Support for Model Checking Z Specifications

Maria Ulfah Siregar[1,2]

[1] PhD Student in the Department of Computer Science, The University of Sheffield
Regent Court, 211 Portobello, Sheffield, S1 4DP, United Kingdom
acp12mus@sheffield.ac.uk
[2] Lecturer in Informatics Department, UIN Sunan Kalijaga
Yogyakarta, Indonesia

Abstract. One of the deficiencies of Z tools is that there is limited support for model checking Z specifications. Building a model checker directly for a Z specification will take considerable amount of effort and time due to the abstraction of the language. Translating a Z specification input into a specification in a language that an existing model checker tool accepts is an alternative method. Researchers at the University of Sheffield implemented a translation tool, which they called Z2SAL that takes a Z specification and translates it into the input for Symbolic Analysis Laboratory (SAL), which is a framework for combining different tools for abstraction, program analysis, theorem proving and model checking. This paper discusses support for model checking Z specifications, in which the capability of Z2SAL is extended. This support includes a translation of a generic constant and a schema calculus definition. Instead of translating these aspects of the Z language into the SAL language as Z2SAL does, a Z specification containing these two notations will be pre-processed, in which a generic constant definition is redefined to an equivalent axiomatic definition and a schema calculus definition is expanded to a new schema definition. As a result of a successful redefinition or expansion, a redefined or expanded Z specification is generated, otherwise the Z specification input is returned.

1 Introduction

As a formal language, the use of Z in academia and industry has increased considerably. This is because Z has been used successfully to address a large variety of problems and the international standard was also designed for this language. The use of Z can make a specification more formal and free from ambiguity. In addition, Z allows a specification to be analysed mechanically [1]. Designing a specification of a system enables a user to verify the system since an early stage of the system development. An early verification can avoid high cost in a system implementation and test phases, if the specification was designed correctly [2–4]. Therefore, a specification is crucial for a system, especially if the system relates to safety of property and/or life.

However, there is a lack of tools for this language, especially in model checking Z specifications. Although the Community Z Tools (CZT) project is developing

continuously a set of open source tools for Z, progress of this development is slow [5]. There are many causes of the shortage of Z tools. These are mostly related to the Z language and semantics, such as an inherent expressiveness and a difficulty in deciding effectively any theorem about Z specifications [5, 1]. Another cause is the richness of this language, which can also be the issue of verifying Z [1]. Furthermore, only a few of these tools can be used in validating intended meanings of such Z specifications [6].

The lack of supporting tools for the Z language and the above issues, make researchers suggest an alternative method which is a quick approach: reuse and adapt existing tools. Researchers at the University of Sheffield implemented the Z2SAL translator [5] which generates a SAL specification from a Z specification input. The generated SAL file can be model checked later by the SAL model checker. A brief introduction to Z2SAL and SAL is given in the Section 2.

In our study, several experiments using Z2SAL and SAL are performed. Our finding is Z2SAL supports many Z tags, but not all. Furthermore, sometimes several generated SAL files could not be verified or simulated by the SAL tool.

Therefore, this paper intends to address problems as stated below:

1. What are crucial features of Z should be implemented to enhance the ability of Z2SAL and why?
2. How is the implementation that supported by Z2SAL and SAL?

These questions will be explored in the Section 3. Our aim is to support model checking Z specifications so as to broaden the applicability of model checking.

The paper is organized as follows. Section 2 describes briefly an introduction to Z2SAL and SAL. Section 3 contains our support for model checking Z specifications. This section is divided into two sub-sections. Section 3.1 presents our support for generic constant definitions. Section 3.2 explains another support which is schema calculus definitions. Section 4 concludes this paper and offers several future works.

2 A Brief Introduction to Z2SAL and SAL

Several tools in Z were developed based on the quick approach, such as *ProZ* [6] which is a translator of Z into the existing Alloy Analyser tool, *ProB* [7]; data refinement verification [8] which uses Alloy SAT-solver based on a counter-example finder; and Z2SAL [5] which is a translator of a Z language specification into a SAL language specification [9].

Smith and Wildman at the University of Queensland, Australia, described how to translate a Z language specification into a SAL input language specification [10]. This basic idea was implemented in a tool set [11] and the current Z2SAL extends it in a different direction, and to tackle optimization issues [5].

In providing a translator of Z into an input language of existing tools, SAL was chosen since it has an *equivalent representation* of many aspects of Z [11]. Moreover, *many different tools exist, which use the SAL input language such as simulator, model checker either symbolic or bounded, deadlock checker, etc.*

[5], which are offered freely by Stanford Research Institute (SRI) International under academic licences.

A generated SAL file consists of a SAL module and/ or several SAL contexts. This module describes a transition system of Z states [11]. The simple SAL module has a general format as follows:

```
State : MODULE =
  BEGIN
    INPUT ...
    LOCAL ...
    OUTPUT ...
    INITIALIZATION [ ... ]
    TRANSITION [
       ...
    ]
END
```

The SAL context is a place to declare types, constants, modules and modules properties [9]. Z2SAL formulates several Z mathematical tool-kits, which are necessary for a generated SAL specification, in separate but integrated SAL context files.

Translating a Z language specification into a SAL input language specification requires several adjustments due to the number of differences of both languages [5]. These adjustments are discussed briefly as given below:

First, it is *bounding the infinite*. Z supports *fully abstract* (non-grounded, non-constructive) specification styles, whereas SAL is a *concrete and grounded language*. For example, Z supports the built-in numerical types $\mathbb{Z}$, $\mathbb{N}$ and $\mathbb{N}_1$, whose ranges are infinite. On the other hand, SAL has similar unbounded types `INTEGER`, `NATURAL` and `NZNATURAL`, which can be used only as base types of finite sub-ranges in a SAL specification. Z supports also given types, which have semantics of an un-interpreted set, such as `[TAPE, NAME]`. Therefore, the translations provided by Z2SAL should specify a finite number for sizes of these sets.

The *mismatched formal paradigms* are the second difference. Z and SAL have very different styles of specifications and descriptions. The Z specification, which consists of state schemas and operational schemas, is built-up increasingly. It views locally and functionally such that every operational schema operates on its input and output variables, or on variables of state schemas. In contrast to SAL, the SAL specification is created as a 'monolithic finite state automaton' (FSA) such that all inputs, outputs and local variables are compiled into aggregate states and all operations act upon guard transitions from one state configuration to another state configuration [5]. Thus, this mismatch can be approached by re-ordering all information in the Z specification. A further mismatch is that Z specifications often use partial functions. On the other hand, as SAL is based on *Binary Decision Diagrams* (BDDs), SAL always requires a representation of function as a total function. Thus, a work-around is necessary in order to present a partial function in Z specifications as a total function in SAL. Furthermore, a

set cannot be treated as a monolithic FSA of SAL, but as a 'poly-lithic collection of judgements' over its elements instead. Thus, several operations in sets are necessary to be expressed differently, such as the cardinality of a set, which is not supported by SAL.

The last difference is an issue of *non-computable specifications*. A Z specification naturally supports non-constructive styles of a specification. These styles should be expressed in computable styles of a specification in SAL. Both styles essentially are different indeed. Normally, a SAL specification consists of a set of update assignments to primed variables, which indicates posterior variable states. In contrast to Z, a direction of a constructive approach is not necessary in a Z specification. Z2SAL asserts posterior existences of variables and restricts their values on preconditions. This requires a search for suitable precondition values.

More information relating to Z2SAL is provided in [12]. It includes also a downloadable version of this translation tool.

SAL is a framework of several different tools such as abstraction, program analysis, theorem proving and model checking, which is used to change perceptions and implementations of model checkers and theorem provers. These perceptions and implementations at first were based on verification, but they were changed to a calculation of properties or symbolic analysis such as abstraction, slicing and composition [9, 13].

The SAL language can be used as a specification language, a target language for several translators, or a common source of several analysis tools. It originated of a collaboration of two researchers, David Dill from Stanford University and Thomas Henzinger from the University of California at Berkeley. These collaborations evolved and included Verimag in it. SAL is developed at SRI now and its current version is 3.3. The SAL language syntax can be found in [9].

The next section describes our support for model checking Z specifications.

3 Support for Model Checking Z Specifications

Based on our experiences using Z2SAL, two aspects of the Z notation were chosen to study in. The first one is a generic constant, which will be described in the following sub-section.

3.1 Generic Constant Definitions

Our first support is to aid Z2SAL to translate generic constant definitions. The following sub-sections describe reasons why this aspect of Z was taken, introduce a generic constant briefly and discuss results of several examples.

Introduction Based on our experiments with Z2SAL, especially with Z specifications that have generic constructs, Z2SAL could not translate these specifications, error files were generated instead. Our finding is that Z2SAL cannot recognize a generic constant though it has been declared in a generic constant

definition; Z2SAL treats a generic constant as a new identifier. Z2SAL has not encountered any generic construct on Z specifications before, so this part of Z has not been implemented yet.

A generic constant, which is one part of generic constructs, was taken as one of our aim to extend the ability of Z2SAL. It is because the current Z2SAL does not support a translation of either a generic constant or a generic schema definition. Although Z2SAL can implement them some time during our research on this redefinition of a generic constant.

A generic constant is used to introduce a new constant which uses generic parameters [14]. By using a generic parameter, different types of a parameter can be specified. They are specified by using different literals such as `X`, `Y`, `Z` and others. A generic constant has a global scope in a Z specification, whereas a generic parameter has a local scope in the particular generic constant definition.

An example of a generic constant definition is formulated as below:

$$[X]$$
$$monoSequence : \mathbb{P}(\operatorname{seq} X)$$
$$monoSequence = \{s : \operatorname{seq} X \mid \#(\operatorname{ran} s) \leq 1\}$$

The above definition has `monoSequence` as the generic constant and one generic parameter `X`. This generic constant definition results a set of a sequence `s` which just has at the most one element.

Since a generic constant is specified in terms of generic parameters, this constant is commonly used in formulating mathematical tool-kit operators [14], in which these operators do not depend on the particular type of its elements in its construction [15]. Another usage of a generic constant is to specify a general notion which is used frequently in a system.

In a case there is no generic constant, several equivalent functions should be formulated because each function is dedicated to one set of types of parameters; it is such a useless and time-wasting work. Thus, a generic constant is quite beneficial to a Z specification.

Based on our review on the SAL literature itself, a generic form cannot be found either. Thus, it seems that Z2SAL does not support a generic constant definition in order to be consistent with the SAL language.

A Generic Constant Redefinition System Our approach to support Z2SAL in translating generic constant definitions is to implement a tool which will redefine a generic constant definition to an equivalent axiomatic definition based on usages of this generic constant. This approach originated of a similar behaviour between a generic constant and an axiomatic definition; they declare a global variable inside a Z specification. This redefinition is called an actualization process, in which a generic type of a parameter will be actualised to its actual type of a parameter.

Plagge and Leuschel in [7] also proposed the same method as our method for translating a generic definition defined in a Z specification. As discussed on their paper, generic constant definitions had not been added to a Z specification.

Our system specifies different types of generic constants. These types can be identified based on the generic constant declarations, as given below:

- a function; the outermost operator is one of infix generic functions. A complete set of these functions is "$\nrightarrow$", "$\rightarrow$", "$\rightarrowtail$", "$\rightarrow$", "$\twoheadrightarrow$", "$\twoheadrightarrow$" and "$\rightarrowtail$". These functions are collected in one token, namely `INGEN`. As a function, it will have at least one input parameter and one output parameter. This type can be generic.
- a relation; a declaration uses a tag "$\leftrightarrow$" in its outermost operator. This tag has a string `REL` as its token. As a relation, there is no output parameter type. In other word, the output is the relation itself; a pair of types.
- a constant; a constant means it does not require any input. Thus, a declaration of this generic constant only gives us generic output parameters. This declaration denotes none of the above tags in its outermost declaration.

The above three types of generic constants are parts of a variable declaration grammar in the Z language. This grammar, which refers to [15], was specified in our parser as below:

```
expr1:  expr1.word REL decor expr1.word
      | expr1.word INGEN decor expr1.word
      | expr2.chain
      | expr2
      ;
```

The first production rule indicates a relation, whereas the second one is a function. The third production rule contains `CROSS` obtained from `expr2.chain`. Thus, this production rule can either be a function or a relation depending on which of those first two production rules is fired previously. The last one is a constant; both function and relation production rules are not matched.

Inevitably, a constant actualization is not always straightforward, especially a constant implicit type. In this case, a solution is to infer an actual type from its surrounding.

Our redefinition system is intended as a pre-processing tool which can aid Z2SAL. An input Z specification, which consists of generic constant definitions and usages, will be pre-processed by this tool in order to redefine its generic constant definitions.

This tool was implemented in Java programs. It has a simple GUI to interact with users and has also two preliminary processes: the scanner and the parser generation. These two generators were implemented by using the JFlex scanner generator [18] and the BYACC/J parser generator [17] respectively.

The current version of our system implemented several Z tokens which refer to [15, 16] and several production rules of the Z grammar which refer to [15]. It was experimented also with simple variable types.

The next sub-section discusses an example of the redefinition process. This Z specification was taken from [19], namely **the function swap**.

An Example of the Redefinition Process This specification has one given type, `NAME`. There are two generic constant definitions for swapping process specified on this specification. These functions, which each have two parameters, swap an order of its parameters. Thus, after a swap, an element in the second position will be shifted such that this element is in the first position. The first position is vice versa.

The first definition, as shown below, has two different generic parameters: `X` and `Y`. A generic constant name is `swap2`.

$$[X, Y]$$
$$swap2 : X \times Y \rightarrow Y \times X$$
$$\forall x : X;\ y : Y \bullet swap2(x, y) = (y, x)$$

The second has one generic parameter, `X`. Its name is `swap1` and it is shown as below:

$$[X]$$
$$swap1 : X \times X \rightarrow X \times X$$
$$\forall x, y : X \bullet swap1(x, y) = (y, x)$$

A state schema, namely `State`, has only one state variable, `name`, which is an instance of the defined given type. There is no predicate specified on this schema.

The initialization schema, `Init`, refers to the post state of the state schema. This schema does not declare its own variable and predicate.

There is one operational schema, `Swap`, which calls these generic constants. This schema does not change a state of this system indicated by Ξ `State`. This schema was specified as below:

Swap
$$a? : NAME;\ a!, b! : NAME;\ c? : \mathbb{N};\ c! : \mathbb{N};\ \Xi State$$
$$(b!, a!) = swap1[NAME, NAME](name, a?)$$
$$(c!, a!) = swap2(name, c?)$$

The first usage of generic constant definitions uses explicit type of parameters in addition to parameters required by this function. Our system generated two axiomatic definitions for these usages as shown below:

$$swap1 : NAME \times NAME \rightarrow NAME \times NAME$$
$$\forall x, y : NAME \bullet swap1(x, y) = (y, x)$$

$$swap2 : NAME \times \mathbb{N} \rightarrow \mathbb{N} \times NAME$$

$$\forall\, x : NAME;\ y : \mathbb{N} \bullet swap2(x, y) = (y, x)$$

Furthermore, the explicit type has been deleted from the first usage since Z2SAL does not support this type of parameter. Thus, the first usage has also been modified by our system as below:

```
(b!,a!) = swap1(name,a?)
```

Result and Discussion The generated specification of the above example could be translated by Z2SAL. A SAL file, which was generated by Z2SAL, could also be verified by the SAL model checker, but it failed to be simulated by the SAL model checker. This simulator generated an unsupported error of a failure to convert function application.

Furthermore, if a theorem was added to the generated SAL file, this SAL file could not be verified either by the SAL model checker. Thus, it is an issue of the redefinition system.

The current Z2SAL translates a function, relation and constant in the base module, in which Z2SAL defines `State` as a default name for a module and puts variable declarations inside a definition clause. As a result, an error of incompatible type in the equality operator or a fail to convert function application produced by the SAL model checker or simulator, as given earlier, was sometimes experienced during our experiments with user-defined functions.

Based on our review on the SAL literature, a user defined function, relation and constant are always declared outside the module and are put inside a context clause, specifically in a constant declaration, instead. Thus, the same method as SAL's method was proposed. This method can be applied to a user defined function and constant, but it is not applicable to a user defined relation since a relation does not have a type for its output parameter.

A constant declaration has a syntax as below [9]:

$$\mathit{ConstantDecl} := \mathit{Identifier}[(\mathit{VarDecls})] : \mathit{Type}[= \mathit{Expr}]$$

Thus, the generated SAL file was modified to adapt a constant declaration formulated by SAL. Both the above function definitions were formulated manually on the generated SAL file. They are shown below:

```
swap1(q__1 : NAME, q__2 : NAME): B__NAME__X__B__NAME = (q__2,q__1);

swap2(q__3 : NAME, q__4 : NAT): B__NAT__X__B__NAME = (q__4,q__3);
```

Original declarations generated by Z2SAL on these functions were deleted.

A few theorems were added to this specification as shown below:

```
th1: theorem State |- G(FORALL (i: NAME,j: NAT): swap2(i,j) = (j,i));
```

```
th2: theorem State |- G(FORALL (i,j: NAME):
i = j => swap1(i,j) = swap1(j,i));

th3: theorem State |- G(FORALL (i,j: NAME): swap1(i,j) = swap1(j,i));
```

The first two theorems are valid; the swapping system could satisfy both properties. The last theorem is invalid since the swap function will not give us the same result for different parameters.

There is another issue relating to an abbreviation definition and a lambda expression during redefinition of a generic constant definition. Both these issues will be discussed in the next sub-sections.

Generic Abbreviation Definitions Z2SAL supports an abbreviation definition, but not the generic one. Declaring a global constant by using an abbreviation definition is common in writing Z specifications. Thus, a generic abbreviation definition was taken also into our consideration.

In a case of a generic abbreviation, it is not enough just to work with an actualization of a generic type. Other issue here is a set comprehension definition because the generic abbreviation definition is usually defined by using a set comprehension definition. However, Z2SAL does not support an abbreviation definition consisting of a set comprehension either.

For example, look at a generic abbreviation definition as below [14]:

$$monoSequence[X] == \{s : \operatorname{seq} X \mid \#(\operatorname{ran} s) \leq 1\}$$

Based on the Z literature, a generic abbreviation definition can be rewritten to a generic constant definition. Both these definitions declare global constants in the related Z specification, in this case the type of the generic constant is a constant.

The expression in the right hand side of a tag "==" uses a set comprehension definition, which denotes that `monoSequence` is a set of a sequence of `X`. The body of this generic definition is obtained from the expression after the tag "==".

Thus, a generic abbreviation definition is firstly rewritten to a generic constant definition. This rewriting is performed manually and automatically in order to prove its correctness. This equivalent definition was given in the Section 3.1. Afterwards, this generic constant definition is redefined to an axiomatic definition.

Lambda Expressions Another kind of generic forms is an expression λ, which is used to define a function without specifying a name [14]. Z2SAL does not support this expression which is common in generic constant definitions or in other definitions in a Z specification generally. Our approach is to rewrite a lambda expression automatically and manually to an equivalent expression without any lambda expression. Then, it is redefined to an axiomatic definition.

For example, a generic constant definition as formulated below consists of the lambda expression [14]:

$$[X]$$

$$commonSubseq : ((\operatorname{seq} X) \times (\operatorname{seq} X)) \rightarrow \mathbb{P}(\operatorname{seq} X)$$

$$commonSubseq = (\lambda\, s, t : \operatorname{seq} X \bullet allSubseqs \cap allSubseqt)$$

The lambda expression in the above definition can be rewritten to an equivalent definition as below:

$$commonSubseq = \{s, t : \operatorname{seq} X \bullet ((s, t), allSubseq(s) \cap allSubseq(t))\}$$

or another equivalent one as given below:

$$\forall\, s, t : \operatorname{seq} X \bullet commonSubseq(s, t) = allSubseqs \cap allSubseqt$$

A lambda expression definition, (λ `S` $\bullet$ `E`), represents a function and has arguments which are taken from `S`. An output of this expression is the value of `E` [15]. As given by the first equivalent definition above, the lambda expression is equivalent to a set comprehension, $\{S \bullet (T, E)\}$, in which `T` is a characteristic tuple of `S`. In a set comprehension, a characteristic tuple is obtained from its declaration. Thus, `(s,t)` was a characteristic tuple of the above set comprehension.

During our experiment, a set comprehension definition, which has a declaration of many parameters of the same type, cannot be translated by Z2SAL. Based on the SAL literature, only one parameter can be declared in one definition of a set comprehension. The SAL syntax [9] for a set expression is given as below:

$SetExpression := SetListExpression \mid SetPredExpression$
$SetListExpression := \{\{Expression\}_{,}^{+}\}$
$SetPredExpression := \{Identifier : Type = Expression\}$

Thus, our approach is to rewrite the first equivalent lambda expression to the second equivalent one.

Several results collected from our experiments are given and discussed on the next sub-section.

Summaries of Experiments on the Redefinition System The number of experiments on several Z specifications are presented on Table 1. These experiments run on a laptop with a 1.30GHz Genuine Intel(R) CPU U7300 and 2.00 GB RAM.

The second column of Table 1 indicates whether the SAL file, generated by Z2SAL from the Z specification resulted by the redefinition system, required a modification to be verified by the SAL model checker or simulated by the SAL simulator. This modification was accomplished manually on the SAL file. It involved rewriting a user defined function and placing this function on which SAL put its function. A couple of examples of this rewriting was given earlier in this section. This modification also involved rewriting other parts of a SAL file.

Table 1. Several Experiments with the Redefinition System

Z Specification	**Details**	**Verification time in secs**	
(*.tex)		**#Theorem = 0**	**#Theorem > 0**
`bbook`	Modified SAL function		0.842
`bbook_map`	Modified SAL function	0.016	0.25
`bbook_uni`	Modified SAL function and other parts of SAL file	0.031	0.406
`bbook_map_uni`	Modified SAL function and other parts of SAL file		0.359
`fDomRan`	Modified SAL function	0.015	
`fEmpty`	OK		0.093
`fEmptyImpl`	OK		0.109
`fFirst`	Modified SAL function	0.015	0.187
`fHead`	Modified SAL function	0.031	
`fHeadFunc`	Modified SAL function and could not be simulated: The set of initial states is empty	0.031	
`fMaxComSubSeq`	Modified other parts of SAL file and could not be simula ted: An out of memory error	0.047	
`fMaxComSubSeq_1`	Modified other parts of SAL file and could not be simula ted: An out of memory error	0.032	
`fMaxComSubSeq_orig`	Modified other parts of SAL file and could not be simula ted: An out of memory error	0.032	
`fMonoSeq`	OK. Long simulation	0.047	
`fMonoSeq_1`	OK. Long simulation	0.031	
`fSwap`	Modified SAL function	0.016	0.141
`fUniqSeq`	Ok. Could not be simulated: An out of memory error	0.062	
`fUniq1Seq`	Ok. Could not be simulated: An out of memory error	0.031	
`fUniq2Seq`	Ok. Could not be simulated: An out of memory error	0.015	
`tn`	Modified other parts of SAL file and could not be simula ted: An out of memory error	0.03	
`tnImpl`	Modified other parts of SAL file and could not be simula ted: An out of memory error	0.0	
`fFileStorage`	Could not be translated by Z2SAL	N/A	
`fSet`	Modified SAL function and other parts of SAL file	0.0	

Such a modification could be a bug in the translation of associated Z specification by Z2SAL. It could also be a mismatch between the Z language and the SAL language.

The third column shows verification times of each SAL file. A SAL file which has one verification time means that it was not verified for another case of the number of theorems. A SAL file which has two verification times is a SAL file which at first could be verified by the SAL model checker, but later it could not be verified if at least one theorem was added to this SAL file. Such a SAL file usually could not be simulated either by the SAL simulator even there is no theorem.

Based on our experiments as shown in Table 1, there were three SAL files which could not be verified by the SAL model checker, though their functions have been modified to adapt a method used by SAL as discussed earlier. These files are **output_bbook_uni** as the SAL file generated from the output of **bbook_uni.tex**, **output_bbook_map_uni** as the SAL file generated from the output of **bbook_map_uni.tex** and **output_fSet** as the SAL file generated from the output of **fSet.tex**. The error related to incompatible types in the equality operator. This error could be seen as a mismatch as mentioned above.

The SAL model checker identified that the type of `birthday` is not compatible with the type of the first argument of a function `uniSet` in the first and second SAL files. The function `uniSet` which is a generic constant definition was specified as below:

$$[X]$$

$$uniSet : (\mathbb{P}\, X) \times (\mathbb{P}\, X) \rightarrow (\mathbb{P}\, X)$$

$$\forall\, S, T : (\mathbb{P}\, X) \bullet uniSet(S, T) = \{x : X \mid x \in S \lor x \in T\}$$

This function combines two sets of elements which have the same types. As can be seen, this function requires two parameter inputs. Both of them have the same types which are the same as the type of the output.

A usage of the above generic constant was specified as follows:

```
birthday' = uniSet(birthday, { name? ↦ date? })
```

As can be seen from the above generic constant definition, the type for the first parameter is a set of `X` and it is said as an expected type. On the other hand, `birthday` is the first parameter passed to `uniSet`; the type of `birthday` will be the actual type for this parameter. A declaration of the function `birthday` is as below:

```
birthday: NAME ⇸ DATE
```

`birthday` is a state variable, which is a partial function from `NAME` to `DATE`.

Our system generated the axiomatic definition `uniSet` as below:

$$
\begin{array}{|l}
uniSet : (\mathbb{P}(NAME \times DATE)) \times (\mathbb{P}(NAME \times DATE)) \\
\rightarrow (\mathbb{P}(NAME \times DATE)) \\
\hline
\forall S, T : (\mathbb{P}(NAME \times DATE)) \bullet \\
uniSet(S, T) = \{x : (NAME \times DATE) \mid x \in S \lor x \in T\}
\end{array}
$$

As can been from the above definition, the type of `birthday` has been modified to its equivalent type. It is done so to ease the unification of the expected type `X` and the actual type `NAME ↠ DATE`.

Based on the Z literature, a function type can be rewritten to a relation type. Several constraints should also be added to present behaviour of a related function. Furthermore, a relation is equivalent to a set of a pair of types.

$$X \leftrightarrow Y \equiv \mathbb{P}(X \times Y)$$

Thus, it seems that SAL failed to recognize that `birthday` had an equivalent type to the first argument of the user-defined function `uniSet`. This error said that there was incompatible type between the output of `uniSet`, which was `Set__C__B__NAME__X__B__DATE__I` and the right hand side of the equality operator, which was `[NAME__X__DATE -> bool]`. Afterwards, a sequence of modifications was performed to the associated SAL file lines.

The last error produced by the SAL model checker is as below:

```
Error: [Context: output_bbook_uni_mod, line(62), column(29)]:
Type mismatch in the function application.
Expected type:
[set{output_bbook_uni_mod!NAME__X__DATE}!Set,
set{output_bbook_uni_mod!NAME__X__DATE}!Set]
Actual type:
[output_bbook_uni_mod!Set__C__NAME__X__B__DATE__I,
set{output_bbook_uni_mod!NAME__X__DATE}!Set]
```

The related SAL lines are as follows:

```
61 NOT set {NAME;} ! contains?(known, name?) AND
62 birthday' = uniSet((birthday, set {NAME__X__DATE;} !
63 singleton((name?, date?)))) AND
64 invariant__'
```

In line 62, the type of `uniSet` after modification is a pair of `set {NAME__X__DATE;} ! Set` and `set {NAME__X__DATE;} ! Set`. This type was not compatible with the actual type passed to `uniSet` which was a pair of `Set__C__NAME__X__B__DATE__I` and `set {NAME__X__DATE;} ! Set`. The type `Set__C__NAME__X__B__DATE__I` is an alias for `[NAME -> B__DATE]`, specified by Z2SAL.

Although a function is special type of a relation and a relation is a set of a pair of types in the Z language, it seems that SAL did not think both types of the first argument of `uniSet` were the same. Thus, this incompatible type

was solved manually. This is because our tool has not been able to perform this modification automatically.

Our last modification kept the same alias for `birthday`, but this time the alias represents a relation, not a function any more. It is shown below:

```
Set__C__NAME__X__B__DATE__I : TYPE = set {NAME__X__DATE;} ! Set;
```

.

This change affects the usage of `birthday`; it cannot any longer be used as a function.

```
function {NAME, B__DATE; DATE__BB} ! partial?(birthday) AND
```

As one effect, the above line was deleted from the old SAL file.

```
known = relation {NAME, DATE;} ! domain(birthday) AND
```

Another effect is the above line was replaced by a line below:

```
known = function {NAME, B__DATE; DATE__BB} ! domain(birthday) AND
```

As well as a line as below:

```
date_' = birthday(name?) AND
```

was replaced by a line below:

```
set {NAME__X__DATE;} ! contains? (birthday, (name?, date_')) AND
```

Finally, the modified SAL file could be verified by the SAL model checker and simulated by the SAL simulator.

The same function was also a source of the error in the third SAL file, but this time its first actual parameter is `used`. A usage of this function in the associated Z specification is as follows:

```
used' = uniSet(used, {n})
```

A state variable `used` is a set of $\mathbb{N}_1$ and a variable `n` is an instance of $\mathbb{N}_1$.

An axiomatic definition generated for the above usage is as below:

$$\begin{array}{|l} \mathit{uniSet} : (\mathbb{P}\,\mathbb{N}_1) \times (\mathbb{P}\,\mathbb{N}_1) \to (\mathbb{P}\,\mathbb{N}_1) \\ \hline \forall\, S, T : \mathbb{P}\,\mathbb{N}_1 \bullet \mathit{uniSet}(S, T) = \{x : \mathbb{N}_1 \mid x \in S \lor x \in T\} \end{array}$$

After a similar modification as performed in both SAL files above, the modified SAL file could be verified and simulated by the SAL tool.

Another finding is that a few SAL files, which could be verified by the SAL model checker, could not be simulated by the SAL simulator due to an out

of memory error as can be seen on the Table 1. These SAL files usually have sequences or a set inside other sets. Currently, this error has not been solved.

Inevitably, there is one of our experiments which could not be translated by Z2SAL as shown by "`N/A`" in the Table 1. It is because this Z specification contains a function which its range is also a function. Z2SAL does not support such a type. A quick solution is to rewrite such a function. However, another error relating to a tag ". ." was experienced which seems it is another bug in Z2SAL.

Another aspect of the Z notation in our study is the schema calculus. This aspect is described in the next sub-section.

3.2 Schema Calculus Definitions

This sub-section discusses other type of our support which is a schema calculus expansion.

Introduction Z2SAL supports a translation of several schema calculus such as a schema inclusion, the operator Δ and the operator Ξ, but they must be specified either vertically or horizontally in a schema. However, if a new schema is constructed from earlier schemas, Z2SAL does not support this schema construction. Thus, it seems that Z2SAL does not support schema calculus definitions.

The constructed schema is specified by using $\widehat{=}$. It is the same as the supported schema calculus, but the constructed schema does not use $[$ and $]$ to surround its declaration of variables and predicates.

The constructed schema is used commonly to define a more complex, modular and a huge specification of a system. Schemas that have been specified can be reused to specify a new schema. It is because every schema has its distinctive operation in a specification, called 'schema separation' [2].

A Schema Calculus Expansion System Our approach is to construct a new schema by expanding other schemas, in which they are connected by schema operators. This system was included in the support tool for model checking Z specifications, as well as the redefinition system.

Since every schema operator has its own definition, a schema operator affects how the expansion is done. The expansion means that all unique variables of involved schemas are listed in the new schema. It also means that predicates which are read from the involved schemas are added. These predicates are combined using schema operators.

There is a prerequisite for operating two schemas; the same or common variables should have the same type. Furthermore, in a case of the negation operator, normalisation is also required.

Normalisation is to define explicitly the constraint given by the declaration part of the related schema. This constraint is specified in the predicate part. Normalisation should be performed just before the negation. This process is applied

also to other schema operators for the sake of easiness. Several normalisation rules were specified in our system as below:

- Every $\mathbb{N}$ or $\mathbb{N}_1$ in a declaration part is rewritten to a type of $\mathbb{Z}$.
- Every seq or seq_1 is changed to $\mathbb{P}(\mathbb{Z} \times newVal)$, *newVal* is a type which comes after seq or seq_1. The previous rule is applied also to *newVal*.
- Every function is changed to a pair of its left hand side type and its right hand side one. Both the above rules are also applied to the type in the left and in the right.

In general, after each schema is expanded, variables and predicates will be collapsed to a reference of a state schema. This collapse benefits the new schema to get a more compact schema and to avoid re-declarations of state variables.

Our system can expand several schema operators such as conjunction $\wedge$, disjunction $\vee$, negation $\neg$, implication $\Rightarrow$, bi-implication $\Leftrightarrow$, hiding $\setminus$, renaming $/$, composition $\fatsemi$, universal quantifier $\forall$ and existential quantifier $\exists$. Our system can also perform a simple simplification over a predicate part. However, a schema calculus definition cannot be a complex definition.

The next sub-section describes an example from our experiments with this system.

An Experiment with the Schema Calculus Expansion System An example, **expandingschema_3.tex**, will be presented in this sub-section. This example was taken from [2], but has been modified in some places for our experiments.

This example represents a library system specification. The specification has four state variables:

- `stock` is a partial function from `COPY` to `BOOK`. It gives us information about what copies a book has.
- `issued` is a relation between a copy of a book and a reader. It gives us information about which copy of a book each reader has.
- `shelved` is finite set of `COPY`.
- `readers` is a finite set of `READERS`.

This specification has also three given types: `COPY, BOOK, READER`.

There is one schema calculus definition specified in this specification, which uses the Z schema composition operator, $\fatsemi$. It is shown as below:

$$\texttt{Donate} \mathrel{\widehat=} \texttt{EnterNewCopy} \fatsemi \texttt{RegisterReader}.$$

This operator will combine the second schema with the first schema, in which the result of the first schema is an input for operating the second schema.

The schema composition consists of the number of operations taken from other schema operators. Renaming is the first operation to take place: rename the same state variables so as the primed ones in the first schema and non-primed ones in the second schema have the same name of variables. Afterwards, these

renamed schemas are combined using a conjunction operator. The next process is to hide the common renamed variables in a declaration part of the new schema and add an existential quantification which binds these hidden variables in a predicate part of the new schema.

The new schema, `Donate`, was constructed by our system as given below:

Donate

$\Delta Library;\ b? : BOOK;\ r? : READER;\ rep! : Report$

$\exists\, c : COPY \mid c \notin \mathrm{dom}\, stock \bullet (stock' = stock \oplus \{c \mapsto b?\} \land$
$shelved' = shelved \cup \{c\}) \land r? \notin readers \Rightarrow$
$(readers' = readers \cup \{r?\} \land rep! = Ok) \land$
$r? \in readers \Rightarrow (readers' = readers \land rep! = ReaderAlreadyRegistered)$
$\land\ issued' = issued$

A theorem as given below was added to the generated SAL:

```
th1: theorem State |- G(shelved = set{COPY;}!empty);
```

It says that `shelved` is always empty, which is invalid since a `c` of type `COPY` can be added to `shelved` by performing `EnterNewCopy` or `Donate`. Indeed, the SAL model checker reported a counter-example on the verification of this SAL file.

Result and Discussion This specification requires a simplification which is applied to the final output, otherwise there will be re-declared state variables. Our system could perform a simple simplification to collapse all state variables and predicates to a reference of the state schema.

As mentioned previously, the first process of a schema composition is renaming which is to rename several state variables to the common names. In this system, the common name is specified to be the same as the name of the state variable, but 0 will be added at the end of this variable. This simplification is achieved by substituting all renamed common variables for their appropriate values obtained from related predicates.

The above example could be translated by Z2SAL. It could also be verified and simulated by the SAL tool.

Below sub-section summarizes results obtained from our experiments with this system.

Summaries of Experiments with the Expansion System Re-declaration of state variables is also an issue of implementations of renaming and hiding operations. Since a simplification is hard to apply on both operations, these operations cannot be further implemented at the moment.

The current Z2SAL assumes that the first schema definition in a Z specification is a state schema and the second one is an initialization schema. Z2SAL defines also one base module in each SAL specification and accepts only one state schema in each Z specification input, though both SAL and Z allow many modules and state schemas respectively in one specification.

A SAL module specifies a transition system of a finite-state automaton. A Z schema represents a state of a system and a collection of these schemas models behaviour of the system. A state schema is a combination of state variables and predicates of a system.

A restriction on the number of state schemas in a Z specification is an issue of performing a negation in a schema expansion. Variables and negated predicates in the constructed schema cannot be collapsed into a state schema inclusion; a problem of re-declared variables. The only way to solve this problem is to define at least two state schemas: the first state schema just defines state variables; the second one defines an inclusion to the first schema as well as defines state predicates. However, Z2SAL does not support many state schemas either as discussed earlier.

This restriction affects also how a renaming and a hiding are applied to. Both schema operators cannot be applied to the initialization schema and operational schemas due to the above same problem and instead to the state schema. Furthermore, Z2SAL also enforces us to define the same name for both the constructed schema and the state schema. Thus, the application of these two operators will modify the whole specification.

Another issue in a schema expansion is the order or the binding of schema operators, especially when brackets are not added in a definition of schema calculus. Fortunately, operators bindings and associativities can be defined by using built-in options of the BYACC/J parser generator [17]: `left`, `right` and `nonassoc`, which mean left, right and no grouping respectively. Afterwards, several actions can be added in associated grammars to define information about these orders. The order of operators tells us the precedence among them, which is getting higher position, the lower the precedence. Several of these orders are given in the [15].

Table 2 shows us several results from our experiments. As can be seen from Table 2, a simplification has been performed on an output of specification `expandingschema_1`. It is indicated by two verification times in associated columns. Outputs of `expandingschema_3`, `expandingschema_4` and `expandingschema_5` have two verification times in one column. The first time is a verification time with no theorem and the second one is a time with one theorem. There are two specifications that have many schema calculus definitions: `expandingschema_4` and `expandingschema_8`. "`N/A`s" in several rows mean that the related specification could not be translated by Z2SAL. All of these specifications contain re-declarations of state variables. It is because these variables could not be collapsed by our system to references of a state schema.

4 Conclusion and Future Work

It has been shown in both tables that all our running examples could be redefined or expanded by our system. Several of them could also be translated by Z2SAL, verified by the SAL model checker, or simulated by the SAL simulator.

Table 2. Several Experiments with the Expansion System

Z Specification	**Details**	**Verification time in secs**	
(.tex)		**Non-simplified**	**Simplified**
expandingschema_1	"$\vee$"	0.063	0.031
expandingschema_2	"$\wedge$"	0.062	
expandingschema_3	"$\circ_9$"	0.03	
		0.733	
expandingschema_4	"$\wedge$"	0.016	
	"$\vee$, $\vee$"		
	"$\vee$"		
		2.044	
expandingschema_5	"$\wedge$, $\neg$, $\wedge$"	0.031	
		1.654	
expandingschema_6	"$\wedge$, $[$, $]$"	0.031	
		0.686	
expandingschema_7	"$\neg$, $\wedge$, $[$, $]$"	N/A	
expandingschema_8	"$\wedge$, $[$, $]$"	N/A	
	"$\neg$, $\wedge$, $[$, $]$"		
	"$\vee$"		
expandingsch2_4	"$\neg$"	N/A	
expandingsch3_1	"$\Rightarrow$"	0.015	
expandingsch3_2	"$\wedge$, $\Rightarrow$"	0.032	
expandingsch3_4	"$\Rightarrow$, $\wedge$"	0.016	
expandingsch4_1	"$\Leftrightarrow$"	0.015	
expandingsch4_2	"$\wedge$, $\Leftrightarrow$"	0.031	
expandingsch5_1	"$[$, /, $]$"	N/A	
expandingsch5_2	"$[$, /, /, $]$"	N/A	
expandingsch6_1	"$\setminus$"	N/A	
expandingsch6_2	"$\setminus$"	N/A	
expandingsch7_1	"$\circ_9$"	0.031	
expandingsch8_1	"$\forall$"	N/A	
expandingsch8_2	"$\forall$"	N/A	
expandingsch8_3	"$\forall$, $\wedge$"	N/A	
expandingsch8_6	"$\exists$"	N/A	

Redefinition and schema expansion, which pre-process a Z specification, can benefit the scope of translation of Z2SAL. It is because a Z specification can consist of a generic constant, or a schema calculus definition. This fact can support Z2SAL to translate a variety of Z specifications, which at the end can also support model checking Z specifications. However, our method on implementing this system, especially the schema calculus expansion, seems that it is not feasible to a huge and complex specification. Expanded schemas can make the specification be more complex.

Regarding an out of memory error which is often encountered during the simulation, this issue can be put as a future work. One idea here is to apply abstraction to the related specification.

Re-declaring state or global variables can be approached by implementing a better simplification in predicates. It can also include upgrading Z2SAL to a version that accepts many state schemas and references to them.

Our approach to a SAL translation of a user defined function or constant can be automated to get big advantage of it. There are two options for this automation: implementing it as an extension to this system or adding it as an extension to Z2SAL system. It seems that the second option is an easier method to implement.

Another future work is to be able to run a more complex, to some extent, Z specification input. This fact requires also an extension to this system.

Acknowledgment

The author would like to thank John Derrick, Siobhán North and Anthony J.H. Simons for giving the author a chance to work with their Z2SAL and discussions on this tool. A lot of thanks are dedicated to ISIHEMORA The Republic of Indonesia for its financial support on this study.

References

1. Jackson, D.: Abstract model checking of infinite specifications. FME'94: Industrial Benefit of Formal Methods. Springer, 519–531 (1994)
2. Potter, B., Till, D., and Sinclair, J.: An introduction to formal specification and Z. Prentice Hall PTR (1996)
3. West, M.M.: Issues in Validation and Executability of Formal Specifications in the Z Notation. Thesis of University of Leeds (2002)
4. Woodcock, J. and Davies, J.: Using Z: specification, refinement, and proof. Prentice-Hall, Inc. (1996)
5. Derrick, J., North, S., and Simons, A.J.H.: Z2SAL: a translation-based model checker for Z. Formal aspects of computing. Springer, 23 1, 43–71 (2011)
6. Malik, P., Groves, L. and Lenihan, C.: Translating z to alloy. ASM, Alloy, b and Z. Springer, 377–390 (2010)
7. Plagge, D. and Leuschel, M.: Validating Z specifications using the ProB animator and model checker. Integrated Formal Methods. Springer, 480–500 (2007)
8. Bolton, C.: Using the alloy analyzer to verify data refinement in Z. Electronic Notes in Theoretical Computer Science. Elsevier, 137 2, 23–44 (2005)
9. De Moura, L., Owre, S. and Shankar, N.: The SAL language manual. Computer Science Laboratory, SRI International, Menlo Park, CA, Tech. Rep. CSL-01-01 (2003)
10. Smith, G. and Wildman, L.: Model checking Z specifications using SAL. ZB 2005: Formal Specification and Development in Z and B. Springer, 85–103 (2005)
11. Derrick, J., North, S., and Simons, A.J.H.: Issues in implementing a model checker for Z. Formal Methods and Software Engineering. Springer, 678–696 (2006)

12. Simons, AJH: The Z2SAL User Guide. Accessed from http://staffwww.dcs.shef.ac.uk/people/A.Simons/z2sal/userguide.html (2012)
13. Bensalem, S., Lakhnech, Y. and Owre, S.: Computing abstractions of infinite state systems compositionally and automatically. Computer Aided Verification. Springer, 319–331 (1998)
14. Barden, R., Stepney, S. and Cooper, D.: Z in Practice. Prentice-Hall, Inc. (1995)
15. Spivey, J.M.: The Z notation. Prentice Hall New York (1989)
16. King, Paul: Printing Z and Object-Z LaTeXdocuments. Department of Computer Science, University of Queensland. (1990)
17. Hurka, T.: BYACC/J. Accessed from http://byaccj.sourceforge.net (2008)
18. Klein, G.: JFlex - The Fast Scanner for Java. Accessed from http://www.jflex.de/index.html (2015)
19. Rann, D. and Turner, J. and Whitworth, J.: Z: a Beginner's Guide. CRC Press (1994)