Model Checking RAISE applicative specifications

Juan I. Perna
Software Engineering Group
Universidad Nacional de San Luis
San Luis – Argentina
jiperna@unsl.edu.ar

and

Chris George
International Institute for Software Technology
United Nations University
Macao, SAR – China
cwg@iist.unu.edu

Abstract

Ensuring the correctness of a given software component has become a crucial aspect in Software Engineering and the Model Checking technique provides a fully automated way to achieve this goal. In particular, the usage of Model Checking in formal languages has been reinforced in the last decades given the fact that specifications provide an abstraction of the problem under study, supplying a model of the system of tractable size given the *state explosion* problem faced by the Model Checking technique.

In this paper we focus on the main issues for adding Model Checking functionalities to the RAISE specification language and present the semantic foundations of our current approach for doing so. An outline of the main problems faced in the process and of the solutions to solve them are also presented.

Keywords: Model Checking, RAISE, formal methods, verification techniques.

1 INTRODUCTION

The utilisation of Model Checking techniques for software components verification has been subject of significant research and study during the last decade [10, 9]. This increasing popularity of Model Checking is due to the high level of automation achieved (compared to other verification techniques such as testing or validation by proof) and to the ability of producing counterexamples when a given property is not satisfied.

Regardless of its context of application, the model checking technique is based on the exploration of all possible reachable states by the system under study and the verification of the satisfaction of properties (expressed in a sub logic of the CTL family [15, 9]) on those states. Due to this exhaustive exploration of the state space of the problem, Model Checking suffers from the *state explosion* problem (i.e. the size of the computation increases exponentially with respect to the size of the original problem). As the state explosion problem is a major limitation for the applicability of model checking in *real* problems (due to their complexity and size), several techniques have been developed to cope with this issue. In particular, *symbolic model checking* [19, 10] and *abstraction* [12, 8] are the most used ones.

In the context of formal or rigorous methods for software development, several attempts have been made to incorporate Model Checking techniques given the advantage that software specifications are, essentially, abstractions of the desired system. In particular, there have been several approaches to the incorporation of model checking techniques in order to verify whether a given property is preserved throughout the whole development process or at a certain abstraction stage. Some well known examples of the incorporation of model checking functionalities into formal languages such as Z [1, 3] or process algebras [11] such as CSP [14, 4] can be analysed from [22, 18, 17].

Regarding RAISE [13], no support for model checking is currently provided. In particular, RAISE provides several tools regarding verification, such as **code generators** to several languages in order to *run* the specification's code [23, 2]; **test cases** [5] (including test coverage analysis and mutation testing) and a **translator to PVS** [6] so important properties or invariants from the specification can be proved correct. However relatively easy to do, testing is necessarily incomplete and it only allows the user to gain certain confidence about the possible satisfaction of the desired properties. Proofs, on the other hand, provide certainty and completeness but are very hard and time consuming to do.

This work presents an ongoing project to incorporate the automated functionalities of model checking to the RAISE language by means of translating specifications into the Symbolic Analysis Laboratory (SAL[7]) model checker. In particular, the aim of the project is to to provide a useful tool that allows one to model check RSL specifications.

The rest of the paper is organised as follows: section 2 describes the main strategies used to translate RSL constructs into SAL constructs, and some of the problems involved; section 3 outlines the extensions incorporated to RAISE to support Model Checking; section 4 shows the limitations of the current work; and section 5 presents the conclusions and future extensions of the present work.

2 TRANSLATION STRATEGY

The task of a translator is to accept as much as possible of the constructs of the input language while generating output that is well-formed, semantically correct and that executes efficiently. But there is typically a choice to be made here, because the goals of completeness and execution efficiency are generally in conflict. Our primary aim is a usable model checker, so efficient execution is the primary goal. Consequently our approach is a *shallow embedding* rather than a *deep embedding* of applicative RSL into SAL.

A *deep embedding* aims to model the input language's semantics in the target language, and so will typically start with the construction within the target language of the input language's semantic domain.This approach will produce a translator that is powerful in terms of what parts of the input language can be covered (ideally all) but typically inefficient in terms of running the translated code. A *shallow embedding*, on the other hand, aims to translate constructs of the input language into corresponding constructs of the target language. It is likely to be less complete but it tends to generate more efficient target code [21].

In this section we consider the constructs from RSL that are translatable to SAL and show how we preserve the RSL semantics in the SAL output.

2.1 Type declarations

As the type system provided by SAL is quite similar to the basic RSL type system, this first version of the translator is currently using the former as a base for the translation of type declarations. There are, however, some exceptions to this rule:

- Integer type. SAL provides an `INTEGER` type that can be used to model the type integer in RSL. Due to the fact that both types are infinite by definition, it is necessary to impose a restriction over the possible values in the type. This is rather a constraint towards the finiteness of the model than a syntactic constraint imposed by SAL language but it should also be addressed by the translator. The solution adopted is to translate the RSL type **Int** as the SAL *subrange* type `[Low .. High]`, where `Low` and `High` can be modified to allow experimentation with the model and properties under verification.
- Natural type. This type is translated to `[0 .. High]`.
- Record type. SAL distinguishes between variants (i.e. `DATATYPES`) and records (identified with the construction `[#{`*`identifier`*` : Type}`$^+_,$`#]`) not only by syntactic means but also by the operations that can be performed over them. On the other hand, RAISE defines records as short variant definitions and thereby allows all the operations from the latter to be applied on the former. Based on this semantic difference the implementation with datatypes was preferred over the (more particular) record construction.
- Union type. Unions are shorthands in RAISE that allow the omission of constructors and destructors that are compulsory in the case of the variant type. This special kind of "implicit constructors" is not accepted in SAL (its syntax requires the presence of a constructor for every possible field in a variant). Due to this restriction, the union type is not translatable into SAL.

Finally, the collection types (sets, maps and lists) require a more complex translation scheme as there is no support in SAL for any of them. In general, the strategy for translating sets and maps relies on an encoding based on total functions. Having the implementation based on this approach allows the definition of the operations over sets and maps by means of intensive use of `LAMBDA` functions. In particular, the translation for sets uses the traditional implementation as a function from the domain of the set into a Boolean value.

In a similar way, maps are also defined as functions but the translation procedure has to handle several issues arising not only from the way functions are handled in SAL but also from RAISE's definition of maps:

- A map might not be defined over all possible values in its domain. In this case, a map application over a value not in the map's domain will return the value **swap**. Moreover, SAL does not provide partial function support, hence, partial constructions are not directly translatable. To solve this problem, the translator modifies the map by creating a "wrapper" over the range by means of a variant declaration of the form:

  LiftedRange == nil | val(Range: OriginalRange)

  This approach turns the map into a total function (the elements in the domain that were not included in the map, are now mapped to the special value `nil`) and allows the encoding of the map as a total function in SAL.

- Maps might be nondeterministic (for example, [x $\mapsto$ y | x,y : **Nat** • {x,y} $\subseteq$ {1,2}]). A map application with a value mapped to multiple values is equivalent to the internal choice among the possible results of the map. Even though it is possible to produce this kind of behaviour in SAL, the required implementation would make the whole model checking process very inefficient hence, they are not accepted in the translator to SAL.
- Maps can be infinite. Due to model checking finiteness requirement, possible infinite constructions can not be translated, so infinite maps are also not accepted in the translator.

2.2 Explicit function definitions

An explicit value definition which defines a function with a name that is unique in the scheme that holds it, translates directly to a SAL explicit function as shown below:

next : **Int** $\rightarrow$ **Int**
next(n) $\equiv$ n + 1

$\Longrightarrow$

```
next(n: Int_) : Int_ =
  n + 1;
```

As SAL does not support overloading, if there is a clash in the function name (i.e. the function is overloaded in the specification), an error is reported during the translation. A similar situation arises when overloading operators (which it is also a feature not supported in SAL). In this case, the most common approach would be to transform the operator into a function declaration and replace all its occurrences in the specification with normal prefix invocations to this new function. Although possible, this feature is also not implemented, because of readability issues.

Partial function encoding. Neither SAL nor any other Model Checker provide support for partial functions given the undefined behaviour they introduce in the system. In this context, it is initially unclear how to encode this kind of construct from RAISE when generating a system for model checking. In this sense, it is possible to consider all the preconditions of a given function (i.e. the conditions over the arguments that make the function partial) as subtype restriction predicates over the arguments' types. With this simple transformation, the problem of partial functions is reduced to an environment of total functions with subtypes as argument types. Thus, the problem can be solved by promoting the preconditions to subtype arguments allowing the translation of partial functions as SAL total functions leaving the verification of preconditions to the model-checking time type verifier built into the SAL model checking engine. Technically this involves using *dependent types*, as the type of a parameter may depend on the value of another, but these are supported by SAL.

However simple and efficient, the approach proposed above does not provide the desired level of verification because SAL only provides a minimal set of type verifications during run time and subtype satisfaction is not among the tests performed. Even though this is a serious drawback, it is an efficient solution to the encoding of partial functions and, by *lifting* the preconditions to the type system of the specification, it allows their verification within the planned extension to model check RAISE *confidence conditions*. In this context, the current version (without precondition verification) will be safely applicable after confidence condition verification and will allow a more efficient model checking procedure.

2.3 Function type expressions

Function type expressions are, in general, translated as regular SAL functions. There are, however, two exceptions to this rule as described below.

- Curried functions are transformed on-the-fly into lambda functions.
- Function-type declared values are declared to be of function type (using the total function type provided in SAL) and the value expression (usually, a lambda abstraction expression) is assigned to it. An example of the translation of this kind of expressions is shown below.

value
  in_range : **Int** → **Bool** =
    λ f : **Int** •
      f < 4

$\Longrightarrow$

```
in_range : [Int_ -> Bool_] =
  LAMBDA (f: IT_AN!Int_):
    f < 4
  ENDIF;
```

2.4 Set expressions

As mentioned in section 2.1, sets are modelled as total functions that return `true` when applied to a member of the set, and `false` otherwise.

All set expressions are accepted, as illustrated in table 1. Note that the context name "`SET_OPS`" is used only for illustrative purposes (in practise, the context name will be automatically generated from the base type of the set).

RSL	SAL declaration
{}	`SET_OPS!emptySet`
{x, y}	`SET_OPS!add(x, SET_OPS!add(y, SET_OPS!empySet))`
{x .. y}	`LAMBDA (z : int): x <= z AND z <= y`
{ b | b : T • p(b) }	`LAMBDA (b : T): p(b)`
{ f(b) | b : T • p(b) }	`LAMBDA (u : U): EXISTS (b : T) : f(b) = u AND p(b)`

Table 1: Set expressions

2.5 Map expressions

Almost all map expressions are accepted, but no verification is carried out over them for checking if the resulting map will be deterministic. In particular, overlapping domain values are resolved by overwriting (i.e. the value returned in the overlapping case is the one of the last modification over the map). Examples are shown in the table table 2.

Note that map expressions that involve complex comprehension expressions (i.e. those that match the pattern [e1(x) ↦ e2(x) | x : T • p(x)] where e1 : T → U1, e2 : T → U2) are not accepted in the translator. The reason for this restriction is the need of a way to generate an *inverse function* of the function "e1" (in order to obtain the original *x* value and then generate the *e2(x)* mapped value). Due to the lack of support for this functionality in SAL, it is impossible to encode this kind of comprehended map, making this feature unavailable in the translator.

RSL	SAL
$[\,]$	`MAP_OPS!emptyMap`
$[x \mapsto p,\, y \mapsto q]$	`MAP_OPS!add(x,p,MAP_OPS!add(y,q,MAP_OPS!emptyMap))`
$[b \mapsto e \mid b : T \bullet p]$	`LAMBDA (b :  T): IF p THEN m(e)=b ELSE nil ENDIF`

Table 2: Map expressions

2.6 Let expressions

As SAL supports let expressions that introduce simple bindings, the translation mechanism for the simplest case of let expressions is straightforward.

On the other hand, the translation of case expressions with bindings involving products is the most complex one due to SAL constraint to only single bindings in let expressions (i.e. only bindings of the form $\{$*`Identifier :  type = Expression`*$\}^{+}$). This restriction prevents let expressions of the form "**let** (a,b) = P **in**" (where P is of product type) being directly translatable into SAL. To solve this problem, the translator uses SAL's feature to access product fields (in SAL, product fields can be accessed by an index associated according the field's position inside the product). Using this approach, the translation mechanism is performed as follows. Suppose `e` is an expression representing a pair of type `Prod`:

let (a,b) = e **in** a > 1 **end** $\Longrightarrow$

```
LET LetId3_ : Prod = e IN
    LetId3_.1 > 1;
```

2.7 Case expressions

As SAL does not provides a case construction, case expressions are translated as a nested sequence of if expressions.

The case when the pattern in the case expression is a product is handled in a field-by-field manner. In particular, when the inner patterns are literal values, the expression is handled in a similar way to the value literal one.

For example, assuming that `x` is an **Int** $\times$ **Int** product, the translation will be performed as follows:

case x **of**
  (2,1) $\rightarrow$ 0,
  _ $\rightarrow$ 2
end

$\Longrightarrow$

```
IF (x.1 = 2 AND x.2 = 1)
   THEN 0
   ELSE 2
ENDIF
```

On the other hand, record patterns require additional tasks to be performed during the translation process. This is essentially due to the fact that a record patterns over non-empty constructors are not only introducing a binding with the components inside a particular constructor but also matching the constructor. This matching of the constructor requires a previous condition to verify that the value in the case actually matches the required constructor. This difficulty can be overcame by using SAL's *recognisers* (specially created predicates that return true only if their argument matches the proper constructor). An example of the translation procedure is shown below.

```
type
   Nonce == na | nb | nc,
   Ag == a | b | c,
   Key == ka | kb | kc,
   Message ==
      m1(n1 : Nonce, a1 : Ag, k1 : Key) |
      m2(n2a, n2b : Nonce, k2 : Key) |
      m3(n3 : Nonce, k3 : Key)

value
key : Message → Key
            key(m) ≡
               case m of
                  m1(_, _, k) → k,
                  m2(_, _, k) → k,
                  m3(_, k) → k
               end,
```

⟹

```
  Message: TYPE = DATATYPE
    m1(n1: Nonce, a1: Ag, k1: Key),
    m2(n2a, n2b: Nonce, k2: Key),
    m3(n3: Nonce, k3: Key)
  END;

key(m : Message): Key =
    IF m1?(m)
    THEN LET k : Key = k1(m)
         IN k
    ELSE
      IF m2?(m)
      THEN LET k : Key = k2(m)
           IN k
      ELSE LET k : Key = k3(m)
           IN k
      ENDIF
    ENDIF;
```

2.8 Define-before-use rule

This is a syntactic restriction that simplifies many tasks during compilation and SAL adopts it but RAISE does not. As the goal is to translate from a language where ordering is not important to one where it is, a sorting procedure must be implemented before translation in order to cope with the define-before-use rule in SAL.

The approach taken to solve this problem is to generate an extra pass over the AST extracted from the RAISE code and to generate a new syntax tree using an intermediate format, where the declarations are sorted according to the dependency among declarations in the RAISE original specification.

In particular, the sorting procedure that is used during this pass is based on the sorting algorithm used in the PVS translator [6]. Roughly speaking, the algorithm collects the set of declarations for each module and tries to reduce it iteratively until reaching the empty set (successful termination) or a point where the set can not be reduced any further (a circular dependency exists among the declarations).

Essentially, the reduction algorithm tries to reduce the set by calculating the dependencies of each element in the set and intersecting it with the set of unprocessed declarations. If the intersect operation results in an empty result, then the declaration's dependencies have already been sorted and the current declaration can be add to the sorted set. On the other hand, if, at any iteration, the result after processing the set of pending declarations is still the same, then there is a circular dependency among declarations that cannot be sorted and an error is reported.

This transformation of the original AST is carried out at the very beginning of the translation procedure, guaranteeing that the define-before-use rule is satisfied for all subsequent steps during the translation.

3 EXTENDING RAISE

Model checking techniques are essentially based on transition systems to represent the system under analysis and Linear Time Logic (LTL) to state the property that must be verified. As none of these theories has a direct representation in RAISE, a way to describe them must be added to the language. This section presents the extensions incorporated to RAISE in order to cope with these two required features in order to allow model checking of specifications.

3.1 Transition systems

It is well known that the model checking approach is based on the creation of a sound representation of the system under analysis and in computing the possible future states of the system by following all possible actions from every reachable state. It is then fundamental to be able to describe/define the transition system that the user wants to be taken as a model in order to verify properties using model checking techniques.

Due to the abstract level at which specifications are written in RAISE, the specification's underlying transition system is, in most of the cases, not obvious and, in general, not derivable by automatic means from the specification's code. It is because of this, that a mechanism for basic transition systems description is needed and there is no construction in RAISE that could serve this purpose.

With this shortcoming in mind, an extension to the RAISE language was devised in order to allow the user to express transition systems referred to the specification's code. A detailed explanation and the grammar defining this extension can be found in [20].

As an example of what can be expressed in this extension, consider a specification where a bounded stack of elements of (finite) type `T` is defined. The following transition system describes a possible model for the system description:

```
transition_system
[TRANS]
local stack : Stack := empty
in
   ([=] e : T •
      [push_trans]
         ∼full(stack) ⟶ stack′ = push(e,stack))
   [=]
      [pop_trans]
         ∼empty(stack) ⟶ stack′ = pop(stack)
end
```

From this short example, it is possible to see the declaration of a local variable (`stack`) that models the *state* of the transition system, initialised with the `empty` value. It is also possible to observe the guarded transitions that determine the evolution of the system. In particular, the first transition corresponds to a *comprehended transition*, a shorthand that is expanded, during model checking time, to a choice of the guarded transitions obtained by instantiating `e` with each of the values in `T`. The last transition, on the other hand, shows a single transition that can be triggered when the stack is not empty.

3.2 LTL properties

No axiom declaration is allowed in the translator to SAL because there is no such construction in SAL's language. There is, however, a built in way of stating properties of a model within SAL. Due to the fact that the logical operators provided by SAL are the same as the ones included in RAISE, one possible translation mechanism could be the encoding of the specification's axioms into SAL's theorems. As a matter of fact, adding the Linear Time Logic (LTL) operator `G` in front of the translated axiom will force the model checking tools to verify that the property is globally (i.e. along all steps in the execution path) true.

However effective, this approach does not allow a maximal advantage of the expressive power of LTL logic supported by the model checking tools provided in the SAL toolkit. Following this reasoning, an extension was incorporated in RAISE in order to allow the specification writer to state the desired properties of the specification under analysis or development.

Following the stack example, the following properties can be stated from the transition system above:

ltl_assertion
[stack_live]
  TRANS ⊢ G(∼full(stack))
[pop_push]
  TRANS ⊢ G(∀ e : T • pop(push(e, stack)) = stack),
[top_push]
  TRANS ⊢ G(∀ e : T • top(push(e, stack)) = e)

Note, from the example above, that all assertions must refer to a transition system (`TRANS` in this case) and that LTL temporal operators `G`, `F` and `X` (`G` in this case) can be used.

In this particular case, the first assertion should be invalid (the stack can reach the full state) and a counter-example will be generated by the model checker. The other two assertions, on the other hand, are standard stack axioms and should be shown correct by the model checker.

A fully detailed grammar of the extension to state LTL properties in RAISE and some examples can be found in [20].

4 LIMITATIONS

4.1 Recursion

Recursive constructions are a very important resource, specially in the context of applicative style specifications. It is so because recursion is the only means provided to traverse data structures of non-static or unknown dimension. In the RAISE case, it is also possible to use recursion as way to define types (defined, in an inductive way).

4.1.1 Recursive types

The only data declaration in SAL that has the syntactic expressiveness necessary to allow a recursive definitions is the `datatype` that is used to encode variants during the translation. However syntactically possible, none of the SAL tools allow the usage of this construction if it involves some kind of recursion.

The reason for this restriction is that there is no way to statically (i.e. during compilation time) resolve the recursion associated with the structure in order to calculate the set of possible values in the type. It is easy to realise that this is a serious shortcoming that can not be overcame if it is taken into account that a finite representation must be available to every type if model checking techniques are going to be applied.

It is important to highlight that this is neither a restriction of the model checking tool nor of the paradigm of model checking chosen for the translation, but a model checking theory restriction that cannot be overcame by any existing tool.

4.1.2 Recursive functions

Recursive functions, on the other hand, do not constitute a serious limitation for model checking techniques but in general, a way to determine the length of the recursion or the so called *measure* of the function is required. In the SAL case, according to the authors in [16], the SAL's type checker was supposed to be able to automatically calculate the measure of a function (provided that the language was initially devised to be very simple). This assertion proved to not to be true in the general case and providing the user with means to state recursive function measures is regarded as future work for the SAL development team.

4.2 Implicitly defined values

Implicitly defined values are a very valuable resource when developing abstract specifications because they allow the introduction of components that will be properly defined in successive refinement steps. On a more conceptual level, implicitly defined values can be seen as a reference to functionality with nondeterministic behaviour at the current level of abstraction.

On the other hand, values are used in in the model checking paradigm as a means to define how the system under study must evolve. In this context, the introduction of undefined values in a model checking transition system is not conceivable if it is taken into account that undefined behaviour would be introduced in the evolution of the system under study.

5 CONCLUSIONS AND FUTURE WORK

In this paper we have explained the problems when constructing a transformation from a formal language into a model checking language. In particular, the transformation described is from RAISE into SAL.

We have also covered the main problems faced during the translation process and, when possible, we describe the translation technique applied in those cases with a justification of how the RAISE semantics are preserved during these non trivial transformations.

Having adopted a practical approach for our translation, we have gone systematically over the most important RAISE language constructs showing whether the construct can be transformed into SAL and then providing a mapping into the semantic equivalent SAL construction or whether the construct can not be transformed and then we have said so and why.

We have also constructed a tool that implements the translation of nearly all the constructs that have been shown previously in the report that they can be translated. The tool, in particular, has served itself as a verification mean of the suitability of the approach mentioned in this report and can now be used to model check specifications in the applicative style.

Regarding as future work, an framework for confidence condition verification should be considered for implementation (again, with the double purpose of validating the translation mechanism and to provide extended model checking facilities for RAISE). Actually, we can say that the translator for confidence condition verification is work in progress.

REFERENCES

[1] J.-R. Abrial, S. A. Schuman, and B. Meyer. Specification language. In R. M. McKeag and A. M. Macnaghten, editors, *On the Construction of Programs: An Advanced Course*, pages 343–410. Cambridge University Press, 1980.

[2] Univan Ahn and Chris George. C++ translator for RAISE Specification Language. Technical report, International Institute for Software Technology - United Nations University, November 2000.

[3] J. P. Bowen, R. B. Gimson, and S. Topp-Jørgensen. Specifying system implementations in Z. Technical Monograph PRG-63, Oxford University Computing Laboratory, February 1988.

[4] Jonathan P. Bowen and Michael G. Hinchey. *High-Integrity System Specification and Design.* Springer Verlag, 1999.

[5] Li Dan and Bernhard K. Aichernig. Automatic Test Case Generation for RAISE. Technical report, International Institute for Software Technology - United Nations University, January 2003.

[6] Aristides Dasso and Chris George. Translating RSL into PVS. Technical report, International Institute for Software Technology - United Nations University, March 2002.

[7] Leonardo de Moura et al. SAL 2. In Rajeev Alur and Doron Peled, editors, *Computer-Aided Verification, CAV 2004*, volume 3114 of *Lecture Notes in Computer Science*, pages 496–500, Boston, MA, July 2004. Springer-Verlag.

[8] Jurgen Dingel and Thomas Filkorn. Model cheking for infinite state systems using data abstraction, assumption-commitment style reasoning and theorem proving. In Pierre Wolper, editor, *Computer Aided Verification*, volume 939 of *Lectures Notes in Computer Science.* 7th International Conference in Computer Aided Verification (CAV '95), Springer, 1995.

[9] B. Berard et al. *Systems and Software Verification, Model Checking Techniques and Tools.* Springer-Verlag, 1998.

[10] Edmund M. Clarke Jr. et al. *Model Checking.* The MIT Press, 1999.

[11] Wan Fokkink. *Introduction to Process Algebra.* Springer-Verlag, Berlin, Germany, 2000.

[12] S. Graf. Verification of a distributed cache memory by using abstractions. In *Computer Aided Verification*, volume 697 of *Lectures Notes in Computer Science.* 5th International Conference in Computer Aided Verification, Springer, 1994.

[13] The RAISE Language Group. *The RAISE Specification Language.* Prentice Hall International (UK), 1992.

[14] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice Hall International Series in Computer Science, 1985.

[15] Michael R. A. Huth and Mark D. Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems.* Cambridge University Press, Cambridge, England, 2000.

[16] N. Shankar Leonardo de Moura, Sam Owre. *The SAL Language Manual.* SRI International, revision 2 edition, August 2003. http://sal.csl.sri.com/doc/language-report.pdf.

[17] Michael Leuschel, Thierry Massart, and Andrew Currie. How to make FDR spin LTL model checking of CSP by refinement. *Lecture Notes in Computer Science*, 2021:99+, 2001.

[18] Formal Systems (Europe) Ltd. Failures-divergence refinement – FDR 2 user manual, 1997.

[19] Kenneth L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, 1993.

[20] Juan I. Perna and Chris George. Model Checking RAISE specifications. Technical report, International Institute for Software Technology - United Nations University, December 2005.

[21] R. Boulton, A. Gordon, M.J.C. Gordon, J. Herbert, and J. van Tassel. Experience with embedding hardware description languages in HOL. In *Proc. of the International Conference on Theorem Provers in Circuit Design: Theory, Practice and Experience*, pages 129–156, Nijmegen, 1992. North-Holland.

[22] Graeme Smith and Luke Wildman. Model Checking Z Specifications Using SAL. In *ZB 2005*, pages 85–103. International Conference of Z and B Users, Springer, 2005.

[23] Ke Wei and Chris George. An RSL to SML Translator. Technical report, International Institute for Software Technology - United Nations University, May 2001.