The United Nations University

UNU-IIST

International Institute for Software Technology

Model checking RAISE specifications

Juan Ignacio Perna and Chris George

November 2006

UNU-IIST Report No. 331

UNU-IIST and UNU-IIST Reports

UNU-IIST (United Nations University International Institute for Software Technology) is a Research and Training Centre of the United Nations University (UNU). It is based in Macao, and was founded in 1991. It started operations in July 1992. UNU-IIST is jointly funded by the government of Macao and the governments of the People's Republic of China and Portugal through a contribution to the UNU Endowment Fund. As well as providing two-thirds of the endowment fund, the Macao authorities also supply UNU-IIST with its office premises and furniture and subsidise fellow accommodation.

The mission of UNU-IIST is to assist developing countries in the application and development of software technology.

UNU-IIST contributes through its programmatic activities:

1. Advanced development projects, in which software techniques supported by tools are applied,
2. Research projects, in which new techniques for software development are investigated,
3. Curriculum development projects, in which courses of software technology for universities in developing countries are developed,
4. University development projects, which complement the curriculum development projects by aiming to strengthen all aspects of computer science teaching in universities in developing countries,
5. Schools and Courses, which typically teach advanced software development techniques,
6. Events, in which conferences and workshops are organised or supported by UNU-IIST, and
7. Dissemination, in which UNU-IIST regularly distributes to developing countries information on international progress of software technology.

Fellows, who are young scientists and engineers from developing countries, are invited to actively participate in all these projects. By doing the projects they are trained.

At present, the technical focus of UNU-IIST is on formal methods for software development. UNU-IIST is an internationally recognised center in the area of formal methods. However, no software technique is universally applicable. We are prepared to choose complementary techniques for our projects, if necessary.

UNU-IIST produces a report series. Reports are either Research R, Technical T, Compendia C or Administrative A. They are records of UNU-IIST activities and research and development achievements. Many of the reports are also published in conference proceedings and journals.

Please write to UNU-IIST at P.O. Box 3058, Macao or visit our home page: http://www.iist.unu.edu, if you would like to know more about UNU-IIST and its report series.

G. M. Reed, Director

The United Nations University

UNU-IIST

International Institute for Software Technology

P.O. Box 3058
Macao

Model checking RAISE specifications

Juan Ignacio Perna and Chris George

Abstract

This report presents the basic foundations for the verification by means of model checking techniques of formal specifications expressed in RAISE.

During this work, third party model checkers are briefly discussed and analysed for suitability under two main criteria: (a) syntactic/semantic restrictions imposed by the model checker's language and (b) the applied representation technique for the system (i.e. symbolic or explicit).

Then, the selection of Symbolic Analysis Laboratory (SAL) as the model checking tool is justified and all RAISE syntactic constructions are analysed for transformation into SAL. Foundations for the semantic preservation during the translation are provided in the cases where the justification is not a trivial one.

Finally, the design of extensions to RAISE to define transition systems and to support temporal logic formulas is described and the tool that implements the first version of the described translation procedure is also reported.

Juan Ignacio Perna is a member of the Computer Science Department at Universidad Nacional de San Luis, Argentina. He was a Fellow at UNU-IIST from March 2005 to December 2005.

Chris George joined UNU-IIST as a Senior Research Fellow on 1 September 1994, and is currently Associate Director. He is one of the main contributors to RAISE, particularly the RAISE method, and that remains his main research interest. Before coming to UNU-IIST he worked for companies in the UK and Denmark.

Copyright © 2006 by UNU-IIST

Contents

Chapter 1

Introduction

Presentation of the current work

Formal methods can be defined as mathematically based techniques for the specification, development and verification of software and hardware systems. The approach is especially relevant in high-integrity systems, for example where safety or security is important, to help ensure that errors are not introduced into the development process. Formal methods are particularly effective early in development at the requirements and specification levels, but can be used for a completely formal development of an implementation (e.g., a program). In this sense, the Rigorous Approach to Industrial Software Engineering (RAISE) is a "formal specification language that aims at providing a sound notation (with a semantics and a proof system) for capturing requirements and expressing the functionality of software"[18].

Regarding the goal of software components/systems verification, there have been several approaches, mainly differing in the degree of involvement of the user and the completeness and soundness of the assertions that they allow to deduce. In particular, *testing*, *deductive verification* and *model checking* can be mentioned as the main techniques within the field. Briefly speaking, "simulation and testing both involve making experiments before deploying the system in the field..." [13], while *deductive verification* normally refers to "the use of axioms and proof rules to prove the correctness of critical systems" [13].

Model checking, on the other hand, is a method to algorithmically verify formal systems. This is achieved by verifying if the model, often deriving from a hardware or software design, satisfies a formal specification. The specification is often written as temporal logic formulas. The model is usually expressed as a transition system, i.e directed graph consisting of nodes (or vertices) and edges. A set of atomic propositions is associated with each node. The nodes represent states of a system, the edges represent possible transitions which may alter the state, while the atomic propositions represent the basic properties that hold at a point of execution. Formally stated, the problem can described as follows: given a desired property, expressed as a temporal logic formula p, and a model M with initial state s, decide if $M, s \models p$.

In this context, the main advantage of model checking over other approaches is the much higher degree of automation achieved (almost no human interaction is needed in order to verify properties in the system). On the other hand, as the model checking technique is based in the calculation (and some times, also the representation) of all possible reachable states by the system under study, it 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* [30] and *abstraction* [16, 12] are the most used ones. Symbolic model checking tries to solve the state explosion problem by avoiding the construction of an explicit representation of the system by using boolean formulas to implicitly represent the system under analysis. Abstraction, on the other hand, is more complex for automation, but is essentially based on system simplification (either manually or, in some cases, automatically) by increasing the level of abstraction used for describing the system.

Regarding RAISE, specifications several tools for its verification are already available such as **code generators** to several languages in order to *run* the specification's code [43, 3]; **test cases** [8] (including test coverage analysis and mutation testing) and a **translator to PVS** [9] so important properties or invariants from the specification can be proved correct. There is, however, no support for Model Checking, even considering the fact that, in the case of RAISE, software is specified in a step-wise way [19]. In particular, the development process (following the *RAISE development method*) is carried out as a sequence of steps, starting from the specification of the system at a high level of abstraction and progressing by successively adding details towards a more concrete (and thereby closer to the implementation) specification. This way of development (getting closer to the implementation by decreasing the level of abstraction) is particularly suitable for model checking because it allows the verification of properties in early stages in the development process, where an abstract level description is obtained "for free" with the specification (actually, the specification itself is the abstract description of the system under study). Once verified, the RAISE development process warranties the preservation of the properties until the actual implementation of the system.

The main objective of this report is, then, the construction of a semantic framework and a tool that uses it to enable the application of the Model Checking technique to specifications written in the RAISE language (RSL).

State of the art

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 functionality into formal languages can be found in relation to the CSP [20, 4] process algebra [14]. In particular, a number of tools for analysing and understanding systems described using CSP have been produced over the years. Early tool implementations used a variety of machine-readable syntaxes for CSP, making input files written for different tools incompatible. However, most CSP tools have now standardised on the machine-readable dialect of CSP devised by Bryan Scattergood, sometimes referred to as CSP_M [38]. The CSP_M dialect of CSP possesses a formally defined operational semantics, which includes an embedded functional programming language.

The most well-known CSP tool is probably Failures/Divergence Refinement 2 (FDR2) [29], which is a

commercial product developed by Formal Systems Europe Ltd. FDR2 is often described as a model checker, but is technically a refinement checker, in that it converts two CSP process expressions into labelled transition systems, and then determines whether one of the processes is a refinement of the other within some specified semantic model (traces, failures, or failures/divergence). This refinement-based approach is particularly suitable when trying to verify the stepwise development of systems, but it has, however, obvious disadvantages over other temporal-logic-based approaches, where the specification of temporal properties is done in a more natural way. This issue was covered by Leusche *et al* in [27] by first (unsuccessfully) exploring the possibility of translating CSP into the SPIN [23] model checker and then, by providing a semantic framework to express temporal properties inside CSP, allowing the encoding of LTL properties behind a specially kind of refinement testing (performed using FDR). For this approach to be effectively applicable, the FDR tool must modified in order to allow it to automatically translate the LTL formulas in the CSP extension into an encoding based on refinement. This approach is incompatible with the goal of this report if the significant syntactic and semantic differences between RAISE applicative language and CSP are taken into account. It is, in particular, not clear how to extract the process-like description from RAISE specifications, making it hard to state the specification in the way required by FDR.

Another important case (because it uses the different approach of translating the specification into a model checking language) is the one developed by Smith and Wildman regarding Model Checking Z [1, 2] specifications [40]. In this work, the authors adopt a blocking semantics for the operations[1] and present a whole set of mathematical libraries in order to provide support for sets, relations, functions (encoded as relations and thus, providing support for partial functions), sequences and bags. Under this approach, schemes are translated using SAL's `MODULES` and channel interactions among them are modelled using `INPUT` and `OUTPUT` variables. As RAISE is much closer to Z than to CSP, this seems to be an important model to follow with the aim of adding model checking functionality to the RAISE language. There are, however, two main disadvantages in this approach:

- *Performance.* Even though the encoding functions as relations allows a straightforward translation of partial functions, this implementation is extremely inefficient when compared with the performance of SAL's regular functions.
- *Predicate-based encoding.* The approach taken to translate most Z constructions is based on *predicates*, i.e. boolean expressions that take an extra argument and return true if the that argument correspond with the actual result of the application of that function. This approach is totally unsuitable for modelling RAISE applicative constructs.

Regarding Z, there have also been other efforts to translate it into languages such as PVS [41] and Isabelle/HOL [25] but these tools are proof languages, not suitable for temporal logic verification and, thus, not comparable with the present work[2]. A similar approach has also been used for Z extensions such as CSP-OZ, CSP-Z and Object-Z but using FDR as the target language in the translation. Given the fact that FDR's input language is quite expressive, this approach allows a straightforward translation of most Z predicates (sequences and sequence operations for example). As mentioned above, FDR is not a temporal logic model checker but a refinement checker. Thus, for a given model M, the temporal logic

[1] As discussed by Smith and Winter [39], applying temporal logic model checking to Z specifications is not possible unless either all operations are totalized, i.e., have a true precondition, or a blocking semantics of operations is adopted, i.e., where operations are unable to occur outside their calculated preconditions.

[2] The main difference in this context is that these approaches follow the *proof-based* approach to verification. In a proof-based approach the system under analysis is described as a set of formulas Γ (in a suitable logic) and the specification in another formula ϕ. The verification method consists of trying to find a proof that $\Gamma \vdash \phi$.
The approach used in Model Checking, on the other hand, relies on a *model-based* approach, i.e. the system is represented in a model $\mathcal{M}$ for an appropriate logic and the specification as a formula ϕ and the verification method consists of computing whether $\mathcal{M} \vDash \phi$.

assertion that should be validated on it must be transformed itself in a model in order to then use the FDR tool to verify if this new model refines M.

In the context of RAISE, there is also a translator to PVS in order to prove invariants or theorems about the specification. Again, PVS is a proof tool that doesn't allow the verification of temporal logic assertions.

Organisation of this report

This report is organised as follows: Chapter 2 explores the two possible approaches to including the Model Checking technique to formal languages that have been recently used (implementation of a model checker tailored to the specification language or translation of the specification into some model checker's language). The chapter also presents a brief survey of the most used model checkers presenting their main characteristics and describing the expressive power of their input languages. The advantages and disadvantages of each approach are assessed and the selection of the option of translating into the SAL model checker is justified.

Then, Chapter 3 introduces the syntactic and semantic foundations that makes the translation from RSL to SAL a sound one. The main problems faced during translation (differences in the type system, limitations on bindings, define-before use rule, etc.) are presented and the implemented solution described and justified. Finally, the limitations of the translation technique are described and the extensions performed to RSL in order to make full usage of model checking verifying power are justified and fully detailed.

Chapter 4 presents a brief introduction to an extension of the present work in order to allow the usage of the model checking technique to verify the preservation of RSL *confidence conditions*. This extension will complete the minimum set of verifications in order to guarantee that the verification performed over the specification satisfies all the quality properties asserted in RSL.

Finally, conclusions of the present work and possible lines of extensions are presented.

Chapter 2

Deciding the Model Checking approach

Considering the previous works on the field, it is possible to conclude that two approaches to model checking are possible: to implement a brand new model checker tailored specifically to the formal method or to try to use an already developed model checking tool/engine and to translate the language's specifications into the model checker's language.

About developing a brand new model checking engine for RAISE, it has the advantages of the eventual better efficiency (compared with other more general model checkers if some particularities from RAISE's language can be used in order to achieve more efficient algorithms) and a closer "fit" of the model checker's language with RAISE's one (no additional syntactic or semantic restrictions but the ones inherent to the model checking techniques). On the other hand, using an already developed model checker provides the security of a well tested verification engine combined with the possible advantages from future developments of these tools that can be immediately used if backward compatibility is preserved.

In order to allow a proper evaluation of the real implications of the usage of a third party model checker, several model checkers where evaluated to analyse the real restrictions that a general purpose model checking engine will impose if used to verify properties in RAISE specifications. Towards this goal, SPIN [23], Symbolic Analysis Laboratory (SAL) [11] and SMV [30] were analysed. The first differs from the others in that it is based in the construction of an explicit state representation of the system, while the others are based on the symbolic approach to model checking.

The modelling language provided by the model checkers was also taken into account (the expressiveness of this language has a major incidence in the kind of constructions that can be translated into it). In this direction, both SPIN's language (PROMELA) and SMV's one are imperative and not very suitable for modelling applicative descriptions. SAL's language, on the other hand, is similar to PVS and provides an applicative modelling language combined with very powerful constructions to describe transition systems. Another interesting feature in SAL's language is that it is used as a front end to a whole verification toolset [10] (symbolic, infinite-bounded, backwards model checkers, deadlock analyser, simulator and path finder) that can be used according to the user's needs.

The following section explores the three model checkers mentioned above, outlines their most important features and presents the same case-study encoded in each of the respective modelling languages.

2.1 Main existing model checkers

This section presents a brief overview of the main characteristics of the three model checkers evaluated as possible target languages for the translator. In particular, SPIN, SMV and SAL are described and a case study described in all of them in order to assess their expressive power and its suitability to encode RSL applicative specifications.

2.1.1 SPIN

SPIN is a generic verification system that supports the design and verification of asynchronous process systems [22, 21]. SPIN verification models are focused on proving the correctness of process interactions, and they attempt to abstract as much as possible from internal sequential computations. Process interactions can be specified in SPIN with rendezvous primitives, with asynchronous message passing through buffered channels, through access to shared variables, or with any combination of these. In focusing on asynchronous control in software systems, rather than synchronous control in hardware systems, SPIN distinguishes itself from other well-known approaches to model checking, e.g., [7, 26].

As a formal methods tool, SPIN aims to provide:

1. An intuitive, program-like notation for specifying design choices unambiguously, without implementation detail.
2. A powerful, concise notation for expressing general correctness requirements.
3. A methodology for establishing the logical consistency of the design choices from 1 and the matching correctness requirements from 2.

Many formalisms have been suggested to address the first two items, but rarely are the language choices directly related to a basic feasibility requirement for the third item. In SPIN the notations are chosen in such a way that the logical consistency of a design can be demonstrated mechanically by the tool. SPIN accepts design specifications written in the verification language PROMELA (a Process Meta Language) [22], and it accepts correctness claims specified in the syntax of standard Linear Temporal Logic (LTL) [33].

There are no general decision procedures for unbounded systems, and one could well question the soundness of a design that would assume unbounded growth. Models that can be specified in PROMELA are, therefore, always required to be bounded, and have only countably many distinct behaviours. This means that all correctness properties automatically become formally decidable, within the constraints that are set by problem size and the computational resources that are available to the model checker to render the proofs.

Verification method and main characteristics

Unlike many model-checkers, SPIN does not actually perform model-checking itself, but instead generates C sources for a problem-specific model checker. This rather antique technique saves memory and improves performance, while also allowing the direct insertion of chunks of C code into the model. SPIN also offers a large number of options to further speed up the model-checking process and save memory, such as:

- Partial order reduction;
- State compression;
- Bitstate hashing (instead of storing whole states, only their hash code is remembered in a bitfield; this saves a lot of memory but destroys completeness);
- Weak fairness enforcement.

Some of the features that set SPIN apart from related verification systems are:

- SPIN targets efficient software verification, not hardware verification. The tool supports a high level language to specify systems descriptions, called PROMELA (a PROcess MEta LAnguage). SPIN has been used to trace logical design errors in distributed systems design, such as operating systems, data communications protocols, switching systems, concurrent algorithms, railway signalling protocols, etc. The tool checks the logical consistency of a specification. It reports on deadlocks, unspecified receptions, flags incompleteness, race conditions, and unwarranted assumptions about the relative speeds of processes.
- SPIN (starting with version 4) provides direct support for the use of embedded C code as part of model specifications. This makes it possible to directly verify implementation level software specifications, using SPIN as a driver and as a logic engine to verify high level temporal properties.
- SPIN works on-the-fly, which means that it avoids the need to preconstruct a global state graph, or Kripke structure, as a prerequisite for the verification of system properties.
- SPIN can be used as a full LTL model checking system, supporting all correctness requirements expressible in linear time temporal logic, but it can also be used as an efficient on-the-fly verifier for more basic safety and liveness properties. Many of the latter properties can be expressed, and verified, without the use of LTL. Correctness properties can be specified as system or process invariants (using assertions), as linear temporal logic requirements (LTL), as formal Büchi Automata, or more broadly as general omega-regular properties in the syntax of never claims.
- The tool supports dynamically growing and shrinking numbers of processes, using a rubber state vector technique.
- The tool supports both rendezvous and buffered message passing, and communication through shared memory. Mixed systems, using both synchronous and asynchronous communications, are also supported. Message channel identifiers for both rendezvous and buffered channels, can be passed from one process to another in messages.
- The tool supports random, interactive and guided simulation, and both exhaustive and partial proof techniques, based on either depth-first or breadth-first search. The tool is specifically designed to scale well, and to handle even very large problem sizes efficiently.
- To optimise the verification runs, the tool exploits efficient partial order reduction techniques, and (optionally) BDD-like storage techniques.

To verify a design, a formal model is built using PROMELA, SPIN's input language. PROMELA is a non-deterministic language, loosely based on Dijkstra's guarded command language notation and borrowing the notation for I/O operations from Hoare's CSP language.

SPIN can be used in three basic modes:

- As a simulator, allowing for rapid prototyping with a random, guided, or interactive simulations.
- As an exhaustive verifier, capable of rigorously proving the validity of user specified correctness requirements (using partial order reduction theory to optimise the search)
- As proof approximation system that can validate even very large system models with maximal coverage of the state space.

A simple case study in SPIN

This section presents the specification of a lift expressed in PROMELA and aims to show how a simple model can be encoded using SPIN's language. The following subsections (regarding SMV and SAL) will also cover this example in order to show the expressive power of the languages associated with each of the model checkers under evaluation.

The description of the system in PROMELA starts with constant definitions and global variables declarations. These global variables can be accessed by all processes, and therefore they are the *shared memory* of the system.

Let us consider a small elevator with three floors. The specification will, then, start with the following declarations:

```
bit doorisopen[3];
char openclosedoor=[0] of {byte,bit};
```

The bits array `doorisopen` indicates for the door of each floor, whether it is open (1) or closed (0). The `openclosedoor` channel provides communication means between the elevator and the floor doors. The declaration states that the associated buffer has length 0, i.e. that the communication will be a rendezvous. Moreover, he messages exchanged contain a byte (number of the floor to which the operation applies) and a bit (order sent to the door: 1 to open, 0 to close).

Each process starts with the keyword `proctype` followed by its name and its (possibly empty) list of arguments.

For example, process `door` takes a floor number as a parameter. It waits for an order to open, indicates that the door is open, and then that it is closed. Afterwards it signals the closing to the elevator:

```
proctype door(byte i) {
do
:: openclosedoor?eval(i),1;
   doorisopen[i-1]=1;
   doorisopen[i-1]=0;
   openclosedoor!i,0
od
}
```

The classical instructions for changing the control flow are available to describe a process. The `do` loop which is used here is an infinite loop in which a non-deterministic choice is made between all instructions sequences starting with "`::`" (there is only one in the door process).

More generally, PROMELA's `do...od` provides a form of *guarded commands* in a CSP-like manner, where guards restrict the set of sequences that can be chosen.

This mechanism is used to control the elevator process:

```
proctype elevator() {
show byte floor = 1;

do
:: (floor != 3) -> floor++
:: (floor != 1) -> floor--
:: openclosedoor!floor,1;
   openclosedoor?eval(floor),0
od
}
```

This process contains two guards: `(floor != 3)` and `(floor != 1)` which check that the elevator is no at the top floor when going up nor at the bottom when going down. The third operation which can be performed by the elevator is to send an order to open to the door of the current floor. It then has to wait for the same door to close before moving.

The next step consists in joining all these parts. The system execution starts with an initialisation process `init`:

```
init{
atomic{
  run door(1); run door(2); run door(3);
  run elevator()}
}
```

The instructions `run door(1)...` instantiate the processes corresponding to the three doors and the elevator. Each instance of the processes then runs in parallel with the ones that already exists.

Verification

The verification techniques in SPIN consist in an analysis of the complete system. In particular, it allows one to check that some property is satisfied by all the reachable states or all the possible executions of the system.

SPIN provides a primitive to indicate invariants which should be satisfied when the system is in a given state. This, it is possible to specify that when a door is open, both others should be closed, by the adding the following assertion:

```
assert(doorisopen[i-1] &&
  !doorisopen[i%3] && !doorisopen[(i+1)%3]);
```

between the instructions to open and close door number i.

Regarding temporal properties, they are specified in SPIN as a linear temporal logic (LTL) formula. For example, it is possible to prove that when the first floor door is open, it will necessarily be closed in the following state. This property can be written as a LTL formula: $\texttt{G}(open_1 \Rightarrow \texttt{X}\ closed_1)$. With SPIN, the formula is written:

```
[] (open1 -> X closed1)
```

where the corresponding atomic propositions are defined by:

```
#define open1 doorisopen[0]
#define closed1 !doorisopen[0]
```

When the property is violated, SPIN keeps track of a counter-example execution, which can be replayed. For example, consider the following property: $\texttt{F}(open_1 \vee open_2 \vee open_3)$, which means "eventually one of the three doors will be open", written:

```
<> (open1 || open2 || open3)

#define open1 doorisopen[0]
#define open2 doorisopen[1]
#define open3 doorisopen[2]
```

SPIN diagnostic indicates that the property is not satisfied and offers to replay a violating sequence. The simulation shows that our elevator can go up and down without ever stopping.

SPIN also provides other verifications, such as *useless code detection*, finding *terminal states* and *undesirable states*.

The bit state technique. This method is one the special features of SPIN. The bit state method should be used when the amount of memory available is not large enough to perform an exhaustive search. The exhaustive verification algorithm used in SPIN to compute the set of all reachable states is a classical depth first search algorithm. The set of states already encountered is stored in a hash table with collision lists: these require to store all (or almost all) the information concerning a state, in order to check whether two states are identical or not.

The bit state algorithm is similar to the classical one, but without collision list: all the available memory is used for the hash table. A state of the system is then identified by its hash key and only one bit per state is needed, to indicate whether the state has already been encountered or not. The size of the hash table is then very large and collisions are unlikely to happen. In case two states collide (share the same hash table entry), they are merged (which can be seen as a folding of the reachability graph) and all the successors of only one of these states will be examined.

Thus, this technique leads to a partial search in the reachability graph. It allows one to find errors (it is always possible to check later on that the given execution is sensible), but it doesn't guarantee the validity of the system. For example, this technique is particularly well-suited to the detection of undesired loops. In practise, it makes it possible to check many more states (by hundreds) than an exhaustive verification.

2.1.2 SMV

The SMV system is a tool for checking finite state systems against specifications in the temporal logic CTL. The input language of SMV is designed to allow the description of finite state systems that range from completely synchronous to completely asynchronous, and from the detailed to the abstract. One can readily specify a system as a synchronous Mealy machine, or as an asynchronous network of abstract, non deterministic processes. The language provides for modular hierarchical descriptions, and for the definition of reusable components. Since it is intended to describe finite state machines, the only data types in the language are finite ones - Booleans, scalars and fixed arrays. Static, structured data types can also be constructed. The logic CTL allows a rich class of temporal properties, including safety, liveness, fairness and deadlock freedom, to be specified in a concise syntax. SMV uses the OBDD-based symbolic model checking algorithm to efficiently determine whether specifications expressed in CTL are satisfied.

The primary purpose of the SMV input language is to describe the transition relation of a finite Kripke structure. Any expression in the propositional calculus can be used to describe this relation. This provides a great deal of flexibility, and at the same time a certain danger of inconsistency. For example, the presence of a logical contradiction can result in a deadlock - a state or states with no successor. This can make some specifications vacuously true, and makes the description unimplementable. While the model checking process can be used to check for deadlocks, it is best to avoid the problem when possible by using a restricted description style. The SMV system supports this by providing a parallel-assignment syntax. The semantics of assignment in SMV is similar to that of single assignment data flow languages. By checking programs for multiple parallel assignments to the same variable, circular assignments, and type errors, the interpreter insures that a program using only the assignment mechanism is implementable. Consequently, this fragment of the language can be viewed as a hardware description language, or a programming language. The SMV system is by no means the last word on symbolic model checking techniques, nor is it intended to be a complete hardware description language. It is simply an experimental tool for exploring the possible applications of symbolic model checking to hardware verification.

SMV's most important features

Some of the most important features of SMV's specification language are described below:

- *Modules.* The user can decompose the description of a complex finite-state system into modules. Individual modules can be instantiated multiple times, and modules can reference other modules. Standard visibility rules are used for naming variables in hierarchically structured designs. Modules can have parameters, which may be state components, expressions, or other modules. Modules can also contain fairness constraints which can be arbitrary CTL[1] formulas.

- *Synchronous and interleaved composition.* SMV modules can be composed either synchronously or using interleaving. In a synchronous composition, a single step in the composition corresponds to a single step in each of the components. With interleaving, a step of the composition represents a step by exactly one component. If the keyword `process` precedes an instance of a module, interleaving is used; otherwise synchronous compositions is assumed.

- *Non-deterministic transitions.* The state transitions in a model may be either deterministic or non-deterministic. Non-determinism can reflect actual choice in the actions of the system being

[1] See section 3.6.2 for a more extensive treatment of CTL logic.

modelled, or it can be used to describe a more abstract model where certain details are hidden. The ability to specify non-determinism is missing from many hardware description languages, but it is crucial when making high-level models.

- *Transition relations.* The transition relations of modules can be specified either explicitly in terms of boolean relations on the current and next state values of state variables, or implicitly as a set of parallel assignment statements define the values of variables in the next state in terms of their values in the current state.

The lift case-study in SMV

SMV's language for the description of automata (i.e. transition system) adopts a very declarative viewpoint that can be confusing. Its conception is clearly oriented towards describing a "possible next state" relation between states seen as tuples of values.

Regarding the simplified elevator model treated in SPIN, the following lines:

```
MODULE main
VAR
  cabin : 0 .. 3;
  dir : {up, down};
```

declare two variables: `cabin` that records the position of the elevator cabin, and `dir` that indicates its direction. The position is a floor number between 0 and 3. In SMV, all variables have a finite domain, thus guaranteeing the finiteness of the state space.

The global state of the system also indicates, for each $0 \leq i \leq 3$, whether there is a pending request for floor i. This is stored in an array of booleans, another finite domain variable:

```
VAR
   request : array 0 .. 3 of boolean;
```

Operational description of transitions. After the states, the transition relation must be defined. The simplest way SMV offers to this is to define a `next` relation. Here, the `next` notation applies to variables and not to the complete state, which allows the user to decompose the changes into several parts. Let us start with the description of the cabin moves in our model: in the simplified model, the cabin goes up and then down according to the infinite sequence: 0, 1, 2, 3, 2, 1, 0, 1, 2, 3, 2, etc.

```
ASSIGN
  next(cabin) := case
                    dir = up & cabin < 3 : cabin + 1;
                    dir = down & cabin > 0 : cabin -1;
                    1 : cabin;
                esac;

  next(dir) := case
```

```
                dir = up & cabin = 2 : down;
                dir = down & cabin = 1 : up;
                1 : dir;
        esac;
```

Assigning a value to "`next(cabin)`" as a function of `cabin` and `dir` really describes how the value of the `cabin` variable *in a next state* depends on the *current values* of `cabin` and `dir`. The "`case ... esac`" construction is a disjunction over cases where the first valid left-hand side is selected, and in which the final "`1`" is an always-valid left-hand side, indicating the default clause. There is a similar description of how `dir` evolves.

An equivalent way of describing the evolution of `dir` would be:

```
next(dir) := case
          dir = up & next(cabin) = 3 : down;
          dir = down & next(cabin) = 0 : up;
          1 : dir;
      esac;
```

This definition relies on the value of `next(cabin)` to define the value of `next(dir)`. It is important to note that SMV allows such a definition to occur before the definition of `next(cabin)`. In fact, these `next(...)` definitions should not be understood with an imperative assignment semantics. SMV has a very declarative semantics and it understands any boolean formula linking current and future values of the variables.

The evolution of requests is described in the same way. The simplified model allows a request for some floor to appear at any time *except if the cabin is actually on that floor*. Then a request cannot disappear until the cabin does reach that floor, in which case the request immediately disappears (that is, is satisfied).

```
next(request[0]) := case
                      next(cabin) = 0 : 0;
                      request[0] : 1;
                      1 : {0,1};
                esac;

next(request[1]) := case
                      next(cabin) = 1 : 0;
                      ...
                esac;
```

The definition uses the "`{0,1}`" construct to indicate a set of possible values for `next(request[0])`. The system is non-deterministic. In this example, non-determinism is used to model a freely evolving environment.

Description of initial states. The initial values of variables are declared according to a similar mechanism:

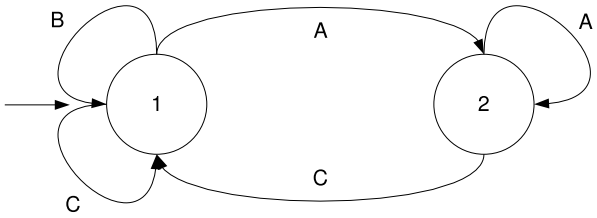

Figure 2.1: Example control automaton

```
init(cabin) := 0;
init(dir) := up;
init(request[0]) := 0;
...
```

Description of control states. SMV lacks the concept of "control state" that is generally used. For example, for the digicode pad (generally used to control reactive systems) such as the one described in figure 2.1 the usage of "current state" variables is required, all of them ranging over a finite set (e.g. an enumerated type). Then an error state must be introduced in order to describe that the automata is in some state that have no successors.

For example, we would write:

```
VAR
   state : {q1, q2, qerror};
   key : {A, B, C};

ASSIGN
   next(state) := case
                    state = q1 & (key = B | key = C) : q1;
                    state = q1 & key = A : q2;
                    state = q2 & key = A : q2;
                    state = q2 & key = C : q1;
                    1 : qerror;
                  esac;
```

It would still be necessary to define how `key` evolves with time. Obviously, this machinery is quite heavy when it comes to describing transitions between control states of automata.

Declarative description of transitions. SMV offers another more declarative way of defining initial states and transition relations.

Initial states can be denied by giving characteristic property (an arbitrary boolean formula), for example:

```
INIT
  (dir = up & cabin = 0)
```

This formula does not constrain the initial values of the `request[i]`, thus all values are possible.

A similar construction can be used for the transition relation. For example, one can characterise the set of situations where the direction of the cabin does not change, by writing:

```
TRANS
   (next(dir) = dir) <-> (!(next(cabin) = 0) & !(next(cabin) = 3))
```

Observe that all these declarations, together with the definition introduced by `ASSIGN`, are combined. They then "add up" meaning that each new declaration further restricts the set of initial states and/or possible transitions.

Verification

Having now completely defined the automaton that we wish to analyse, it is possible to submit it to the CTL model checker. The syntax for CTL is the one presented in section 3.6.2.

Let us verify that the elevator model has no deadlock[2]:

```
SPEC
  AG EX 1
```

SMV provides the answer:

```
-- specification AG EX 1 is true
```

Similarly, one can check that "all requests are eventually satisfied":

```
AG(AF!request[0] & AF!request[1] & AF!request[2] & AF!request[3])
```

Finally, it is possible to check whether "all requests are eventually satisfied (simultaneously)" with:

```
SPEC
  AG AF(!request[0] & !request[1] & !request[2] & !request[3])
```

SMV provides an error diagnostic under the form of an (ultimately) cyclic run in which requests for floors 2 and 3 appear as soon as the elevator leaves one of the floors. In this run, states are listed in succession until the loop is closed. Each state is concisely described by only giving the values of the components that have been modified while moving from the previous state.

[2]The value "1" denotes `TRUE` in SMV

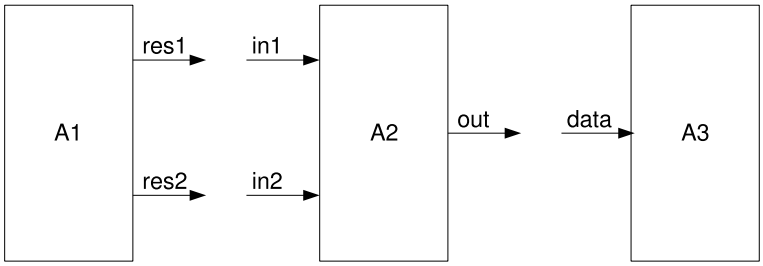

Figure 2.2: A communicating automata network

Synchronising Automata

For the construction of networks of automata, the viewpoint adopted by SMV is inspired by logical circuits. As an example, consider the network schematised in figure 2.2.

SMV describes each component of this network as an automaton in which output channels are actually state variables in which the automaton can write, and where input channels are parameters that stand for other variables in other components.

Automaton `A2` from figure 2.2 is described as follows:

```
MODULE A2(in1, in2)
VAR
  out : boolean;
  othervar : 0 .. 9;
ASSIGN
  next(out) := case
                 in2 = in2 : 0;
                 ...
               esac;
```

The global architecture of the network is defined via:

```
MODULE main
VAR
   elem1 : A1;
   elem2 : A2(elem1.res1, elem1.res2);
   elem3 : A3(elem2.out);
   other_local_vars : ...
SPEC
   (elem2.out = 1) -> ...
```

Thus modules are seen as variables of "record" type, that import their own variables. On can use several instances of the same module.

Synchronous and asynchronous networks. The example we just considered assumes a network of three synchronous modules: a transition in the global system involves a transition in every component. This point of view is suitable for logical circuits.

In SMV modules can evolve in an asynchronous way: a transition of the global system is a transition of one and only one component. This behaviour is obtained by using the keyword `PROCESS` in place of `MODULE` in the example above.

In order to allow stating assumptions on the way they are scheduled, each component provides an additional boolean variable called "`running`". In a given state, `P.running` is true if and only if `P` is the component involved in the transition. Then, temporal logic formulae can state properties and assumptions about scheduling by referring to these `running` variables.

2.1.3 SAL

SAL stands for Symbolic Analysis Laboratory. It is a framework for combining different tools for abstraction, program analysis, theorem proving, and model checking toward the calculation of properties (symbolic analysis) of transition systems. A key part of the SAL framework is a language for describing transition systems. This language serves as a specification language and as the target for translators that extract the transition system description for popular programming languages such as Esterel, Java, and Statecharts. The language also serves as a common source for driving different analysis tools through translators from the SAL language to the input format for the tools, and from the output of these tools back to the SAL language. The basic high-level goals of the SAL language are:

- *Generality.* It should be possible to effectively capture the transition semantics of a wide variety of source languages.
- *Minimality.* The language should not have redundant or extraneous features that add complexity to the analysis. The language must capture transition system behaviour without any complicated control structures.
- *Semantic Regularity.* The semantics of the language ought to be standard and straightforward so that it is easy to verify the correctness of the various translations with respect to linear and branching time semantics. The semantics should be definable in a formal logic such as PVS.
- *Language Modularity.* The language should be parametric with respect to orthogonal features such as the type/expression sub-language, the transition sub-language, and the module sub-language.
- *Compositionality.* The language must have a way of defining transition system modules that can be composed in a meaningful way. Properties of systems composed from modules can then be derived from the individual module properties.
  - Synchronous composition: In this form of composition, modules react to inputs synchronously or in zero time, as with combinational circuitry in hardware. In order to achieve semantic hygiene, causal loops arising in such synchronous interactions have to be eliminated. The constraints on the language for the elimination of causal loops should not be so onerous as to rule out sensible specifications.
  - Asynchronous composition: Modules that are driven by independent clocks are modelled by means of interleaving the atomic transitions of the individual modules.

SAL language

The SAL language is presented in terms of its concrete or presentation syntax but only the internal or abstract syntax is really important for tool interaction. The SAL language is not that different from the input languages used by various other verification tools such as SMV, Murphy [31] and SPIN. Like these languages, SAL describes transition systems in terms of initialisation and transition commands. These can be given by variable-wise definitions in the style of SMV or as guarded commands in the style of Murphy.

The SAL intermediate language is a rather rich language. In the sequel, we give an overview of the main features of the SAL type language, the expression language, the module language, and the context language. For a precise definition and semantics of the SAL language, including comparisons to related languages for expressing concurrent systems, see [10].

The type system of SAL supports basic types such as booleans, scalars, integers and integer sub-ranges, records, arrays, and abstract datatypes. Expressions are strongly typed. The expressions consist of constants, variables, applications of Boolean, arithmetic, and bit-vector operations (bit-vectors are just arrays of Booleans), and array and record selection and updates. Conditional expressions are also part of the expression language and user-defined functions may also be introduced.

A module, on the other hand, is a self-contained specification of a transition system in SAL. Usually, several modules are collected in a context. Contexts also include type and constant declarations. A transition system module consists of a state type, an initialisation condition on this state type, and a binary transition relation of a specific form on the state type. The state type is defined by four pairwise disjoint sets of input, output, global, and local variables. The input and global variables are the observed variables of a module and the output, global, and local variables are the controlled variables of the module. It is good pragmatics to name a module. This name can be used to index the local variables so that they need not be renamed during composition. Also, the properties of the module can be indexed on the name for quick look-up.

The transitions of a module can be specified variable-wise by means of definitions or transition-wise by guarded commands. Henceforth, primed variables x denote next-state variables. A definition is of the form $x = f(y, z)$. Both the initialisations and transitions can also be specified as guarded assignments. Each guarded command consists of a guarded formula and an assignment part. The guard is a boolean expression in the current controlled (local, global, and output) variables and current-state and next-state input variables. The assignment part is a list of equalities between a left-hand side next-state variable and a right hand side expression in both current-state and next-state variables.

Parametric modules allow the use of logical (state-independent) and type parametrisation in the definition of modules. Furthermore, modules in SAL can be combined by either synchronous composition `||`, or asynchronous composition `[]`. Output and global variables can be made local by the `HIDE` construct. In order to avoid name clashes, variables in a module can be renamed using the `RENAME` construct.

Besides declaring new types, constants, or modules,SAL also includes constructs for stating module properties and abstractions between modules. The form of composition in SAL supports a compositional analysis in the sense that any module properties expressed in linear-time temporal logic or in the more expressive universal fragment of CTL* are preserved through composition. A similar claim holds for asynchronous composition with respect to stuttering invariant properties where a stuttering step is one where the local and output variables of the module remain unchanged.

Because SAL is an environment where theorem proving as well as model checking is available, absence of causal loops in synchronous systems is ensured by generating proof obligations, rather than by more restrictive syntactic methods as in other languages. Consider the following definitions:

```
X = IF A THEN NOT Y ELSE C ENDIF
Y = IF A THEN B ELSE X ENDIF
```

This pair of definitions is acceptable in SAL because it is possible to prove that `X` is causally dependent on `Y` only when `A` is true, and vice-versa only when it is false – hence there is no causal loop. In general, causality checking generates proof obligations asserting that the conditions that can trigger a causal loop are unreachable.

SAL components

SAL is built around a blackboard architecture centred around the SAL intermediate language [11]. Different back-end tools operate on system descriptions in the intermediate language to generate properties and abstractions. The core of the SAL toolset includes the usual infrastructure for parsing and type-checking. It also allows integration of translators and specialised components for computing and verifying properties of transition systems. These components are loosely coupled and communicate through well-defined interfaces. An invariant generator may expect, for example, various application specific flags and a SAL base module, and it generates a corresponding assertion in the context language together with a justification of the invariant. The SAL toolset keeps track of the dependencies between generated entities, and provides capabilities similar to proof-chain analysis in theorem proving systems like PVS.

The main ingredients of the SAL toolset are specialised components for computing and verifying properties of transition systems. There are various components providing basic capabilities for analysing SAL specifications, such as validation based on theorem proving, model checking, and animation; abstraction and invariant generation; generation of counterexamples; and slicing.

Theorem Proving: SAL to PVS. PVS is a specification and verification environment based on higher-order logic [32]. SAL contexts containing definitions of types, constants, and modules, are translated into PVS theories. This translation yields a semantics for SAL transition systems. Modules are translated as parametric theories containing a record type to represent the state type, a predicate over states to represent the initialisation condition, and a relation over states to represent the transition relation.

Compositions of modules are embedded as logical operations on the transition relations of the corresponding modules: disjunction for the case of asynchronous composition, conjunction for the case of synchronous composition. Hiding and renaming operations are modelled as morphisms on the state types of the modules. Logical properties are encoded via the temporal logic of the PVS specification language.

Model Checking: SAL to SMV. SAL modules are mapped to SMV modules. Type and constant definitions appearing in SAL contexts are directly expanded in the SMV specifications. Output and local variables are translated to variables in SMV. Input variables are encoded as parameters of SMV modules.

The non-deterministic assignment of SMV is used to capture the arbitrary choice of an enabled SAL transition. Roughly speaking, two extra variables are introduced. The first is assigned non-deterministically with a value representing a SAL transition. The guard of the transition represented by this variable is the first guard to be evaluated. The second variable loops over all transitions starting from the chosen one until it finds a transition which is enabled. This mechanism assures that every transition satisfying the guard has an equal chance to being fired in the first place. Composition of SAL modules and logical properties are directly translated via the specification language of SMV.

Slicing. Program analyses like slicing can help remove code irrelevant to the property under consideration from the input transition system which may result in a reduced state-space, thus easing the computational needs of subsequent formal analysis efforts. SAL's slicing tool is based on the ideas in [15] and it accepts an input transition system which may be synchronously or asynchronously composed of multiple modules written in SAL with the property under verification. The property under verification is converted into a slicing criterion and the input transition system is sliced with respect to this slicing criterion. The slicing criterion is merely a set of local/output variables of a subset of the modules in the input SAL program that are not relevant to the property.

The output of the slicing algorithm is another SAL program similarly composed of modules wherein irrelevant code manipulating irrelevant variables from each module has been sliced out. For every input module there will be an output module, empty or otherwise. In a nutshell the slicing algorithm does a dependency analysis of each module and computes backward transitive closure of the dependencies. This transitive closure would take into consideration only a subset of all transitions in the module.

Boolean abstraction. SAL toolset also implements boolean abstraction of specifications. In this sense, the tool is based on the idea of the boolean abstraction scheme defined in [17] that uses predicates over concrete variables as abstract variables to abstract infinite or large state systems into finite state systems analysable by model checking. The advantage of using boolean abstractions can be summarised as follows:

- Any abstraction to a finite state system can be expressed as a boolean abstraction.
- The abstract transition relation can be represented symbolically using Binary Decision Diagram (BDDs). Thus, efficient symbolic model checking can be effectively applied.
- There exists an efficient algorithm for the construction of boolean abstractions [37].

Explicit Model Checking. Finite-state SAL modules can be translated to SMV for model checking as explained above. However, model checkers usually do not allow to access their internal data structures where intermediate computation steps of the model-checking process can be exploited. For this reason, the SAL toolset also implements an explicit-state model checker for SAL systems obtained by boolean abstraction. The abstract SAL description is translated into an executable Lisp code that performs the explicit state model checking procedure allowing the tool to explore about twenty thousand states a second. This procedure builds an abstract state graph that can be exploited for further analysis. Furthermore, additional abstractions can be applied on the fly while the abstract state graph is being built.

Automatic refinement and abstraction. When model checking fails to establish the property of interest, it is also possible to indicate SAL to use the results developed in [36, 37] to decide whether

the constructed abstraction is too coarse and needs to be refined, or that the property is violated in the concrete system and that the generated counterexample corresponds indeed to an execution of the concrete system violating the property. This is done by examining the generated abstract state graph. The refinement technique computes the precondition to a transition where non-deterministic assignments occur. The preconditions corresponding to the cases where the variables get either `TRUE` or `FALSE` define two predicates that are used as new abstract variables.

The lift example in SAL

As a first approach, it is possible to use a modelling style very similar to the one used in SMV for the lift example. In this sense, it is possible to state in SAL a declaration of a module (under SMV's semantics defined above) that encapsulates the lift's behaviour.

```
lift : CONTEXT =
BEGIN

  Direction : TYPE = {up,down};
  Floors: TYPE = [0 .. 3];

  lift : MODULE =
  BEGIN
  LOCAL
    request : array [0 .. 3] of BOOLEAN,
    cabin : Floors,
    dir : Direction
```

In this case, the lift is abstracted as a variable `cabin` (that models the floor where the cabin is passing by/stopping at a given time), a variable `dir` that memorises the actual direction of the lift and an array of users' requests (modelled as a boolean array in the variable `request`). It is important to note that in SAL, local variables (declared inside the `LOCAL` environment) constitute the "state variables" the transition system modelled in the module. Other variable qualifiers are also possible, such as `INPUT` or `OUTPUT` (used as an explicit communication mean among modules when networks of transition systems are built), or as `GLOBAL` (used to implement some sort of "shared memory" communication style under certain conditions, see [11]).

Initial states. The description of the initial state for the transition system is specified in the `INITIALISATION` section. Several styles may be used for the initialisation of `LOCAL`, `GLOBAL` or `OUTPUT` variables (`INPUT` variables are not controlled by the module and, thus, not initialisable).

```
  INITIALISATION
    cabin = 0;
    dir = up;
    (FORALL (f : Floors) : request[f] = FALSE);
```

In fact, explicit initialisations (such as for the variables `dir` and `cabin`) and quantified initialisations (as used to initialise the `request` array). Unspecified initialisation values are also possible (every variable

left out of the `INITIALISATION` section will be randomly initialised by the model checker).

Operational description of transitions. SAL also endorses the idea of specifying the system's evolution by means of a transition relation among states. In this case, the way used for achieving this goal consists of a `TRANSITION` section inside the module that states the possible next states of the system.

```
TRANSITION
[
   go_up_no_top:
   dir = up AND cabin < 3 -->
              request'[cabin] = FALSE;
              cabin' = cabin + 1;
[]
```

Transitions in SAL can be stated atomically (as above), in the form of an optional label (useful when interpreting the transitions that leaded to a counter-example after model checking a system), a guard (`dir = up AND cabin < 3` in the example) and a set of actions performed atomically if the guard is true.

It is also possible to specify comprehended transitions (as the one used below to update the `request` array) that are based on the declaration of a comprehended variable and a set of commands that are expanded on model checking time following all the possible instantiations of the variable.

```
[]
  ([] (f : Floors) :
   update_request_floor_f:
     request[f] = FALSE AND f /= cabin --> request'[f] = TRUE;)
```

Finally, "default transitions" can be stated as follows:

```
[]
  idle:
  ELSE --> cabin' = cabin;
           dir' = dir;
]
```

It is important to note that when multiple guards are true at a given time, one of them is selected non-deterministically by the model checker. This enables to model non-deterministic behaviour (i.e. *internal choice* following CSP's semantics).

Verification. In SAL, properties are stated in LTL and must be bounded to a given `MODULE`. Only a minimal set of LTL operators are implemented (`G`, `F` and `X`) but they can be combined in any possible way.

As an example, consider the "liveness" property that states that the lift will eventually satisfy any request. It can be stated in SAL as:

```
liveness : THEOREM
   lift |- G(FORALL (f : Floors) : request[f] => F(NOT request[f]));
END
```

Regarding the deadlock-freedom property stated in SMV, SAL provides an specific tool (`sal-deadlock-checker`) that verifies that a given transition system never reaches a deadlock state.

Applicative functions inside the model. SAL also provides the user with the possibility of declaring a whole applicative set of functions and data types in order to simplify the declaration of transition systems and to increase the expressive power of the language. In fact, high level constructs such as anonymous functions (the so called *lambda functions*), *total-function* type declarators, let expressions, quantifiers in logical expressions and context instantiations.

With all these constructs, it will be possible to specify a far more complex model of the lift case study. In particular, the lift may be defined now as a motor (to control the movement and direction of the lift), the doors (to cover both, the doors in the cabin and the individual doors on each floor) and the buttons (in the inner control panel inside the lift and the up/down caller on each floor). This extended definition of the lift can be specified in SAL with the following declarations:

```
DS  : CONTEXT = doors{MIN_FLOOR, MAX_FLOOR};
BS  : CONTEXT = buttons{MIN_FLOOR, MAX_FLOOR};
M   : CONTEXT = motor{MIN_FLOOR, MAX_FLOOR};

Lift : TYPE = [# mtr : M!Motor,
                 drs : DS!Doors,
                 btns: BS!Buttons #];
```

The fragment of code shows how parametric contexts (`doors`, `buttons` and `motor`) can be instantiated and, thus, imported to another context. The extended `Lift` type is also declared by means of the record declarator.

In this context, the function to calculate the next state of the system in function of the current state of the system and the requirements that may arise can be encoded in a function `next` as follows:

```
next(r : T!Requirement, l : Lift) : Lift =
       LET d : T!Direction = M!direction(l.mtr) IN
       IF (M!movement(l.mtr) = T!halted)
          THEN IF (r.after = TRUE)
               THEN
                  move(d, T!halted, l)
               ELSIF (r.before = TRUE)
               THEN
                  move(T!invert(d), T!halted, l)
               ELSE l
               ENDIF
          ELSE % T!moving...
               IF (r.here = TRUE)
```

```
                    THEN
                       halt(l)
                    ELSIF (r.before = FALSE AND r.after = FALSE)
                    THEN
                       halt(l)
                    ELSIF (r.after = TRUE)
                    THEN
                       move(d, T!moving, l)
                    ELSE
                       move(T!invert(d), T!moving, l)
                    ENDIF
            ENDIF;
```

The complete applicative encoding of the extended lift can be found on appendix A at the end of this report.

Automata networks

SAL also uses the idea of connecting several modules in a circuit-based way where each component in the network is an automaton (`MODULE`). The main difference with SMV is that state variables used as a communication mechanism are declared with its functionality explicitly specified (i.e. `INPUT` or `OUTPUT`). Another significant difference is that channels are not explicitly supported by SAL, hindering SAL's ability to model systems that use this kind of semantics.

Following the example from figure 2.2, automaton `A2` would be described in SAL as:

```
A2 : MODULE =
BEGIN
  INPUT
    in1 : BOOLEAN,
    in2 : BOOLEAN
  OUTPUT
    out : BOOLEAN
  LOCAL
    othervar : ...
INITIALISATION
    othervar = ...
TRANSITION
[
  in1 = in2 --> ...
[]
  ...
]
END;
```

Then, the whole network described in figure 2.2 can be described as:

```
main : MODULE =
```

```
    WITH
      LOCAL
        res1Toin1 : BOOLEAN,
        res2Toin2 : BOOLEAN,
        outToData : BOOLEAN
      (RENAME
        res1 TO res1Toin1, res2 TO res2Toin2 IN A1)
      ||
      (RENAME
        res1Toin1 TO in1, res2Toin2 TO in2, out TO outToData IN A2)
      ||
      (RENAME outToData TO Data IN A3);
END;
```

In the example above, a set of local variables (`res1Toin1`, `res2Toin2` and `outToData`) are used to communicate the different automata in the network. Synchronous composition (indicated by the `||` operator) is used to synchronise all the automata in the network.

2.1.4 Translating RSL to SAL

After analysing the possible options to implement the Model Checking technique for RAISE the decision was made in favour of using the Symbolic Analysis Laboratory (SAL), a third party model checker. The decision was made based on the following considerations:

- The fact that an in-home development of a model checker customised particularly for RSL may have less syntactic/semantic restrictions than translating to an already developed model checking tool has been proved not too relevant given the expressive power of the input language of modern model checking engines. In particular, SAL's input language is particularly similar in the applicative constructions provided, allowing a shallow or deep encoding [35] of most of RAISE applicative constructs in a very efficient way.
- The possibles advantages regarding efficiency that may be achieved by a particular solution (that might take advantage of certain constructs in RAISE) are overcame by the multiple advantages in performance provided by SAL's multiple optimisation techniques mentioned in the previous sections (slicing, boolean abstraction, etc.).
- The fact that SAL is based on the symbolic approach to model checking and that the symbolic technique is not suitable for software analysis where dynamic creation of objects/data loses relevance in the context of applicative RAISE specifications, where no dynamic object creation is possible. Together with this idea are the multiple advantages that the symbolic management of states can offer: sets of states can be handled collectively (better temporal performance in some cases) and the ability to handle much bigger state spaces due to symbolic representation (compared with the explicit state representation approach).
- An in-house development of a model checker will imply a huge implementation effort and time. Moreover, issues such as maintenance, error fixing and updates (as new approaches and techniques are very often created, making existing approaches obsolete) became a major issue regarding the usability of the outcome of this report. Using a third-party model checker, on the other hand, transfers all the maintenance and bug fixing responsibilities to the tool developing team. Regarding

SAL, given the fact that a common language was designed as an intermediate language for several tools, backward compatibility of the input language is guaranteed, ensuring that future (possibly updated) versions of the model checker will still be able to run the current generated code for the SAL toolset.

- As mentioned in the previous section, SAL provides a whole set of model checking verification techniques, based on different verifying approaches: forward model checking, backward model checking, infinite-bounded model checking, path finder, translation to other model checkers (such as SMV), graphical simulation (in Java) and deadlock checking. All of these tools using several optimisation engines and techniques (such as incidence cones, slicing and boolean abstraction).
- SAL provides good quality in the counter-examples produced.
- SAL is free (for academic purposes).

Chapter 3

Translation issues

As SAL and RAISE are languages created with very different purposes, it is reasonable to find differences in the kind and expressive power of certain constructions/restrictions among them. The differences arise, mainly, in areas where RAISE provides more powerful constructions (like the case of the bindings) that are not provided/supported in SAL due to the complexity that this constructions will imply during the model checking procedure.

The following sections explore the main differences found while designing the translation from RAISE to SAL and explains the approach adopted to solve them.

3.1 Syntactic issues

The most important step when translating one language to other is to properly convert the syntactic structure from the former language to the one of the latter. In particular, using the appropriate constructions in the target language in order to achieve semantic equivalence between the original specification and the generated model checking specification is a crucial goal to ensure sound results from the verification.

This section explores the translation approach used for each of RAISE's syntactic constructions and provides the foundations for the semantic equivalence in the cases where SAL and RAISE's semantics are not totally compatible.

3.1.1 Declarations

A declaration translates to one or more type, constant, function or variable declarations as described for the various kinds of declarations.

Scheme declarations

The scheme declaration is directly translated to a `CONTEXT` construction with the same name.

Object declarations

As the current implementation of the translator is only handling the applicative specification style, object declarations are translated as instantiations of the proper SAL `CONTEXT`.

This approach is sufficient because object instantiations, when using the applicative style, are used only to introduce a name space in the current schema. The same functionality is achieved in SAL either by context inclusion or by context qualification. Even though the translator is currently using the later approach, it can be adapted to use the former if needed (for a more comprehensive explanation of the translation of objects – including the planned translation approach for object declarations in other specification styles – see 4.4).

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:

Sorts SAL does not support this kind of declaration[1]. Generating (by macro expansion) an enumerated or union type with possible values was the first choice to solve this problem. This approach has, however, the disadvantage that the values in the type will not exist in the RAISE's specification environment, so they won't be "addressable" in the specification. Moreover, all functions involving this type (either as a type for a parameter or for the result) must be also abstractly specified and this kind of function is also not translatable. Thus, the final approach taken with respect to abstract type declarations is to not to allow them in the translator.

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 use a special integer type generated automatically during translation time as a subtype (by means of using the *subrange* structure in SAL) of the `INTEGER` basic type in SAL. The subtyping used is of the form (in RAISE):

Int_ = {| x : **Int** • x **isin** {DefaultLow .. DefaultHigh} |}

where `DefaultLow` and `DefaultHigh` can be modified to allow experimentation with the model and properties under verification.

[1] This is because sort types will generate empty types in the model checker that, in turn, will lead to spurious results under some quantification cases. For a more detailed explanation, see [11].

Natural type This type is translated similarly to **Int**, as if defined in RSL by

Nat_ = {| x : **Int** • x ∈ {0 .. DefaultNatHigh} |}

Records 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 hereby allows all the operations from the latter to be applied on the former. Based on this semantic difference is that the implementation with datatypes was preferred over the (more particular) record construction.

Variants Variant definitions are translated to the SAL type declarator `DATATYPE`. For example, the construction

type
V ==
Vconst |
Vint(T0) |
Vrec(d1Vrec : T1, d2Vrec : T2 ↔ r2Vrec)

is translated to the SAL declaration:

```
V : TYPE = DATATYPE
  Vconst,
  Vint(acc_1_ : T0),
  Vrec(d1Vrec : T1, d2Vrec : T2)
END;

r2Vrec(d2Vrec_1_: T2, v_1_ : V) : V =
  Vrec(d1Vrec(v_1_), d2Vrec_1_);
```

From the translation example, it is important to notice that:

- The constructors will translated directly to constructors in the SAL model.
- The destructors will also be translated with the same name in the proper datatype field. Although destructors are optional in RAISE, SAL does not allow this feature to be optional in the `DATATYPE` declarations. Due to this restriction, an explicit destructor is automatically generated during the translation process with the rule: `acc_n_` (where `n` is an integer used for making the accessor unique inside the variant) when no name is provided in the specification.
- Reconstructor declarations are directly translated as explicit functions in the model.

Unions 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.

Collection types 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 in total functions. Having the implementation based in 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 this way, the type declaration:

MySet = **Int-set**

will be translated into SAL as:

```
MySet : TYPE = [Int_ -> Bool_]
```

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 `nil` especial value) and allows the encoding the map as a total function in SAL.

- Maps might be non deterministic (for example, [x ↦ y | x,y : **Nat** • {x,y} ⊆ {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.

  The only construction in SAL that allows non deterministic behaviour (i.e. *internal choice* in CSP terms) is a `MODULE` where multiple guarded transitions are ready to be triggered simultaneously (i.e. their respective guards are true at the same time). Even though the intended behaviour (non determinism) is achieved with this approach, it is impossible to create an interaction between standard functions (they belong to the *applicative* set of constructions) and modules (inside the *transition system* set of constructions) in SAL. Under this restriction, the implementation of map applications (and of maps in general) will lead to the total translation of the specification's code

into `MODULE` constructions, making the resulting code much bigger (considering the amount of state-variables in the model) and preempting the advantages gained by having as much code as possible outside the transition system in the model[2].

There is also another alternative using a similar approach to the one used in [40] to model relations as a set of pairs, where the elements in the pair belong to the domain and range of the relationship (map in this case) respectively. This approach, however, only allows the definition of a predicate for the map application of the form

$$\text{map\_app} : \text{Map\_Dom} \times \text{Map\_Rng} \times \text{Map\_Dom} \xrightarrow[m]{} \text{Map\_Rng} \rightarrow \textbf{Bool}$$

that does not provide a way to actually "calculate" the result of the application. As it is necessary to provide a type definition that supports a way to obtain the result of a map application, this approach is also not feasible as the translation for maps.

From the above observations it follows that non deterministic maps can not be translated (at least, not in an efficient way) 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.

Finally, lists are not accepted in the translation. This design decision is not based in the impossibility to find an adequate translation for them but in the severely restricted amount of functionalities expressible with lists without having recursion (see subsection 3.7.1) or iteration (see subsection 3.1.5) as traversing mechanisms over them.

As a matter of fact, two approaches were devised as translation strategies for lists: either as total functions from a subset of **Int** to the list domain or using the array data structure provided in SAL. Both approaches heavily rely on the finiteness (and statically known) length of the list in order to allow macro processing expansion to generate the list operations. In particular, restrictions over lists can create "holes" in the representation (i.e. values in the **Nat** domain of the list implementation that are not mapped to any value, making the list discontinuous) that can be solved by explicitly reconstructing the resulting list with on-the-fly shifting (to cover the holes).

Value declarations

Typings. Typings are not accepted.

Explicit value definitions. An explicit value definition translates to a constant declaration in the model.

Implicit value definitions. Implicit value definitions are not accepted.

[2]Essentially, the advantage lies in that all the behaviour encoded in functions in SAL is not explicitly represented inside the kripke structure that evolves during the model checking process and thus, does not contribute to the state explosion problem while verifying properties of the system.

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. For example, the RAISE declaration

next : **Int** → **Int**
next(n) ≡ n + 1

translates to the SAL function (the actual qualification for the `Int_` type is not shown for simplicity):

```
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. Another approach to solve this problem will be to generate a new identifier associated with this function by including strings describing either the operator's and the result's type (encoded in the same way used in the translator to SML [43]) or to include the position where the identifier was found in the specification code. However valid, both approaches tend to decrease the translated code's readability and, as function overloading is not an essential feature, readability is preferred over it.

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. However possible, due to readability issues, this feature is also not implemented.

Regarding sets and maps, there are no predefined operators over them (as SAL does not directly support them as built-in types). For this two particular cases, the translator will generate a file with proper macro declarations that will be expanded before model checking the specification. In particular, the names for the set and map operations that will be generated during the macro expansion are as described in tables 3.1 and 3.2.

Operator	Function name	Operator	Function name
x = y	`=`	x ∉ y	`not_isin(x, y)`
x ≠ y	`/=`	x ∩ y	`intersection(x, y)`
x ⊃ y	`strict_supset?(x, y)`	x \ y	`difference(x,y)`
x ⊂ y	`strict_subset?(x, y)`	x ∪ y	`union(x, y)`
x ⊇ y	`supset?(x, y)`	x ∈ y	`isin(x, y)`
x ⊆ y	`subset?(x, y)`		

Table 3.1: Function names for set operators

Partial function encoding. As mentioned above, SAL does not provide support for partial functions. This version of the translator assumes that all preconditions are verified so translates partial functions as SAL total functions.

It is planned, as an extension of the translator, a new version of the tool to generate a model for confidence condition verification (precondition satisfaction is included among them) so it will be possible to model

Operator	Function name	Operator	Function name
x = y	`=`	x / y	`restriction_to(x, y)`
x ≠ y	`/=`	x \ y	`restriction_by(x, y)`
x † y	`override(x, y)`	**dom** x	`dom(x)`
rng x	`rng(x)`		

Table 3.2: Function names for map operators

check confidence conditions over a special extended model (see subsection 4.1). This extended model for confidence condition verification will be much more inefficient than a model without code for these verifications and, hence, not suitable for properties verification over the specification. 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.

Recursive functions. Recursive functions are not accepted. This restriction is inherited from a similar restriction in SAL that does not allow recursive constructions.

Implicit function definitions. Implicit function definitions are not accepted.

Variable declarations

As the current version of the translator operates only on the applicative specification style, variable declarations are not accepted (variables are allowed in RAISE in specifications that use one of the imperative styles only).

A scheme for variable declarations translation was already devised and is stated in subsection 4.4 where the extensions for imperative sequential and concurrent styles are described.

Channel declarations

Again, as channel declarations belong to the concurrent style of specification, channel declarations are not translated in this version of the tool. It is planned, however, to provide an extension to include this style enabling the translation of this construction. A possible implementation for this feature is described in subsection 4.4.

Axiom declarations

Axiom declarations are not accepted. However, as will be explained in subsection 3.6.2 a more suitable model checking construction for expressing properties is provided. With this constructions, most axioms reflecting desired properties could be easily rewritten as SAL properties in the translated model.

Test case declarations

Test case declarations are ignored.

3.1.2 Class expressions

A class expression translates to the declarations and statements which the translation of the contents of the class expression results in.

Basic class expressions

A basic class expression translates as its declarations.

Extending class expressions

An extending class expression translates as a new class declaration that includes all the extended class declarations.

Hiding class expressions

Hiding class expressions are ignored.

Renaming class expression

Renaming class expressions are ignored.

With expression

With expressions are ignored: names are qualified as if they had been in RSL.

Scheme instantiations

Scheme instantiations in the applicative style have the property that they do not introduce variables (as there are no variable declarations in the applicative style). Under this conditions, the role of schemes is just to provide type and value declarations and they are made available in any environment by instantiating the scheme. In particular, as the translator only works with specifications written in the applicative style, there is no need give any special treatment to scheme instantiation. As a matter of fact, scheme instantiations do not generate any construction from the point of view of the generated code. There are,

however, some actions that are performed internally by the translator when a scheme instantiation is found in order to collect the object qualification that is introduced in the context.

3.1.3 Object expressions

Object expressions are partially supported by the translator.

Object names. Are accepted and translated as the proper `CONTEXT` reference as a qualification to the expression originally quantified by the name in the RAISE specification. For example, given the following RAISE fragment of code (assuming the existence of a scheme called NAMES that provides a type *Name* and a function *to_bool*):

object N : NAMES
value
  f : N.Name → **Bool**
  f(n) ≡ N.to_bool(n)

will be translated as:

```
f(n: SAL_TYPES!Name) : BT_AN!Bool_ =
NAMES!to_bool(n);
```

From the example note that several context qualifications have been introduced:

- The `Name` type is qualified but with a different context (`SAL_TYPES`). This is due to the type declaration relocation used to avoid circular dependencies (see subsection 3.3)
- The **Bool** type is qualified in SAL (while it was not in RAISE). The basic types are declared in special contexts during the translation procedure so, there is a need for additional qualification when using them in the SAL code. This is also a consequence of the relocation of declarations introduced to avoid circular dependencies in SAL.
- The function *to_bool* is qualified with the context `NAMES`. Note that the object name (`N`) is neither declared nor used for qualification. This is an example of how this approach is consistent with the translation approach used for scheme instantiation and shows how qualifications are resolved (internally, an association was made between the object name '`N`' and the context (`NAMES`) where the class/object was declared).

Object array expressions. As object arrays are not supported in this version of the translator, *element object* expressions and *array object* expressions are also not accepted in the translator.

Fitting Object expressions. As parametrized schemes are not accepted there is no real use in Fitting Object expressions inside the generated SAL code. Thus, they are not accepted in the translator.

3.1.4 Type expressions

As mentioned in subsection 3.1.1, the SAL type system is used to translate RAISE's basic type system. This approach is planned to be modified when extending the translator to verify confidence condition satisfaction (see subsection 4.1 for a more detailed description about type expressions in the extension).

Type Names

A type name translates to the correspondent SAL type name.

Product type expressions

Product type expressions are directly translated into SAL's tuple declarations. For example, the declaration

Prod = **Nat** × **Nat** × **Bool**

translates to the SAL declaration

```
Prod : TYPE = [NT_AN!Nat_, NT_AN!Nat_, BT!Bool_];
```

Set type expressions

As the translations of sets does not rely on SAL's basic type system, set type expressions are handled in a very different way than other basic type constructions. In particular, the policy used in the translator for sets (and also for maps) is to generate a new *set context* for every set declaration/type expression found. To avoid multiple set declarations of the same domain type, information is kept internally during the translation process in order to reuse existing sets in case that the same set occurs in multiple type expressions/declarations.

In general, declarations of the form:

MySet = T**-set**

Translate to the generation of a new set type (inside the `SAL_TYPES` context):

```
T_set :  TYPE = [SAL_TYPES!T -> BT_AN!Bool_];
```

and to the declaration (also in the `SAL_TYPES` context) of:

```
MySet : TYPE = T_set;
```

As a record of the existing set types is created and searched on the occurrence of every new set type expression, an efficient way of matching types is needed (in order to determine if a set with the current domain type has already been declared). Moreover, due to the fact that SAL rules for type equivalence are not based in the *maximal* criterion used in RAISE, a more restrictive type equivalence relationship is needed during the translation. In particular, SAL distinguishes subtypes from the subtyped type (see subsection 3.5 for a more in-depth treatment of this topic) and as information to make this kind of distinction possible is, in the general case, not available in RAISE, a restriction need to be imposed over the valid types that can be used in the domain of sets. In particular, only those set expressions that involve a set which domain is either a basic or a defined type are accepted. Under this restriction, the declaration:

MySet = (**Bool** × **Int**)**-set**

will be rejected by the translator because (**Bool** × **Int**) it is neither a basic type nor a defined one. On the other hand, this problem can be easily overcame by turning the former declaration into:

MyData = **Bool** × **Int**,
MySet = MyData**-set**

List type expressions

List type expressions are not accepted (as the list type is, in general, not translated).

Map type expressions

Finite map expressions are translated using the same rules explained for sets to a new map type declaration (if there is not another previously declared map with the same domain and range). As with any other type declaration, the domain of the map should be bounded not only to a type, but to a finite (possibly reduced) subrange of it. As expressed in subsection 3.1.1, neither infinite maps nor non-deterministic map type expressions are allowed.

Function type expressions

Function type expressions are, in general, translated as described in subsection 3.1.1. 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. For example, the declaration:

  value
  in_range : **Int** → **Bool** = λ f : **Int** • **if** f < 4 **then true else false end**

will be translated into:

```
in_range : [IT_AN!Int_ -> BT_AN!Bool_] =
  LAMBDA (f: IT_AN!Int_):
      IF f < 4
      THEN TRUE
      ELSE FALSE
      ENDIF;
```

Subtype expressions

Subtype expressions are translated using SAL subtype declarations.

Bracketed type expressions

A bracketed type expression translates as its constituent type expression.

3.1.5 Value expressions

Value literals

Value literals of type **Bool**, **Int** or **Nat** translate to the corresponding value in SAL. On the other hand, values of literal of type **Unit**, **Real**, **Char** and **Text** are not accepted.

Notice: This way of translation is planned to change in the extensions of the tool in order to allow confidence condition generation.

Names

A value name translates as a name.

Pre names

Pre names are not accepted.

Basic expressions

The **skip** expression is ignored.

stop, **chaos** and **swap** are not accepted.

Product expressions

Product expressions are translated as SAL tuples.

Set expressions

As mentioned in subsection 3.1.4, 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 3.3. Note that the context name "`SET_OPS`" is used only for illustrative purposes (in practice, 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 3.3: Set expressions

List expressions

List expressions are not accepted.

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 3.4 below. Again note the the context name "`MAP_OPS`" is used only with illustrative purposes (in the case of maps, the name of the operations context will be generated by concatenating the domain and range types' names).

RSL	SAL
[ ]	`MAP_OPS!emptyMap`
[x ↦ p, y ↦ q]	`MAP_OPS!add(x,p,MAP_OPS!add(y,q,MAP_OPS!emptyMap))`
[b ↦ e | b : T • p]	`LAMBDA (b : T): IF p THEN m(e)=b ELSE nil ENDIF`

Table 3.4: Map expressions

Note that the examples in the table only include cases of simple comprehended map expressions (i.e. those that can be matched with the pattern [x ↦ f(x) | x : T • p(x)] where the domain value is just an occurrence of x).

On the other hand, 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 maps making this feature not available in the translator.

Function expressions

Function expressions are directly translated to SAL's `LAMBDA` abstraction. In particular, the RAISE expression

λ x : **Int** • x + 1

translates into SAL's construction:

```
LAMBDA (x : IT_AN!Int_) : x + 1
```

Application expressions

A valid application expression may be translated to a function call or a map application. List application expressions are not accepted.

Quantified expressions

Quantified expressions are accepted by the translator. In particular, for those quantifiers that are directly supported in SAL (∀ and ∃), the translation mechanism is straightforward (i.e. the proper SAL quantifier is used).

Only the ∃! quantifier is not supported by SAL. For its translation, the quantifier is translated as `EXISTS (x: T) : p(x) AND (FORALL (x1: T) : p(x1) => x = x1)`, where `x` and `p` are, respectively, the variable (or set of variables) and the restriction bounded by the quantifier.

Equivalence expressions

Equivalence expressions translate as equalities in SAL.

Post expressions

Not accepted.

Disambiguation expressions

Disambiguation expressions are, in general, ignored in all cases except in set or map expressions involving empty sets/maps. In this case, the disambiguation is needed in order to cope with the type restrictions for collections explained in subsection 3.1.4.

Bracketed expressions

A bracketed expression translates to a SAL bracketed expression.

Infix expressions

Statement infix expressions. As this kind of infix expressions belong to the imperative style, they are not accepted in the current version of the translator. For an explanation of the approach that was designed to translate this kind of construction, see subsection 4.4.

Axiom infix expressions. As SAL provides a larger set of logical combinator than the one provided in RAISE, the translation of the axiom infix expressions is straightforward. In this way, the infix operators $\Rightarrow$, $\wedge$ and $\vee$ are directly translated into SAL's built-in operators (`=>`, `AND` and `OR`).

Value infix expressions. Due to the binding created among RAISE's and SAL's type systems by the translator, expressions involving infix operations over elements of any basic type are directly translated into their SAL counterparts.

The case of infix expressions that involve set or map operations are handled differently, according to the following rules:

- Equality/Inequality. These are the only set/map infix operations that remain as infix operations in the translated code. This is possible due to the implementation of the collections as functions (as lambda functions in particular). Under this particular implementation, it is possible to use the equality/inequality operations in SAL to compare sets/maps.

- Other operations. The rest of the operations are provided in a different context generated for each individual set/map used in the specification. The names of the operations correspond to the names described in tables 3.1 and 3.2 and, as SAL does not support infix operator definition. They are turned into prefix operations during the translation process.

However efficient and simple, this approach is not extensible for translator version that supports confidence condition verification. A more general approach is proposed in subsection 4.1.

Prefix expressions

A prefix expression generally translates to the corresponding SAL expression. The axiom prefix expression $\sim$ translates to the expression `NOT`. On the other hand, the universal prefix expression ($\Box$) is not accepted by the translator.

The rest of the value prefix expressions translate to function calls, using the function names described in table 3.5.

RSL	SAL	RSL	SAL
abs	`abs`	**elems**	`not accepted`
int	`not accepted`	**hd**	`not accepted`
real	`not accepted`	**tl**	`not accepted`
card	`not accepted`	**dom**	`dom`
len	`not accepted`	**rng**	`rng`
inds	`not accepted`		

Table 3.5: Translation of built-in prefix operators

The **int** and **real** operators are not accepted because there is no real type translation. As a matter of fact, due to absence of the **Real** type, there is no need for explicit casting functions between these types. Following the same idea, since lists are not accepted, neither are the list prefix operators (like **len**, **hd**, **tail** and **inds**). **card** is not accepted because of the lack of support for recursion in SAL.

Comprehended expressions

Comprehended expressions are not accepted in the translator.

Initialise expressions

Initialize expressions are not accepted in the translator.

Assignment expressions

In the current version of the translator, assignment expressions are only allowed when describing transition systems (see subsection 3.6.1). In this context, assignment expressions are translated as SAL assignments. In the other possible environments inside the specification, assignment expressions are reported as errors (as there are no variables in the applicative style).

Channel expressions

Both Input and Output expressions are not accepted in the current version of the translator. For an explanation of the proposed translation mechanism to handle this construction in the extension covering

the imperative sequential and concurrent style, see subsections 4.4 and 4.4.

Local expressions

Local expressions are not accepted.

Let expressions

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

let h = 1 **in** h + 1 **end**

is translated as:

```
LET h : INTEGER = 1 IN h + 1;
```

From this simple example note that the type of the bounded name in the let definition must be explicitly stated in SAL's model. The translator will use the maximal type of the value expression in the binding for the type declaration.

This slightly more complex case (involving bindings to product types) is handled in the same way:

type
  Prod = **Int** × **Int**
value
  cons : **Int** × **Int** → Prod
  cons(a,b) ≡ **let** res = (a,b) **in** res
  end

In this case, the translation will include the type declaration (in the `SAL_TYPES` context) of:

```
Prod: TYPE = [IT_AN!Int_, IT_AN!Int_];
```

And the translation of the function as:

```
cons(a : IT_AN!Int_, b : IT_AN!Int_): SAL_TYPES!Prod =
    LET res : [IT_AN!Int_, IT_AN!Int_] = (a, b) IN res;
```

On the other hand, the translation of the remaining case of binding involving products is the most complex one. In this case, a problem arises 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's fields can be accessed by an index associated according the field's position inside the product). Under this approach, the declaration:

test : Prod → **Bool**
test(p) ≡ **let** (a,b) = p **in** a > 1 **end**

will be translated (assuming that the type *Prod* is the same than the one defined in previous examples in this subsection) into:

```
test(p : SAL_TYPES!Prod): BT_AN!Bool_ =
  LET LetId3_ : SAL_TYPES!Prod = p IN
      LetId3_.1 > 1;
```

If expressions

The if expression translation is also straightforward because SAL provides `IF-THEN-ELSE` and `ELSIF` constructions. There is, however, a syntactic limitation due to the `ELSE` non optionality. This means that in SAL, every `IF` must have an explicit `ELSE` branch declaration. RAISE, on the other hand, allows optional else branch declaration in the case of **Unit** type returning expressions. As this case is explicitly banned by the translator, the remaining if expressions in RAISE are directly translatable using SAL's `IF-THEN-ELSE` constructions.

Case expressions

As SAL does not provides a case construction, case expressions are translated as a nested sequence of if expressions. For example, the expression:

k :=
  case x **of**
    2 → 0,
    1 → 1,
    _ → 2
  end

translates to:

```
k = IF (x = 2)
       THEN 0
       ELSIF (x = 1) THEN 1
       ELSE 2
    ENDIF
```

The case when the pattern in the case expression is a product is handled in a field-by-field base. 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** × **Int** product. Then, the following specification:

case x **of**
  (2,1) → 0,
  _ → 2
end

Translates to:

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

Following this approach, when the wildcard pattern is used within a product pattern, the wildcarded field is simply removed from the condition in the if clause. According with this idea, the following code:

case x **of**
  (_,**true**) → 0,
  _ → 2
end

will translate into:

```
IF (x.2 = TRUE)
   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 *recognizers* (specially created predicates that return true only if their argument matches the proper constructor).

Under this approach, the declaration:

type
  Nonce == na | nb | nc,
  Agent == a | b | c,

```
  Pub_key == ka | kb | kc,
  Message ==
    m1(n1 : Nonce, a1 : Agent, k1 : Pub_key) |
    m2(n2a : Nonce, n2b : Nonce, a2 : Agent, k2 : Pub_key) |
    m3(n3 : Nonce, k3 : Pub_key)
value
key : Message → Pub_key
        key(m) ≡
          case m of
            m1(_, _, k) → k,
            m2(_, _, _, k) → k,
            m3(_, k) → k
          end,
```

will translate into the type declarations (in the context `SAL_TYPES`)

```
Nonce: TYPE = DATATYPE
  na,
  nb,
  nc
END;

Agent: TYPE = DATATYPE
  a,
  b,
  c
END;

Pub_key: TYPE = DATATYPE
  ka,
  kb,
  kc
END;

Message: TYPE = DATATYPE
  m1(n1: Nonce, a1: Agent, k1: Pub_key),
  m2(n2a: Nonce, n2b: Nonce, a2: Agent, k2: Pub_key),
  m3(n3: Nonce, k3: Pub_key)
END;
```

and into the function declaration:

```
key(m : SAL_TYPES!Message): SAL_TYPES!Pub_key =
    IF SAL_TYPES!m1?(m)
    THEN LET k : SAL_TYPES!Pub_key = SAL_TYPES!k1(m)
         IN k
    ELSE
      IF SAL_TYPES!m2?(m)
      THEN LET k : SAL_TYPES!Pub_key = SAL_TYPES!k2(m)
```

```
        IN k
    ELSE LET k : SAL_TYPES!Pub_key = SAL_TYPES!k3(m)
        IN k
    ENDIF
ENDIF;
```

Finally, list patterns are not accepted in the translator.

Iterative expressions

Expressions that introduce iterations (i.e. **while**, **until** and **for**) are not accepted. The reason for this restriction is that SAL provides, essentially, an applicative language for function description and this construction does not belong to this kind of language. Again, it would be possible to implement this constructions inside a module using an assembly-like construction (i.e. using the transition subsection with proper guards in order to implement some kind of program counter) but this approach is extremely inefficient and it does not allow the usage of this kind of construction combined with the applicative functions (as it would make it necessary to translate the whole specification in a module-like style).

3.2 Define before use rule

This is a syntactic restriction that simplifies many tasks during compilation and SAL endorses 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 sorting pass is based in the sorting algorithm used in the PVS translator [9]. 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 has 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 can not be sorted out and an error is reported.

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

3.3 Solving the circular dependencies among SAL contexts

In this case both, RAISE and SAL do not allow circular dependencies among modules. Moreover, both also use similar approaches to import functionalities defined in other modules. Following this reasoning, the rules concerning circular dependencies in SAL should be trivially satisfied by just preserving the specification's module structure during the translation process. As a matter of fact, this translation scheme is sufficient for the current version of the tool (model checking applicative style specifications ignoring confidence conditions).

On the other hand, the minimal set of functionalities that should be provided in order to make the specification's model checking meaningful include confidence condition verification and support for imperative style. Considering this reality, support should be present in the current version of the tool in order to allow an easy extension of it to meet this requirements. Taking this into account, support for a "lifted" type system (see section 4.1) together with a global state model (in order to encode variable and object declarations in SAL's applicative language) are needed and basic support for them was already included in the current version of the tool.

Including support for the confidence condition verification extension introduced many design changes in the current version of the tool (that would not have been necessary otherwise) but is worth the effort considering the need for this extensions in a short period of time. The most important problem introduced for the extension support inclusion is the possible violation of the non circular dependencies restriction. The problem is evident when considering the new global state in the imperative styles to cope with variable declarations (see section 4.4). In particular, the global state will need to reference the types involved in all variable declarations in the specification. On the other hand, the global state needs to be accessed by every function when using confidence condition verification (in order to signal confidence condition violation when necessary). Combining all this observations with the fact that all these individual declarations may exist in different context leads to a circular dependency inclusion among the modules (see figure 3.1 for a graphical explanation).

In this case, including the *sys* type definition in one of the modules will solve the conflict with that module in particular, but the circular dependency will remain existing with the rest of the modules. The approach that was devised to cope with this problem was to collect all type declarations in the specification (including the *sys* type) in only one SAL module placed almost at the top of the module hierarchy (the topmost places are occupied by contexts providing the `Int_` and `Bool_` declarations). An overview of this structure is showed in the figure 3.2.

In this way, all circular dependencies that could have been generated by the introduction of the new *sys* type are avoided by having all type declarations in the same file. Even though this approach solves the problem of circular dependencies in a simple way, it introduces another one. The problem arises when a type depends on some value for its definition (one example is the subtype case, where the restriction over the original type may involve some function application). The case is depicted in figure 3.3.

Again, the solution is to extend the original approach from moving only types to the `SAL_TYPES` context to move type declarations plus all values involved in those type declarations. In this way, all declarations (either values or types) necessary to define the type system of the specification are contained inside one individual file (`SAL_TYPES`) avoiding any kind of dependency and allowing the existence of the *sys* type.

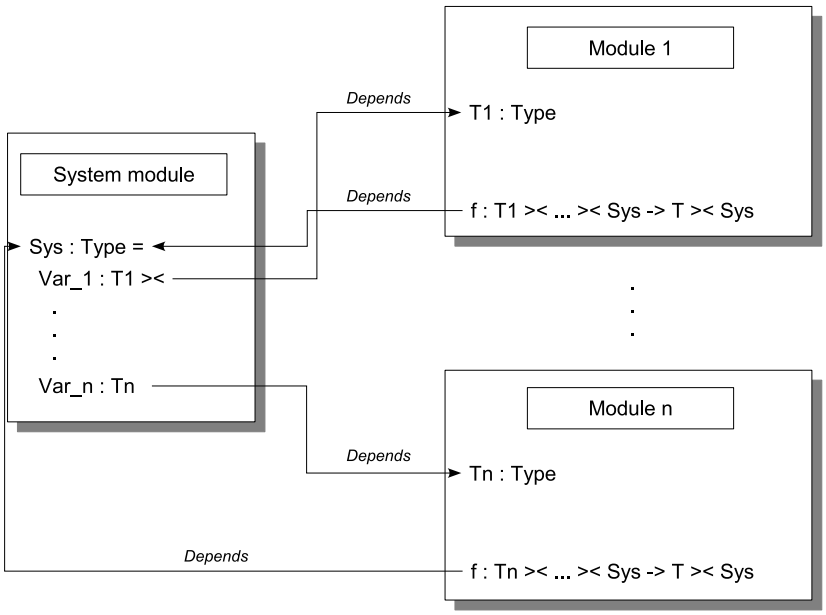

Figure 3.1: Circular dependency generation

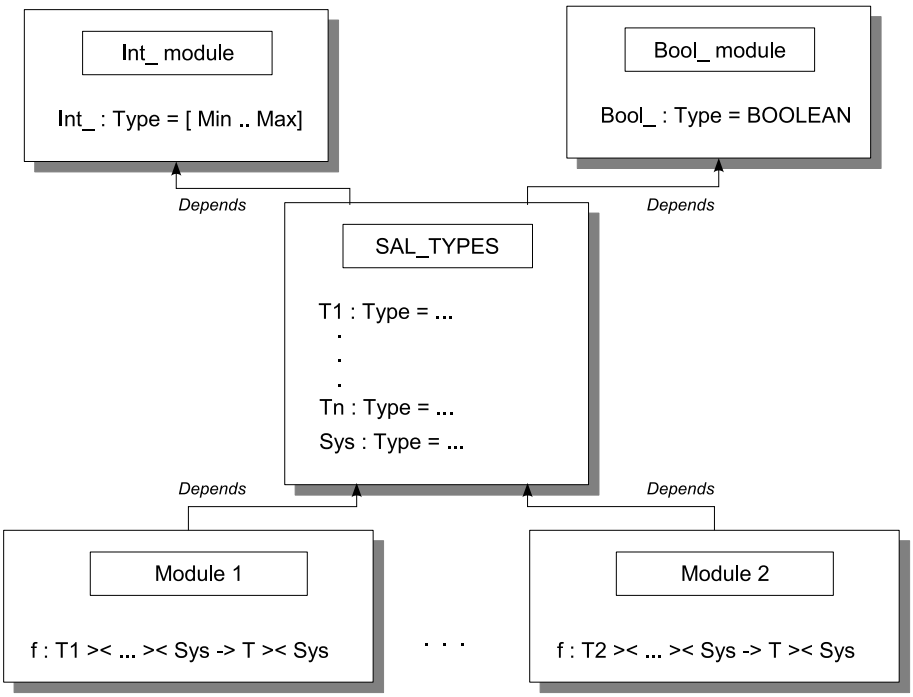

Figure 3.2: New structure induced by the translator

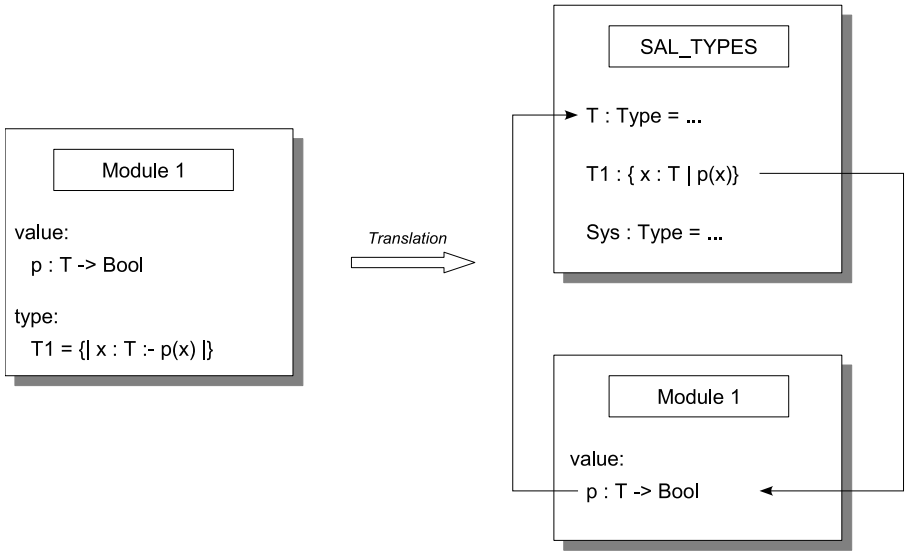

Figure 3.3: Circular module dependency induced by the new translation scheme

3.4 Bindings in functions' signature/arguments

Bindings in general are a very important feature of RAISE and allow from very simple associations (with simple variables) to complex product ones. Bindings typically occur inside let expressions (see section 3.1.5), case expressions (section 3.1.5) but also in functions. The latter can, in turn, be divided into two independent cases: bindings introduced in the formal parameters during the function declaration, like the names a and b in:

type:
Prod = **Int** × **Int**
value:
f : Prod → **Bool**
f(a,b) ≡ a < b

And the ones introduced by actual parameters during function application like in the code (using the definition of the type *Prod* and function *f* from the previous example):

value:
f1 : **Int** × **Int** → **Bool**
f1(a,b) ≡ f((a,b)),

f2 : Prod → **Bool**
f2(p) ≡ f1(p)

Bindings in this two particular cases are restricted to single or product bindings in RAISE. On the other hand, SAL does not allow any kind of bindings but the simple ones for functions.

The approach used to solve the problem depends on the situation and the rules are as follows:

- Bindings introduced in the function's formal declaration (i.e. unfolding one of the formal arguments like in the case of the function *f* in the first example). This case is solved with the same strategy used for product bindings used for let expressions. In particular, a new argument name is generated for the formal argument and product field's identifiers are assigned to the identifiers introduced in the bindings. This association (from names in the bindings to product fields) is pushed during the function's body processing and used for look up every time a name occurrence is found inside the function. Following this approach, the first example will be translated as:

  ```
  f(Function_id_n : SAL_TYPES!Prod) : BT_AN!Bool_ =
    Function_id_n.1 < Function_id_n.2;
  ```

- Bindings introduced during function application. This case can be treated as two separated problems. The first arises when the actual arguments match with the unfolding of the formal parameters of the applied function (like in the case of function *f1* in the second example). This case is the simplest one and can be solved by direct translation of the product expression (i.e. (a,b) in the example) as a product in SAL. Translated in this way, the product expression matches structurally with the product/abbreviated type in the argument.

  Using this rule, the translation of the function *f1* in the second example would be:

  ```
  f1(a: IT_AN!Int, b: IT_AN!Int) : BT_AN!Bool_ =
    f((a,b));
  ```

  On the other hand, the second case arises when the actual parameters must be unfolded to match the formal parameters (like in the case of function *f2* in the second example). In this case, the solution is slightly more complex because SAL does not allow this kind of binding (the applied function is expecting more parameters than the provided ones and this is a syntactic error for SAL type checker). The approach taken is to unfold on the fly the argument into its constituent fields in order to match the formal parameter list.

  In the case of the function *f2*, for example, the translator will generate:

  ```
  f2(p : SAL_TYPES!Prod) : BT_AN!Bool_ =
    f1(p.1, p.2);
  ```

3.5 The RSL Maximal-type approach

It is well known that proving type correctness of arbitrary expressions cannot be automatically decided, especially if the type theory includes subtyping functionalities. This is a major shortcoming and creates the need for type checking tools to avoid the problem in some way. In RAISE, the type relationship among expressions is simplified by the concept of *maximal types* which relies on the elimination of the restrictions in any subtype when considering type equivalences [18].

On the other hand, the SAL type system is slightly different. The approach used in all SAL tools is to distinguish among subtypes and maximal types only when working with `DATATYPES`. In this context, the kind of verification that is used is a syntactic one (instead of a structural one as with rest of the constructions). This particular way of type checking has the effect that code that involves variants (and

that is valid according to RAISE's type checker) might be translated into a construction that does not satisfy SAL's type rules.

For example, the following type and value declarations

```
type
   Database = Key ~m~> Data,
   Key = Int,
   Key1 = {| x : Int • x < Max |},
   Data ...

value
   Max : Int,

   init : Int → Database
   init(x) ≡ [ i ↦ map(i) | i : Key • i < x ],

   init_error : Int → Database
   init_error(x) ≡ [ i ↦ map(i) | i : Key1 • i < x ],
```

do satisfy the RAISE type checking rules. In particular, even though the expression *init_error* has type **Int** → Database that is not syntactically the type in its signature, the construction does still satisfy the RAISE type rules given the fact that, by the maximal type rule, Key1 is equivalent to Key when the restriction over its values is removed.

On the other hand, Key and Key1 do have a different syntactic declaration and are different types for SAL's type checker. In this case, if the translator generates the normal construction for the function *init_error* a type error will be reported when type checking the code with the SAL checker.

To solve this problem, a complete new type verification must be carried out over the already type checked code (but under the RAISE's maximal type relationship) in order to detect expressions where a sub-typed value was used in a place where a non-subtyped value was expected or vice versa. This approach is, as mentioned above, not completely decidable and might, in some cases, require the insertion of coercion constructions in the RAISE specification in order to disambiguate an expression where the proper type is not decidable.

Using the translator with the new type verification will report, then, a type error when processing the function *init_error*.

3.6 RSL extensions

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 this two required features in order to allow model checking of specifications.

3.6.1 Transition systems

It is well known that the model checking approach is based in 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 model in order to verify properties using model checking techniques.

Due to the abstract level in 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 to this purpose.

One simple alternative to solve this problem will be to not to allow transition system descriptions inside RAISE and force the specification writer to state the transition system after translation (i.e. by directly extending/modifying the SAL code). This approach, however simple to implement, has the disadvantage of forcing the RAISE user to write SAL code (which will require SAL syntax understanding).

Based in this observation is that an extension to RAISE language was devised in order to allow transition system description inside RAISE's specification code. This approach has several advantages, as described below:

- It is possible to modify the specification code and use the translator as many times as needed. As the transition system description is part of the code, it will be also re-generated every time. On the other hand, all modifications made directly over SAL's code will be erased on every new execution of the translator (when generating the translated code).
- The named entities in the RAISE specification can be addressed with their own names inside the RAISE code. This avoids the problem of dealing with the possible name transformation or declaration relocation that happens during the translation process.
- SAL code and execution remains totally hidden behind the RAISE tools interface. This avoids the need to know or even understand the SAL code but still providing the model checking features available in SAL.

The grammar of the RAISE extension that allows transition system descriptions is described below.

Transition_system_decl ::“transition_system” “[” id “]”{Base_module}$^{+}_{,}$

Base_module ::= [“in” variable_decl “end”]
“local” variable_decl “[” Transition_declarations “]”

Transition_declarations ::= {Transition_decl}$^{+}_{[=]}$

Transition_declaration ::= Single_guarded_command
| Multiple_guarded_command

Single_guarded_command ::= [“[” id “]”] *logical*−value_expr “==>” Update_exprs
| [“[” id “]”] “else” “==>” Update_exprs

Multiple_guarded_command ::= "([=]" variable_decl • Single_guarded_command ")"

Update_exprs ::= {id "=" value_expr}$^{+}_{,}$

It is important to note that:

- Variables declared in the **in** subsection are considered as inputs to the transition system. In particular, the value for these variables will be assigned by the model checker and no initialisation nor assumption can be made about their value.
- Variables declared as **local** comprise the actual state of the transition system. Initialisation of this variables is not compulsory but it is important to note that the model checker can otherwise initialise them to any value. This last situation may lead to spurious results (i.e. the model checking assigning a value that, from initialisation, violates some of the properties, making them false). Considering this, the RAISE extension for transition systems imposes the existence of an initial value for every local variable.

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.

3.6.2 Temporal logic and temporal properties

The model checking technique is based on *temporal logic*. The idea of temporal logic is that a formula is not *statically* true or false in a model, as it is in propositional and predicate logic. Instead, the models of temporal logic contain several states and a formula can be true in some states and false in others.

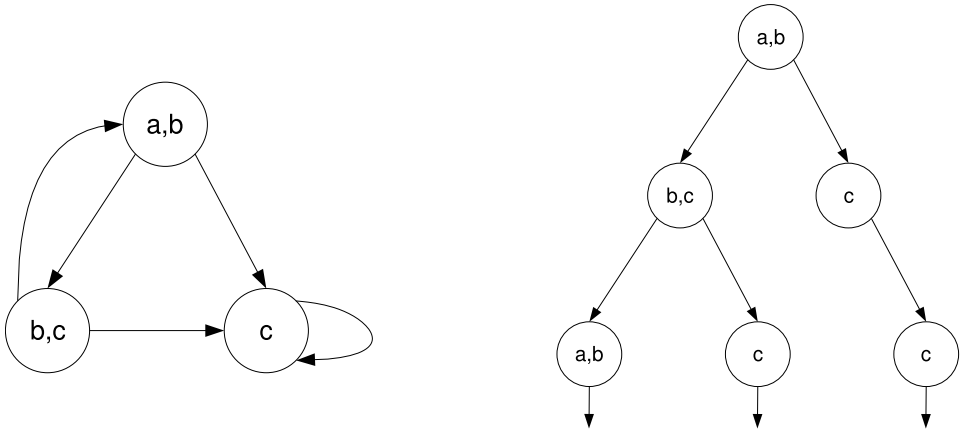

Figure 3.4: Computation trees

Thus, the static notion of truth is replaced by a *dynamic* one, in which the formulas may change their truth values as the system evolves from state to state. As mentioned in the previous subsection, in model checking, the models $\mathcal{M}$ are *transition systems* and, the properties ϕ to be verified are formulas in temporal logic.

In this context, the *Computation Tree Logic* (CTL*) is the most expressive temporal logic and, conceptually, it is used to describe properties of *computation trees*. As exemplified in figure 3.4, the tree is formed by starting from a given initial state from some kind of transition system and then unwinding the structure into an infinite tree following the system possible ways of evolution.

By definition[3], CTL* formulas are composed of *path quantifiers* and *temporal operators*. The path quantifiers are used to describe the branching structure in the computational tree. There are two such quantifiers `A` (for "for all computational paths") and `E` ("for some computational path"). These quantifiers are used in a particular state to specify that all of the paths or some of the paths starting at that state have some property. The temporal operators describe properties of a path through the tree. There are five basic operators:

- `X` ("next time") requires that a property holds in the second state of the path.
- `F` ("eventually" or "in the future") operator is used to assert that a property holds at some state on the path.
- `G` ("always" or "globally") specifies that a property holds at every state on the path.
- `U` ("until") is used to combine two properties. It holds if there is a state on the path where the second property holds, and at every preceding state on the path, the first property holds.
- `R` ("release") is the logical dual of the `U` operator. It requires that the second property holds along the path up to and including the first state where the first property holds. However, the first property is not required to hold eventually.

[3]Following the definition by Clarke *et al.* in [13], pages 28 to 30.

There are two types of formulas in CTL*: *state formulas* (which are true in a specific state) and *path formulas* (which are true along a specific path). Let AP be the set of atomic proposition names. The syntax of state formulas is given by the following rules:

- If $p \in AP$, then p is a state formula.
- If f and g are state formulas, then $\neg f$, $f \wedge g$ and $f \vee g$ are state formulas.
- If f is a path formula, then $\mathtt{E}f$ and $\mathtt{A}f$ are state formulas.

In order to specify the syntax of path formulas, two more rules are needed:

- If f is a state formula, then f is also a path formula.
- If f and g are path formulas, then $\neg f$, $f \wedge g$, $f \vee g$, $\mathtt{X}\ f$, $\mathtt{F}\ f$, $\mathtt{G}\ f$, $f\ \mathtt{U}\ g$ and $f\ \mathtt{R}\ g$ are path formulas.

The definition of the semantics of CTL* is generally associated with a *Kripke structure* [13, 30]. A Kripke structure M is a triple $\langle S, R, L\rangle$, where S is the set of states; $R \subseteq S \times S$ is the transition relation that must be total (i.e., for all states $s \in S$ there exists a state $s' \in S$ such that $(s, s') \in R$); and $L : S \longrightarrow 2^{AP}$ is a function that labels each state with a set of atomic propositions that are true in that state. A *path in* M is an infinite sequence of states, $\pi = s_0, s_1, ...$ such that for every $i \geq 0, (s_i, s_{i+1}) \in R$.

If f is a state formula, the notation $M, s \models f$ means that f holds at state s in the Kripke structure M. Similarly, if f is a path formula, $M, \pi \models f$ means that f holds along the path π in the Kripke structure M. The relation $\models$ is defined inductively as follows[4] (assuming that f_1 and f_2 are state formulas and that g_1 and g_2 are path formulas):

$$
\begin{array}{lcl}
M, s \models p & \iff & p \in L(s) \\
M, s \models \neg f_1 & \iff & M, s \not\models f_1 \\
M, s \models f_1 \vee f_2 & \iff & M, s \models f_1 \text{ or } M, s \models f_2 \\
M, s \models f_1 \wedge f_2 & \iff & M, s \models f_1 \text{ and } M, s \models f_2 \\
M, s \models \mathtt{E}\ g_1 & \iff & \text{there is a path } \pi \text{ from } s \text{ such that } M, \pi \models g_1 \\
M, s \models \mathtt{A}\ g_1 & \iff & \text{for every path } \pi \text{ starting from } s,\ M, \pi \models g_1 \\
M, \pi \models f_1 & \iff & s \text{ is the first state of } \pi \text{ and } M, s \models f_1 \\
M, \pi \models \neg g_1 & \iff & M, \pi \not\models g_1 \\
M, \pi \models g_1 \vee g_2 & \iff & M, \pi \models g_1 \text{ or } M, \pi \models g_2 \\
M, \pi \models g_1 \wedge g_2 & \iff & M, \pi \models g_1 \text{ and } M, \pi \models g_2 \\
M, \pi \models \mathtt{X}\ g_1 & \iff & M, \pi^1 \models g_1 \\
M, \pi \models \mathtt{F}\ g_1 & \iff & \text{there exists a } k \geq 0 \text{ such that } M, \pi^k \models g_1 \\
M, \pi \models \mathtt{G}\ g_1 & \iff & \text{for all } i \geq 0, M, \pi^k \models g_1 \\
M, \pi \models g_1\ \mathtt{U}\ g_2 & \iff & \text{there exists a } k \geq 0 \text{ such that } M, \pi^k \models g_2 \text{ and} \\
 & & \text{for all } 0 \leq j < k, M, \pi^j \models g_1 \\
M, \pi \models g_1\ \mathtt{R}\ g_2 & \iff & \text{for all } j \geq 0 \text{ if for every } i \leq j\ M, \pi^i \not\models g_1 \text{ then} \\
 & & M, \pi^j \models g_2
\end{array}
$$

[4] Note that π^i is used to denote the suffix of π starting at s_i.

Expressively of CTL*

When a given property concerns the execution tree of a given automaton (or, more generally speaking, transition system), CTL* is an expressive enough logic to describe the most relevant properties for the model[5]. Under this assumption, theoreticians have obtained many theorems uncovering the fundamental aspects regarding the expressiveness of CTL*. The two which seem to be of particular interest in this context are:

- Any property of the form "as seen from the outside world, the transition system $\mathcal{A}$ being studied behaves like the reference transition system $\mathcal{B}$" can be expressed in CTL*. Concretely, for any $\mathcal{B}$, a CTL* formula $\phi_{\mathcal{B}}$ can be produced to mean "to be like $\mathcal{B}$", that is, such that for every $\mathcal{A}, \mathcal{A} \models \phi_{\mathcal{B}}$ if and only if $\mathcal{A}$ and $\mathcal{B}$ are indistinguishable as seen from the outside [5]
- The CTL* combinators are sufficiently expressive. A theorem due to Kamp [24] shows that any new temporal combinator whose semantics can be expressed as a first order logic having $\leq$ as the only predicate can be defined as an expression based on `X` and `U`. For example, a `Z` combinator defined by $\theta \models \phi$ `Z` ψ if and only if between each pair of states satisfying ϕ it is possible to find a state (strictly in between) satisfying ψ can be expressed using `U` and `X` as[6]:

$$\phi \texttt{ Z } \psi \equiv \texttt{G}(\phi \Rightarrow \texttt{X}(\neg\phi \texttt{ W}(\psi \vee \neg\phi)))$$

CTL* sublogics

In this subsection two useful sublogics of CTL* are considered: *branching-time logic* and *linear-time logic*. The distinction between them is in how they handle branching in the underlying computation tree. In branching-time temporal logic temporal operators quantify over the paths that are possible from a given state. In linear-time temporal logic, operators are provided for describing events along a single computation path.

Computation Tree Logic (CTL) is a branching-time logic and it is a restricted subset if CTL* in which each of the temporal operators `X`, `F`, `G`, `U` and `R` must be immediately preceded by a path quantifier. More precisely, CTL is a subset of CTL* that is obtained by restricting the syntax of path formulas using the following rule:

- If f and g are state formulas, then `X` f, `F` f, `G` f, f `U` g and f `R` g are path formulas.

Linear Temporal Logic (LTL), on the other hand, is a linear time logic and consists of formulas that have the form `A` f, where f is a path formula in which the only state subformulas permitted are atomic propositions. More precisely, an LTL path formula is either:

[5]It is important to note here that no logic can express anything not taken into account by the modelling decisions made, for example it will be impossible to state properties about time constraints if the model used to describe a system consider atomic transitions with no *duration* associated to them.

[6]The `W` operator is used for simplicity of the whole expression. It can be defined as a *weak until* operator, i.e. ϕ_1 `W` ϕ_2 still expresses "ϕ_1 `W` ϕ_2", but without the inexorable occurrence of ϕ_2 (and if phi_2 never occurs, then ϕ_1 remains true until the end). It can be shown that the following equivalence holds:

$$\phi_1 \texttt{ W } \phi_2 \equiv (\phi_1 \texttt{ U } \phi_2) \vee \texttt{G } \phi_1$$

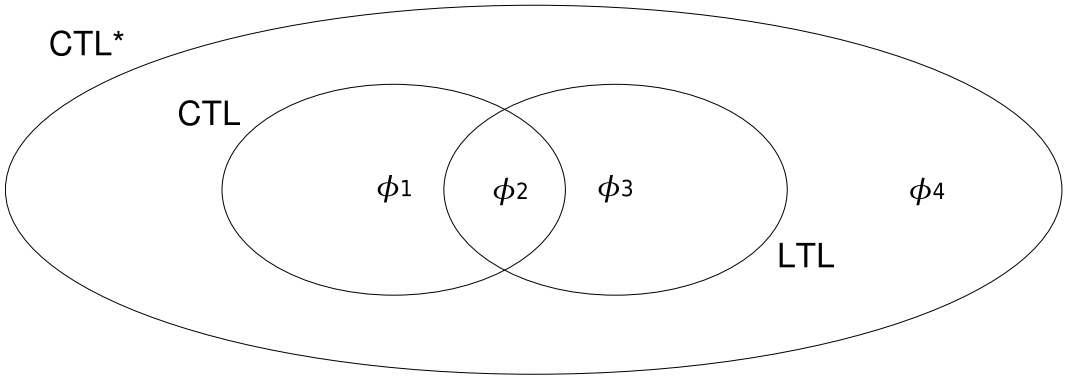

Figure 3.5: The expressive power of CTL, LTL and CTL*

- If $p \in AP$, then p is a path formula.
- if f and g are path formulas, then $\neg f$, $f \vee g$, $f \wedge g$, $\mathtt{X}\ f$, $\mathtt{F}\ f$, $\mathtt{G}\ f$, $f\ \mathtt{U}\ g$ and $f\ \mathtt{R}\ g$ are path formulas.

As LTL formulas are evaluated over paths, a state of a system is defined to satisfy an LTL formula if *all paths* from the given state satisfy it. Thus, LTL implicitly quantifies universally over paths. Therefore, properties which assert the existence of a path cannot be expressed in LTL. This problem can be partly alleviated by considering the negation of the property in question, and interpreting the result accordingly. To check whether there exists a path from s satisfying the LTL formula ϕ, one can check whether all paths satisfy $\neg\phi$; a positive answer to this is a negative answer to the original question and vice versa. However useful in many cases, this approach is not powerful enough to allow the expression of properties which mix universal and existential path quantifiers.

On the other hand, CTL allows explicit quantification over paths, and, in this respect, it is more expressive that LTL. However, it does not allow one to select a range of paths by describing them with a formula, as LTL does. In that respect, LTL is more expressive than CTL.

Moreover, both CTL and LTL can be considered to be subsets of CTL*. On one hand, even though LTL's syntax does not include `A` and `E`, the semantic viewpoint of LTL is that the formulas are considered over all possible paths. Therefore, the LTL formula ϕ is equivalent to the CTL* formula $\mathtt{A}\ \phi$. Thus, LTL can be viewed as a subset of CTL*. CTL, on the other hand, is also a subset of CTL* since it is the fragment of CTL* in which formulas are restricted to $\phi ::= (\psi\ \mathtt{U}\ \psi) \mid (\mathtt{G}\ \psi) \mid (\mathtt{F}\ \psi) \mid (\mathtt{X}\ \psi)$.

Figure 3.5 shows the relationship among the expressive powers of CTL, LTL and CTL*.

There are model checking algorithms for both CTL and LTL. The model checking algorithm for CTL was initially developed by Queille, Sifakis, Clarke, Emerson and Sistla [34, 6]. The algorithm runs in time linear to its component (the automaton on the one hand, and the CTL formula on the other). The algorithm relies on the fact that CTL can only express state formulas and its fundamental component is a procedure that marks each state of an automaton $\mathcal{A}$ with the formulas and subformulas that are valid on that state.

LTL's model checking algorithm, on the other hand, is essentially due to Lichtenstein, Pnueli, Vardi and Wolper [28, 42]. In the case of LTL, the problem does not deal with state formulas and it is no longer possible to rely on marking the automaton states. The formulas of LTL are *path formulas* and a finite automaton will generally give raise to infinitely many different executions, themselves often infinite in

length. In this context, the viewpoint adopted will be language theory. In practise, the algorithm is based on the idea of constructing, for every property ϕ referring to an automaton $\mathcal{A}$, an automaton $\mathcal{B}_{\neg\phi}$ that recognises precisely the executions which do not satisfy ϕ. Then, a product automaton $\mathcal{A} \otimes \mathcal{B}_{\neg\phi}$ (made by the synchronising $\mathcal{A}$ and $\mathcal{B}_{\neg\phi}$) is constructed, whose sole behaviours are the behaviours of $\mathcal{A}$ accepted by $\mathcal{B}_{\neg\phi}$, in other words, the executions of $\mathcal{A}$ which do not satisfy ϕ.

Temporal logic in RAISE

RSL provides a natural way of stating properties about the specifications: the **axiom** construct. However powerful, this construct does not allow the usage of any temporal operator given the fact that it was meant to express invariant properties of the system being described.

On the other hand, as mentioned in subsection 3.1.1, no axiom declaration is allowed in the translator given the fact that SAL language does not support this kind of construct. There is, however, a built in way of stating properties of a model within SAL and it is by means of the usage of a `THEOREM` definition. 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. In fact, using a model checker to verify invariants is a very useful way to gain confidence/certainty about the system satisfying them but, at the same time, it is not making full usage of the expressive power of the temporal logic embedded in the model checking paradigm. 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.

The extension, then, is meant to add temporal logic operators to RSL and to allow the specification writer to state properties about a given transition system. The temporal logic chosen for the extension is LTL (given the fact that it is the only CTL* sublogic supported by the SAL toolset) and a minimum set of operators was devised (in order to keep the amount of reserved keywords to a minimum).

The following grammar rules describe the allowed syntactic constructions for the properties statement:

LTL_Property_decl ::= "ltl_assertion" $\{\text{LTL\_assertion}\}^{+}_{,}$

LTL_assertion ::= "["id"]" id "|–" LTL_expr

LTL_expr ::=
  logical−value_expr

In addition, the LTL temporal operators "G", "F", and "X" are allowed in LTL_exprs as function symbols.

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 complete example

This subsection is meant to show a real example of how the new constructs incorporate into a specification and how temporal properties can be stated about an existing system.

```
scheme TOKENS =
  class
    type
      Token == a | b | c | d | e | f,
      TSet = Token-set,
      State ::
        S1 : Token-set ↔ re_S1
        S2 : Token-set ↔ re_S2

    value
      init : State =
        mk_State({a,b,c}, {d,e,f}),

      give_to_1 : Token × State ~→ State
      give_to_1(t, s) ≡
        re_S1({t} ∪ S1(s), re_S2(S2(s) \ {t}, s))
      pre t ∈ S2(s),

      give_to_2 : Token × State ~→ State
      give_to_2(t, s) ≡
        re_S2({t} ∪ S2(s), re_S1(S1(s) \ {t}, s))
      pre t ∈ S1(s)

    transition_system
      [ sys ]
        local
          state : State := init
        in
            ( [] tok : Token •
```

```
                [ give_to_S1 ]
                    tok ∈ S2(state)  ==>
                        state′ := give_to_1(tok, state)
            )
        □
            ([] tok : Token •
                [ give_to_S2 ]
                    tok ∈ S1(state) ==>
                        state′ := give_to_2(tok, state)
            )
    end

  ltl_assertion
    [ consistent ] sys ⊢ G(S1(state) ∩ S2(state) = {}),
    [ no_loss ] sys ⊢ G(S1(state) ∪ S2(state) =  {a, b, c, d, e, f}),
    [ empty_S1_reachable ] sys ⊢ G(S1(state) ≠ {}),
    [ empty_S2_reachable ] sys ⊢ G(S2(state) ≠ {})
end
```

The example shows a very simple example where the interchange of tokens among two repositories is modelled. The repositories are modelled as sets and the operations for token interchange are also shown. Then, one of the new extensions to RAISE is used in order to describe the transition system that models how the top level functions from the specification are used to describe the evolution of the system under study. In particular, note the usage of the preconditions of each function as guards for its execution, warrantying correct evolution of the system.

Also note the comprehended multiple guarded transitions (by means of the creation of a quantified variable over the type Token in the example). This construction allows the definition of transition systems that are expanded at run time depending on the types in the specification.

Finally, four properties are asserted about the system, one that states that there is no token replication due to the interchange (i.e. the system remains consistent), and the second one expresses that no token will be lost due to the interchange of tokens. These are typical safety properties.

The last two properties are typical liveness properties. We cannot guarantee that either set will become empty (since, for example, the token "a" might be exchanged to and fro for ever), but we do expect that it is *possible* for each of the sets to become empty. We show this be asserting the *opposite*, that S1 for example must always be non-empty. When we run SAL we intend that it will find a counter example, thus demonstrating the liveness property that S1 can become empty.

3.7 Limitations

3.7.1 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). Based in these facts, it is possible to assert that recursion is a key feature in RAISE if recursive functions or data structures are going to be used.

On the other hand, these features are not provided in SAL and, together with the lack of support for iterative constructions, result in a important limitation to the portion of the space of valid RAISE specifications that can be model checked. This section explores the reasons for the exclusion of recursive declarations both in SAL and, hence, in the translator.

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 realize 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. In particular, the symbolic model checking theory relies in the representation of the system as strings of binary (boolean in the general case) values and it is impossible to find a codification for the state of the system if one of the types that comprises it has an unknown/undefined length.

As SAL is based in the symbolic model checking theory, no recursive types are allowed and this limitation logically propagates to the translator from RAISE to SAL.

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 can not be overcame by any existing tool.

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 [11], the SAL's type checker was supposed to be able to automatically figure out 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.

3.7.2 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 unknown 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 check-

ing transition system is not conceivable if it is taken into account that undefined behaviour would be introduced in the system under study's evolution.

Chapter 4

Extensions to the tool

4.1 Model checking confidence conditions

As mentioned before, the first version of the tool is meant to verify certain properties of the specification under the assumption of confidence condition satisfaction along the whole evolution of the model. Under this assumption, all partial functions can be considered as total and type well-formedness can be taken for granted.

Besides the fact that this assumption allows a more optimized analysis of the system, it is a very strong hypothesis and, in general, it can not be taken for granted. As a matter of fact, it would be very important to be able to use the automatic verification power of model checking to first certify the satisfaction of confidence conditions (in a suitable model of the system) in order to then verify the desired property (in the model generated by the current version of the tool). The tool therefore generates three translations to SAL:

1. The normal translation described in the previous chapter.
2. A "CC" version that checks confidence conditions.
3. A "simple CC" version that also checks confidence conditions but gives less diagnostic information. It is provided in case the CC version is too large to run, since it is smaller.

This section presents the major sections of the grammar where the designed extensions to the current version of the tool must be applied in order to provide confidence condition verification.

Extended type system

The main problem faced when model checking confidence conditions is what action to take when some of the confidence conditions for some value expression are not met. In particular, what value should be

returned when the precondition of a function is not satisfied or what action to take when one of the arguments in a function application is outside of the expected domain.

The solution proposed to this problem is to use a deeper embedding of type system in order to get a better partial function translation. As a matter of fact, a slightly more complex translation mechanism for the types was devised, by extending the SAL basic type system to a "lifted" new one (by means of SAL basic types and the `DATATYPE` constructor).

In particular, this new type system includes a special datatype, called `Not_a_value_type`, in all SAL types (and also in the possible new, user-defined ones). This extended type system allows a very simple partial function translation (see the function section in 4.1) and also provides support for explicit subtype and precondition verification to be performed by the model checking tools. The main drawback of this approach is that all built-in SAL infix operators can not be directly used in the model. Instead of that, a new set of prefix functions must be used. This new set of operators provides the basic operations on each type, with the proper extensions to handle values of type `Not_a_value_type`. We shall henceforth call such values "navs".

For example, given the following RSL declarations in a module M

type
  PersonId = **Nat**
value
  me : PersonId = 0

we get the following SAL declarations in the CC version:

```
Not_a_value_type: TYPE = DATATYPE
  Value_M_me_not_in_subtype,
  ...;

Int__cc: TYPE = DATATYPE
  Int__cc(Int__val: Int_),
  Int__nav(Int__nav_val: Not_a_value_type)
END;

me : Int__cc =
  LET RSL_res_ : Int__cc = Int__cc(0) IN
  IF is_true(RSL_is_Nat(RSL_res_))
  THEN RSL_res_
  ELSE Int__nav(Value_M_me_not_in_subtype)
  ENDIF;
```

We see that

- Occurrences of the type PersonId are translated as the maximal type of **Nat**, ie to the (lifted) type `Int__cc`.
- The lifted type `Int__cc` is constructed by creating a datatype consisting of integer values or navs.

- Constants like "me" are defined as lifted values of their maximal type. If the defined value is not in the relevant subtype then they are bound to a nav.
- `Not_a_value_type` is defined as a datatype which is really an enumeration of identifiers that indicate where the nav was generated.

Partial functions

As mentioned before, partial functions were translated as total functions in the basic version of the tool under the assumption that the precondition that made them partial was satisfied. This was done in this way because SAL does not provide a partial function construction and thus, it was not clear which value a partial function must return when invoked with arguments outside its domain. In the context of the extension, this limitation is bypassed, however, with the the incorporation of lifted types. With this approach, code can be added at the beginning of the function's body (i.e. the SAL conditional statement `IF - THEN - ELSE`) in order to verify the precondition satisfaction. In this way, RAISE's partial functions are embedded within SAL total functions For example, the RAISE partial function (in a module M, say)

value
diff : **Nat** × **Nat** $\xrightarrow{\sim}$ **Nat**
diff(x, y) ≡ x − y
pre x ≥ y

is translated to something like:

```
diff(x: Int__cc, y: Int__cc) : Int__cc =
  IF Int__nav?(x) THEN x
  ELSE
    IF Int__nav?(y) THEN y
    ELSE
      IF is_true(cc_and(RSL_is_Nat(x), RSL_is_Nat(y)))
      THEN
        IF is_true(cc_ge(x, y))
        THEN LET RSL_res_ : Int__cc = cc_sub(x,y) IN
          IF is_true(RSL_is_Nat(RSL_res_)) THEN RSL_res_
          ELSE Int__nav(Result_of_function_M_diff_not_in_subtype)
          ENDIF
        ELSE Int__nav(Precondition_of_function_M_diff_not_satisfied)
        ENDIF
      ELSE Int__nav(Argument_of_function_M_diff_not_in_subtype)
      ENDIF
    ENDIF
  ENDIF;
```

Notice from the example that:

- All functions take values of lifted types as arguments and return values of lifted types.

- All functions are made strict: if any argument is a nav then the nav is returned. This ensures that a nav, once generated, is not changed by having other functions applied to it. So when the user sees a nav reported it will be the first nav generated.
- A nav is returned if any argument is not in its subtype, or if the precondition is violated, or if the result of the function is not in its subtype.
- Functions like `RSL_is_Nat` are used to check that their arguments are in subtypes. For any RSL type T that is defined as a subtype there will be a corresponding function `RSL_is_T`.
- The function `is_true()` converts lifted boolean values into SAL `BOOLEAN` ones (this function returns `TRUE` only if its argument is a lifted `TRUE`).
- Functions like `cc_ge` are strict versions of the corresponding operators such as `>=`

Type expressions

As mentioned in section 3.1.1, all RAISE basic types are lifted by the translator in order to handle the non-determined return value of partial functions (when the precondition is not met).

Type literals

The general rule for type literal construction is the incorporation of the `Not_a_value_type` type and then, the declaration of a constructor (with the suffix `_cc`) and an accessor (with the suffix `_val`).

Transition systems

The transition systems in the CC and simple CC versions have extra guards added to the transitions that there is no nav in any local variable. This is done merely to reduce the size of the model.

4.2 Detecting confidence condition violations

To check for confidence condition violation a single LTL assertion is generated for each transition system which asserts that all the local variables in the transition system are not navs. If this is not true then SAL will produce a trace showing how the nav was generated. Since all functions are strict this will, as noted above, be the first nav generated.

This is not a perfect solution since, for example, if the nav occurs in the definition of of a constant “me”, for example, but the value “me” does not contribute to any local variable, no failure will be recorded. So what we can say if the confidence condition check succeeds is that no confidence condition violation occurs in the evolution of the transition system

4.3 The simple CC version

The simple CC version is essentially the same as the CC version except that `Not_a_value_type` is reduced to the single constant `nav`. This reduces the sizes of all the lifted types, and hence the size of the model, but provides less diagnostic information: the trace showing how the nav was generated will still be produced, but instead of, for example, showing the nav `Precondition_of_function_M_diff_not_satisfied` it will just show the `nav`.

4.4 Imperative and concurrent style

Together, the first version of the tool and the confidence condition verifier, span most of the constructions that can be model checked under the applicative style. On the other hand, RAISE provides a whole set of imperative (both sequential and concurrent) constructions in order to allow the specifications to became closer to the actual implementation languages. Providing support for imperative-style specifications model checking is a future extension of the current tool. This section explores how the key imperative constructions in RAISE can be encoded in SAL's applicative language.

Object declarations

Object declarations are not an exclusive feature in the imperative style (they are also used in the applicative style), but the possibility of including variables in objects in the imperative paradigm makes the initial approach for objects insufficient. In this new context, the translation of a global object declaration depends, mainly, on its semantics. In particular, the selection of the translation technique is based in the presence (or absence) of explicit internal state representation inside the object:

- The object does not have an internal state (i.e. it does not contain any variable declaration). This is essentially the same case faced in the applicative style, where object declarations where acting an exporting mechanism for types and functions. The approach used in the applicative case can, then, be used in this case.
- The global object does contain state information. At this point it is important to distinguish between the different imperative specification styles (it is impossible for this situation to arise in the applicative one). The reason for the separation of the concurrent specification style from the sequential one is based on the substantial differences in the models that the translator will generate from them:
  - Imperative sequential specifications: In this case, the global object is translated as a record with the same name (no renaming is necessary due to the type correctness assumption). This new record is included in the explicit global state variable generated for the model.
  - Concurrent specifications: For this style, a new module (with all the functionality of its class) will be generated for each global object instantiation. This way of translation is necessary to enable the interaction with channels translation (see 4.4), in order to preserve the channel semantics.

    Besides the rapid model size growth when using this approach (the number of variables is significantly increased by the way in which functions inside objects are handled), this translation

policy also incurs function replication due to SAL semantic restrictions. In particular, SAL rules forbid the declaration of components inside the model that are supposed to be accessed by multiple modules. To be more precise, the whole model could deadlock if multiple sources attempt to use the same communication mechanism to modify the internal state of another module. However syntactically allowed (an example of this situation occurs when a function within an object is invoked from more than one external entity), this kind of situation is not supposed to arise according to the RAISE Development Method [19](modules containing variables are only allowed if they are located in the leaves of the dependency tree), so specifications with this characteristic are not translated.

For the case of object array declaration, it is translated to a SAL `ARRAY` declaration for the cases of the imperative sequential specification styles.

In the case of the concurrent style, the declaration of an array of objects is translated to an array of `INPUT` and/or `OUTPUT` declarations and the creation of a new module that (implicitly) produces the replication of the global object (by means of synchronous composition) and the mapping between the declarations inside each individual object and the array (using the `RENAME` and `IN` statements).

Note that for none of the possible specification styles a multidimensional array declaration is allowed (i.e only uni-dimensional arrays are translatable to SAL).

For example, the RAISE specification

```
scheme C_DOOR1 = hide CH, door_var, door in class
   object CH : class
            channel
               open, close, open_ack, close_ack : Unit,
               door_state : T.Door_state
         end
   variable door_var : T.Door_state
   value
      door : Unit → in any out any write any Unit
      door() ≡
         while true do
            CH.open? ; CH.open_ack ! () ; door_var := T.open
            []
            CH.close? ; CH.close_ack ! () ; door_var := T.shut
            .
            .
            .
         end
end
```

is translated to the SAL model (assuming that the Unit type was implemented in a context imported to this environment under the "unitType" name):

```
door : MODULE =
```

```
BEGIN
INPUT
        open_start : unitType!Unit,
        close_start : unitType!Unit
        ...
OUTPUT
        open_finished : unitType!Unit,
        close_finished : unitType!Unit,
        ...
INITIALIZATION
        open_finished = rsl!nil;
        close_finished = rsl!nil;
        ...
TRANSITION
[
        open_start:
        open_start = unitType!set AND
        open_finished = unitType!nil -->
                door_state' = T!open;
                open_finished' = unitType!set;
[=]
        open_finished:
        open_start = unitType!nil AND
        open_finished = unitType!set -->
                open_finished' = unitType!nil;
[=]
        ...
]
END;
```

Note that a complete treatment of channel translation can be found in 4.4

Then,

scheme C_DOORS1 = **hide** DS **in class**
  object DS[f : T.Floor] : C_DOOR1
  value
  .
  .
  .
end

translates to (assuming that there exists a context that provides the implementation of the Floor type and that it was imported with the name T):

```
c_doors : MODULE =
WITH
```

```
INPUT
        DS_open_start : ARRAY T!Floor OF unitType!Unit,
        DS_close_start : ARRAY T!Floor OF unitType!Unit;
OUTPUT
        DS_open_finished : ARRAY T!Floor OF unitType!Unit,
        DS_close_finished : ARRAY T!Floor OF unitType!Unit,
        .
        .
        .
(|| (f : T!Floor) :
    (RENAME open_start to DS_open_start[f],
            open_finished to DS_open_finished[f],
            close_start to DS_close_start[f],
            close_finished to DS_close_finished[f],
            .
            .
            .
        IN door));
```

Note from the last example that the replication of the door module is actually produced when the synchronous composition is used (by means of the || operator). In particular, each individual `INPUT` or `OUTPUT` is mapped (through the renaming procedure) to one particular position within the `DS_<operation>_<event>` array. The new module (c_doors) is a the (synchronous) parallel composition of multiple "doors" module instances (the amount is determined, in this case, by the T!Floor set cardinality).

Variable declarations

For the variable declarations translation, the procedure, again, depends on the style of the specification. For the imperative sequential style, the new variable is added to the global state record. Due to this enlargement in the scope (and the obvious need for name uniqueness), the variable must be renamed. The new identifier for the variable will be composed by its actual position into the specification code concatenated with the name. A more detailed discussion about the renaming procedure can be found in section 4.4.

On the other hand, a variable declaration in the concurrent style will be translated to a proper declaration inside the module that represents the translation of the constituent class. In particular, the rules for translation are:

- Hidden variables are translated as SAL `LOCAL` declarations. In this style, variable initializations are collected and incorporated to a global initial state declaration (initialization, is, as a matter of fact, an advisable component to the soundness of the model checking results).
- Non hidden variables are translated to be `OUTPUT` declarations in the model. This allows external reading from many modules but forbids external (i.e. from another scheme) **write** operations. This policy seems to be too restrictive, but it forces external modification of variables through the class interface (i.e. functions) which is a good programming practice. Finally, initializations are gathered and translated to a `INITIALIZATION` section inside the module that contains the variables.

Channel declarations

Channel declarations will be translated as pairs of variables. Each variable in the pair is used, respectively, to model the reader's and the writer's attempt to communicate over the channel. Thus variables are used as a communication mechanism to instrument the channel synchronization semantics. In particular, each module involved in the channel communication has its own (writable) variable declared as `OUTPUT`. A read-only copy of its counterpart variable is also included by means of an `INPUT` declaration. Finally, the semantics of the channel operation is preserved with the incorporation of guarded commands in the `TRANSITION` section of both modules. In order to reduce the amount of variables in the model, an automatic planarization of the system is performed. With this approach, there is no channel explicit construction in the generated model. Instead, all the functions are absorbed by the scheme's translation. For example, the RAISE specification:

```
scheme
  C_BUTTON1 =
      class
        object
          CH : class channel push end
        variable button_var : T.Button_state
        value
          /∗ main ∗/
          button : Unit → in any out any write any Unit
          button() ≡
            while true do
              CH.push? ; button_var := T.lit
            end,
          /∗ generators ∗/
          push : Unit → in any out any Unit
          push() ≡ CH.push ! (),
      end
```

Is translated to

```
button : CONTEXT =
BEGIN
        Unit : CONTEXT = UnitType;
        T    : CONTEXT = types;

        c_button : MODULE =
        BEGIN
        INPUT
                push_start : Unit!Unit,
        OUTPUT
                push_finished : Unit!Unit,
        INITIALIZATION
                push_finished = Unit!nil;
        TRANSITION
        [
                push_start:
```

```
        push_start = Unit!set AND
        push_finished = Unit!nil -->
                check' = T!lit;
                push_finished' = Unit!set;
[]
        push_finished:
        clear_finished = Unit!nil AND
        push_finished = Unit!set -->
                push_finished' = Unit!nil;
[]
        ELSE -->
]
END;
```

From the example, it is important to notice that the function "push" from the specification is absorbed in the motor context within the SAL model. This approach can be generalized to handle a multiple channel declaration and the functions that encapsulates it. As mentioned above, with this technique the variable declarations for this function (needed for beginning/ending signalling) are avoided, allowing a more compact model generation.

Object expressions

Object expressions are a subtle point in the translation mechanism. As there is neither class nor object concepts inside SAL, an object exists in the model in different ways, depending on the style of the original specification.

Object expressions in the imperative sequential style For the imperative sequential case, an object is split in two parts:

1. The data, which is modelled as a record holding all the attributes and no functionality.
2. The functions (the so called "methods" in the object oriented programming paradigm) and types, which are translated as functions and type declarations respectively.

This two parts are, in turn, incorporated in the `CONTEXT` that holds the class declaration. As was mentioned before, an object instantiation is translated by the incorporation of the "data" part of the object in the global state record (with a proper, unique name). On the other hand, the operations involving the object instance are translated as a context instantiation and the proper function is called with the object data part as a parameter.

According to this convention, object names can be translated as the "data" part of the object, inside the global state record. The case of object array declarations can be handled in a similar way: a global array of the "data" part is incorporated, in this case, to the global state record. With this in mind, object expressions are translated as an indexed access to this global array.

For the case of array object expressions, the declaration is translated as a new type generation (with the name uniqueness property) for the binding involved in the expression. Then, an array of the "data"

component will be declared and incorporated to the global record, indexed by the type defined in the previous step.

With respect to fitting object expressions, SAL only supports context instantiation by type or value fitting. This functionality is not powerful enough for translating object fitting directly into it. One possible solution to this problem could be to perform the translation by unfolding the scheme on the fly, generating a whole new context each time a fitting is found.

Object expressions in the imperative concurrent style Element object expression are translated to an indexed access to the array of `INPUT` or `OUTPUT` that constitutes the interface to the object array.

Array object expressions are allowed (with the uni-dimensional indexing restriction stated in section 4.4). In particular, the typing within the expression is directly used in the SAL translator for defining the index type over the multi sequential composition operator in the generated module.

Fitting object expressions can be handled in the same way as in the imperative sequential style. In particular, the unfolding technique can also be applied but with the logical differences in the function invocation (due to the `MODULE`-based implementation).

Function type expressions

In the imperative sequential styles, access descriptors will be ignored because the function has complete access to the global state variable. On the other hand, for the imperative concurrent style, the variables with the **read** access mode will be translated as `INPUT` declarations in the `MODULE` that represents the function. Similarly, **write** accessed variables will be translated as `OUTPUT` variables. Variables with the **any** access descriptor will not be accepted (as there is no way to translate it directly into SAL).

Infix expressions

The translation of infix expressions is handled in very different ways, depending mainly on the type of infix expressions. In particular, statement infix expressions are the distinguishing feature of imperative (and, in most of the cases) concurrent styles. Due to this observation, the translation mechanism is not a simple one.

On the other hand, axiom and value infix expressions are handled, in general, in a straightforward way, based on the operation involved.

Statement infix expressions A statement involving external choice will be translated as a pair of guarded commands (the guarded expression is obtained from the condition included on each value expression involved in the external choice). This translation fits particularly well with the semantics of external choice due to SAL's guarded commands semantics. In particular, if at any moment, more than one choice is selectable for execution (i.e. the guards became true) only one of them will non-deterministically be selected for execution (behaviour that exactly matches RAISE's semantics of external choice). For example, the declaration:

x := c? [] d!5

will translate to a MODULE with a INITIALIZATION and TRANSITION sections like the following:

```
INITIALIZATION
    c_reader = unitType!set;
    d_writer = unitType!set;
    d_value  = 5;
TRANSITION
[
    c_reader = unitType!set AND
    c_writer = unitType!set -->
        x' = c_value;
        c_reader' = unitType!nil;
[]
    d_writer = unitType!set AND
    d_reader = unitType!set -->
        d_writer' = unitType!nil;
]
```

Even though the example includes some coding that is generated by the channel translation (for example, the initialization section) it is relevant to note that there is a guard for each possible option in the external choice (in this case, the channel requirement that both, the reader and the writer are present over the channel).

The parallel combinator operator is not supported within SAL's module language. However, it is possible to compose different modules in a parallel way. With this in mind, the translation of parallel operator could be translated as an independent module for each value expression involved in the statement. Then, the modules are joined together by means of SAL's synchronous composition operator (||). With this construction, both modules are forced to communicate if they both share the same channel and attempt to perform opposite operations over it. This behaviour is, in the general case, essentially different from the RAISE's parallel combinator one. In particular, according to RAISE's rules[18], the following equivalence holds:

x := c? || c!e $\equiv$ (x := e)⌈⌉ ((x := c? ; c!e) [] (c!e ; x := c?) [] (x := e))

On the other hand, according to SAL's semantics and the translation performed, the actual equivalence is:

x := c? || c!e $\equiv$ (x := e)

Despite the difference, in the case in which there is only one writer and only one reader over a channel, the former equivalence reduces to:

x := c? || c!e $\equiv$ (x := e)

which is exactly the semantic obtained in SAL by means of the parallel combinator. With this in mind, the translator will only accept specifications on which channels are input/output only by one process.

Finally, if there is no common event, both modules will execute their default, none action (implemented by means of an `ELSE` guarded empty transition). This last option does not lead the model to an explicit deadlock state (this is not a desirable behaviour because the model checking tools will not be allowed to make any further evaluation from this point) but both modules may become livelocked.

The interlocking combinator will not be accepted.

Lastly, the sequential combinator will be handled in two different ways, depending on the specification style in which it is used. At this point it is relevant to recall that the imperative sequential style will be encoded in SAL's applicative-styled model. In particular, as SAL functions are provided for an applicative behaviour, it is not possible to use the sequential combinator within them (there is no need for such a combinator in the applicative style). Due to this constraint, in the case of the imperative sequential style, the semantics of this operator is achieved by means of successive application of different, individual, functions. Note that, to preserve the evaluation order of the sequentially combined expressions, the order of invocation of the functions must be the inverse of the one used in the sequential composition.

The case of the imperative concurrent style, on the other hand, presents essentially the same situation. The case here is not the inability to perform more than one step inside an applicative function, but the atomicity of the statements included inside a guarded command (that inherently leads to a parallel behaviour). In particular, SAL semantic rules specify that all actions inside a guard are performed "atomically" and this restriction is reinforced by syntactic rules (by forbidding more than one statement inside a guard to modify the same variable).

Input expressions

Input expressions will be translated as a variable assuming the `set` value. The variable is the one used for signalling to the channel the presence of a reader willing to perform an input operation.

Output expressions

As with inputs, output expressions will be modelled as a variable set to the `set` value. In the case of output expressions, the variable used for the value transference is also set to the proper value (i.e. the outputted value). Once the channel detects both sides (by means of proper variable reading) set to `set`, the transference over the channel is performed.

Unique global variable names

Unique global variable names are generated by prepending the position in the specification's code to the identifier given by the user.

Chapter 5

Conclusions

Summary. In this report we have explained the problems when facing a transformation from one 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 every RAISE language construct 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.

Future work. Regarding as future work, the extensions described in section 4 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.

It will also be interesting to explore the SAL's construction `IMPLEMENTS` that allows the usage of model checking exhaustive state exploration in order to verify if there is a refinement/abstraction relationship between two modules. With this construction, it will be possible to actually verify (without the need of a proof) specifications that are derived from each other.

Appendix A

An applicative example of the lift example in SAL

```
types{; MIN_FLOOR : INTEGER, MAX_FLOOR: INTEGER}: CONTEXT =
BEGIN
    min_floor : INTEGER = MIN_FLOOR;
    max_floor : INTEGER = MAX_FLOOR;

    is_floor(f: INTEGER) : BOOLEAN =
        f >= MIN_FLOOR AND f <= MAX_FLOOR;

    MYINT : TYPE = [0..10];

    Floor : TYPE = [MIN_FLOOR .. MAX_FLOOR];
    Lower_floor : TYPE = {f  : Floor|f < MAX_FLOOR};
    Upper_floor : TYPE = {f: Floor | f > MIN_FLOOR};
    Door_state : TYPE = {open, shut};
    Button_state : TYPE = {lit, clear};
    Direction : TYPE = {up, down};
    Movement : TYPE = {halted, moving};
    Requirement : TYPE = [# here : BOOLEAN,
                            after: BOOLEAN,
                            before: BOOLEAN #];

    is_next_floor(d : Direction, f : Floor) : BOOLEAN =
       IF d = up
       THEN f < MAX_FLOOR
       ELSE f > MIN_FLOOR
       ENDIF;

    next_floor(d: Direction, f: Floor) : Floor =
       IF d = up
       THEN f+1
       ELSE f-1
       ENDIF

    invert(d: Direction) : Direction =
       IF d = up
       THEN down
       ELSE up
       ENDIF;

END
```

```
buttons{MIN_FLOOR : INTEGER, MAX_FLOOR: INTEGER} : CONTEXT =
BEGIN
      T   : CONTEXT = types{MIN_FLOOR,MAX_FLOOR};

      Buttons : TYPE = [# lift_btns : [T!Floor -> T!Button_state],
                     up_btns: [T!Lower_floor -> T!Button_state],
                     down_btns : [T!Upper_floor -> T!Button_state]#];

      init : Buttons = (# lift_btns := LAMBDA (f_prime : T!Floor) : T!clear,
                     up_btns:= LAMBDA (f_prime : T!Lower_floor): T!clear,
                     down_btns:= LAMBDA (f_prime : T!Upper_floor): T!clear#);

      push_lift(f: T!Floor, bs : Buttons) : Buttons =
            (# lift_btns := LAMBDA (f_prime : T!Floor) :
                            IF f = f_prime
                                      THEN T!lit
                               ELSE bs.lift_btns(f_prime)
                            ENDIF,
               up_btns:= bs.up_btns,
               down_btns := bs.down_btns#);

      push_up(f: T!Floor, bs : Buttons) : Buttons =
            (# lift_btns := bs.lift_btns,
               up_btns:= LAMBDA (f_prime : T!Lower_floor) :
                            IF f = f_prime
                               THEN T!lit
                               ELSE bs.up_btns(f_prime)
                            ENDIF,
               down_btns := bs.down_btns #);

      push_down(f: T!Floor, bs : Buttons) : Buttons =
            (# lift_btns := bs.lift_btns,
               up_btns:=bs.up_btns,
               down_btns := LAMBDA (f_prime : T!Upper_floor) :
                            IF f = f_prime
                               THEN T!lit
                               ELSE bs.down_btns(f_prime)
                            ENDIF#);

      clear(f: T!Floor, bs: Buttons) : Buttons =
             (# lift_btns := LAMBDA (f_prime : T!Floor) :
                            IF f = f_prime
                                      THEN T!clear
                               ELSE bs.lift_btns(f_prime)
                            ENDIF,
               up_btns:= LAMBDA (f_prime : T!Lower_floor) :
                            IF f = f_prime
                               THEN T!clear
                               ELSE bs.up_btns(f_prime)
                            ENDIF,
               down_btns := LAMBDA (f_prime : T!Upper_floor) :
```

```
                        IF f = f_prime
                           THEN T!clear
                           ELSE bs.down_btns(f_prime)
                        ENDIF#);


      required_here(d: T!Direction, f: T!Floor, bs: Buttons) : BOOLEAN =
            bs.lift_btns(f) = T!lit OR
            d = T!up AND
            (f < T!max_floor AND bs.up_btns(f) = T!lit OR
             f > T!min_floor AND bs.down_btns(f)= T!lit AND NOT
             (EXISTS (f_prime: T!Upper_floor) :
              f_prime > f AND
              (bs.lift_btns(f_prime) = T!lit OR
              (f_prime < T!max_floor AND bs.up_btns(f_prime) = T!lit) OR
              (f_prime > T!min_floor AND bs.down_btns(f_prime) = T!lit)))
            ) OR
            d = T!down AND
            (f > T!min_floor AND bs.down_btns(f) = T!lit OR
             f < T!max_floor AND bs.up_btns(f) = T!lit AND NOT
             (EXISTS (f_p : T!Lower_floor) :
                   f_p < f AND
                   (bs.lift_btns(f_p) = T!lit OR
                   (f_p < T!max_floor AND bs.up_btns(f_p) = T!lit) OR
                   (f_p > T!min_floor AND bs.down_btns(f_p) = T!lit))));

      required_beyond(d: T!Direction, f: T!Floor, bs: Buttons) : BOOLEAN =
            T!is_next_floor(d,f) AND
            LET f_prime : T!Floor = T!next_floor(d,f) IN
                required_here(d,f_prime,bs)
                OR
                (d = T!up AND
                 (EXISTS (f_2prime : T!Upper_floor) :
                   f_2prime > T!next_floor(d,f) AND
                   required_here(d,f_2prime,bs)))
                OR
                (d = T!down AND
                 (EXISTS (f_2prime : T!Lower_floor) :
                   f_2prime < T!next_floor(d,f) AND
                   required_here(d,f_2prime,bs)));


      check(d: T!Direction, f: T!Floor, s: Buttons) : T!Requirement =
             (# here   := required_here(d,f,s),
                after  := required_beyond(d,f,s),
                before := required_beyond(T!invert(d),f,s)#);
END
```

```
doors{MIN_FLOOR : INTEGER, MAX_FLOOR: INTEGER} : CONTEXT =
BEGIN
      rsl : CONTEXT = rsl_types;
      T   : CONTEXT = types{MIN_FLOOR,MAX_FLOOR};


      Doors : TYPE = [T!Floor -> T!Door_state];

      init : Doors = LAMBDA (e : T!Floor) : T!shut;

      open(f : T!Floor, s: Doors, f_prime: T!Floor) : Doors =
             IF f = f_prime THEN LAMBDA (e: T!Floor):
                           IF (e = f)
                            THEN T!open
                            ELSE s(e)
                         ENDIF
                         ELSE s
            ENDIF;

      close(f : T!Floor, s: Doors, f_prime: T!Floor) : Doors =
             IF f = f_prime THEN LAMBDA (e: T!Floor):
                           IF (e = f)
                            THEN T!shut
                            ELSE s(e)
                         ENDIF
                         ELSE s
            ENDIF;

      door_state(s: Doors, f: T!Floor) : T!Door_state =
            s(f);
END
```

```
motor{MIN_FLOOR : INTEGER, MAX_FLOOR: INTEGER}: CONTEXT =
BEGIN
      T : CONTEXT = types{MIN_FLOOR, MAX_FLOOR};
      rsl : CONTEXT = rsl_types;

      Motor : TYPE = [# direction : T!Direction,
                      movement: T!Movement,
                      floor : T!Floor#];

      init : Motor = (#direction := T!up,
                       movement := T!halted,
                       floor := T!min_floor#);

      move(d: T!Direction, mtr : Motor) : Motor =
            (# direction := d,
               movement := T!moving,
               floor := T!next_floor(d, mtr.floor)#);

      halt(m : Motor) : Motor =
              (# direction := m.direction,
               movement:= T!halted,
               floor := m.floor#);

      direction(m : Motor) : T!Direction =
            m.direction;

      movement(m : Motor) : T!Movement =
            m.movement;

      floor(m : Motor) : T!Floor =
            m.floor;
END
```

```
lift{MIN_FLOOR : INTEGER, MAX_FLOOR: INTEGER}: CONTEXT =
BEGIN
      T   : CONTEXT = types{MIN_FLOOR, MAX_FLOOR};
      DS  : CONTEXT = doors{MIN_FLOOR, MAX_FLOOR};
      BS  : CONTEXT = buttons{MIN_FLOOR, MAX_FLOOR};
      M   : CONTEXT = motor{MIN_FLOOR, MAX_FLOOR};

      Lift : TYPE = [# mtr : M!Motor,
                       drs : DS!Doors,
                     btns: BS!Buttons #];

      init_lift : Lift = (#mtr := M!init, drs := DS!init, btns := BS!init#);

      move(d: T!Direction, m: T!Movement, l: Lift) : Lift =
            (# mtr := M!move(d, l.mtr),
               drs := IF (m = T!halted)
                        THEN DS!close(M!floor(l.mtr),l.drs,M!floor(l.mtr))
                        ELSE l.drs
                        ENDIF,
               btns:= l.btns #);

      halt(l : Lift) : Lift =
             (# mtr := M!halt(l.mtr),
               drs := DS!open(M!floor(l.mtr),l.drs,M!floor(l.mtr)),
               btns:= BS!clear(M!floor(l.mtr),l.btns) #);

      next(r : T!Requirement, l : Lift) : Lift =
             LET d : T!Direction = M!direction(l.mtr) IN
             IF (M!movement(l.mtr) = T!halted)
              THEN IF (r.after = TRUE)
                   THEN
                    move(d, T!halted, l)
                   ELSIF (r.before = TRUE)
                   THEN
                    move(T!invert(d), T!halted, l)
                   ELSE l
                   ENDIF
               ELSE % T!moving...
                   IF (r.here = TRUE)
                   THEN
                    halt(l)
                   ELSIF (r.before = FALSE AND r.after = FALSE)
                   THEN
                    halt(l)
                   ELSIF (r.after = TRUE)
                   THEN
                      move(d, T!moving, l)
                   ELSE
                    move(T!invert(d), T!moving, l)
                   ENDIF
             ENDIF;
```

```
safe(l : Lift) : BOOLEAN =
       G(FORALL (f: T!Floor) : DS!door_state(l.drs,f) = T!open
       => (M!movement(l.mtr) = T!halted AND M!floor(l.mtr) = f));

liveness_lift_btns(l : Lift) : BOOLEAN =
       G(FORALL (f: T!Floor) : l.btns.lift_btns(f) = T!lit =>
                       F(M!floor(l.mtr) = f));

liveness_up_btns(l : Lift) : BOOLEAN =
       G(FORALL (f: T!Lower_floor) : l.btns.up_btns(f) = T!lit =>
                       F(M!floor(l.mtr) = f));

liveness_down_btns(l : Lift) : BOOLEAN =
       G(FORALL (f: T!Upper_floor) : l.btns.down_btns(f) = T!lit =>
                       F(M!floor(l.mtr) = f));

Option : TYPE = {PRESS, IDLE};

system : MODULE =
BEGIN
INPUT
      f_l : T!Floor,
      f_u : T!Lower_floor,
      f_d : T!Upper_floor,
      sel_lift,
      sel_up,
      sel_down : Option
LOCAL
      lift : Lift,
      req: T!Requirement
INITIALIZATION
      lift = init_lift;
      req.here = FALSE;
      req.before = FALSE;
      req.after = FALSE;
TRANSITION
[
      lift_button_selected:
      sel_lift = PRESS AND
      lift.btns.lift_btns(f_l) = T!clear -->
            lift' = (# mtr := lift.mtr,
                     drs := lift.drs,
                     btns:= BS!push_lift(f_l,lift.btns)#);
            req' = BS!check(M!direction(lift.mtr),M!floor(lift.mtr),
                            BS!push_lift(f_l,lift.btns));
[]
      up_button_selected:
      sel_up = PRESS AND
      lift.btns.up_btns(f_u) = T!clear -->
            lift' = (# mtr := lift.mtr,
```

```
                        drs := lift.drs,
                        btns:= BS!push_up(f_u,lift.btns)#);
                req' = BS!check(M!direction(lift.mtr),M!floor(lift.mtr),
                                BS!push_up(f_u,lift.btns));
      []
            down_button_selected:
            sel_down = PRESS AND
            lift.btns.down_btns(f_d) = T!clear -->
                  lift' = (# mtr := lift.mtr,
                           drs := lift.drs,
                           btns:= BS!push_down(f_d,lift.btns)#);
                  req' = BS!check(M!direction(lift.mtr),M!floor(lift.mtr),
                                  BS!push_down(f_d,lift.btns));
      []
            calculating_next_state:
            ELSE --> lift' = next(BS!check(M!direction(lift.mtr),
                                           M!floor(lift.mtr),
                                           lift.btns),lift);
      ]
      END;

safety : THEOREM
   system |- safe(lift);

liveness: THEOREM
   system |- liveness_lift_btns(lift) AND liveness_up_btns(lift) AND
              liveness_down_btns(lift);
END
```

Appendix B

Operator precedence in the translator

Table B.1: RSL and corresponding SAL

Precedence for Individual Operators in RSL and corresponding SAL					
RSL			SAL		
Prec	Operators	Assoc	Prec	Operators	Assoc
14	$\Box$	Right			
14	λ	Right	21	`LAMBDA`	None
14	$\forall$	Right	21	`FORALL`	None
14	$\exists$	Right	21	`EXISTS`	None
14	$\exists!$	Right	21	`EXISTS_ONE`	None
13	$\equiv$		18	`<=>`	Right
13	**post**				
12	$[]$	Right			
12	Π	Right			
12	$\parallel$	Right			
12	$\#$	Right			
11	;	Right			
10	:=				
9	$\Rightarrow$	Right	17	`IMPLIES, =>`	Right
8	$\vee$	Right	16	`OR`	Right
7	$\wedge$	Right	15	`AND`	Right
6	$= \neq > < \geq \leq$		13	`=, /=, <, <=, >, >=`	Left
6	$\subset \subseteq \supset \supseteq$			`subset()`, etc.	
6	$\in \notin$			`member()`, etc.	
5	$+ \quad -$	Left	8	`+, -`	Left
5	$\setminus$	Left		`difference()`	
5	$\hat{}$	Left		`append()`	
5	$\cup$	Left		`union()`	
5	$\dagger$	Left		`override()`	
4	$* \quad /$	Left	7	`*, /`	Left
4	$\circ$	Left	5	`o`	Left
4	$\cap$	Left		`intersection()`	
3	$\uparrow$		2		
2	:		4	:	Left
1	$\sim$		14	`NOT`	None
	prefix_op			functions	

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] J.-R. Abrial and I. H. Sørensen. KWIC-index generation. In J. Staunstrup, editor, *Program Specification: Proceedings of a Workshop*, volume 134 of *Lecture Notes in Computer Science*, pages 88–95. Springer-Verlag, 1981.

[3] Univan Ahn and Chris George. C++ translator for RAISE Specification Language. Technical Report 220, International Institute for Software Technology - United Nations University, November 2000.

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

[5] M. C. Browne, E. M. Clarke, and O. Grumberg. Characterizing finite Kripke structures in propositional temporal logic. In *Theoretical Computer Science*, pages 115–131, 1988.

[6] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. In *ACM Transactions on Programming Languages and Systems*, pages 8(2):244–263, 1986.

[7] E.M. Clarke and Emerson. Synthesis of synchronization skeletons for branching time temporal logic. In *Lecture Notes in Computer Science 131*. Springer-Verlag, 1981.

[8] Li Dan and Bernhard K. Aichernig. Automatic Test Case Generation for RAISE. Technical Report 273, International Institute for Software Technology - United Nations University, January 2003.

[9] Aristides Dasso and Chris George. Translating RSL into PVS. Technical Report 256, International Institute for Software Technology - United Nations University, March 2002.

[10] Leonardo de Moura, Sam Owre, Harald Rueß, John Rushby, N. Shankar, Maria Sorea, and Ashish Tiwari. 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.

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

[12] 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.

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

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

[15] Vijay Ganesh, Hassen Saïdi, and Natarajan Shankar. Slicing sal.

[16] 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.

[17] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In O. Grumberg, editor, *Proc. 9th INternational Conference on Computer Aided Verification (CAV'97)*, volume 1254, pages 72–83. Springer Verlag, 1997.

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

[19] The RAISE Method Group. *The RAISE Development Method*. Prentice Hall International (UK), 1995.

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

[21] G. Holzmann, P. Godefroid, and D. Pirottin. Coverage Preserving Reduction Strategies for Reachability Analisys. In *Symp. Protocol Specification, Testing and Verification (PSTV92)*, Holland, 1995. Addison Wesley Professional.

[22] Gerard Holzmann. *Algorithms for Automated Protocol Verification*. Prentice Hall, New Jersey, 1991.

[23] Gerard J. Holzmann. *The SPIN Model Checker*. Addison-Wesley, 2003.

[24] J. A. Kamp. *Tense logic and the theory of linear order*. PhD thesis, UCLA, Los Angeles, California, 1968.

[25] Kolyang, T. Santen, and B. Wolff. A structure preserving encoding of Z in isabelle/hol. In J. von Wright, J. Grundy, and J. Harrison, editors, *Higher Order Logics (TPHOLs 96)*. Springer-Verlag, 1996.

[26] R.P. Kurshan. *Computer-Aided Verification of Coordinating Processes*. Princeton Univ. Press, New Jersey, 1994.

[27] 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.

[28] O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *ACM Symp. Principles of Programming Languages (POPL'85)*, pages 97–107, 1985.

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

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

[31] Ralph Melton and David L. Dill. *Murϕ Annotated Reference Manual*. Computer Science Department, Standford University, California, 1993.

[32] Sam Owre, John Rushby, Natarajan Shankar, and Friedrich von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, feb 1995.

[33] A. Pnuelli. The Temporal Logic Programs. In *18th IEEE Symp. Foundations of Computer Science*, New Jersey, 1977.

[34] P.-P. Queille and J. Sifakis. Specificationfication of concurrent systems in CESAR. In *Lecture Notes in Computer Science*, pages 337–351, 1982.

[35] 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.

[36] Hassen Saïdi. Modular and incremental analysis of concurrent software systems. In *ASE'99: Proceedings of the 14th IEEE international conference on Automated software engineering*, page 92, Washington, DC, USA, 1999. IEEE Computer Society.

[37] Hassen Saïdi and Natarajan Shankar. Abstract and model check while you prove. In Nicolas Halbwachs and Doron Peled, editors, *Computer-Aided Verification (CAV'99)*, number 1633 in Lecture Notes in Computer Science, pages 443–454, Trento, Italy, jul 1999. Springer-Verlag.

[38] J. Scattergood. *The Semantics and Implementation of Machine-readable CSP*. PhD thesis, Oxford University Computing Laboratory, 1998.

[39] G. Smith and K. Winter. Proving temporal properties of Z specifications using abstractions. In D. Bert, J. P. Bowen, S. King, and M. Waldén, editors, *3rd International Conference of Z and B users (ZB 2003)*. Springer-Verlag, 2003.

[40] 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.

[41] D. Stringer-Calvert, S. Stepney, and I. Wand. Using PVS to prove a Z refinement: A case study. In J. Fitzgerald, C. Jones, and P. Lucas, editors, *Formal Methods Europe (FME 97)*. Springer-Verlag, 1997.

[42] M. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *1st IEEE Symp. Logic in Computer Science (LICS'86)*, pages 332–344, 1986.

[43] Ke Wei and Chris George. An RSL to SML Translator. Technical Report 208, International Institute for Software Technology - United Nations University, May 2001.