The SAL Language

N.Shankar[1], David Dill[2], Thomas Henzinger[3], and Sam Owre[1]

Draft Technical Report
March, 2001

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.

[1]Computer Science Laboratory, SRI International, Menlo Park, California
[2]Stanford University, Stanford, California
[3]University of California at Berkeley

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 an intermediate language for describing transition systems. This language serves as the *target* for translators that extract the transition system description for popular programming languages such as Esterel, Java, and Statecharts. The intermediate language also serves as a common *source* for driving different analysis tools through translators from the intermediate language to the input format for the tools, and from the output of these tools back to the SAL intermediate language.

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 an intermediate language for describing transition systems. This language serves as the *target* for translators that extract the transition system description for popular programming languages such as Esterel, Java, and Statecharts. The intermediate language also serves as a common *source* for driving different analysis tools through translators from the intermediate language to the input format for the tools, and from the output of these tools back to the SAL intermediate 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 within the SAL intermediate format.

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 hygiene:** 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. Correspondingly, tools such as parsers and typecheckers can be made similarly parametric so that they can be constructed by plugging together the tools for the corresponding sublanguages.

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.

6. **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.

7. **Asynchronous composition:** Modules that are driven by independent clocks are modelled by means of interleaving the atomic transitions of the individual modules.

We present the SAL intermediate 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 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 intermediate language is not that different from the input languages used by various other verification tools such as SMV, Murphi, Mocha, and SPIN. 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

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*.

As noted, the SAL intermediate language needs to be liberal in order to accomodate 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, semicolon, single quote, exclamation point, and hash. The special symbols are `(`, `)`, `[`, `]`, `{`, `}`, `%`, `,`, `.`, `;`, `:`, `'`, `!`, `?`, `_`, and `#`. Tokens can be separated by *WhiteSpace*, which consists of spaces, tabs, carriage returns, and line feeds.

$$
\begin{array}{rcl}
\mathit{SpecialSymbol} & := & \texttt{(} \mid \texttt{)} \mid \texttt{[} \mid \texttt{]} \mid \texttt{\{} \mid \texttt{\}} \mid \texttt{\%} \mid \texttt{,} \mid \texttt{.} \mid \texttt{;} \mid \texttt{:} \mid \texttt{'} \mid \texttt{!} \mid \texttt{\#} \mid \texttt{?} \mid \texttt{\_} \\
\mathit{Letter} & := & \texttt{a} \mid \ldots \mid \texttt{z} \mid \texttt{A} \mid \ldots \mid \texttt{Z} \\
\mathit{Digit} & := & \texttt{0} \mid \ldots \mid \texttt{9} \\
\mathit{Identifier} & := & \mathit{Letter}\{\mathit{Letter} \mid \mathit{Digit} \mid \texttt{?} \mid \texttt{\_}\}^{*} \\
 & \mid & \{\mathit{Opchar}\}^{+} \\
\mathit{Numeral} & := & \{\mathit{Digit}\}^{+}
\end{array}
$$

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.

2.1 Types

Minimally, the SAL intermediate language should support the basic types such as booleans, scalars, integers and integer subranges, records, and arrays. A bit-vector is just an array of Booleans. The grammar for types is given by

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

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 subrange bounds, and the unfolding of the definitions of defined types that are not scalar types or datatypes. Equivalence for types that are defined to be scalar types and datatypes is just name equivalence. Name equivalence is not a simple concept because compound names consist of the context name, actual 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.

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. 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 type `NATURAL` is merely an abbreviation for `SUBRANGE 0 TO _]`. Any subrange is a subtype of a larger subrange. It is also a subtype of `INTEGER`. An array type A is a subtype of another array 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 the set of labels is identical and the type A_i corresponding to each label l_i in A is a subtype of the corresponding type B_i in B.

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[1] as

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

2.2 Expressions

Expressions in the expression 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 and record selection and updates. Conditional (if-then-else) expressions are also part of the expression language.

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

$$
\begin{array}{rcl}
\mathit{NextVariable} & := & \mathit{Identifier}\,\texttt{'} \\
\mathit{Argument} & := & \texttt{(}\{\mathit{Expression}\}^{+}_{,}\texttt{)} \\
\mathit{Function} & := & \mathit{Expression} \\
\mathit{Application} & := & \mathit{Function}\ \mathit{Argument} \\
\mathit{InfixApplication} & := & \mathit{Expression}\ \mathit{Identifier}\ \mathit{Expression} \\
\mathit{ArraySelection} & := & \mathit{Expression}\texttt{[}\mathit{Expression}\texttt{]} \\
\mathit{RecordSelection} & := & \mathit{Expression}\texttt{.}\mathit{Identifier} \\
\mathit{TupleSelection} & := & \mathit{Expression}\texttt{.}\mathit{Numeral} \\
\mathit{RecordEntry} & := & \mathit{Identifier}\ \texttt{:=}\ \mathit{Expression} \\
\mathit{RecordLiteral} & := & \texttt{(\#}\{\mathit{RecordEntry}\}^{+}_{,}\texttt{\#)} \\
\mathit{TupleLiteral} & := & \mathit{Argument} \\
\mathit{UpdatePosition} & := & \{\mathit{Argument} \mid \texttt{[}\mathit{Expression}\texttt{]} \mid \texttt{.}\mathit{Identifier} \mid \texttt{.}\mathit{Numeral}\}^{+} \\
\mathit{Update} & := & \mathit{UpdatePosition}\ \texttt{:=}\ \mathit{Expression} \\
\mathit{UpdateExpression} & := & \mathit{Expression}\ \texttt{WITH}\ \mathit{Update} \\
\mathit{VarDecl} & := & \{\mathit{Identifier}\}^{+}_{,}\ \texttt{:}\ \mathit{Type} \\
\mathit{VarDecls} & := & \{\mathit{VarDecl}\}^{+}_{,} \\
\mathit{IndexVarDecl} & := & \mathit{Identifier}\ \texttt{:}\ \mathit{IndexType} \\
\mathit{ArrayLiteral} & := & \texttt{[[}\mathit{IndexVarDecl}\texttt{]}\ \mathit{Expression}\texttt{]} \\
\mathit{LambdaAbstraction} & := & \texttt{LAMBDA (}\mathit{VarDecls}\texttt{) :}\ \mathit{Expression} \\
\mathit{Quantifier} & := & \texttt{FORALL} \mid \texttt{EXISTS} \\
\mathit{QuantifiedExpression} & := & \mathit{Quantifier}\ \texttt{(}\mathit{VarDecls}\texttt{) :}\ \mathit{Expression} \\
\mathit{LetDeclarations} & := & \{\mathit{Identifier}\ \texttt{:}\ \mathit{Type}\ \texttt{=}\ \mathit{Expression}\}^{+}_{,} \\
\mathit{LetExpression} & := & \texttt{LET}\ \mathit{LetDeclarations}\ \texttt{IN}\ \mathit{Expression} \\
\mathit{SetExpression} & := & \texttt{\{}\ \mathit{Identifier}\ \texttt{:}\ \mathit{Type}\ \texttt{|}\ \mathit{Expression}\ \texttt{\}} \\
 & \mid & \texttt{\{}\ \{\mathit{Expression}\}^{+}_{,}\ \texttt{\}} \\
\mathit{Conditional} & := & \texttt{IF}\ \mathit{Expression}\ \mathit{ThenRest} \\
\mathit{ThenRest} & := & \texttt{THEN}\ \mathit{Expression} \\
 & & [\mathit{ElsIf}] \\
 & & \texttt{ELSE}\ \mathit{Expression}\ \texttt{ENDIF} \\
\mathit{ElsIf} & := & \texttt{ELSIF}\ \mathit{Expression}\ \mathit{ThenRest}
\end{array}
$$

Expression := *Name*
| *NextVariable*
| *Numeral*
| *Application*
| *InfixApplication*
| *ArraySelection*
| *RecordSelection*
| *TupleSelection*
| *UpdateExpression*
| *LambdaAbstraction*
| *QuantifiedExpression*
| *LetExpression*
| *SetExpression*
| *ArrayLiteral*
| *RecordLiteral*
| *TupleLiteral*
| *Conditional*
| (*Expression*)

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`. Though the PVS notion of predicate subtypes is not a part of the SAL type system, division is treated specially and the type rules generate a proof obligation requiring nonzero divisors. 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 proof obligations generated during typechecking are called type correctness conditions (TCCs). In addition to arithmetic opreations such as division, the sources of TCCs include expressions of subrange types, recursive datatypes, recursive definitions, and type nonemptiness.

An expression without nextvariables 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 next variables. 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 next variables 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 next variables, we write the meaning of e as $\mathcal{M}[\![e]\!]_r$.

Chapter 3

The Transition Language

A transition system *module* consists of a *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 Section 4.

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'`.

A variable having a type that is an aggregate structure like a record or array is also treated as a collection of variables. Thus it is possible to have left-hand sides to assignments that are array selections or record selections. When an array selection occurs on the left-hand side, the index must not contain any state variables.

3.1 Definitions

Definitions can be used to specify the trajectory of variables in a computation by initialization and transition rules that are given variable-wise for each of 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. The left-hand side of definitions is given by the nonterminal *Lhs*.

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

There are two kinds of definitions. The simpler of these are invariant definitions that are of the form

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

The right-hand side expression must not contain any *NextVariable* occurrences, i.e., it must be a *current expression*. The interpretation of such a definition is that the defining equation is an invariant and is hence true in every state of the computation.

The second kind of variable definition is given in terms of an initialization and a transition equation. An initialization definition is just a *SimpleDefinition* that occurs in the `INITIALIZATION` section of transition system. Whereas a transition equation given by the nonterminal *TransDefinition* defines a *NextVariable* on the left-hand side in terms of an expression that can contain *NextVariable* occurrences. A *TransDefinition* or 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.

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

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, $GC(\texttt{X},\texttt{Y})$ for the above definitions of `X` yields `A`. Then $GC^*(\texttt{X},\texttt{Y})$ 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 $GC^*(\texttt{X},\texttt{Y})$ is just $GC(\texttt{X},\texttt{Y}) \wedge GC(\texttt{X},\texttt{Z}) \wedge GC(\texttt{Z},\texttt{Y})$. Thus, in the above definitions of `X` and `Y`, $GC^*(X, X)$ is $\texttt{A} \wedge \neg\texttt{A}$. The dependency conditions can be used to generate the conditions $C_{\texttt{X}}$ under which a variable `X` could depend on itself. For such dependency loops to be avoided, the condition $C_{\texttt{X}}$ must be shown to be invariantly false in the transition system. In the above example, $C_{\texttt{X}}$ would be the obviously unreachable assertion $\texttt{A} \wedge \neg\texttt{A}$. The dependency analysis generates proof obligations to this effect. A similar dependency analysis can be carried out for initialization definitions and transition definitions.

3.2 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. The transitions specified by them are deterministic. 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 guarded formula and an assignment part. The guard is a boolean expression in the current controlled (local, global, and output) variables and current and next input variables. The assignment part is a list of equalities between a left-hand side next variable and a right-hand side expression in current and next variables.

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

Note that both the initializations and transitions are specified by guarded assignments. No variable that is defined in a definition can be assigned in either a guarded initialization or transition. The well-formedness checks for each guarded initialization are that the guard must not contain any controlled variables so that they are boolean conditions on the input variables, and the initialization assignments must contain exactly one assignment per controlled variable. The well-formedness checks on the guarded transitions are that the guard must not contain controlled next 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.

3.3 Semantics

We have already described how the semantics of expressions are given with respect to a model $\mathcal{M}$ that assigns meaning to the types, constant, and function symbols and an environment ρ that assigns values to the free logical variables in an expression. The semantics of a transition system is given by a Kripke model $\mathcal{K}$ consisting of:

- A state space Σ consists of valuations s for the input, local, global, and output variables.
- A set of initial states I.
- A transition relation between states N.

Given a program with definitions (initialization, invariant, and transition), guarded initializations g_0 `-->` a_0, and guarded transitions g_i `-->` a_i, the Kripke model $\mathcal{K}$ must be such that

1. Every state s in I satisfies
   - the invariant definitions
   - the initialization definitions, and

- the guard and assignments of some guarded initialization or the negation of each of the guards in a guarded initialization.

2. Every pair of states $\langle r, s \rangle$ in N satisfies

   - the invariant definitions
   - the transition definitions, and
   - the guard and assignments of some guarded assignment, or the negations of the guards in the guarded assignment and s leaves the controlled variables that are not specified by definitions unchanged from r.

Chapter 4

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. A simple module

has the form

$$
\begin{array}{rcl}
\mathit{NamedCommand} & := & [\mathit{Identifier}\texttt{:}]\mathit{GuardedCommand} \\
\mathit{MultiCommand} & := & \texttt{([] (}\mathit{VarDecls}\texttt{):}\ \mathit{SomeCommand}\texttt{)} \\
\mathit{SomeCommand} & := & \mathit{NamedCommand} \mid \mathit{MultiCommand} \\
\mathit{SomeCommands} & := & \{\mathit{SomeCommand}\}^{+}_{\texttt{[]}} \\
\mathit{DefinitionOrCommand} & := & \mathit{Definition} \\
 & \mid & \texttt{[}\ \mathit{SomeCommands}\ \texttt{]} \\
\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}\}^{+}_{;} \\
\mathit{BaseDeclaration} & := & \mathit{InputDecl} \\
 & \mid & \mathit{OutputDecl} \\
 & \mid & \mathit{GlobalDecl} \\
 & \mid & \mathit{LocalDecl} \\
 & \mid & \mathit{DefDecl} \\
 & \mid & \mathit{InitDecl} \\
 & \mid & \mathit{TransDecl} \\
\mathit{BaseDeclarations} & := & \{\mathit{BaseDeclaration}\}^{*} \\
\mathit{BaseModule} & := & \texttt{BEGIN} \\
 & & \mathit{BaseDeclarations} \\
 & & \texttt{END} \\
\mathit{SynchronousComposition} & := & \mathit{Module}\texttt{||}\mathit{Module} \\
\mathit{AsynchronousComposition} & := & \mathit{Module}\texttt{[]}\mathit{Module} \\
\mathit{Hiding} & := & \texttt{LOCAL}\ \{\mathit{Identifier}\}^{+}_{,}\ \texttt{IN}\ \mathit{Module} \\
\mathit{NewOutput} & := & \texttt{OUTPUT}\ \mathit{VarDecls}\ \texttt{IN}\ \mathit{Module} \\
\mathit{NewVarDecls} & := & \{\mathit{InputDecl} \mid \mathit{OutputDecl} \mid \mathit{GlobalDecl} \mid \mathit{LocalDecl}\}^{+}_{;} \\
\mathit{Renames} & := & \{\mathit{Lhs}\ \texttt{TO}\ \ \mathit{Lhs}\}^{+}_{,} \\
\mathit{Renaming} & := & [\texttt{WITH}\ \mathit{NewVarDecls}] \\
 & & \texttt{RENAME}\ \mathit{Renames}\ \texttt{IN}\ \mathit{Module} \\
\mathit{ModuleName} & := & \mathit{Name}[\texttt{[}\{\mathit{Expression}\}^{+}_{,}\texttt{]}] \\
\mathit{Module} & := & \mathit{BaseModule} \\
 & \mid & \mathit{ModuleName} \\
 & \mid & \mathit{SynchronousComposition} \\
 & \mid & \mathit{AsynchronousComposition} \\
 & \mid & \mathit{MultiSynchronous} \\
 & \mid & \mathit{MultiAsynchronous} \\
 & \mid & \mathit{Hiding} \\
 & \mid & \mathit{NewOutput} \\
 & \mid & \mathit{Renaming} \\
 & \mid & \mathit{ObserveModule} \\
 & \mid & \texttt{(}\mathit{Module}\texttt{)}
\end{array}
$$

A base module identifies the pairwise distinct sets of input, output, and local variables. The initialization and transition sections are marked by the keywords `INITIALIZATION` and `TRANSITION`, respectively. Output and global variables can be made local by the `LOCAL` 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 declared by `INPUT`, `OUTPUT`, `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.

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. For a synchronous composition $M_1 \,||\, M_2$ to be well-formed, the modules must contain no global variables so that G_1 and G_2 have to be empty. There is no such restriction on the global variables in an asynchronous composition. Variables with the same identifier are treated as identical. 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 syntax for N-fold synchronous and asynchronous composition (or a *multicomposition*) is specified below.

$$\begin{array}{rcl} \mathit{MultiSynchronous} & := & \texttt{(||}\ \texttt{(}\mathit{Identifier}\ \texttt{:}\ \mathit{IndexType}\texttt{)}\ \texttt{:}\ \mathit{Module}\texttt{)} \\ \mathit{MultiAsynchronous} & := & \texttt{([]}\ \texttt{(}\mathit{Identifier}\ \texttt{:}\ \mathit{IndexType}\texttt{)}\ \texttt{:}\ \mathit{Module}\texttt{)} \end{array}$$

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.

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 parametrization in the definition of modules. A parametric module is defined as

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

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

Chapter 5

Eliminating Guarded Commands

We identify a fragment called L0 of the language consisting of modules where the `INITIALIZATION` and `TRANSITION` sections are empty so that the module transitions are given solely in terms of definitions. This fragment is useful since it supports certain kinds of analysis more directly. As we already noted, it is easy to eliminate definitions in favor of guarded commands. The converse is also possible. The SAL language described above can be translated into the L0 fragment by systematically eliminating guarded commands.

The guarded command presentation of a module in the L1 language can be translated into L0 form while preserving the structure of the state. This is done by adding an extra variable that captures the choice of the true guard. Let $\tau_1, \ldots, \tau_N$ be the transitions of the module with guards $g_1, \ldots, g_N$. A boolean array `guard` over the subrange `[1..N]` is used to save the evaluations of the guards $g_1, \ldots, g_N$. A variable `start` over the subrange `[1..N]` can be first nondeterministically selected to indicate the first guard in the sequence of guard evaluations. Another variable `chosen`, also of subrange `[1..N]` type, is computed as follows

```
guard' IN {g_1, ..., g_N} ;
chosen' = (IF guard[chosen] THEN chosen
                 ELSE IF guard[mod(chosen + 1, N+1)]
                       THEN mod(chosen + 1, N+1)
                        ...
               ENDIF) ;
X' = (IF chosen = 1 THEN e_1
          ELSE IF chosen = 2 THEN e_2
           ...
          ENDIF)
```

Then a variable `X` is assigned by collecting together all the assignments for `X` from each of the `N` guarded commands in one conditional expression that is controlled by the value of `chosen`.

This translation works only when either no next variables occur in guards or there is a uniform causal ordering on the variables. Otherwise, the guards cannot be evaluated in the assignment to the variable `guard` without creating a causal dependency cycle.

Chapter 6

SAL Contexts

The language so far can only 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. We leave the syntax for module properties open for now but present the syntax for contexts containing declarations for constants, types, and other (imported) theories. Sal contexts are read from left to right, top to bottom, and an entity must be declared before it is referenced.

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

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

One would write

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

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

Chapter 7

Assertion Expressions

Assertion expressions allow properties of modules to be stated. The syntax says nothing about the possible temporal operators; this is defined in a separate context (perhaps a prelude?).

AssertionExpression	:=	*ModuleAssertion* | *PropositionalAssertion* | *QuantifiedAssertion*
ModuleAssertion	:=	*ModuleModels* | *ModuleImplements* | *ModuleRefines*
ModuleModels	:=	*Module* `|-` *Expression*
ModuleImplements	:=	*Module* `IMPLEMENTS` *Module*
ModuleRefines	:=	*Module* `REFINES` *Module*
PropositionalAssertion	:=	*PropOp* `(` *AssertionExpression* `,` *AssertionExpression* `)`
		| `NOT (` *AssertionExpression* `)`
QuantifiedAssertion	:=	*Quantifier* `(` *VarDecls* `) :` *AssertionExpression*
PropOp	:=	`AND` | `OR` | `=>` | `<=>`

Chapter 8

Abstraction and Refinement

These are represented by means of the module language itself so that abstractions and refinements are given by a kind of (vertical) composition that indicates how certain variables are mapped. This is different from the horizontal composition used in module composition.

Given a module M, we construct an abstraction of it by essentially taking a view of M through another observer module MA. M is a module with input I, output O, global G, and local variables L. MA is a module with abstract variables IA, OA, GA, and LA, and definitions of these variables in terms of each other as well as the concrete variables I, O, G, L, which are essentially all inputs to MA. The rules are that

LA is defined solely in terms of L, IA in terms of I and L, OA in terms of O and L, and GA in terms of G, O, I, and L.

An observer module is one that contains no guarded commands so that all variables are defined solely through definitions given in terms of the observed variables (and observing variables).

If there is another module N that is similarly abstracted by NA, then it is easy to see that the composition of MA and NA is sensible as long as the definitions coincide on the common variables. This is because the inputs of MA and NA can be connected. There is a problem with the synchronous composition of MA and NA when there are shared global variables, but this is acceptable since the definitions coincide.

Furthermore, MA composed with NA is an abstraction of M composed with N.

The notation for such composition is OBSERVE M WITH MA.

We would like to claim that such a composition of a concrete module and actually implements another directly constructed abstract module.

For this we have a relation M IMPLEMENTS N, where the interface (I,O, G, L) of N must be a subset of that of N. The meaning is that under all possible behaviors of the variables outside of M, every behavior of M is a behavior of N.

This gives rise to three new constructs: the *ObserverModuleDeclaration*, the *ObserveModule*, and the *ImplementsAssertion*.

ObserverModuleDeclaration := *Identifier* [`[` *VarDecls* `]`] :
`OBSERVER_MODULE =` *ObserverModule*

ObserveModule := `OBSERVE` *Module* `WITH` *ObserverModule*

ObserverModule	:=	*BaseObserverModule*
		| *ObserverModuleName*
		| *SynchronousObserverComposition*
		| *AsynchronousObserverComposition*
		| *MultiSynchronousObserver*
		| *MultiAsynchronousObserver*
		| *ObserverHiding*
		| *ObserverNewOutput*
		| *ObserverRenaming*
		| `(` *ObserverModule* `)`
BaseObserverModule	:=	`BEGIN` *BaseObserverDeclarations* `END`
BaseObserverDeclarations	:=	$\{$*BaseObserverDeclaration*$\}^*$
BaseObserverDeclaration	:=	*ObservedDecl*
		| *InputDecl*
		| *OutputDecl*
		| *GlobalDecl*
		| *LocalDecl*
		| *DefDecl*
ObservedDecl	:=	`OBSERVED` *VarDecls*
SynchronousObserverComposition	:=	*ObserverModule* `||` *ObserverModule*
AsynchronousObserverComposition	:=	*ObserverModule* `[]` *ObserverModule*
ObserverHiding	:=	`LOCAL` $\{$*Identifier*$\}_{,}^{+}$ `IN` *ObserverModule*
ObserverNewOutput	:=	`OUTPUT` *VarDecls* `IN` *ObserverModule*
ObserverRenaming	:=	[`WITH` *NewVarDecls*] `RENAME` *Renames* `IN` *ObserverModule*
ObserverModuleName	:=	*ModuleName*

Chapter 9

SAL Examples

9.1 A 3-bit Counter

We start with a simple example of a 3-bit counter taken from the `Mocha` reference manual. The design is meant to synchronously take a `tick` input and generate an `out` output signal on every eighth tick.

```
countercell: CONTEXT =
 BEGIN
  countercell : MODULE =
    LOCAL sum : boolean
    INPUT tick : boolean
    OUTPUT carryout : boolean
    INITIALIZATION
      sum = FALSE;
      carryout = FALSE
    TRANSITION
      [tick' -->  sum' = NOT sum;
                  carryout' = sum]
   END
 END

threebitcounter : MODULE =
  LOCAL out0, out1 IN
    (  (RENAME carryout TO out0 IN
          countercell)
    || (RENAME tick TO out0, carryout TO out1 IN
          countercell)
    || (RENAME tick TO out1, carryout TO out IN
          countercell)
    )
END
```

The `countercell` module is a single-bit counter that computes the `sum'` and `carryout'` bits every time `tick'` is `TRUE`. The module `threebitcounter` is defined to be a serial connection of three countercells with the input `tick` connected to the first countercell whose output `carryout` is renamed `out0` and is connected to the `tick` input of the second countercell. Similarly, the output `carryout` of the second countercell is renamed `out1` and connected to the `tick` input of the third countercell whose output is renamed `out`.

9.2 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`. 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`.

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

  Nbitadder [N : natural] : MODULE =
   WITH INPUT A, B : ARRAY SUBRANGE 0 TO N OF BOOLEAN, carryin: boolean;
        OUTPUT S, C : ARRAY SUBRANGE 0 TO N 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 : SUBRANGE 1 TO N):
          (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
```

9.3 A Mutual Exclusion Scheme

We show another example of a SAL specification of a variant of Peterson's mutual exclusion algorithm. Here the state of the 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.

```
mutex : CONTEXT =
 BEGIN
  PC: TYPE = {trying, critical, sleeping}
  mutex [tval:boolean] : MODULE =
   BEGIN
    INPUT  pc2: PC, x2: boolean
    OUTPUT pc1: PC, x1: boolean
    INITIALIZATION
      pc1 = sleeping
    TRANSITION
     [pc1 = sleeping -->
           pc1' = trying;
           x1' = (x2 = tval)
      []
      pc1 = trying AND (pc2 = sleeping OR x1 = (x2 /= tval)) -->
           pc1' = critical
      []
      pc1 = critical -->
           pc1' = sleeping;
           x1' = (x2 = tval)
     ]
   END
  ...
 END
```

Two instances of the above mutex module can be combined to produce a mutual exclusion algorithm.

```
 system : MODULE
  =
  LOCAL x1, x2
   (mutex[FALSE]
    ||
    (RENAME pc2 TO pc1, x2 TO x1,
           pc1 TO pc2, x1 TO x2
      mutex[TRUE]))
```

The above modules can also be combined asynchronously by using [] instead of || as the composition operator.

Chapter 10

Comparison with Related Work

We compare SAL to other transition system formalisms, notably UNITY, TLA, I/O automata, SMV, Murphi, and Reactive Modules.

UNITY. Chandy and Misra's UNITY is a notation for describing transition systems by means of guarded commands. A module state in UNITY is the global state shared by all the modules. Modules are composed by interleaving. The guarded commands are all assumed to be executed according to a weakly fair schedule. There is a simplified temporal logic for proving non-nested temporal properties such as invariance and progress of UNITY programs.

SAL state is more modular consisting of input, local, and output variables. This constrains the interference between modules and allows stronger properties to be derived at the module level. SAL has both synchronous and asynchronous (interleaving) composition. The assertion language and logic for SAL transition system properties has not yet been finalized but it will be richer than UNITY in allowing linear and branching-time temporal logics, mu-calculi, and various refinement relations between modules.

TLA. Lamport's TLA or the temporal logic of actions is an expressive temporal logic that allows transition system modules to be specified in terms of an initialization predicate, next-state relation, and a fairness constraint. Modules are composed by means of conjunction and variables in the state are hidden by means of existential quantification. The temporal logic provides inference rules for reasoning about safety and liveness in the using the fairness constraints.

TLA is a logically expressive medium for describing and reasoning about transition systems. Conjunction captures synchronous composition in an obvious way. Asynchronous composition is also defined by conjunction by adding an awkward restriction that the output variables of two modules must not both be changed in a single step. The primary problem with TLA as an intermediate language is that it does not have a simple operational reading needed for driving model checkers and analysis tools. SAL uses the familiar guarded commands which do have a simple operational reading and can form the basis for effective tools that rely on precondition and postcondition calculations.

Murphi. The Murphi model checker of David Dill and his colleagues uses a description language that is based on the UNITY model. Transition systems are given by transition rules and rulesets

that are defined as guarded commands. Murphi does not have a module system nor any mechanisms for synchronous and asynchronous composition.

SAL can serve as a front-end for a Murphi model checker since any SAL module can be flattened into a Murphi like representation consisting of a list of guarded commands applied to a global state.

Reactive Modules. Alur and Henzinger defined Reactive Modules as a way of giving modular definitions of synchronous and asynchronous transition systems. Reactive Modules is the description language used by the model-checking tool MOCHA. Specifications are divided into modules with input, output, and local variables (as in SAL). Modules consist of atoms which control specific variables. An atom specifies the causal dependency between the variables it controls and other variables, and it also specifies the transitions for these variables using guarded commands. Reactive Modules require a fixed causal ordering of variables that has proved awkward for many real-life applications where the dependency orderings need to be more dynamic.

SAL relaxes the fixed causal ordering in Reactive Modules by allowing causal cycles in the syntax but ruling them out semantically by generating proof obligations demonstrating that the guards of possibly cyclic guarded commands are unreachable. Reactive Modules lacks an asynchronous composition operator and implements it by means of a scheduler module that ensures mutually exclusive scheduling of two modules. SAL includes an asynchronous composition operator since the explicit schedule involves changing the definitions of composed processes whenever they are asynchronously composed. For transition systems that have a fixed causal dependency ordering between the variables, there is an easy translation from SAL to Reactive Modules.

Chapter 11

SAL: Frequently Asked Questions

Why does SAL need an intermediate language? Why don't the analysis techniques support real languages and not toy ones? The trouble is that the individual tools operate on their own languages anyway. If there are m source languages and n target tools, then we'll need $m * n$ translations to these tools. The mapping between the source languages and the intermediate language ought to be straightforward enough that any results (properties/counterexamples) will be easily interpretable at the source level. The semantics of real source languages can be slippery. The use of an intermediate language helps to fix the semantics with respect to which bugs and properties are generated.

Is SAL targeted at hardware or software? Both. SAL is broadly targeted at transition systems, i.e., any computation system whose computation with observable global states such that the computation evolves by means of observable state changes. These include hardware, software, hybrids, protocols, discrete control systems, among other things.

Why does the SAL language need modules? More generally, why does SAL need high-level features since it is only meant to be an intermediate format? One of the challenges for SAL is to be able to reason about large systems by composing properties of smaller ones. The smaller systems are more easily analyzed and model checked. The purpose of modules in the intermediate language is to promote more modular design and analysis. Otherwise, it will only be possible to prove global properties of closed systems, and these properties will have to be discarded as unreusable once the analysis is complete.

Why doesn't SAL take a shared variable view of the state? The module system in SAL divides variables into local, input, and output variables for modularity reasons. With synchronous composition, this separation ensures that it is not the case that a single variable is driven by multiple sources (possibly inconsistently). With asynchronous composition, this separation ensures that the only interference is through inputs so that a class of module properties is preserved even under composition.

Why doesn't SAL just use predicates and relations to capture transition systems? The main reasons are

1. This would be okay for theorem provers but few model checkers can take arbitrary relations as transitions so this would not be a good choice for an intermediate language.
2. Static analysis would be harder with relations since computing weakest preconditions and post-condition assertions would be more complicated.
3. The composition of two modules given by predicates can be inconsistent.
4. The fairness constraints involve the enabledness of relations and this could complicate the proof rules (see Unity versus TLA). With guarded commands, a transition is enabled when its guard is true.
5. Causal loops are hard to resolve since there is no causal ordering on the variables.

Why does the composition of modules need to be a module? A module is a self-contained unit of a transition system specification that can be analyzed for properties. If the composition of modules is a transition system that cannot be captured in the language, then the language has a gap in its expressivity.

Why does SAL need both synchronous and asynchronous composition? A common trick in many formalisms is to simulate interleaving of modules M_1 and M_2 using synchronous composition by postulating an arbiter that nondeterministically selects between M_1 and M_2. This kind of reduction is inherently noncompositional since the composition operator has to change the modules M_1 and M_2 in order to subject them to the control of the arbiter. There is no advantage gained from eliminating asynchronous composition in this way since it only complicates both deductive and algorithmic verification. In the deductive case, one has to reason about the arbiter than about arbitrary interleavings, and in the algorithmic case, the arbiter contributes extra state.

What is a stuttering-invariant property? An LTL or CTL property without a next-state modality is stuttering invariant, when stuttering means that all the variables retain their prior value. The interpretation of stuttering here is that the controlled variables are stuttered whereas the inputs vary freely. These properties have to be verified with an explicit stuttering step added to the transition relation. Then, when M_1 and M_2 are asynchronously composed, the stuttering-invariant component properties of each hold in the composition. In a shared variable model, this form of modularity is ruled out since M_2 can interfere with M_1 in arbitrary ways.