SRI International

CSL Technical Report SRI-CSL-01-02 (Rev. 2) • August, 2003

The SAL Language Manual

Leonardo de Moura
Sam Owre
N. Shankar

This report was developed and is maintained by SRI International. SRI's part of the SAL project is funded by DARPA/AFRL contract numbers F30602-96-C-0204 and F33615-00-C-3043.

Computer Science Laboratory • 333 Ravenswood Ave. • Menlo Park, CA 94025 • (650) 326-6200 • Facsimile: (650) 859-2844

Abstract

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 SAL language was originally designed in collaboration with David Dill of Stanford University and Thomas Henzinger of the University of California at Berkeley. The version presented here is the one currently accepted by the tools developed at SRI.

Contents

Chapter 1

Introduction

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 requirements on the SAL language are

1. **Generality:** It should be possible to effectively capture the transition semantics of a wide variety of source languages.

2. **Minimality:** The language should not have redundant or extraneous features that add complexity to the analysis. The language must capture transition system behavior without any complicated control structures.

3. **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.

4. **Language Modularity:** The language should be parametric with respect to orthogonal features such as the type/expression sublanguage, the transition sublanguage, and the module sublanguage.

5. **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 modeled by means of interleaving the atomic transitions of the individual modules.

We present the SAL language in stages consisting of the type system, the expression language, the transition language, modules, synchronous and asynchronous composition of modules, and the specification of systems. The language is largely modular in these choices in the sense that many of the language choices can be independently modified without affecting the other choices. The 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 [3], Murphi [4], Mocha [1], and SPIN [2]. Like these languages, SAL describes transition systems in terms of initialization and transition commands. These can be given by variable-wise definitions in the style of SMV or as guarded commands in the style of Murphi.

Chapter 2

A Simple Example: An N-bit Adder

An N-bit ripple-carry adder module is specified from a one-bit adder module by composing a base one-bit adder module with the synchronous multicomposition of $N-1$ one-bit adder modules. The one-bit adder takes three inputs: the two input bits `a` and `b` and the carry-in bit `cin`, and returns two outputs: the sum bit `sum` and the carry-out bit `cout`. See Figure 2.1. The N-bit adder takes three inputs: the two input bit-vectors `A` and `B` and the carry-in bit `carryin`, and returns two outputs: the sum vector `S` and the carry-out vector `C`. See Figure 2.2.

The `adder` module is definitional, as is usual for a purely combinational circuit description. This means there are no guarded commands, and the adders are synchronously composed.

Note that the requirement that types be nonempty means that the N-bit adder cannot be used to model a 1-bit adder. We plan on allowing empty types in the future, see Section 8.4.

```
adder: CONTEXT =
 BEGIN
  onebitadder: MODULE =
   BEGIN
    INPUT cin, a, b: BOOLEAN
    OUTPUT cout, sum: BOOLEAN
    DEFINITION
      sum = (a XOR b) XOR cin ;
      cout = (a AND b) OR (a AND cin) OR (b AND cin)
   END;

  Nbitadder [N : {n: NATURAL | n > 1}] : MODULE =
   WITH INPUT  A, B : ARRAY [0 .. N-1] OF BOOLEAN, carryin: BOOLEAN;
        OUTPUT S, C : ARRAY [0 .. N-1] OF BOOLEAN
    RENAME a TO A[0], b TO B[0], cin TO carryin,
           sum TO S[0], cout TO C[0] IN
      onebitadder
       ||
       (|| (i : [1 .. N-1]):
          (RENAME a TO A[i], b TO B[i], cin TO C[i-1],
                  sum TO S[i], cout TO C[i] IN
           onebitadder));
END
```

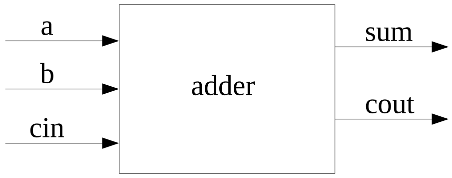

Figure 2.1: Module `adder`

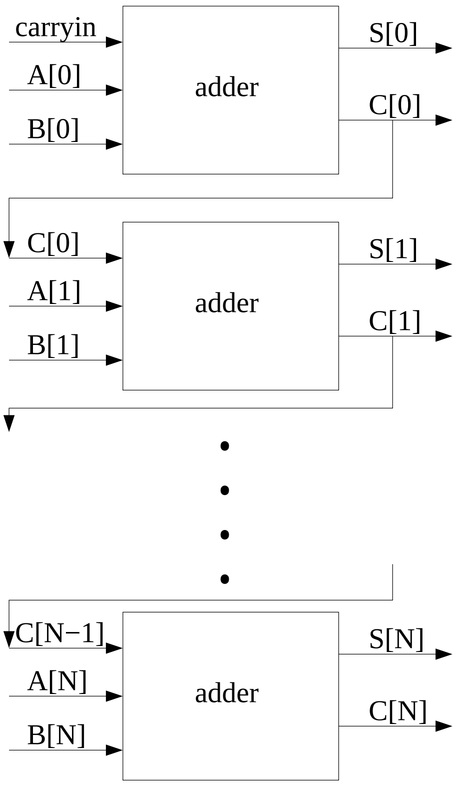

Figure 2.2: Module `Nbitadder`

Chapter 3

The Expression Language

The conventions used in presenting the SAL grammar are that tokens are given in `teletype` font, [*optional*] indicates that *optional* is optional, $\{category\}^{+}_{,}$ indicates one or more occurrences of the syntactic category *category* separated by commas, and $\{category\}^{*}_{,}$ indicates zero or more repetitions of *category* separated by commas. Separators other than comma can be used so that a transition given by a set of named guarded commands separated by the choice operator `[]` can be written as $\{NamedCommands\}^{+}_{\texttt{[]}}$. Nonterminals are written in *italics*.

The SAL language needs to be liberal in order to accommodate translations from other source languages. For this reason, identifiers include a large number of operators. The *special symbols* are parentheses (`(`, `)`), brackets (`[`, `]`), braces (`{`, `}`), the percent sign (`%`), comma (`,`), period (`.`), colon (`:`), semi-colon (`;`), single quote (`'`), exclamation point (`!`), hash (`#`), question mark (`?`), and underscore (`_`). Tokens can be separated by *WhiteSpace*, which consists of spaces, tabs, carriage returns, and line feeds.

SpecialSymbol := `(` | `)` | `[` | `]` | `{` | `}` | `%` | `,` | `.` | `;` | `:` | `'` | `!` | `#` | `?` | `_`
Letter := `a` | ... | `z` | `A` | ... | `Z`
Digit := `0` | ... | `9`
Identifier := *Letter* {*Letter* | *Digit* | `?` | `_`}*
| {*Opchar*}+
Numeral := {*Digit*}+

An *Opchar* is any character that is not a *Letter*, *Digit*, *SpecialSymbol*, or *WhiteSpace*. For example, `f1_3` and `+-+` are identifiers, but `a+-1` is three tokens: two identifiers (`a` and `+-`), and a numeral.

The grammar is case-sensitive. The reserved words must be in upper case. The reserved words are:

`AND`, `ARRAY`, `BEGIN`, `BOOLEAN`, `CLAIM`, `CONTEXT`, `DATATYPE`, `DEFINITION`, `ELSE`, `ELSIF`, `END`, `ENDIF`, `EXISTS`, `FALSE`, `FORALL`, `GLOBAL`, `IF`, `IN`, `INITIALIZATION`, `INPUT`, `INTEGER`, `LAMBDA`, `LEMMA`, `LET`, `LOCAL`, `MODULE`, `NATURAL`, `NOT`, `NZINTEGER`, `NZREAL`, `OBLIGATION`, `OF`, `OR`, `OUTPUT`, `REAL`, `RENAME`, `THEN`, `THEOREM`, `TO`, `TRANSITION`, `TRUE`, `TYPE`, `WITH`, `XOR`.

Comments in SAL are preceded by the `%` symbol and terminated by an end-of-line.

3.1 Types[1]

The SAL language supports the built-in basic types for booleans, natural numbers, integers, and reals. New basic types may be introduced using uninterpreted type declarations. Types may be used in type constructions to create subtype, subrange, array, function, tuple, and record types. Function, tuple, and record types may be dependent. In addition to uninterpreted type declarations, that introduce a name without a defining form, type declarations may be used to introduce names for existing types, as well as scalars and datatypes. The grammar for types is given by

$$
\begin{array}{rcl}
\mathit{TypeDef} & := & \mathit{Type} \\
 & & \mid \mathit{ScalarType} \\
 & & \mid \mathit{DataType} \\
\mathit{Type} & := & \mathit{BasicType} \\
 & & \mid \mathit{Name} \\
 & & \mid \mathit{Subrange} \\
 & & \mid \mathit{SubType} \\
 & & \mid \mathit{ArrayType} \\
 & & \mid \mathit{TupleType} \\
 & & \mid \mathit{FunctionType} \\
 & & \mid \mathit{RecordType} \\
 & & \mid \mathit{StateType} \\
\mathit{BasicType} & := & \texttt{BOOLEAN} \mid \texttt{REAL} \mid \texttt{INTEGER} \mid \texttt{NZINTEGER} \mid \texttt{NATURAL} \mid \texttt{NZREAL} \\
\mathit{Name} & := & \mathit{Identifier} \\
\mathit{QualifiedName} & := & \mathit{Identifier}[\ \{\mathit{ActualParameters}\}\ ]\,\texttt{!}\,\mathit{Identifier} \\
\mathit{Subrange} & := & \texttt{[}\ \mathit{Bound}\ \texttt{..}\ \mathit{Bound}\ \texttt{]} \\
\mathit{SubType} & := & \texttt{\{}\ \mathit{Identifier}\ \texttt{:}\ \mathit{Type}\ \texttt{|}\ \mathit{Expression}\ \texttt{\}} \\
\mathit{Bound} & := & \mathit{Unbounded} \mid \mathit{Expression} \\
\mathit{Unbounded} & := & \texttt{\_} \\
\mathit{ArrayType} & := & \texttt{ARRAY}\ \mathit{IndexType}\ \texttt{OF}\ \mathit{Type} \\
\mathit{IndexType} & := & \texttt{INTEGER} \mid \mathit{Subrange} \mid \mathit{ScalarTypeName} \\
\mathit{ScalarTypeName} & := & \mathit{Name} \\
\mathit{TupleType} & := & \texttt{[}\ \mathit{VarType}\ \texttt{,}\ \{\mathit{VarType}\}^{+}_{,}\ \texttt{]} \\
\mathit{FunctionType} & := & \texttt{[}\ \mathit{VarType}\ \texttt{->}\ \mathit{Type}\ \texttt{]} \\
\mathit{VarType} & := & [\,\mathit{Identifier}\ \texttt{:}\,]\ \mathit{Type} \\
\mathit{RecordType} & := & \texttt{[\#}\ \{\mathit{Identifier}\ \texttt{:}\ \mathit{Type}\}^{+}_{,}\ \texttt{\#]} \\
\mathit{StateType} & := & \mathit{Module}\ \texttt{.}\ \texttt{STATE} \\
\mathit{ScalarType} & := & \texttt{\{}\{\mathit{Identifier}\}^{+}_{,}\texttt{\}} \\
\mathit{DataType} & := & \texttt{DATATYPE}\ \mathit{Constructors}\ \texttt{END} \\
\mathit{Constructors} & := & \{\mathit{Identifier}[\ \texttt{(}\ \mathit{Accessors}\texttt{)}\ ]\}^{+}_{,} \\
\mathit{Accessors} & := & \{\mathit{Identifier}\ \texttt{:}\ \mathit{Type}\}^{+}_{,}
\end{array}
$$

A *TypeDef* is a type expression that can occur as the body of a type declaration, whereas a *Type* is more restrictive and circumscribes the types that can be used within an expression or a transition system module. Two types are equivalent if they are identical modulo the renaming of bound variables, the rearrangement of record labels, the equality of subtype predicates, and the unfolding of the definitions of defined types that are not scalar types or datatypes. Equivalence for types that are defined to be uninterpreted, scalar types, and datatypes is just name equivalence. Name equivalence is not a simple concept because compound names consist of the context name, actual

[1]SAL types are very similar to PVS types, both syntactically and semantically. See the PVS Language Reference [5].

parameters, and the identifier. Two names are equivalent if they agree on the context name, and the identifier, and the actual parameters, which are either types or expressions, are equivalent. Types in SAL (as in PVS) are modeled as sets, and two types are equivalent when every element of one is an element of the other. Thus the dependent types

```
[# a: INTEGER, b: {x: INTEGER | x < a} #]
[# b: INTEGER, a: {x: INTEGER | b < x} #]
```

are equivalent, and similarly for tuples. One way to see this equivalence is to note that each is equivalent to the type

```
{r: [# a: INTEGER, b: INTEGER #] | r‘b < r‘a}
```

Note that in an array type, the index type must either be `INTEGER`, a subrange, or a scalar type. SAL has a higher-order type system since it contains function types between arbitrary domain and range types. SAL types need not be finite, and the `REAL` and `INTEGER` types, for example, are infinite. The `REAL` type is the mathematical reals, not a floating point representation. Arrays with infinite index and range types are also admissible.

There are a fixed set of subtyping relations among the types that naturally corresponds to a subset relation between the denotations of these types. The subrange type `[a .. b]` is an abbreviation for `{x: INTEGER | a <= x AND x <= b}`, `[a .. _]` is an abbreviation for `{x: INTEGER | a <= x}`, and `[_ .. b]` is an abbreviation for `{x: INTEGER | x <= b}`. The type `NATURAL` is merely an abbreviation for `{x: INTEGER | 0 <= x}`. Any subrange is a subtype of a larger subrange. It is also a subtype of `INTEGER`. An array (function) type A is a subtype of another array (function) type B if the index types are identical, and the range type of A is a subtype of the range type of B. Similarly, a record type A is a subtype of another record type B if every element of A is an element of B, which means the label sets must be the same, though as described in type equivalence, the corresponding types do not have to be in the subtype relation.

A *StateType* is a record type representing the state of the specified module. This is described in more detail below.

All types must be checked to be nonempty through the possible generation of proof obligations entailing nonemptiness.

Recursive datatypes can be used to define list and tree-like types. The datatype is specified by a list of constructor operations, each with a list of accessor operations. For example, the list type of integers is constructed as

```
intlist: TYPE = DATATYPE
                  cons(car : INTEGER, cdr : intlist),
                  nil
                END
```

Recognizers are automatically generated by appending a `?` to the corresponding constructor. Thus `cons?` and `nil?` are recognizers for `intlist`. These may be used in definitions. For example, `length` may be defined recursively[2] as

[2] This will lead to proof obligations showing that the function is total, i.e., terminating.

```
length: [intlist -> NATURAL] =
   LAMBDA (lst: intlist):
     IF nil?(lst) THEN 0 ELSE 1 + length(cdr(lst)) ENDIF
```

3.2 Expressions

Expressions in the SAL language have to be type-correct with respect to the types in the type language. The expressions consist of constants, variables, applications with Boolean, arithmetic, and bit-vector operations, and array, function, tuple, and record selection and updates. Conditional (if-then-else) expressions are also part of the expression language.

$$
\begin{array}{lcl}
\textit{Expression} & := & \textit{NameExpr} \\
 & | & \textit{QualifiedNameExpr} \\
 & | & \textit{NextVariable} \\
 & | & \textit{Numeral} \\
 & | & \textit{Application} \\
 & | & \textit{InfixApplication} \\
 & | & \textit{ArraySelection} \\
 & | & \textit{RecordSelection} \\
 & | & \textit{TupleSelection} \\
 & | & \textit{UpdateExpression} \\
 & | & \textit{LambdaAbstraction} \\
 & | & \textit{QuantifiedExpression} \\
 & | & \textit{LetExpression} \\
 & | & \textit{SetExpression} \\
 & | & \textit{ArrayLiteral} \\
 & | & \textit{RecordLiteral} \\
 & | & \textit{TupleLiteral} \\
 & | & \textit{Conditional} \\
 & | & (\ \textit{Expression}\ ) \\
 & | & \textit{StatePred}
\end{array}
$$

NameExpr	:=	*Name*
QualifiedNameExpr	:=	*QualifiedName*
NextVariable	:=	*Identifier* '
Application	:=	*Function Argument*
Function	:=	*Expression*
Argument	:=	($\{Expression\}^+_,$)
InfixApplication	:=	*Expression Identifier Expression*
ArraySelection	:=	*Expression*[*Expression*]
RecordSelection	:=	*Expression*.*Identifier*
TupleSelection	:=	*Expression*.*Numeral*
UpdateExpression	:=	*Expression* `WITH` *Update*
Update	:=	*UpdatePosition* `:=` *Expression*
UpdatePosition	:=	$\{$*Argument* | [*Expression*] | .*Identifier* | .*Numeral*$\}^+$
LambdaAbstraction	:=	`LAMBDA` (*VarDecls*) : *Expression*
VarDecls	:=	$\{VarDecl\}^+_,$
VarDecl	:=	$\{Identifier\}^+_,$: *Type*
QuantifiedExpression	:=	*Quantifier* (*VarDecls*) : *Expression*
Quantifier	:=	`FORALL` | `EXISTS`
LetExpression	:=	`LET` *LetDeclarations* `IN` *Expression*
LetDeclarations	:=	$\{$*Identifier* : *Type* = *Expression*$\}^+_,$
SetExpression	:=	*SetListExpression* | *SetPredExpression*
SetPredExpression	:=	{ *Identifier* : *Type* | *Expression* }
SetListExpression	|	{ $\{Expression\}^+_,$ }
ArrayLiteral	:=	[[*IndexVarDecl*] *Expression*]
IndexVarDecl	:=	*Identifier* : *IndexType*
RecordLiteral	:=	(# $\{RecordEntry\}^+_,$ #)
RecordEntry	:=	*Identifier* `:=` *Expression*
TupleLiteral	:=	*Argument*
Conditional	:=	`IF` *Expression ThenRest*
ThenRest	:=	`THEN` *Expression*
		[*ElsIf*]
		`ELSE` *Expression* `ENDIF`
ElsIf	:=	`ELSIF` *Expression ThenRest*
StatePred	:=	*Module* . (`INIT` | `TRANS`)

The unary operators include boolean negation `NOT`, and integer minus `-`.

The binary operators include

- Polymorphic equality `=` and disequality `/=`. Note that since subtypes are semantically the same as subsets, equality and disequality are defined on the maximal supertype of a type.
- Boolean operations of conjunction `AND`, disjunction `OR`, implication `=>`, equivalence `<=>`, and exclusive-or `XOR`
- Real arithmetic operations of addition `+`, subtraction `-`, multiplication `*`, division `/`, and the comparison operators `<`, `<=`, `>`, `>=`. Note that the divisor type of division is restricted to `NZREAL` and the type rules generate a proof obligation if the divisor is not known to be nonzero. The integer arithmetic operations of `DIV` and `MOD` are included in the binary operations. Both require nonzero integers, i.e., `NZINTEGER`, in the divisor position and they satisfy the equation

$$a = b * (a \texttt{ DIV } b) + (a \texttt{ MOD } b)$$

Although the parser allows any *Identifier* as an infix operator, it is clearly useful to have a standard operator precedence so that expressions such as `y + 1 = x AND A` are not parsed nonsensically, e.g., as `y + (1 = (x AND A))`. The precedence is as follows, from lowest to highest:

`<=>`
`=>`
`OR`, `XOR`
`AND`
`=`, `/=`
`>`, `>=`, `<`, `<=`
OtherIdentifier
`+`, `-`
`*`, `/`

`<=>`, `OR`, `XOR`, `AND`, `+`, infix `-`, `*`, and `/` are all left-associative, `=>` is right-associative, and the rest are non-associative.

The *LetExpression* is parallel, to get the sequential form use nested `LET`s, e.g.,

```
LET a = f(b) IN
 LET b = f(a) IN e
```

The proof obligations generated during typechecking are called type correctness conditions (TCCs). In addition to operations with subtype domains such as division, the sources of TCCs include expressions of subrange types, recursive datatypes, recursive definitions, and type nonemptiness.

An expression without *NextVariable*s is called a *current expression* and is represented by the nonterminal *CExpression*. We will not define its grammar but it essentially corresponds to the grammar for *Expression* with the occurrences of *NextVariable* removed.

SAL expressions contain two kinds of variables: logical variables and state variables. The state variables are either current variables or *NextVariables*. SAL types and expressions are given a semantics with respect to a model $\mathcal{M}$ that fixes the meanings of types, constants, and operators, an assignment ρ of values to the free logical variables, and an assignment of values to the current variables x and the *NextVariable*s x' by a pair of *states* $\langle r, s\rangle$. The meaning of expression e with respect to model $\mathcal{M}$, assignment ρ, and a pair of states $\langle r, s\rangle$, is given by $\mathcal{M}[\![e]\!]^{\rho}_{\langle r,s\rangle}$. If variable x has type A, then the interpretation of x in state s, $s(x)$, must be an element of $\mathcal{M}[\![A]\!]$. If x is a variable in the state type, then $\mathcal{M}[\![x]\!]_{\langle r,s\rangle} = r(x)$, and $\mathcal{M}[\![x']\!]_{\langle r,s\rangle} = s(x)$. The interpretation of types and operators are the standard ones. When expression e does not contain any *NextVariable*s, we write the meaning of e as $\mathcal{M}[\![e]\!]_r$.

The *StatePred* expressions provide access to the initialization predicate and transition relations for a given module M. In particular, M`.INIT` is of type `[`M`.STATE -> BOOLEAN]` and M`.TRANS` is of type `[`M`.STATE, `M`.STATE -> BOOLEAN]`.

Chapter 4

The Transition Language

A transition system *module* consists of a *state* type, an *invariant definition* on this state type, an *initialization 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. The language constructs for defining modules from transition systems are treated in Chapter 5.

The transition rules are constraints on the current and next states of the transition. The current variables are written as `X` whereas the next state variables are written as `X'`.

4.1 Definitions

Definitions are the basic constructs used to build up the invariants, initializations, and transitions of a module. Definitions are used to specify the trajectory of variables in a computation by providing constraints on the controlled variables in a transition system. For variables ranging over aggregate data structures like records or arrays, it is possible to define each component separately. For example,

```
x' = x + 1
```

simply increments the state variable `x`, where `x'` is the newstate of the variable,

```
y'[i] = 3
```

sets the new state of the array `y` to be `3` at index `i`, and to remain unchanged on all other indices, and

```
z.foo.1[0] = y
```

constrains state variable `z`, which is a record whose `foo` component is a tuple, whose first component in turn is an array of the same type as `y`.

The left-hand side of a definition is given by the nonterminal *Lhs*.

$$
\begin{array}{rcl}
\textit{Lhs} & := & \textit{Identifier}[\ \texttt{'}\ ]\ \{\textit{Access}\}^{*} \\
\textit{Access} & := & \textit{ArrayAccess} \mid \textit{RecordAccess} \mid \textit{TupleAccess} \\
\textit{ArrayAccess} & := & \texttt{[}\ \textit{Expression}\ \texttt{]} \\
\textit{RecordAccess} & := & \texttt{.}\ \textit{Identifier} \\
\textit{TupleAccess} & := & \texttt{.}\ \textit{Numeral}
\end{array}
$$

Simple definitions are of the form

$$
\begin{array}{rcl}
\textit{SimpleDefinition} & := & \textit{Lhs RhsDefinition} \\
\textit{RhsDefinition} & := & \textit{RhsExpression} \mid \textit{RhsSelection} \\
\textit{RhsExpression} & := & \texttt{=}\ \textit{Expression} \\
\textit{RhsSelection} & := & \texttt{IN}\ \textit{Expression}
\end{array}
$$

For an *RhsExpression*, the *Lhs* is simply assigned the corresponding value. For an *RhsSelection*, the *Lhs* is assigned any value satisfying the expression, which must be a predicate (a boolean-valued *LambdaAbstraction* or a *SetExpression*). This predicate must be satisfiable; an *invariant obligation* is generated if it cannot be determined to be nonempty.

Note that in an *Access*, all unspecified components are unchanged, thus `x'[i].name = Ed` is equivalent to `x' = x WITH [i].name := Ed`. If the given transition has multiple assignments to `x`, they must all be collected to get the equivalent form, for example, the assignments

```
x'[0].name = Ed;
x'[1].name = Al
```

are equivalent to `x' = x WITH [0].name = Ed WITH [1].name = Al`.

There are other restrictions on the *Access*. Within a given `DEFINITION`, `INITIALIZATION`, or `TRANSITION` section of a module the *Lhs* accesses must all be unique. Thus the assignments

```
x'[3] = 0;
x'[f(3)] = 0
```

will generate a proof obligation that `3 /= f(3)`. Note that it does not matter that these are really the same assignments if they are equal, the obligation will still be generated.

A transition equation in the `TRANSITION` section defines a *NextVariable* on the left-hand side in terms of an expression that can contain *NextVariable* occurrences. A *SimpleDefinition* can occur in the `TRANSITION` section of a transition system. An array index expression on the left-hand side must not contain any state variables.

$$
\begin{array}{rcl}
\textit{Definitions} & := & \{\textit{Definition}\}^{+}_{;} \\
\textit{Definition} & := & \textit{SimpleDefinition} \mid \textit{ForallDefinition} \\
\textit{ForallDefinition} & := & \texttt{(FORALL (}\ \textit{VarDecls}\texttt{):}\ \textit{Definitions}\texttt{)}
\end{array}
$$

In a transition system module, a controlled variable must be defined exactly once. It is easy to write definitions that admit causal cycles such as:

```
X = NOT Y;
Y = X
```

Such causal loops can lead to contradictory or meaningless definitions and have to be ruled out. One way to avoid causal loops is by means of an ordering on the variables so that the right-hand side of a definition can contain only those variables that are lower in the ordering. However, such a restriction would rule out natural definitions where variables can depend on each other without triggering a causal loop, for example

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

Here there is no causal loop since `X` depends on `Y` only when `A` holds, and `Y` depends on `X` only when `NOT A` holds. A dependency analysis generates a Boolean formula indicating the governing conditions *GC(X, Y)* under which a variable `X` immediately depends on another variable `Y`. The governing conditions are required to be current expressions. For example, $GC(\mathtt{X},\mathtt{Y})$ for the above definitions of `X` yields `A`. If there is no assignment defining `X` in terms of `Y` then *GC(X, Y)* is *false*. Then $GC^*(\mathtt{X},\mathtt{Y})$ yields the governing conditions under which a variable `X` could indirectly depend on a variable `Y`. For example, if `X` depends on a variable `Z` that in turn depends on `Y`, then $GC^*(\mathtt{X},\mathtt{Y})$ is just $GC(\mathtt{X},\mathtt{Y}) \vee (GC(\mathtt{X},\mathtt{Z}) \wedge GC(\mathtt{Z},\mathtt{Y}))$. Thus, in the above definitions of `X` and `Y`, $GC^*(X,X)$ is $\mathtt{A} \wedge \neg\mathtt{A}$. The dependency conditions can be used to generate the conditions $C_{\mathtt{X}}$ under which a variable `X` could depend on itself. For such dependency loops to be avoided, the condition $C_{\mathtt{X}}$ must be shown to be invariantly false in the transition system. In the above example, $C_{\mathtt{X}}$ would be the obviously unreachable assertion $\mathtt{A} \wedge \neg\mathtt{A}$. The dependency analysis (causality checks) generate proof obligations to this effect. A similar dependency analysis can be carried out for initialization definitions and transition definitions.

4.2 Guarded Commands

Definitions are convenient for specifying the values taken on by those controlled variables whose transitions can be independently specified in a simple equational form. Definitions have some drawbacks. For variables whose definitions follow a similar case structure, this case structure has to be repeated in each of the definitions. For such controlled variables, it is convenient to specify their initialization and transitions in terms of guarded commands. Each guarded command consists of a *guard* formula and an assignment part. The guard is a boolean expression in the current controlled (local, global, and output) variables and current 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 current and next state variables.

$$
\begin{array}{rcl}
\mathit{GuardedCommand} & := & \mathit{Guard}\ \texttt{-->}\ \mathit{Assignments} \\
\mathit{Guard} & := & \mathit{Expression} \\
\mathit{Assignments} & := & \{\mathit{SimpleDefinition}\}^*_{;}\ [\ ;\ ]
\end{array}
$$

Note that both the initializations and transitions may be specified by guarded assignments. No variable that is defined in the *Lhs* of a definition can be assigned in either a guarded initialization or transition. The initializations must not contain next state variables, whereas the transitions must have next state variables on the left-hand side of assignments, and may have next state variables on the right-hand side. The well-formedness checks on the guarded transitions are that the guard must not contain controlled next state variables, i.e., `X'` for some controlled variable `X`, since these variables are only assigned values in the assignment part. The assignments in the assignment part must ensure that no controlled variable is assigned more than once.

The causality checks and proof obligations corresponding to a guarded initialization or transition are similar to those for definitions. The primary difference is that current conjuncts in the guard can be conjoined to the the conditions when the proof obligations are generated. For example, if there is a guarded command of the form g `-->` *Assignments* where the dependency analysis on the combination of the *Assignments* and the definitions yields the conditions for a causal loop on variable `X` as C_X, then the conjunction $g \wedge C_X$ must be shown to be unreachable.

Note that the initialization and transition sections may contain simple definitions and/or guarded commands. The model of execution is that when the module gets activated, one guarded transition is chosen so that the guard formula holds in the current (and possibly next input) state, and the transition is the conjunction of the associated guarded transition with all the definitions of the transition section(s). If no guard is satisfied, the module may deadlock. A synchronously composed system is deadlocked if any of its component modules is. An asynchronously composed system is only deadlocked if all its components are. If you want to ensure a given module does not deadlock, just make sure that there is always some guard of the module that hlods true (the `ELSE` clause is useful for this).

Chapter 5

The Module Language

A module is a self-contained specification of a transition system in SAL. Modules can be independently analyzed for properties and composed synchronously or asynchronously. Here is a fairly simple module declaration.

```
m : MODULE =
 BEGIN
  INPUT temp: INTEGER
  LOCAL high: BOOLEAN, ctr: NATURAL
  OUTPUT danger: BOOLEAN
  DEFINITION high = i > 100
  INITIALIZATION ctr = 0; danger = FALSE
  TRANSITION [  ctr > 3 --> danger' = danger OR high
             [] ctr <= 3 AND high --> ctr' = ctr + 1
             [] ELSE --> ctr' = 0
              ]
 END
```

Here `m` is a *BaseModule*, that is intended to monitor the temperature and indicate a problem if the temperature stays high for too long. It declares the input variable `temp`, local variables `high` and `ctr`, and output variable `danger`. Initially `danger` is `FALSE` and `ctr` is 0, and when this module is activated it sets `danger` to `TRUE` if `temp` exceeds 100 more than 3 times in a row.

Once base modules are declared, they may be composed synchronously or asynchronously to yield new modules. The grammar for module expressions is given below. The grammars for *Definitions* and *GuardedCommand* are described in the previous chapter, but are repeated here for convenience.

$$
\begin{array}{rcl}
\mathit{Module} & := & \mathit{BaseModule} \\
& | & \mathit{ModuleInstance} \\
& | & \mathit{SynchronousComposition} \\
& | & \mathit{AsynchronousComposition} \\
& | & \mathit{MultiSynchronous} \\
& | & \mathit{MultiAsynchronous} \\
& | & \mathit{Hiding} \\
& | & \mathit{NewOutput} \\
& | & \mathit{Renaming} \\
& | & \mathit{WithModule} \\
& | & \mathit{ObserveModule} \\
& | & \texttt{(}\ \mathit{Module}\ \texttt{)}
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{BaseModule} & := & \texttt{BEGIN}\ \mathit{BaseDeclarations}\ \texttt{END} \\
\mathit{BaseDeclarations} & := & \{\mathit{BaseDeclaration}\}^{*} \\
\mathit{BaseDeclaration} & := & \mathit{InputDecl} \\
& | & \mathit{OutputDecl} \\
& | & \mathit{GlobalDecl} \\
& | & \mathit{LocalDecl} \\
& | & \mathit{DefDecl} \\
& | & \mathit{InitDecl} \\
& | & \mathit{TransDecl} \\
\mathit{InputDecl} & := & \texttt{INPUT}\ \mathit{VarDecls} \\
\mathit{OutputDecl} & := & \texttt{OUTPUT}\ \mathit{VarDecls} \\
\mathit{GlobalDecl} & := & \texttt{GLOBAL}\ \mathit{VarDecls} \\
\mathit{LocalDecl} & := & \texttt{LOCAL}\ \mathit{VarDecls} \\
\mathit{DefDecl} & := & \texttt{DEFINITION}\ \mathit{Definitions} \\
\mathit{InitDecl} & := & \texttt{INITIALIZATION}\ \{\mathit{DefinitionOrCommand}\}^{+}_{;}\ [\ ;\ ] \\
\mathit{TransDecl} & := & \texttt{TRANSITION}\ \{\mathit{DefinitionOrCommand}\}^{+}_{;}\ [\ ;\ ]
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{DefinitionOrCommand} & := & \mathit{Definition} \\
& | & \texttt{[}\ \mathit{SomeCommands}\ \texttt{]} \\
\mathit{Definitions} & := & \{\mathit{Definition}\}^{+}_{;} \\
\mathit{Definition} & := & \mathit{SimpleDefinition} \mid \mathit{ForallDefinition} \\
\mathit{ForallDefinition} & := & \texttt{(FORALL (}\ \mathit{VarDecls}\texttt{):}\ \mathit{Definitions}\texttt{)} \\
\mathit{SimpleDefinition} & := & \mathit{Lhs}\ \mathit{RhsDefinition} \\
\mathit{Lhs} & := & \mathit{Identifier}\,[\ \texttt{'}\ ]\ \{\mathit{Access}\}^{*} \\
\mathit{Access} & := & \mathit{ArrayAccess} \mid \mathit{RecordAccess} \mid \mathit{TupleAccess} \\
\mathit{ArrayAccess} & := & \texttt{[}\ \mathit{Expression}\ \texttt{]} \\
\mathit{RecordAccess} & := & \texttt{.}\ \mathit{Identifier} \\
\mathit{TupleAccess} & := & \texttt{.}\ \mathit{Numeral} \\
\mathit{RhsDefinition} & := & \mathit{RhsExpression} \mid \mathit{RhsSelection} \\
\mathit{RhsExpression} & := & \texttt{=}\ \mathit{Expression} \\
\mathit{RhsSelection} & := & \texttt{IN}\ \mathit{Expression} \\
\mathit{SomeCommands} & := & \{\mathit{SomeCommand}\}^{+}_{\texttt{[]}}\ [\ \texttt{[]}\,\mathit{ElseCommand}\ ] \\
\mathit{SomeCommand} & := & \mathit{NamedCommand} \mid \mathit{MultiCommand} \\
\mathit{NamedCommand} & := & [\ \mathit{Identifier}\ \texttt{:}\ ]\ \mathit{GuardedCommand} \\
\mathit{GuardedCommand} & := & \mathit{Guard}\ \texttt{-->}\ \mathit{Assignments} \\
\mathit{Guard} & := & \mathit{Expression} \\
\mathit{Assignments} & := & \{\mathit{SimpleDefinition}\}^{*}_{;}\ [\ ;\ ] \\
\mathit{MultiCommand} & := & \texttt{([] (}\ \mathit{VarDecls}\texttt{):}\ \mathit{SomeCommand}\texttt{)} \\
\mathit{ElseCommand} & := & [\ \mathit{Identifier}\ \texttt{:}\ ]\ \texttt{ELSE -->}\ \mathit{Assignments}
\end{array}
$$

$$\begin{array}{rcl}
\mathit{ModuleInstance} & := & \{\mathit{ModuleName} \mid \mathit{QualifiedModuleName}\}\ \mathit{Name}[\ \texttt{[}\{\mathit{Expression}\}^{+}_{,}\texttt{]}\ ] \\
\mathit{ModuleName} & := & \mathit{Name} \\
\mathit{QualifiedModuleName} & := & \mathit{QualifiedName} \\
\mathit{SynchronousComposition} & := & \mathit{Module}\ \texttt{||}\ \mathit{Module} \\
\mathit{AsynchronousComposition} & := & \mathit{Module}\ \texttt{[]}\ \mathit{Module} \\
\mathit{MultiSynchronous} & := & \texttt{(||}\ \mathit{(Identifier : IndexType): Module)} \\
\mathit{MultiAsynchronous} & := & \texttt{([]}\ \mathit{(Identifier : IndexType): Module)} \\
\mathit{Hiding} & := & \texttt{LOCAL}\ \{\mathit{Identifier}\}^{+}_{,}\ \texttt{IN}\ \mathit{Module} \\
\mathit{NewOutput} & := & \texttt{OUTPUT}\ \{\mathit{Identifier}\}^{+}_{,}\ \texttt{IN}\ \mathit{Module} \\
\mathit{Renaming} & := & \texttt{RENAME}\ \mathit{Renames}\ \texttt{IN}\ \mathit{Module} \\
\mathit{Renames} & := & \{\mathit{Lhs}\ \texttt{TO}\ \mathit{Lhs}\}^{+}_{,} \\
\mathit{WithModule} & := & \texttt{WITH}\ \mathit{NewVarDecls}\ \mathit{Module} \\
\mathit{NewVarDecls} & := & \{\mathit{InputDecl} \mid \mathit{OutputDecl} \mid \mathit{GlobalDecl}\}^{+}_{;} \\
\mathit{ObserveModule} & := & \texttt{OBSERVE}\ \mathit{Module}\ \texttt{WITH}\ \mathit{Module}
\end{array}$$

5.1 Base Modules

A *BaseModule* identifies the pairwise distinct sets of input, output, global, and local variables. This characterizes the *state* of the module.

As described below, base modules also may consist of several sections. Note that the grammar allows variables and sections to be given in any order, and there may, for example, be 3 distinct `TRANSITION` sections. In every case, it is the same as if there was a prescribed order, with each class of variable and section being the union of the individual declarations.

`DEFINITION` section. Definitions appearing in the `DEFINITION` section(s) are treated as invariants for the system. When composed with other modules, the definitions remain true even during the transitions of the other modules. For this reason, proof obligations may be generated for a composition where definition sections are involved. This section is usually used to define controlled variables whose values ultimately depend on the inputs, for example, a boolean variable that becomes true when the temperature goes above a specified value.

Definition sections must be used with care, especially when modeling asynchronous systems, as this means that in some sense the execution of a module on a remote machine can still be seen locally.

`INITIALIZATION` section. The `INITIALIZATION` section(s) constrain the possible initial values for the local, global, and output declarations. Input variables may not be initialized. The `INITIALIZATION` section(s) determine a state predicate that holds of the initial state of the base module.

Definitions and guarded commands appearing in the `INITIALIZATION` section must not contain any *NextVariable* occurrences, i.e., both sides of the defining equation must be *current expressions*. Guards may refer to any variables, this acts as a form of postcondition when controlled variables are involved. This is like backtracking: operationally a guarded initialization is selected, the assignments made, and if the assignments violate the guard the assignments are undone and a new guarded initialization is selected.

`TRANSITION` section. The `TRANSITION` section(s) constrain the possible next states for the local, global, and output declarations. As this is generally defined relative to the previous state of the module, the transition section(s) determine a state relation. Input variables may not appear on the *Lhs* of any assignments. Guards may refer to any variables, even *NextVariable*s. As with guarded initial transitions, guards involving *NextVariable*s have to be evaluated after the assignments have been made, and if they are false the assignments must be undone and a new guarded transition selected.

5.2 State Variable Manipulation

Output and global variables can be made local by the `LOCAL` construct. Global variables can be made output by the `OUTPUT` construct. In order to avoid name clashes, variables in a module can be renamed using the `RENAME` construct. When the renaming variable is an identifier, its type can be easily inferred from the renamed variable. New state variables used for renaming can be introduced using the `WITH` construct for `INPUT`, `OUTPUT`, and `GLOBAL` declarations. These newly declared variables can be used in the `RENAME` construct to rename the variables in a given module. The renaming should be consistent so that the input variables can be renamed only by input variables, output variables only by output variables, and global variables only by output or global variables. The types of the renamed and the renaming variable should also match.

5.3 Module Composition

Modules can be combined by either synchronous or asynchronous composition.

Let module M_i consists of input variables I_i, output variables O_i, global variables G_i, and local variables L_i. The module $M_1 \,||\, M_2$ and $M_1 \,[\,]\, M_2$ respectively represent the synchronous and asynchronous composition of M_1 and M_2.

Variables with the same identifier are treated as identical, and it is an error to compose modules that assign different types to the same identifier. The syntactic constraints on both synchronous and asynchronous composition are that the output variable sets must be disjoint from the global and output variables of the other module ($O_1 \bigcap (O_2 \bigcup G_2) = \emptyset$, $(O_1 \bigcup G_1) \bigcap O_2 = \emptyset$), the local variables must be disjoint from the other variables ($L \bigcap (I \bigcup O \bigcup G) = \emptyset$), but need not be disjoint from each other.

The input variables I, the output variables O, global variables G, and the local variables L of $M_1 \,||\, M_2$ and $M_1 \,[\,]\, M_2$ are given by

$$
\begin{aligned}
I &= (I_1 \bigcup I_2) - (O \bigcup G) \\
O &= (O_1 \bigcup O_2) \\
G &= (G_1 \bigcup G_2) \\
L &= (L_1 \bigcup L_2)
\end{aligned}
$$

The semantics of synchronous composition is that the module $M_1 \,||\, M_2$ consists of initializations that are the combination of initializations from the two modules, and the transitions are the combinations of the individual transitions of the two modules. The definitions of $M_1 \,||\, M_2$ are simply the union of the definitions in M_1 and M_2. The initializations of $M_1 \,||\, M_2$ are the pairwise combination of the initializations in M_1 and M_2. Two guarded initializations are combined by conjoining the guards and by taking the union of the assignments. Let $g_{1,i}$ `-->` $a_{1,i}$ be an initialization from M_1 and $g_{2,j}$ `-->` $a_{2,j}$ be an initialization from M_2. The guard $g_{1,i}$ might contain output variables of M_2, and similarly, guard $g_{2,j}$ might contain output variables of M_1. For the combination to be sensible, only at most one of these guards, say $g_{1,i}$, is allowed to contain output variables of the other module. If we take $\overline{a_{2,j}}$ as the union of the assignments in $a_{2,j}$ with the initialization definitions of M_2, then we can repeatedly apply $\overline{a_{2,j}}$ as a substitution. It should then be the case that the repeated application $\overline{a_{2,j}}^*(g_{1,i})$ converges. The combination of the two initializations is then $\overline{a_{2,j}}^*(g_{1,i}) \wedge g_{2,j}$ `-->` $a_{1,i}\,;a_{2,j}$. The resulting combination might not be sensible since the conjunction of the guards could be inconsistent. The combination of the assignments $a_{1,i}\,;a_{2,j}$ might also be causally inconsistent and proof obligations have to be generated to ensure that such combinations do not occur. The dependency analysis in the case of synchronous composition is similar to that for a single module with the restriction that only cycles involving variables from both modules need be considered.

The consistency and dependency analysis for combinations of guarded transitions in a synchronous composition is similar to that for guarded initializations. In this manner, the synchronous composition $M_1 \,||\, M_2$ of two modules M_1 and M_2 can be expressed as a single module combining the definitions, initializations, and transitions from the individual modules. If there are n_1 guarded commands in M_1 and n_2 in M_2, the composition $M_1 \,||\, M_2$ could have up to $n_1 * n_2$ guarded commands. Thus it is not always feasible to expand out the module corresponding to such a composition. The expectation is that this will rarely be necessary since the modules can be individually analyzed and the properties composed.

The semantics of asynchronous composition of two modules is given by the conjunction of the initializations and the interleaving of the transitions of the two modules. For this purpose, the definitions in M_1 and M_2 must first be eliminated by including them in the guarded initializations and transitions. The module corresponding to M_1 `[]` M_2 is obtained by combining the initializations as in synchronous composition and taking the union of the transition definitions and the guarded transitions. The combination of initializations can generate proof obligations but there are no new proof obligations arising from the union of the module transitions.

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.

The causality analysis for synchronous multicompositions is carried out inductively by unfolding the multicomposition into a composition of a single module and a smaller multicomposition.

5.4 Module Declarations

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. Parametric modules allow the use of logical (state-independent) and type parameterization in the definition of modules. A parametric module is defined as

$$\textit{ModuleDeclaration} \quad := \quad \textit{Identifier}[\,\texttt{[}\,\textit{VarDecls}\texttt{]}\,]\;\texttt{:}\quad \texttt{MODULE = } \textit{Module}$$

Parametric modules allow modules to be defined with some open parameters that can be instantiated when the module is used.

Chapter 6

SAL Contexts

The language so far can describe transition system modules but has no way of declaring new types or constants or asserting properties of these modules. The SAL context language provides the framework for declaring types, constants, modules, and module properties. Below we present the syntax for contexts containing declarations for constants, types, modules, assertions, and other (imported) contexts. SAL contexts are read from left to right, top to bottom, and an entity must be declared before it is referenced.[1]

There is no name overloading in SAL. An unqualified name always refers to the local context. Qualified names must provide both the context and the parameters. Because of this, explicit importings are not needed.[2]

$$
\begin{array}{rcl}
\mathit{Context} & := & \mathit{Identifier}\ [\ \{\mathit{Parameters}\}\ ]\ :\ \texttt{CONTEXT}\ =\ \mathit{ContextBody} \\
\mathit{Parameters} & := & [\ \mathit{TypeDecls}\ ]\ \texttt{;}\ \{\mathit{VarDecls}\}^{*}_{,} \\
\mathit{TypeDecls} & := & \{\mathit{Identifier}\}^{+}_{,}\ \texttt{: TYPE} \\
\mathit{ContextBody} & := & \texttt{BEGIN}\ \mathit{Declarations}\ \texttt{END} \\
\mathit{Declarations} & := & \{\mathit{Declaration}\ \texttt{;}\}^{+} \\
\mathit{Declaration} & := & \mathit{ConstantDeclaration} \\
 & | & \mathit{TypeDeclaration} \\
 & | & \mathit{AssertionDeclaration} \\
 & | & \mathit{ContextDeclaration} \\
 & | & \mathit{ModuleDeclaration} \\
\mathit{ConstantDeclaration} & := & \mathit{Identifier}\ [\ \texttt{(}\,\mathit{VarDecls}\texttt{)}\ ]\ \texttt{:}\ \mathit{Type}\ [\ \texttt{=}\ \mathit{Expression}\ ] \\
\mathit{TypeDeclaration} & := & \mathit{Identifier}\ \texttt{: TYPE}\ [\ \texttt{=}\ \mathit{TypeDef}\ ] \\
\mathit{AssertionDeclaration} & := & \mathit{Identifier}\ \texttt{:}\ \mathit{AssertionForm}\ \texttt{=}\ \mathit{AssertionExpression} \\
\mathit{AssertionForm} & := & \texttt{OBLIGATION}\ |\ \texttt{CLAIM}\ |\ \texttt{LEMMA}\ |\ \texttt{THEOREM} \\
\mathit{ContextDeclaration} & := & \mathit{Identifier}\ \texttt{: CONTEXT =}\ \ \mathit{Identifier}\{\mathit{ActualParameters}\} \\
\mathit{ActualParameters} & := & \{\mathit{Type}\}^{*}_{,}\ \texttt{;}\ \{\mathit{Expression}\}^{*}_{,}
\end{array}
$$

[1] For those readers familiar with PVS, a SAL context is very similar to a PVS theory, but with different sets of allowable declarations.

[2] We are considering adding IMPORTINGs for convenience in the concrete language, but the parser should always be able to generate fully qualified names in the abstract syntax. See Section 8.2.1

6.1 Context Parameters

Context parameters allow for generic contexts that may be used from other contexts with different instances. Thus a context may be parameterized by a positive integer `N` that gives the number of processes, and a modelchecker may instantiate this to 6, in order to make it finite.

Within the given context, parameter types are treated as uninterpreted types, and parameter variables are treated as uninterpreted constants. Note that distinct type parameters are treated as distinct types, although they may be instantiated to the same type.

6.2 Constant Declarations

The simplest constant declaration provides an uninterpreted constant, e.g.,

```
c: INTEGER
```

Note that because all types must be nonempty, no proof obligation will be generated for the constant, though there may be one generated for the type.

Constant declarations may also provide a definition:

```
n: INTEGER = 3
f: [INTEGER -> [INTEGER -> INTEGER]] =
   LAMBDA (x: INTEGER): LAMBDA (y: INTEGER): x + n * y
```

A defining form may be used, which is usually more readable:

```
f(x: INTEGER): [INTEGER -> INTEGER]] =
   LAMBDA (y: INTEGER): x + n * y
```

Although higher-order functions are supported, only the top-level `LAMBDA` may be turned into a defining form. This is not much of an inconvenience, since higher-order functions are not often needed in transition system specifications.

Constant declarations may also be recursive. This is implicit, and the system must be able to determine the measure in order to generate the proper termination obligation:[3]

```
fact(n: NATURAL): NATURAL =
 IF n = 0 THEN 1 ELSE n * fact(n - 1)
```

6.3 Context Declarations

A *ContextDeclaration* provides an abbreviation, e.g., instead of writing

```
lem: LEMMA mycontext{int; 13}!f(3) = mycontext{int; 13}!f(4)
```

[3] As discussed in Section 8.5, this will probably change in the future.

One would write

```
mc: CONTEXT = mycontext{int; 13}
lem: mc!f(3) = mc!f(4)
```

6.4 Assertion Declarations

Assertion expressions allow properties to be stated. In the simplest case these are just boolean-valued expressions, which are thus just logical formulas. The *ModuleModels* form allows properties of modules to be stated. Note that the syntax says nothing about the possible temporal operators; this is defined in a separate context. A *ModuleImplements* assertion M_C `IMPLEMENTS` M_A, says that any possible behavior of M_C is also a behavior of M_A. This allow refinement and abstraction relations to be specified.

$$
\begin{array}{rcl}
\mathit{AssertionExpression} & := & \mathit{ModuleAssertion} \mid \mathit{PropositionalAssertion} \mid \mathit{QuantifiedAssertion} \mid \mathit{Expression} \\
\mathit{ModuleAssertion} & := & \mathit{ModuleModels} \mid \mathit{ModuleImplements} \\
\mathit{ModuleModels} & := & \mathit{Module}\ \texttt{|-}\ \mathit{Expression} \\
\mathit{ModuleImplements} & := & \mathit{Module}\ \texttt{IMPLEMENTS}\ \mathit{Module} \\
\mathit{PropositionalAssertion} & := & \mathit{PropOp}\ \texttt{(}\ \mathit{AssertionExpression}\ \texttt{,}\ \mathit{AssertionExpression}\ \texttt{)} \\
 & & \mid\ \texttt{NOT}\ \texttt{(}\ \mathit{AssertionExpression}\ \texttt{)} \\
\mathit{QuantifiedAssertion} & := & \mathit{Quantifier}\ \texttt{(}\ \mathit{VarDecls}\ \texttt{)}\ \texttt{:}\ \mathit{AssertionExpression} \\
\mathit{PropOp} & := & \texttt{AND} \mid \texttt{OR} \mid \texttt{=>} \mid \texttt{<=>}
\end{array}
$$

Chapter 7

Another SAL Example: Mutual Exclusion

We show another example SAL specification: a variant of Peterson's mutual exclusion algorithm [6]. Here the state of the `process` module consists of the controlled variables corresponding to its own program counter `pc1` and boolean variable `x1`, and the observed variables are the corresponding `pc2` and `x2` of the other process. Initially `process` is `sleeping`. The `process` module is parameterized with a boolean `tval` argument.

The `system` is then the asynchronous composition of two processes, where the variables of the `process[TRUE]` have been renamed in order to make them compatible with `process[FALSE]`, i.e., the outputs of one are wired to the inputs of the other.

The main property of this algorithm is assertion `mutex`, which asserts the safety property that in `system`, it is always true that the two processes are not both in their `critical` sections. The assertion language used here is LTL. `G` represents the henceforth modality and `F` represents eventually. Other properties are given, for example `livenessbug1` states the liveness property that it is always possible for `process[FALSE]` to reach its `critical` section. This property is false, because there is no fairness built-in to SAL, so `process[TRUE]` can simply run forever. The same is true for `livenessbug2`. The other liveness properties bring in fairness constraints explicitly, and are provable.

```
peterson: CONTEXT =
BEGIN
  PC: TYPE = {sleeping, trying, critical};

  process [tval : BOOLEAN]: MODULE =
    BEGIN
      INPUT pc2 : PC
      INPUT x2 : BOOLEAN
      OUTPUT pc1 : PC
      OUTPUT x1 : BOOLEAN
      INITIALIZATION pc1 = sleeping
      TRANSITION
        [
```

```
        wakening:
          pc1 = sleeping --> pc1' = trying; x1' = x2 = tval
        []
        entering_critical:
          pc1 = trying AND (pc2 = sleeping OR x1 = (x2 /= tval))
          --> pc1' = critical
        []
        leaving_critical:
          pc1 = critical --> pc1' = sleeping; x1' = x2 = tval
       ]
  END;

 system: MODULE =
    process[FALSE]
    []
    RENAME pc2 TO pc1, pc1 TO pc2,
           x2 TO x1, x1 TO x2
     IN process[TRUE];

 mutex: THEOREM system |- G(NOT(pc1 = critical AND pc2 = critical));

 invalid: THEOREM system |- G(NOT(pc1 = trying AND pc2 = critical));

 livenessbug1: THEOREM system |- G(F(pc1 = critical));

 livenessbug2: THEOREM system |- G(F(pc2 = critical));

 liveness1: THEOREM system |- G(pc2 = trying => F(pc2 = critical));

 liveness2: THEOREM system |- G(pc1 = trying => F(pc1 = critical));

 liveness3: THEOREM system |- G(F(pc1 = trying)) => G(F(pc1 = critical));

 liveness4: THEOREM system |- G(F(pc2 = trying)) => G(F(pc2 = critical));

END
```

Note: the assertions in the THEOREMS are not technically type correct, because the LTL operators `G` and `F` are not defined locally. They are built-in to the SALENV tools descibed in `http://sal.csl.sri.com/salenv.html`. To make this valid would require defining a LTL context, then including the context name (along with the parameters) in the references to `G` and `F`. In addition, `G` and `F` technically operate on path formulas, so giving them a type that allows them to operate on boolean formulas is a problem. Sections 8.2.1 and 8.3 address these issues.

Chapter 8

Future Work

This language manual and SAL itself are a work in progress.

8.1 SAL as an Intermediate Language

SAL was originally intended to be an intermediate language, but as work progressed it became clear that many users were going to use the language directly, not as an internal representation for some front end. In addition, the desire to create a SAL tool bus, and to keep it language independent, led to the decision to create an abstract syntax in XML, and treat that as the intermediate form. XML was chosen because it is widely used, extensible, and most popular programming languages have direct support for reading and representing XML datas structures.

We have thus defined an abstract syntax in XML by a document type description (DTD), available at `http://sal.csl.sri.com/documentation.html`. The SAL parser (`http://sal.csl.sri.com/salparser.html`) simply reads the concrete syntax and generates an XML file that satisfies the SAL DTD. The separation of the abstract and concrete syntax has many benefits, in that the concrete language may be extended in various ways for convenience, yet map to a more restricted set of data structures, which means that tools do not need to be modified everytime something is added to the concrete language. In addition, users may create their own concrete languages, as long as there is a mapping to the SAL XML abstract syntax.

A general rule followed by the SAL parser is that any transformations done by the parser in creating the abstract structures must, in principle, be invertible. In other words, it should be possible to prettyprint the abstract syntax and get back the original form, ignoring whitespace.

8.2 A SAL Prelude

The language described here has many built-ins, such as `INTEGER`, `AND`, `+`, etc. In principle, these could be defined in a separate context, and imported. This would make the language cumbersome, so instead they were built-in. In our opinion a better choice would be to define these in a prelude, that is automatically imported and provides types, constants, and lemmas. For example, various logics such as CTL, CTL*, and LTL can be defined in the prelude, and even given semantics.

The main advantage of a prelude is that it separates the core language from entities built on the core language. This means that changes to the language can be kept to a minimum, while still allowing new types and constants to be treated as if they were built-in. This is simlar to the separation of the core language of C from its numerous libraries.

Any given SAL tool should be able to read the prelude, and build a symbol table, so it should not be difficult to support.

8.2.1 Libraries, Importings, and Logics

The language defined here may only refer to names outside the context using the fully qualified name. This is helped somewhat with *ContextDeclaration*s, but if a large hierarchy is built up, even this will lead to specifications that are difficult to write and to read. In moving away from the view that this is solely an intermediate language, we feel that the addition of libraries, importings, and logics would be useful, at least in the concrete language.

A library is really just an extension of the idea of a prelude, and allows sets of contexts to be defined in a separate directory, and packaged for broad use and distribution, as with PVS libraries.

Importing a context instance allows the names from that context to be used without a qualifier. There would be restrictions: name conflicts will not be allowed, even if the entities are not comparable. If a referenced name has an associated declaration both in the current context and an imported one, the local one always is used. If a referenced name is common to two separate contexts (including different instances of the same context), then it is an error, and the name must be fully qualified.

Importing a logic is similar, but the idea here is that a logic may be parameterized with the transition system defined by a module, and many instances may be needed for multiple module expressions. A logic declaration would be similar to an importing, but the information needed to instantiate it is derived from the module assertions, for example, a `CTL` context could be defined,

```
ctl{state: TYPE;
    init: [state -> BOOLEAN],
    trans: [[state, state] -> BOOLEAN]} : CONTEXT =
  ...
```

this can then be used as follows:

```
LOGIC ctl
asafety: LEMMA async_bak |- AG(NOT (pc1 = l3 AND pc2 = l3));
```

rather than the error prone and unreadable expanded form:

```
asafety: LEMMA
  async_bak |- ctl{async_bak.STATE;async_bak.INIT,async_bak.TRANS}!
                 AG(NOT (pc1 = l3 AND pc2 = l3));
```

This would address the problem with the peterson specification described in Chapter 7.

Note that in principle a parser for the concrete language can parse the imported contexts, produce name conflict errors, and generate XML files that do not have any importings. This kind of tranformation means that the abstract syntax can be kept minimal, while allowing the concrete syntax to be much more convenient and readable.

8.3 Conversions

The preceding section described a CTL formula. In CTL, `AG` is a predicate transformer, of type `[[STATE -> BOOLEAN] -> [STATE -> BOOLEAN]]`. But of course `NOT` and `AND` are `BOOLEAN` operators, so there is a mismatch. PVS provides a mechanism, called *lambda conversion*, that is very effective in lifting such operators, in this case the result would be as follows:

```
AG(LAMBDA (s: STATE): NOT (pc1(s) = l3 AND pc2(s) = l3))
```

Of course, if SAL was only intended for CTL, this could simply be built-in, but SAL is intended to be logic-independent. For LTL, the formulas are path formulas, not state formulas. In fact, LTL often treats state formulas as path formulas. So a more comprehensive treatment is needed, and conversions look like a reasonable approach.

8.4 Empty Types

As discussed in the `adder` example in Chapter 2, the restriction to nonempty types can actually get in the way of succinct specifications. In the adder case, there is no real problem with having the empty type, it simply means that the `onebitadder` is composed with a module that always skips. Thus in a *MultiSynchronous* composition if the index type is empty, the result is a module with no state variables that always skips. If it is a *MultiAsynchronous* composition, the result is an empty module with no transitions (i.e., it is deadlocked). PVS allows empty types, and there is no logical difficulty. One must, of course, be careful with applying logical rules, in particular those involving quantifiers. For example, one can usually ignore quantifiers whose bound variables do not occur in the underlying expression, but if empty types are allowed, this is unsound. Thus `FORALL (x: T): FALSE` could naively be reduced to `FALSE`, but if the type `T` is empty it is actually vacuously `TRUE`. Also allowing a type to be nonempty means that the declaration of a constant may entail a nonemptiness obligation on the type.

8.5 Recursive Function Termination

In PVS, recursive functions must include a measure, and optionally a well-founded ordering. In earlier discussions of SAL it was thought that functions would be simple enough that the typechecker would always be able to figure out the measure, but this is clearly not true; even the usual definition for GCD requires a measure on the difference of the arguments, and it is not at all clear how a typechecker would be able to determine this. In the future we plan to allow a measure and ordering to be optionall provided by the user.

8.6 State-Dependent Types

Types in SAL are static, but there are situations where having a type that depends on the state is more expressive. In effect, it means that the type can change as the system progresses. The typechecker would generate proof obligations that in every reachable state all state variables satisfy their types. State-dependent types might be useful, for example, in modeling adjustable arrays, where an array may change size dynamically, but it is preferable to prove that a runtime arrays-bound check is not necessary.

Bibliography

[1] R. Alur, T. A. Henzinger, F. Y. C. Mang, S. Qadeer an d S. K. Rajamani, and S. Tasiran. MOCHA: Modularity in model checking. In Alan J. Hu and Moshe Y. Vardi, editors, *98*, volume 1427, pages 521–525. Vancouver, Canada, June 1998.

[2] G. J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, 1991.

[3] Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Boston, MA, 1993.

[4] Ralph Melton and David L. Dill. *Murϕ Annotated Reference Manual*. Computer Science Department, Stanford University, Stanford, CA, March 1993.

[5] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS Language Reference*, September 1999.

[6] G. L. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 12(3):115–116, June 1981.

Index