment, our parser rejects recursive definitions.

4.2. Translating Z constants and axiomatic definitions

Any *literal* Z constant may be translated directly as a grounded SAL constant. Any *uninterpreted* symbolic constant in the Z specification is handled according to the strategy described in Sect. 3. By default, a symbolic constant declared in Z as: $max : \mathbb{N}$ will, if no further optimisation is possible, be translated as a *local* variable, part of the state of the automaton, defined in the SAL *module* clause:

```
State : MODULE = BEGIN
 LOCAL max : NAT
 ...
END;
```

If the symbolic constant is introduced as part of an *axiomatic definition*, then its range will be restricted by a predicate. For example, the type of max has the range [0..3], but the following declaration restricts this interval to the smaller range [2..3]:

$$max : \mathbb{N}$$
$$\hline$$
$$max > 1$$

However, if the interval of the constant max is further constrained to a single value (by symbolic reasoning about interval constraints), then it is converted into a grounded SAL constant, for example:

```
max : NAT = 3;
```

In this case, nothing need be generated for the *axiomatic definition*, which is now redundant.

4.3. Translating the Z state schema

For the sake of the examples below we assume the principal Z state schema is called *State* and currently the tool identifies it from its place in the order of parsed schemas. *State* is also used as the name of the cognate SAL *module* defining the finite-state automaton. The following illustrates a very simple state schema, which defines one state variable *level*, and restricts the range of this with a state predicate (two separate, implicitly conjoined inequalities):

State
$$level : \mathbb{N}$$
$$\hline$$
$$0 \leq level$$
$$level \leq max$$

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 in a particular way. SAL supports the definition of formulae, which establish an equivalence between a variable and a longer expression, possibly consisting of many terms. The variable may serve as an abbreviation for the longer expression, which is useful if the expression occurs in many contexts. Our translation makes use of this facility, by introducing an extra *local* boolean variable, named `invariant__`, and then declaring a formula for this in the *definition* sub-clause, which equates the `invariant__` with all the conjoined terms of the state predicate:

```
State : MODULE = BEGIN
 LOCAL level : NAT
 LOCAL invariant__ : BOOLEAN
 ...
 DEFINITION
   invariant__ = (0 <= level AND level <= max)
 ...
END;
```

The above assumes that max was translated as a grounded SAL constant. If instead max could not be grounded, but was translated by a SAL *local* variable (see above), then the constraint from its associated *axiomatic definition* is added to the conjuncts in the state predicate:

```
State : MODULE = BEGIN
 LOCAL max : NAT
 LOCAL level : NAT
 LOCAL invariant__ : BOOLEAN
 ...
 DEFINITION
   invariant__ = (max > 1 AND 0 <= level AND level <= max)
 ...
END;
```

The invariant may eventually be extended with further terms, to assert total properties of input and output variables, or to assert the semantic properties of Z's different function types (see Sect. 6).

4.4. Translating the Z initialisation schema

The principal Z initialisation schema, which is conventionally named *Init*, is translated into the cognate SAL *initialization* sub-clause of the *module* clause. Initial values may optionally be assigned to any, or all SAL variables. Variables which are not initialised will range freely over all values in their type. Initialisation reduces the number of initial states, possibly to a single state configuration.

The typical SAL initialisation style is to declare a list of initial assignments to variables. For example, the *level* variable may be set to be zero initially:

```
State : MODULE = BEGIN
 LOCAL level : NAT
 ...
 INITIALIZATION
   level = 0;
 ...
END;
```

However, this assumes that initial values can always be asserted directly, rather than derived from the model constraints. This is not always possible, for example, we cannot assert by assignment that the `invariant__` holds initially, since this property must be derived by a formula from other model constraints. In any case, we prefer to mimic the non-constructive style of Z specifications, which allows the model constraints to influence variable bindings (see the discussion in Sect. 2).

The alternative SAL style for assignment uses a *guarded command*. This has the usual syntactic form: `guard --> assignments`, in which the `guard` expresses a triggering condition and the `assignments` express state updates to perform, when the `guard` holds. The trick used by our translation is to force variable bindings to be resolved in the `guard`, giving full play to the model constraints:

```
State : MODULE = BEGIN
 LOCAL level : NAT
 LOCAL invariant__ : BOOLEAN
 ...
 INITIALIZATION [
   level = 0 AND invariant__
   -->
 ]
END;
```

In this case, the set of updates (after the arrow) is empty, since we are in the initial state. This style also forces the invariant to hold, as a precondition for entering the initial state. The *initialization* sub-clause may later contain further terms that act to constrain the initial state of the system (see below).

4.5. 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 prequel 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.

Continuing with the above example, two simple operation schemas are defined in Z, named *Increment* and *Decrement*. These respectively add or subtract an input amount from the state variable *level*, which is imported from the *State* schema. The operations are robust, and report success, overflow or underflow as an output:

Increment

$$\Delta State$$
$$n? : \mathbb{N}$$
$$r! : REPORT$$
$$level + n? \leq max \Rightarrow level' = level + n? \land r! = ok$$
$$level + n? > max \Rightarrow level' = max \land r! = message(overflow)$$

Decrement

$$\Delta State$$
$$n? : \mathbb{N}$$
$$r! : REPORT$$
$$level - n? \geq 0 \Rightarrow level' = level - n? \land r! = ok$$
$$level - n? < 0 \Rightarrow level' = 0 \land r! = message(underflow)$$

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.

Previously [DNS06], we adopted the conservative policy (following [SW05]) of synthesising unique names for all input and output variables, by prefixing their local name with the name of the schema in which they appeared. This was to ensure that no variable names were accidentally aliased, with the undesired effect that constraints might propagate beyond their intended scope:

```
State : MODULE = BEGIN
 ...
 INPUT Increment__n?
 INPUT Decrement__n?
 OUTPUT Increment__r_
 OUTPUT Decrement__r_
 ...
END;
```

Unfortunately, creating many unique variable names also increased the variable product-space, thereby greatly increasing the state-space for model-checking. After consultation with colleagues working on the CZT project [MFMU05] and some careful experiments, we have established that input and output variable names may safely be coalesced across all operation schemas, since only one set of guard constraints are actually *enforced* in any one cycle (although many are *enabled*). This subtle decision has greatly improved the performance of model-checking (see Sect. 7).

Furthermore, the initial state space may be further reduced by clamping the *output* variables to arbitrary initial values (the *input* variables must be free to range over all legal inputs). The optimised translation is given as:

```
State : MODULE = BEGIN
 ...
 INPUT n?
 OUTPUT r_
 INITIALIZATION [
   ... AND r_ = ok
   -->
 ]
END;
```

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 conventional SAL style would be to express preconditions on the unprimed (prior state) variables in the `guard`, then assert a series of updates to primed (posterior state) variables in the `assignments`. Following the non-constructive style of specification (see above), we prefer instead to express the relationship between primed and unprimed variables in the guard, to give full play to the model constraints. However, the consequent may not be left empty (unlike the case with initialisation, above), since SAL requires all primed variables to appear here, if their values are to change. The unusual format of the update expression asserts that the primed variables still exist, and potentially range over their whole type, in the posterior state.

```
State : MODULE = BEGIN
 ...
 TRANSITION [
   Increment : ((level + n? <= max) => ((level' = level + n?)
     AND (r_' = ok)) AND ((level + n? > max) => ((level' = max)
     AND (r_' = message(overflow))) AND invariant__'
   -->
     level' IN {x : NAT | TRUE};
     r_' IN {x : REPORT | TRUE};
 []
   Decrement : ((level - n? >=0) => ((level' = level - n?)
     AND (r_' = ok)) AND ((level - n? < 0) => ((level' = 0)
     AND (r_' = message(underflow))) AND invariant__'
   -->
     level' IN {x : NAT | TRUE};
     r_' IN (x : REPORT | TRUE);
 []
   ELSE
   -->
     level' = level
 ]
END;
```

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. It also illustrates the utility of the abbreviation facility offered by formula definitions.

The *transition* sub-clause essentially describes a control structure analogous to the *alternate* block of non-deterministic programming languages. On any one cycle, just one transition may *fire*, chosen randomly from all those whose guards are *enabled* by the model constraints. From this, it is clear that the constraints are only enforced upon one set of input or output variables in each cycle (the argument for coalescence). The *transition* clause must also include a default *ELSE*-transition, which may always fire, to ensure that the transition relation is total (for soundness of model checking). In this case, we require the automaton's state to remain unchanged.

5. Translating counted sets and relations

The heart of the Z2SAL translation deals with the Z mathematical toolkit, which provides a rich vocabulary of mathematical data types, including sets, products, relations, functions, sequences (and sometimes 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, which may then be included with the master *context*, as and when the specification requires.

5.1. The BDD-optimal encoding for sets

An initial library example is the SAL *context* for the Z set data type. This is a reusable *context*, parameterised over the element-type `T` of the set. It encodes a set as a function from elements to `BOOLEAN` values, which returns `TRUE` if a given element is a member, otherwise `FALSE`. This is a standard encoding for sets [Bry86, Bry92], optimized for symbolic model checkers that use BDDs as the core representation (see the earlier discussion about BDDs in Sect. 2):

```
set {T : TYPE; } : CONTEXT = BEGIN
 Set : TYPE = [T -> BOOLEAN];
 empty : Set = LAMBDA (elem : T) : FALSE;
 ...
 contains? (set : Set, elem : T) : BOOLEAN =
   set(elem);
 subset? (setA : Set, setB : Set) : BOOLEAN =
   FORALL (elem : T) : setA(elem) => setB(elem);
 ...
 union(setA : Set, setB : Set) : Set =
   LAMBDA (elem : T) : setA(elem) OR setB(elem);
 intersection(setA : Set, setB : Set) : Set =
   LAMBDA (elem : T) : setA(elem) AND setB(elem);
 difference(setA : Set, setB : Set) : Set =
   LAMBDA (elem : T) : setA(elem) AND NOT setB(elem);
END
```

A set is not a single, monolithic entity, but rather a polylithic membership predicate over each of its elements. The advantages of this are seen in the encoding of set constants and operations. For example, the `empty` set constant is simply a predicate that always returns `FALSE`; and the set `union` operation constructs a new predicate that computes the disjunction of the argument set predicates. A BDD-based model checker turns operations upon sets into ordered binary decisions about elements. Multiple, nested set operations become binary decision trees (BDDs) over each element, which rapidly collapse onto the two outcomes `FALSE` and `TRUE`.

This set context may be used in other contexts by instantiating the type parameter `T` and selecting types, constants or operations from the *context* using the selector: '`!`' (which is the SAL equivalent of the record-type's dot selector in other languages):

```
LOCAL members : set{PERSON; } ! Set
 ... members = set{PERSON; } ! empty ...
```

Here, the type parameter `T` is replaced by the actual `PERSON` type, such that the type given to the set-valued variable `members` is understood as the `Set` type selected from the set-of-`PERSON` context (that is, a function type `[PERSON -> BOOLEAN]` in the Bryant encoding). The `empty` set constant is selected from the context in the same way. The set *context* may instantiated by other element types, and used multiple times, as needed.

5.2. Computing the cardinality of sets

The BDD-encoding for sets causes major problems when seeking to calculate the cardinality of sets, since no monolithic, countable set object exists in the SAL translation (see Sect. 2). The need to determine the size of a set occurs quite frequently in Z, for example, when establishing limits. Earlier work [SW05] attempted to define cardinality as the search for a relation between sets and natural numbers. However, in experiments, we proved that this was inefficient to the point of being intractable [DNS06].

Our eventual preferred solution (after attempting the obvious recursive definition of sets, which failed upon the infinite BDD expansion) was to create bespoke *countN* counting-contexts, parameterised over arbitrary N

elements, according to the declared maximum capacity of the set. For example the *count*3-context supports the brute-force counting of elements in sets containing at most three elements:

```
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
```

The context has a type parameter `T` and three value parameters of this type, `e1, e2, e3`, standing for the set elements. The context declares a `Set`-type and a single operation `size?`, which computes the sum over a membership test by an exhaustive enumeration of each element. The context is used as follows, after instantiation with a type and all the elements of that type:

```
LOCAL num : NAT
LOCAL friends : set{PERSON; } ! Set
 ... num = count3{PERSON; PERSON__1, PERSON__2, PERSON__3}
             ! size? (friends) ...
```

Counting-contexts may be synthesised by the translator for sets with different maximum capacity, by varying the number of value parameters to be generated and the number of terms added to the brute-force summation. The brute-force counting approach has proven in tests to be the most efficient method. Providing separate counting-contexts to define the different versions of `size?` supports the counting of many kinds of sets (and relations) without intruding on the main set *context*.

5.3. The encoding for standard relations

The initial idea for encoding relations was to mimic the set encoding, defining a relation as a set of pairs, to preserve Z's ability to view relations also as sets. So, just as a set translates into an ordered set of propositions over elements, a relation, being a set of pairs, translates into an ordered set of propositions over pairs. Our standard SAL library *context* for the Z relation data type is parameterised over the domain element type `X` and range element type `Y`, and internally defines a number of product-types and set-types, as well as the types of the relation and its inverse:

```
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);
 range (rel : Relation) : Range =
   LAMBDA (y : Y) : EXISTS (x : X) :
     LET pair : XY = (x, y) IN rel(pair);
 ...
 image (rel : Relation, set : Domain) : Range =
   LAMBDA (y : Y) : EXISTS (x : X) : LET pair : XY = (x, y)
     IN set(x) AND rel(pair);
 inverse (rel : Relation) : Inverse =
   LAMBDA (pair : YX) : LET elem : XY = (pair.2, pair.1)
     IN rel(elem);
END
```

This translation makes maximally-efficient use of the direct encoding of relations as boolean functions, for example in the `domain` and `range` operations, where relations are applied directly to pairs, to test whether the pair is a member of the relation. To work around a fault discovered in some implementations of the SAL tool-suite, we always define a new symbolic type name for each pair-type, and, where necessary, bind a new local variable to each constructed pair-value, using the *let*-construction (see Sect. 3).

Initially, we had the option of repeating all of the set-operations in the relation-*context*, but decided against this, to better facilitate the treatment of relations as simple sets. When relations are used in the master *context*, they are declared as sets of pairs, such as `rented` in the following:

```
PERSON__X__TITLE : TYPE = [PERSON, TITLE];
...
LOCAL rented : set {PERSON__X__TITLE; } ! Set
```

This ensures that `rented` has the basic type of a `Set`. A quirk of SAL means that we may only select this type externally from one of the *contexts* (SAL treats type symbols defined in different *contexts* as distinct, even if they denote structurally identical types), so the other type names inside the relation-*context* may never be exported, but they may safely be used internally.

In order to access both set-operations and relation-operations, both of these *contexts* must be instantiated appropriately in the master *context*, for example, the following illustrates how to access the `domain` of the `rented` relation, and how to perform a set membership test for a pair, using the `contains?` operation:

```
... relation {PERSON, TITLE;} ! domain(rented) ...
... set {PERSON__X__TITLE;} ! contains?(rented, (p?, t?)) ...
```

Note that, whereas the relation-*context* is instantiated with two separate types, the set-*context* must be instantiated with the single product-type of the pair.

5.4. Special encodings for relations and sets

The success of this approach to partitioning operations over different *contexts* motivated our decision to split the complete definition of Z relations over three SAL *contexts*, according to the number of element type parameters required to type the basic sets being related. The standard context (above) provides all operations on relations between two distinct sets. A separate closure-*context* was created to provide all operations on relations closed over a single set (such as identity, transitive closure); while a third compose-*context* was created just to handle relational composition, which relates three set types. For example, the latter is a *context* with three type parameters:

```
compose{X, Y, Z : TYPE; } : CONTEXT = BEGIN
 XY : TYPE = [X, Y];
 YZ : TYPE = [Y, Z];
 XZ : TYPE = [X, Z];
 First : TYPE = [XY -> BOOLEAN];
 Second : TYPE = [YZ -> BOOLEAN];
 Composed : TYPE = [XZ -> BOOLEAN];

 compose (relA : First, relB : Second) : Composed =
   LAMBDA (pair : XZ) : EXISTS (elem : Y) :
     LET pairA : XY = (pair.1, elem),
         pairB : YZ = (elem, pair.2)
     IN relA(pairA) AND relB(pairB);
END
```

The advantage of this partitioning is clear: if the standard relation *context* were to declare the `compose` operation, it too would require three type parameters, yet the third type-instantiation would be an unnecessary overhead for most of the time. The translator identifies when specific *contexts* are needed and includes them, as required.

Further special-purpose encodings are also provided within certain *contexts* to handle common Z cases more efficiently. For example, the addition of single elements to sets is conventionally expressed in Z as the union of a set with a constructed singleton set. Likewise, the removal of a single element can only be expressed in Z as the difference of a set with a constructed singleton. The literal SAL translation of this is needlessly inefficient. Instead the set-*context* provides bespoke operations to `insert` and `remove` single elements more efficiently:

```
insert (set : Set, new : T) : Set =
 LAMBDA (elem : T) : elem = new OR set(elem);
remove (set : Set, old : T) : Set =
 LAMBDA (elem : T) : elem /= old AND set(elem);
```

We treat such operations as special optimisations, rather than as first-class extensions to the public API of the set data type. The translator identifies certain inefficient structural patterns in the parse-tree, and generates the optimised code instead.

6. Translating partial functions and sequences

There were two possible approaches to encoding Z functions in SAL. The first followed on from the treatment of relations as sets. A Z function may also be viewed as a set of pairs (but with the additional constraint of unique mappings). This approach would ease the transition between the different views of a function, as a relation, or as a set; and would presumably support dynamic maps more easily. The second approach was to use SAL's built-in function type, with its likely BDD-optimal encoding as a set of ordered judgements over variable mappings. The disadvantages of this encoding included the structural mismatch between SAL's functions and Z's sets; and the difficulty in representing partial functions (see Sect. 2) in a target language that only has total functions. We conducted a series of timing experiments using a relation-style encoding and a native SAL function encoding of Z's function types. The results confirmed that using the native SAL encoding was far more efficient.

6.1. The BDD-optimal encoding for functions

In the light of the above experiments, we selected SAL's native function encoding as the preferred basis for our standard SAL library *context* for the Z function data type. In order to handle Z's commonly-occurring partial functions, we adopted a totalising strategy, in which every type appearing in a function signature is extended with a *bottom* value, denoting the undefined element. Partial functions in Z may therefore be represented as total functions in SAL, in which some domain or range values are *bottom*. The function-*context* is parameterized over the domain element-type `X` and range element-type `Y`, but also accepts value-parameters `xb` and `yb`, denoting the *bottom* element of each of these types. This allows operations on functions to detect undefined elements and treat them specially, where required. Similar to the earlier relation-*context*, the function-*context* declares a number of types for internal use, followed by operations for public use:

```
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];
 Range : TYPE = [Y -> BOOLEAN];
 ...
 empty : Function =
   LAMBDA (x : X) : yb;
 contains? (fun : Function, pair : XY) : BOOLEAN =
   fun(pair.1) = pair.2;
 ...
```

```
 domain (fun : Function) Domain =
   LAMBDA (x : X) : x /= xb AND fun(x) /= yb;
 range (fun : Function) Range =
   LAMBDA (y : Y) : EXISTS (x : X) :
     x /= xb AND fun(x) = y AND y /= yb;
 ...
 override (f, g : Function) : Function =
   LAMBDA (x : X) : IF g(x) = yb
     THEN f(x) ELSE g(x) ENDIF;
 ...
 inverse (fun : Function) : Inverse =
   LAMBDA (pair : YX) : fun(pair.2) = pair.1
     AND pair.2 /= xb AND pair.1 /= yb;
 convert (fun : Function) : Relation =
   LAMBDA (pair : XY) : fun(pair.1) = pair.2
     AND pair.1 /= xb AND pair.2 /= yb;
 ...
END
```

Many of the commonly-used set operations are re-implemented for this different kind of encoding. For example, the `empty` map constant is encoded as a function, which always returns the *bottom* range value, `yb`. The set membership test `contains?` deconstructs the pair-argument to see whether applying the function to the first projection yields the second projection. Many commonly-used relation operations are also re-implemented, such as the operations to extract the `domain` and `range` sets. Other operations are specific to the function data type, such as `override`, which computes the union with override of two functions. Finally, a special operation `convert` is provided, in case it is desired to convert from the BDD-efficient representation back into a set of pairs, for example, when treating a function as a relation, or as a set, prior to counting its maplets.

All of these constructions take suitable note of `xb` and `yb`, the undefined *bottom* values. For example, the `domain` and `range` extractors are careful not to include these sentinels in the result. Some of the above constructions can be simplified if a specification asserts `f(xb) = yb` globally, for any function `f` (see the discussion of Z's function types, below).

6.2. Preserving well-defined inputs and outputs

Whenever the function-*context* is flagged for use by the translator, it takes note of the actual domain and range element-types, so that extra *bottom* values are inserted, when these types are generated. For example, the `TITLE` and `NAT` types are extended below, prior to declaring a function relating these types:

```
TITLE : TYPE = {TITLE__1, TITLE__2, TITLE__3, TITLE__B};
NAT : TYPE = [0..4];
...
LOCAL stockLevel : [TITLE -> NAT];
```

The extra symbolic value `TITLE__B` denotes the *bottom* value for the basic type `TITLE`, whereas the out-of-range sentinel value 4 is used as the *bottom* value for the numeric type `NAT`. Now, it is possible to instantiate the function-*context* appropriately and select library operations, as desired. Of course, the function can always be applied directly to a legal domain value to yield its image.

```
... stockLevel = function {TITLE, NAT; TITLE__B, 4} ! empty ...
... function {TITLE, NAT; TITLE__B, 4} ! domain(stockLevel) ...
... stockLevel(t?) > 0 ...
```

One consequence of extending types to include *bottom* values is that the same types may also be attached to *input* and *output* variables. This causes a new problem, since the inputs and outputs to Z operations are always supposed to be well-defined. One possible approach is to define both the standard type and the extended type, using SAL's set comprehension notation to relate these by subtyping:

```
NAT__X : TYPE = [1..4];
NAT : TYPE = {x : NAT__X | x /= 4};
TITLE__X : TYPE = {TITLE__1, TITLE__2, TITLE__3, TITLE__B};
TITLE : TYPE = {x : TITLE__X | x /= TITLE__B};
```

The notion is that the standard types are attached to inputs and outputs, while the extended types are used with functions. This works in one direction, but fails to type-check in SAL when an extended value is passed from a function operation to a standard variable, since SAL has no built-in notion of retracts.

So, we are constrained by SAL's type system to use the extended types everywhere, but instead adopt the work-around of including extra predicates asserting that *input* and *output* variables never bind to *bottom* values. These constraints are added to the state predicate after the input and output variables have been processed.

6.3. Translating Z's family of function types

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 an injective function just as a plain function, for example. Instead, the Z definitions of each function type are converted into predicates, that may be applied to any function, to assert its family properties.

The function-*context* defines atomic predicates for total and partial functions, which may be combined with the predicates for injective, surjective and bijective functions (it also provides convenient binary combinations, such as `totalSurjective?`, defined out of the more primitive forms):

```
function {X, Y : TYPE; xb : X, yb : Y} : CONTEXT = BEGIN
 ...
 Function : TYPE = [X -> Y];
 ...
 partial? (f : Function) : BOOLEAN =
   f(xb) = yb;
 total? (f : Function) : BOOLEAN =
   FORALL (x : X) : IF x = xb THEN f(x) = yb
     ELSE f(x) /= yb ENDIF;
 injective? (f : Function) : BOOLEAN =
   FORALL (x1, x2 : X) : (f(x1) /= yb AND
     f(x1) = f(x2)) => (x1 = x2);
 surjective? (f : Function) : BOOLEAN =
   FORALL (y : Y) : EXISTS (x : X) : f(x) = y;
 bijective? (f : Function) : BOOLEAN =
   injective?(f) AND surjective?(f);
 ...
END
```

An important property of these predicates, which seems counter-intuitive at first, is that a mapping must always exist in SAL for the *bottom* domain element, even where the Z function is total. This is because the BDD compilation of each function converts into an ordered set of mappings from every domain element to some range element and, from SAL's viewpoint, the *bottom* domain value is like any other. Initially, we had asserted totality using: `FORALL (x : X) : f(x) /= yb`, but later found during testing that this led to deadlock (systems with no future states), hence the more careful translation above.

When translating a Z function with one of these function types, the SAL output includes suitable function family predicates, conjoining these with the state invariant. In our latest translation, one of `total?` or `partial?` is always asserted, such that `f(xb) = yb` is a global property. This allows the removal of the duplicated side conditions: `x /= xb` in some operations of the function-*context*.

6.4. The translation of Z standard sequences

Following the Z treatment of sequences as functions from element-type to $\mathbb{N}_1$, we wanted to model sequences as SAL functions, which are analogous to SAL arrays, in that both compile to an ordered set of judgements on pairs in the BDD representation. However, the ordered property of sequences suggests an optimisation in computing the length of the sequence.

Our SAL *context* for a Z sequence treats this as a tuple of a fill-counter and a function, where the counter is the length (possibly zero) of the sequence, and the function maps from a non-zero index to a value. Typical operations include deconstructing a sequence at the head or tail ends, or concatenating two sequences. An additional constraint for sequences is that all mapped indices must be contiguous, from $1..n$, with no *bottom* mapping appearing medially. Furthermore, the result of concatenating two sequences might exceed the range of the index type, in which case it must be possible to detect overflow in the encoding. All this is facilitated by parameterising the sequence-*context* by the element-type, the bottom element value and the maximum anticipated length:

```
sequence {X : TYPE; xb : X, max : NATURAL} : CONTEXT = BEGIN
 Index : TYPE = [1..max];
 Size : TYPE = [0..max];
 Function : TYPE = [Index -> X];
 Sequence : TYPE = [Size, Function];
 ...
 empty : Sequence = (0, LAMBDA (n : Index) : xb);
 undefined : Sequence =(max, LAMBDA (n : Index) : xb);
 ...
 size? (seq : Sequence) : Size =
   seq.1;
 convert (seq : Sequence) : Function =
   seq.2;
 ...
 head (seq : Sequence) : X =
   seq.2(1);
 tail (seq : Sequence) : Sequence =
   IF seq.1 = 0 THEN undefined ELSE (seq.1 - 1,
     LAMBDA (n : Index) : IF n = seq.1 THEN xb
       ELSE seq.2(add1(n)) ENDIF)
   ENDIF;
 ...
 concat (first, second : Sequence) : Sequence =
   IF (first.1 + second.1 > max) THEN undefined
     ELSE (first.1 + second.1, LAMBDA (n : Index) :
       IF n <= first.1 THEN first.2(n)
         ELSE second.2(n - first.1) ENDIF)
   ENDIF;
 ...
 valid? (seq: Sequence) : BOOLEAN =
   FORALL (i : Index) : IF i > seq.1 THEN seq.2(i) = xb
     ELSE seq.2(i) /= xb ENDIF;
 ...
END
```

The internal types distinguish `Size`, including zero, from `Index`, excluding zero. Apart from simple operations such as `size?`, which projects the fill-counter, and `convert`, which projects the function, most operations have to handle boundary cases. For example, if a sequence has zero length, `head` may return the *bottom* element, and `tail` may return the *undefined* sequence. Technically, the Z mathematical toolkit only defines these operations for seq_1, the non-empty sequence type, which we declare using a predicate constraint, similar to the treatment of function family types; however, these operations must still be total over extended domains in SAL, for completeness of model checking.

Likewise, `concat` may return an `undefined` sequence as a consequence of overflow in the representation. This is an extra consideration forced upon the model by the limitations of the implementation. We assume that, during model checking, we may assert that a sequence is still *valid*, as part of the system invariant. The heart of this validity-check is a contiguous range check for mapped indices, which the explicit fill-counter makes easier to implement. Having `valid?` as a separate predicate also offers the flexibility to distinguish between an inconsistent specification, or a failed representation.

Apart from this, a number of operations from the relation-*context* are duplicated, such as the `domain` and `range` operations, where these may benefit from the fill-counter (like `size?`). Elsewhere, the expectation is that the sequence may be treated as a function (by using `convert`), which in turn may be treated as a set of pairs, as described above.

6.5. Special encodings for sequences and functions

Similar to the function types, Z provides two distinguished sequence types, seq_1, the non-empty sequence, and iseq, the injective sequence. In SAL these are encoded as predicates on sequences, rather than as separate types (see the similar argument above for function types):

```
notEmpty? (seq : Sequence) : BOOLEAN =
 seq.1 > 0 AND valid?(seq);
injective? (seq : Sequence) : BOOLEAN =
 FORALL (i, j : Index) : (seq.2(i) /= xb AND
   seq.2(i) = seq.2(j)) => (i = j);
```

The first of these illustrates the need to handle both parts of the sequence's representation. It is not enough simply to assert that the fill-counter is non-zero in `notEmpty?`, since this might permit the tool-suite to infer representations of the function part which were meaningless; hence the use of `valid?` to ensure that the remaining mappings are contiguous.

In the same way that special encodings were developed to handle the insertion and removal of single elements from sets, cognate operations are also provided for both functions and sequences. Optimised operations are provided in the function-*context* to insert and remove single maplets:

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

which is much more efficient than performing function override with a singleton function. Likewise for sequences, optimised operations are provided to insert (prepend) or append single elements:

```
insert (seq : Sequence, val : X) : Sequence =
 IF (seq.1 = max) THEN undefined
   ELSE (seq.1 + 1, LAMBDA (n : Index) :
     IF n = 1 THEN val ELSE seq.2(sub1(n)) ENDIF)
 ENDIF;
append (seq : Sequence, val : X) : Sequence =
 IF (seq.1 = max) THEN undefined
   ELSE (seq.1 + 1, LAMBDA (n : Index) :
     IF n = seq.1 + 1 THEN val ELSE seq.2(n) ENDIF)
 ENDIF;
```

which is much more efficient than concatenation at the head or tail end with a singleton sequence. The translator identifies structures containing singletons and uses translation templates to replace the less efficient construction with the optimised one.

In early tests, it was found that simple arithmetic on indices did not typecheck in SAL, because of the possibility of a sum or difference going out of the range of the `Index` type. The work-around for this was two helper-functions `add1` and `sub1`, which map arithmetic to the limits of the range, purely for the sake of type checking (the validity of the representation is handled separately).

7. Case study

Here we present a small example to illustrate some of the ideas presented above. First, a Z specification is given, representing the operation of a video shop which rents out videos to its subscribed members. The SAL translation is then used as the input to the SAL tool-suite, both for simulation and for model-checking. The tools are able to verify expected properties, but also discover incomplete properties of the specification. The performance of the tool-suite on our latest improved SAL encodings is also presented.

In the Z specification, $PERSON$ is the type of members and $TITLE$ is the type of videos. The state of the video shop business describes a set of subscribed $members$, a relation $rented$ mapping from the members to the (many) videos that they currently each rent, and a function $stockLevel$ describing how many copies of each video the shop owns. Initially, there are no members and there is no stock. The stock level may be modified by $AddTitle$, which adds a number of copies of a video, and $DeleteTitle$, which removes all copies of a video (provided none are rented). The membership only increases monotonically using $AddMember$. $RentVideo$ loans a video to a member, if the shop has available stock (and the video is not already loaned to the member). There is no facility to return videos to the shop. $CopiesOut$ reports how many copies of a video are currently on loan.

$[PERSON, TITLE]$

State

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

Init

$$State'$$
$$members' = \emptyset$$
$$stockLevel' = \emptyset$$

RentVideo

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

AddTitle

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

DeleteTitle

$$\Delta State$$
$$t? : TITLE$$
$$t? \notin \text{ran}\, rented$$
$$t? \in \text{dom}\, stockLevel$$
$$stockLevel' = \{t?\} \ndres stockLevel$$
$$rented' = rented$$
$$members' = members$$

AddMember

$$\Delta State$$
$$p? : PERSON$$
$$p? \notin members$$
$$stockLevel' = stockLevel$$
$$rented' = rented$$
$$members' = members \cup \{p?\}$$

CopiesOut

$$\Xi State$$
$$t? : TITLE$$
$$copies! : \mathbb{N}$$
$$t? \in \text{dom}\, stockLevel$$
$$copies! = \#(rented \rhd \{t?\})$$

The following describes the SAL output, resulting from the translation of the above Z specification. This uses the latest SAL encodings reported in this paper, and so differs from the example reported in [DNS08], in that the number of SAL *input* and *output* variables is reduced by coalescing (see Sect. 4); and the initial values of output variables are clamped.

```
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 p? : PERSON
   INPUT t? : TITLE
   INPUT level? : NAT
   OUTPUT 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
       partial?(stockLevel) AND
       t? /= TITLE__B AND
       level? /= 4 AND
       copies_ /= 4)
   INITIALIZATION [
     members = set {PERSON;} ! empty AND
     stockLevel = function {TITLE, NAT; TITLE__B, 4} ! empty AND
     copies_ = 1 AND
     invariant__
   -->
   ]
   TRANSITION [
     RentVideo :
       set {PERSON;} ! contains?(members, RentVideo__p?) AND
       set {TITLE;} ! contains?(function {TITLE, NAT; TITLE__B, 4} !
         domain(stockLevel), t?) AND
       stockLevel (t?) > PERSON__X__TITLE__counter !
         size?(relation {PERSON, TITLE;} ! rangeRestrict(rented,
           set {TITLE;} ! singleton(t?))) AND
       NOT set {PERSON__X__TITLE;} ! contains?(rented, (p?, t?)) AND
       rented' = set {PERSON__X__TITLE;} ! insert(rented, (p?, 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, (t?, 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), t?) AND
       set {TITLE;} ! contains?(function {TITLE, NAT; TITLE__B, 4} !
         domain(stockLevel), t?) AND
      stockLevel' = function {TITLE, NAT; TITLE__B, 4} !
        domainSubtract(set {TITLE;} ! singleton(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, p?) AND
      stockLevel' = stockLevel AND
      rented' = rented AND
      members' = set {PERSON;} ! insert(members, 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), t?) AND
      copies_' = PERSON__X__TITLE__counter ! size?(
        relation {PERSON, TITLE;} ! rangeRestrict(rented, set {TITLE;} !
          singleton(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};
      copies_' IN { x : NAT | TRUE}
    []
    ELSE -->
      members' = members;
      rented' = rented;
      stockLevel' = stockLevel
  ]
 END;

END
```

This translation demonstrates a large subset of the SAL encodings described above, including some further relational operations, such as `domainSubtract` and `rangeRestrict`, applied variously to the preferred set-encoding and native SAL function-encoding. The `singleton` operation is a further special-purpose set constructor. Note in the prequel that the translator (only) synthesised two *bottom* values for the types `NAT` and `TITLE`, since these appear in the function signature, whereas `PERSON` does not. Also, a large $count12 - context$ is synthesised to support counting of the `rented` relation; and an abbreviated name is defined for the instantiated context, to facilitate the readability of the translation.

7.1. Simulating with the example

The SAL translation of the example was then animated using the `sal-sim` simulator from the SAL tool-suite. This tool supports interaction with the user, who can step forward in time from the initial state(s), and view the current states, or arbitrary traces. Timing information was also obtained from the tool. We ran all experiments on a shared Unix server running Solaris 9, from a desktop client. Because of shared processor usage, all times reported here and below were the average times taken across five repetitions of each timing experiment.

The example SAL file is loaded using `(import! "example")`, and the tool takes approximately 0.3 s to parse and type-check the various *contexts*. The example is then compiled to BDDs by invoking the command `(start-simulation! "State")`, where *State* is the name of the main SAL *module* driving the simulation. The compilation process takes around 6.15 s for the current example. This involves flattening the module, simplifying the AST, expanding function applications, unfolding quantifiers and eliminating common sub-expressions, converting to a *boolean* flat module, then to the BDD format. This is then optimised by ordering the variables to minimise support, then rearranging and compressing the BDD clusters [dMOS03].

The current number of states in the system may be viewed using the `(display-curr-states N)` command, where N is the maximum number of states to display in full to the user (the tool always reports the total number of states). An arbitrary trace may also be displayed using the `(display-curr-trace)` command. The trace is selected at random; and a different trace is typically displayed on subsequent invocation (repeatedly executing this command is not guaranteed to iterate over all traces). The output is in a particular textual format, close to the BDD representation, which we convert to a more readable tabular format below.

To illustrate the simulator's state-exploring behaviour, the example was advanced through a series of ten state-transitions using the `(step!)` command; and the number of states were counted after each step, using the `(display-curr-states N)` command. To evaluate the effects of the latest revisions to the SAL translation, we simulated three different encodings. The first is the original encoding reported in [DNS08], with replicated *input* and *output* variables for each operation schema and no *output*-clamping. The second version clamps the initial values of *output* variables to the first value in the range of the type (chosen arbitrarily). The third (and current) version also coalesces *input* and *output* variables across all Z operation schemas. These measures resulted in a progressive reduction in the state-space:

Example version:	Original version	+ Clamped init outputs	+ Coalesced i/o variables
	states	states	states
step 0 :	11664	2916	36
step 1 :	221040	55260	600
step 2 :	1752048	438012	4524
step 3 :	7918848	1979712	19752
step 4 :	24593328	6227064	60756
step 5 :	61568640	16516764	152580
step 6 :	134093232	40250196	331212
step 7 :	256801824	87486660	641184
step 8 :	443716992	168270120	1138080
step 9 :	*memory*	292802220	1878780
step 10 :	*memory*	468633996	2888976

From this, it can be seen that clamping initial outputs resulted initially in a 4.0x state reduction, tending to a 2.6x reduction by step 8 (the original version exhausted memory after this step). Since our example only has one *output* variable `copies_`, we would expect greater reductions in examples with more outputs.

The effect of coalescing all similarly-named *input* and *output* variables had a more dramatic effect. Compared to the intermediate version, this resulted initially in a 81.0× state reduction, tending to a 162.2× state reduction by step 10. Coalescing variable names will give the best state reductions for Z authoring styles that choose identical names for similar variables in different operation schemas.

Altogether, our latest translation has therefore reduced memory usage by a factor of something between 324.0× (at step 0) and 384.5× (by step 8) for this example. This reduction is useful, since it will allow the simulation of more complex examples with more states, without exhausting memory.

To illustrate the simulator's trace-reporting behaviour, the example was advanced through a series of ten state-transitions using the `(step!)` command; and arbitrary traces were then displayed using the `(display-curr-trace)` command. One illustrative example trace is the following:

Step	Transition	Updates
0	Init	$members, rented, stockLevel = \emptyset$
1	AddTitle	$stockLevel(TITLE\_2) = 3$
2	AddMember	$PERSON\_2 \in members$
3	RentVideo	(PERSON_2, TITLE_2) $\in rented$
4	AddTitle	$stockLevel(TITLE\_1) = 2$
5	RentVideo	(PERSON_2, TITLE_1) $\in rented$
6	AddTitle	$stockLevel(TITLE\_3) = 0$
7	AddTitle	$stockLevel(TITLE\_1) = 0$
8	AddMember	$PERSON\_1 \in members$
9	RentVideo	(PERSON_1, TITLE_2) $\in rented$
10	Else	no change

From this, it can be seen that the system acquired some videos and members and rented videos to some of the members. These behaviours were as expected, and executed in the logical order of dependency (e.g. in step 1, 3 copies of the video `TITLE__2` were acquired; in step 2, `PERSON__2` joined the video club; and in step 3, `PERSON__2` was rented a copy of `TITLE__2`).

However, the simulation also reveals a semantic fault in the original Z specification, namely that it is possible to reset the quantity of each video in stock independently of the number of copies that are already on loan (in step 5, one copy of `TITLE__1` is rented to `PERSON__2`, but in step 7, the stock level of `TITLE__1` is reset to zero). The semantic fault in the specification is that `AddTitle` merely replaces the previous stock-level, rather than adding to it. It is also arguably pointless, in steps 6 and 7, to permit adding zero copies of a video. This ability to view traces illustrates how useful simulation can be when initially validating a specification, looking for obvious omissions and faults. The detection of less obvious faults may be accomplished through model-checking, described below.

A final point to note is that, in step 10 above, the simulator has arbitrarily selected the default `ELSE`-transition, a nullop that is always possible, in case a simulation would otherwise deadlock. This transition allows a step to be taken, even if the input conditions prevent any other transition from firing. The consequences are that the values of the `LOCAL` state variables must remain unchanged, even if the input conditions would violate the invariant.

7.2. Model-checking with the example

The SAL tool-suite provides several simple and bounded model-checkers that support both LTL and CTL temporal logics. We have used the simple model-checker `sal-smc` and also the bounded model-checker `sal-bmc` for checking to known bounded depths. At the moment, we add theorems by hand to the end of the translated SAL file. Eventually, we propose to extend the Z notation to support temporal logic expressions.

The proof strategy adopted can either be positive, in which theorems are proved or falsified directly, or negative, following a refutation strategy, in which counter-theorems are proposed in the expectation that they will be disproved. The latter approach is often more revealing, since the tools provide traces whenever a counter-example is found. In the positive approach, it is sometimes difficult to distinguish between proof success and mere failure to find a counter-example within a certain depth of exploration. Therefore we always seek to frame queries in such a way as to force the model checker to reveal how it found the solution.

The following illustrates the refutation approach using counter-theorems. Suppose that we want to show that videos eventually get rented to subscribed members of the video shop. In SAL, we propose the negation of this property as a counter-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 using the command to check theorem 1 in the example: `sal-smc exam\-ple th1` and this generates the smallest counter-example that proves the desired property:

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

For this theorem, the counter-example is found in three steps. This is the shortest path which stocks 3 copies of one video title, adds a member, then rents a video to that member. The `sal-smc` model-checker reports that it finds the counter-example within 4.45 s, of which most time is taken up compiling the example. The actual searching time is approximately 0.14 s.

To investigate further the execution times for checking counter-theorems of varying complexity in our different translation models, we devised a number of counter-theorems to test the searching capability of the tools under different conditions:

```
th1 : THEOREM State |- G (set {PERSON__X__TITLE;} ! empty?(rented));
 %% "No video copies are ever rented to people."

th2 : THEOREM State |- G (NOT (members = set {PERSON;} ! full));
 %% "The video club membership is never complete."

th3 : THEOREM State |- G (copies_ /= 3);
 %% "3 copies of a video are never on loan at any time."

th4 : THEOREM State |- G (FORALL (t : TITLE) : stockLevel(t) /= 3);
 %% "There are never 3 video copies stocked for any title."

th5 : THEOREM State |- G (FORALL (t : TITLE) :
 stockLevel(t) >= PERSON__X__TITLE__counter ! size?(
   relation {PERSON, TITLE;} ! rangeRestrict(rented,
     set {TITLE;} ! singleton(t))) );
 %% "The number of copies in stock is never less than the number rented."
```

```
th6 : THEOREM State |- G (NOT (
 (FORALL (p : PERSON) : FORALL (t : TITLE) : (
   t /= TITLE__B => LET pair : PERSON__X__TITLE = (p, t) IN
    set {PERSON__X__TITLE;} ! contains? (rented, pair))
   AND stockLevel(t) >= 3 ) ));
 %% "It never happens that every person rents a copy of every video
 %% and at least 3 copies are stocked of that video."

th7 : THEOREM State |- G (NOT (
 (FORALL (p : PERSON) : FORALL (t : TITLE) : (
   t /= TITLE__B => LET pair : PERSON__X__TITLE = (p, t) IN
    set {PERSON__X__TITLE;} ! contains? (rented, pair))
   AND stockLevel(t) >= PERSON__X__TITLE__counter ! size?(
     relation {PERSON, TITLE;} ! rangeRestrict(rented,
       set {TITLE;} ! singleton(t))) ) ));
 %% "It never happens that every person rents a copy of every video
 %% and the stock of that video equals or exceeds the number of copies
 %% rented."
```

These ranged from simple counter-theorems designed to reveal that extreme states of the specification could be reached, to more complex counter-theorems designed to exercise as many transitions as possible, within the constraints of the (admittedly rather small) pedagogical example. Theorem 1 was a sanity-check, seeking to verify that videos can be rented; Theorems 2–4 were designed to reach the maximum population of sets, or the limits of ranges; Theorem 5 was an example of a plausible positive system property to seek to verify; and Theorems 6–7 were designed to exercise the maximum number of transitions to reach an extreme state. Theorem 7 essentially asks the same question as Theorem 6, but in a more circumlocutory way that exercises more of the functions from the mathematical toolkit.

Timings were collected for checking each counter-theorem in each of the three translation models (the original encoding; the intermediate encoding with clamped initial output values; and the current encoding with clamped initial outputs and coalesced input/output variables).

Example version:	Steps taken	Original version	+Clamped init outputs	+Coalesced i/o variables
theorem 1	3	4.61 s	4.68 s	4.40 s
theorem 2	3	4.53 s	4.51 s	4.37 s
theorem 3	8/0*	4.39 s*	5.48 s	5.10 s
theorem 4	1	4.55 s	4.51 s	4.40 s
theorem 5	4	9.22 s	9.47 s	9.03 s
theorem 6	15	6.91 s	7.47 s	6.52 s
theorem 7	15	11.42 s	12.04 s	10.83 s

All counter-examples were found within zero to fifteen steps. The execution times ranged from 4.37 to 12.04 s. Broadly speaking, the current encoding demonstrates slightly improved execution times over the original encoding (ranging from 0.2 to 0.6 s faster) and the intermediate version made both gains and losses over the original encoding (ranging from 0.6 s slower, to 0.04 s faster). It should be noted that all these execution times were dominated by the compilation time for the example, which also increased with the complexity of the theorem. The actual verification times constitute around 20–25% of the overall execution time; and so we estimate that verification times are around 10% faster for the current encoding.

It is interesting that clamping initial outputs alone appears to have a small time penalty in most cases, reflecting the additional time taken to compile the assignments and execute the constraints. On the other hand, coalescing input/output variables reduces the execution time across all examples. There was one anomaly (theorem 3, original version) where a counter-example was found in zero steps (rather than the expected eight steps) in the original version. This was due to the fact that the output variable `copies_` was not initially clamped, such that the prover could choose to assign any value it liked in the initial state!

All of the refutation examples above showed, by counter-example, that expected properties of the specification were present, in that certain limiting states of the specification could indeed be reached. The one positive proof example (Theorem 5) was devised to detect the semantic fault already identified above by simulation. The theorem asks whether there are always at least as many copies of a video in stock as there are on loan to members. Because of the semantic fault in the original specification, which allows stock levels to be reset independently, this property is violated. The model-checker correctly finds a counter-example within the minimum of four steps:

Step	Transition	Updates
0	Init	$members$, $rented$, $stockLevel = \emptyset$
1	AddTitle	$stockLevel(TITLE\_2) = 3$
2	AddMember	$PERSON\_3 \in members$
3	RentVideo	(PERSON_3, TITLE_2) $\in rented$
4	AddTitle	$stockLevel(TITLE\_2) = 0$

The model-checker had to work hardest for Theorem 7, which forced the system through 15 transitions (viz. 3 members were added, 3 copies of each video were acquired and 9 rentals were offered) to reach the limiting state where each member has a copy of each video. The side condition that the number of copies in stock should exceed the number of copies on loan forced the checker to evaluate an additional range restriction and set element count, compared to Theorem 6, in which the minimum stock level was expressed directly as a constant. Comparing the original encoding with the current encoding, the compilation time was reduced from 8.91 to 8.58 s; and the verification time from 2.51 to 2.25 s.

8. Conclusion

In this paper we have discussed our current approach to translating Z into the input language of the SAL tool-suite, with a view to providing a model-checking capability for Z. Our encoding is close to the optimal format for the tool-suite's internal BDD structures, and is informed by heuristics for reducing the state-space.

Whilst we have demonstrated the feasibility of a translation of a large part of Z there are still some limitations. These fall into two areas—the limitations of our translation and the limitations of SAL. While the SAL tools are reasonably stable, they have some deficiencies from our point of view. The most notable of these are: the failure to cope with recursive types; and an unpredictable error when processing tuples and their types. However, new versions of the tool-suite continue to be released and it is to be hoped that these problems will be resolved in time. Of course there is still work to be done on our translation of Z. For example, we have yet to resolve some issues with the representation of sequences and schema calculus expressions; and have not yet attempted the translation of bags.

Despite these limitations, this still appears to be a promising area to explore. The frequency with which we have discovered new adjustments to the translation which yield greatly improved performance in the model-checker suggests that there is still useful work to be done in this area. It would also be worthwhile to explore how our approach scales up, by conducting some performance comparisons with other similar tools such as ProB. While this paper reports a work in progress, we have established that the approach is feasible and a promising area for further work.

Acknowledgements

This work initially grew out of a collaboration with the University of Queensland, in particular, with Graeme Smith and Luke Wildman. Tim Miller gave valuable advice on the current CZT tools. We are also indebted to Mark Utting for a discussion that led to the optimisation strategy of coalescing input and output variables.

References

[Abr96] Abrial J-R (1996) The B-book: assigning programs to meanings. Cambridge University Press, New York

[Bol05] Bolton C (2005) Using the alloy analyzer to verify data refinement in Z. Electron Notes Theor Comput Sci 137(2):23–44

[Bry86] Bryant RE (1986) Graph-based algorithms for boolean function manipulation. IEEE Trans Comput 35(8):677–691

[Bry92] Bryant RE (1992) Symbolic boolean manipulation with ordered binary-decision diagrams. ACM Comput Surv 24(3):293–318

[AWS08] Chantar H, Wali A, Sosa A, Sharma Y (2008) Translating Z to SAL. Technical report, Department of Computer Science, University of Sheffield, Sheffield, May 2008

[CGL94] Clarke EM, Grumberg O, Long DE (1994) Verification tools for finite-state concurrent systems. In: A decade of concurrency, reflections and perspectives, REX school/symposium. Springer, London, pp 124–175

[dMOS03] de Moura L, Owre S, Shankar N (2003) The SAL language manual. Technical Report SRI-CSL-01-02 (Rev.2), SRI International

[DNS06] Derrick J, North S, Simons T (2006) Issues in implementing a model checker for Z. In: Liu Z, He J (eds) ICFEM, Lecture notes in computer science, vol 4260. Springer, pp 678–696

[DNS08] Derrick J, North S, Simons AJH (2008) Z2SAL—building a model checker for Z. In: Börger E, Butler MJ, Bowen JP, Boca P (eds) ABZ. Lecture Notes in Computer Science, vol 5238. Springer, pp 280–293

[DUT+03] Daley N, Utting M, Toyn I, Dong JS, Martin A, Currie D (2003) ZML: XML support for standard Z. In: 3rd international conference of Z and B users (ZB03). LNCS, Springer, p 2651

[Hol97] Holzmann GJ (1997) The model checker SPIN. IEEE Trans Softw Eng 23(5):279–295

[135] ISO/IEC 13568:2002. Information technology—Z formal specification notation—syntax, type system and semantics. International Standard.

[Jac02] Jackson D (2002) Alloy: a lightweight object modelling notation. ACM Trans Softw Eng Methodol 11(2):256–290

[LB05] Leuschel M, Butler M (2005) Automatic refinement checking for B. In: Lau K, Banach R (eds) International conference on formal engineering methods, ICFEM 2005, LNCS, vol 3785. Springer, pp 345–359

[MFMU05] Miller T, Freitas L, Malik P, Utting M (2005) CZT support for Z extensions. In: Romijn J, Smith G, Pol J (eds) Integrated formal methods, IFM 2005, LNCS, vol 3771. Springer, pp 227–245

[PL07] Plagge D, Leuschel M (2007) Validating Z specifications using the ProB animator and model checker. Integr Form Methods 4591:480–500

[Saa97] Saaltink M (1997) The Z/EVES system. In: Bowen JP, Hinchey MG, Till D (eds) ZUM, Lecture notes in computer science, vol 1212. Springer, pp 72–85

[Saa99] Saaltink M (1999) The Z/Eves 2.0 User's Guide. ORA Canada, http://www.cs.kent.ac.uk/people/staff/gsn2/zeves/usersguide.pdf

[Spi92] Spivey JM (1992) The Z notation: a reference manual. Prentice Hall, Englewood Cliffs

[Spi00] Spivey MJ (1988–2000) The fuZZ Manual, 2nd edn. Spivey Partnership, http://spivey.oriel.ox.ac.uk/mike/fuzz/fuzzman.pdf

[SW05] Smith G, Wildman L (2005) Model checking Z specifications using SAL. In: Treharne H, King S, Henson S, Schneider S (eds) International conference of Z and B users, LNCS, vol 3455. Springer, pp 87–105

[TM95] Toyn I, Mcdermid JA (1995) CADiZ: An architecture for Z tools and its implementation. Softw Pract Exp 25:305–330

Received 4 February 2009

Accepted in revised form 25 August 2009 by Jonathan P. Bowen and Michael J. Butler