# IF: An Intermediate Representation for SDL and its Applications

Marius Bozga[a], Jean-Claude Fernandez[b], Lucian Ghirvu[a*], Susanne Graf[a], Jean-Pierre Krimm[a], Laurent Mounier[a] and Joseph Sifakis[a]

[a]VERIMAG, Centre Equation, 2 avenue de Vignate, F-38610 Gières,
e-mail: Marius.Bozga@imag.fr, http://www-verimag.imag.fr/DIST_SYS/IF

[b]LSR/IMAG, BP 82, F-38402 Saint Martin d'Hères Cedex,
e-mail: Jean-Claude.Fernandez@imag.fr

We present work of a project for the improvement of a specification/validation toolbox integrating a commercial toolset *Object*GEODE and different validation tools such as the verification tool CADP and the test sequence generator TGV.

The intrinsic complexity of most protocol specifications lead us to study combinations of techniques such as static analysis and abstraction together with classical model-checking techniques. Experimentation and validation of our results in this context motivated the development of an intermediate representation for SDL called IF. In IF, a system is represented as a set of timed automata communicating asynchronously through a set of buffers or by rendez-vous through a set of synchronization gates. The advantage of the use of such a program level intermediate representation is that it is easier to interface with various existing tools, such as static analysis, abstraction and compositional state space generation. Moreover, it allows to define for SDL different, but mathematically sound, notions of time.

**Keywords**:
SDL, Time Semantics, Validation, Model-Checking, Test Generation, Static Analysis.

## 1. INTRODUCTION

SDL and related formalisms such as MSC and TTCN are at the base of a technology for the specification and the validation of telecommunication systems. This technology will be developing fast due to many reasons, institutional, commercial and economical ones. SDL is promoted by ITU and other international standardization bodies. There exist commercially available tools and most importantly, there are increasing needs for description and validation tools covering as many aspects of system development as possible. These needs motivate the work for enhancement of the existing standards undertaken by ITU and ETSI, in particular.

Among the work directions for improvement of SDL, an important one is the description of non functional aspects of the behavior, such as performance and timing. Finding a

*Work partially supported by Région Rhône-Alpes, France

"reasonable" notion of time is a central problem which admits many possible solutions depending on choices of semantic models. This is certainly a non trivial question and this is reflected by the variety of the existing proposals.

Choosing an appropriate timed extension for SDL should take into account not only technical considerations about the semantics of timed systems but also more pragmatic ones related to the appropriateness for use in a system engineering context. We believe that the different ideas about extensions of the language must be validated experimentally before being adopted to avoid phenomena of rejection by the users. Furthermore, it is important to ensure as much as possible compatibility with the existing technology and provide evidence that the modified standard can be efficiently supported by tools.

Another challenge for the existing technology for SDL to face the demand for description and validation of systems of increasing size, is to provide environments that allow the user to master this complexity. The existing commercial tools are quite satisfactory in several respects and this is a recognized advantage of SDL over other formalisms poorly supported by tools. However, it is necessary to improve the existing technology to avoid failing to keep up. Mastering complexity requires a set of integrated tools supporting user driven analysis. Of course, the existing tools such as simulators, verifiers, automatic test generators can be improved. Our experience from real case studies shows that another family of tools is badly needed to break down complexity. All the methods for achieving such a goal are important ranging from the simplest and most "naive" to the most sophisticated.

In this paper we present work of a project for the improvement of a specification and validation toolbox interconnecting *Object*GEODE[1] and different validation tools such as CADP[2] developed jointly with the VASY team of Inria Rhône-Alpes and TGV[3] developed jointly with the PAMPA team of IRISA. The project has two complementary work directions. The first is the study and the implementation of timed extensions for SDL; this work is carried out in cooperation with Verilog, Sema Group and CNET within a common project. The second is coping with complexity by using a combination of techniques based on static analysis, abstraction and compositional generation. Achieving these objectives requires both theoretical and experimental work. Experimentation and validation of our results in this context motivated the development of an intermediate representation for SDL called IF. IF is based on a simple, and semantically sound model for distributed timed systems which is asynchronously communicating timed automata (automata with clocks). A translator from a static subset of SDL to IF has been developed and IF has been connected to different tools of our toolbox. The use of such an intermediate representation confers many advantages.

- IF to implement and evaluate different semantics of time for SDL as the underlying model of IF is general enough to encompass a large variety of notions of urgency, time non determinism and different kinds of real-time constructs.
- IF allows a flattened description of the corresponding SDL specification with the possibility of direct manipulation, simplification and generally application of analysis algorithms which are not easy to perform using commercial tools which, in general, are closed.
- IF can be considered as a common representation model for other existing languages or for the combination of languages adopting different description styles.

**Related work**

After its standardization in the eighties, a lot of work has been done concerning the mathematical foundations of SDL. The first complete semantics was given by the annex F to the recommendation Z.100 [4,5] and is based on a combination of CSP [6] and META-IV [7]. Even if it is the reference semantics of SDL (about 500 pages), it is far from being complete and contains many inconsistencies and obscure points.

In [8] is given a semantics for SDL based on *streams* and *stream processing functions*. It deals with a subset of SDL and the timing aspects are simplified. An *operational semantics* which covers SDL systems, processes, blocks and channels is given in [9]. It defines a method to build labeled transition systems from SDL specifications. The approach is relatively complete, however in this case too, time is not handled in a satisfactory manner. An important work is done in [10,11] which gives a semantics based on *process algebra* to a rather simple subset of SDL, called $\varphi^-$SDL. A method is given, for translating each SDL system into a term of $\mathrm{PA}^{psc}_{drt}$-ID which is a discrete time process algebra extended with propositional signals and conditions, counting process creation operator, and a state operator. Finally, we mention the work of [12] which proposes an axiomatic semantics based on *Duration Calculus* and the work of [13] which uses *abstract real time machines*.

Our aim is somewhat different from the one of the above mentioned works, as it is not only to present a sound semantics, especially concerning timing aspects, but also to make forward a step concerning verification of SDL specifications: we give a program level intermediate representation into which a large subset of SDL and also other specification formalisms can be translated — by preserving its semantics — and which is the input language of an open verification environment.

The paper is organized as follows. In the next section, we present an example used throughout the paper to illustrate our work. Then, we describe the main features of the IF formalism used as an intermediate representation for SDL. Finally, we present an open validation environment for SDL specifications and illustrate its usefulness by means of some experimental results.

## 2. AN EXAMPLE: A DISTRIBUTED LEADER ELECTION ALGORITHM

We present a simple example used throughout the paper to illustrate the introduced formalisms and verification methods. We consider a *token ring*, that is a system of $n$ stations $S_1, \ldots, S_n$, connected through a circular network, in which a station is allowed to access some shared resource $R$ only when it "owns" a particular message, the *token*. If the network is unreliable, it is necessary to recover from token loss. This can be done using a *leader election algorithm* [14,15] to designate a station responsible for generating a new token.

Formal specifications and verifications of these algorithms already exist and we consider here an SDL version of the one described in [16]. Figure 1 shows the system view of the specification. The signals `open` and `close` denote the access and the release of the shared resource (here part of the environment). The signals `token` and `claim` are the messages circulating on the ring.

All stations $S_i$ are identical and modeled by the SDL process of Figure 2. On expiration

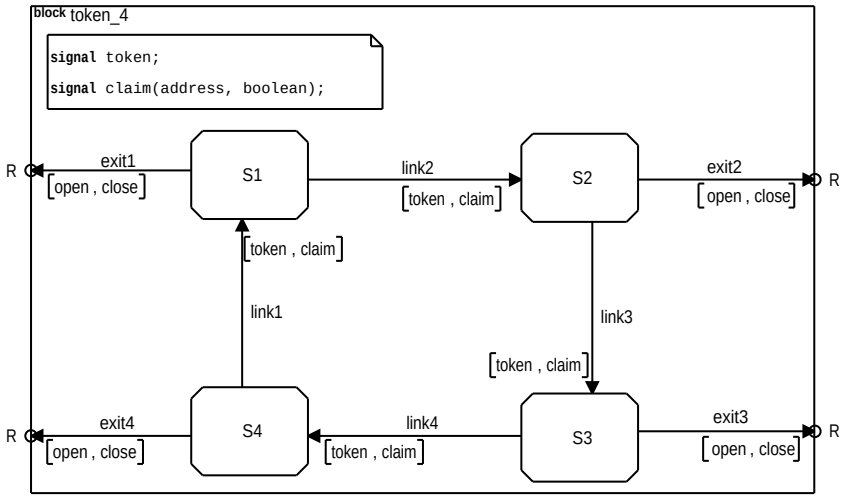


Figure 1. The *token-ring* architecture

of the timer `worried` token loss is assumed: this timer is set when the station waits for the token, and reset when it receives it. The "alternating bit" `round` is used to distinguish between valid claims (emitted during the current election phase) and old ones (cancelled by a token reception). In the `idle` state, a station may either receive the token from its neighbor (then it reaches the `critical` state and can access the resource), receive a timer expiration signal (then it emits a claim stamped with its `address` and the current value of `round`) or receive a claim. A received claim is "filtered" if its associated `address` is smaller than its own address and transmitted unchanged if it is greater. If its own valid claim is received, then this station is elected and generates a new token.

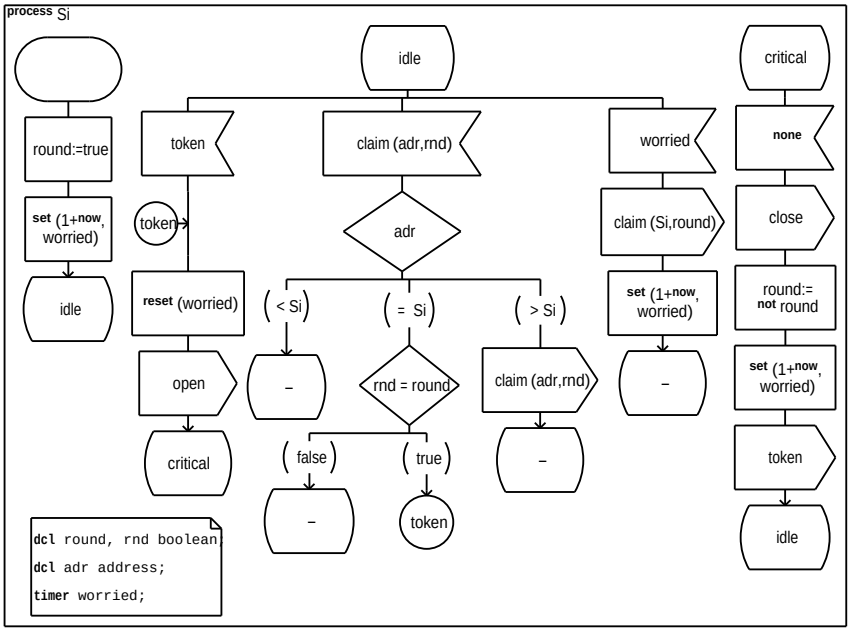


Figure 2. The behavior of station $S_i$

In the SDL specification, message loss must be modeled explicitly (for instance by introducing a non deterministic choice when a token or claim is transmitted by a station). Using the intermediate representation IF, message loss can be expressed simpler by means of lossy buffers, for which exist particular verification techniques.

## 3. IF: AN INTERMEDIATE REPRESENTATION FOR SDL

In the following sections, we give a brief overview of the intermediate representation IF, its operational semantics in terms of labeled transition systems and the translation of a rather extended subset of SDL into IF. A more complete description can be found in [17]. In particular, we do not present the rendez-vous communication mechanism here.

### 3.1. An overview on IF

In IF, a system is a set of *processes* (state machines as SDL processes) communicating *asynchronously* through a set of *buffers* (which may be lossy/reliable and bounded/unbounded). Each process can send and receive messages to/from any buffer. The timed behavior of a system can be controlled through *clocks* (like in timed automata [18,19]) and *timers* (SDL timers, which can be set, reset and expire when they reach a value below 0).

#### 3.1.1. IF system definition

A system is a tuple $Sys = (glob\text{-}def, \text{PROCS})$ where

- *glob-def* = (*type-def*,*sig-def*,*var-def*,*buf-def*) is a list of global definitions, where *type-def* is a list of type definitions, *sig-def* defines a list of parameterized signals (as in SDL), *var-def* is a list of global variable definitions, and finally, *buf-def* is a list of buffers through which the processes communicate by asynchronous signal exchange.
- PROCS defines a set of processes described in section 3.1.2.

#### 3.1.2. IF process definition

Processes are defined by a set of local variables, a set of control states and a set of control transitions. A process P∈PROCS is a tuple P= (*var-def*, Q, CTRANS), where:

- *var-def* is a list of local variable definitions (including timers and clocks)
- Q is a set of control states on which the following attributes are defined:
  - *stable*(q) and *init*(q) are boolean attributes, where the attribute *stable* allows to control the level of atomicity, which is a useful feature for verification: only *stable* states are visible on the semantic level.
  - *save*(q), *discard*(q) are lists of `filter`s of the form
    `signal-list` **in** `buf` **if** `cond`.
    *save*(q) is used to implement the **save** statement of SDL; its effect is to preserve all signals of the list in `buf`, whenever the condition `cond` holds.
    *discard*(q) is used to implement the implicit discarding of non consumable signals of SDL. When reading the next input signal in `buf`, all signals to be discarded in `buf` preceding it are discarded in the same atomic transition.
- CTRANS is a set of control transitions, consisting of two types of transitions between two control states q,q′∈Q:

– input transitions which are triggered by some signal read from one of the communication buffers as in SDL:

$$\mathtt{q} \xrightarrow[(\mathtt{u})]{\mathtt{g} \mapsto \mathtt{input}\ ;\ \mathtt{body}} \mathtt{q}'$$

– internal transitions depending not on communications:

$$\mathtt{q} \xrightarrow[(\mathtt{u})]{\mathtt{g} \mapsto \mathtt{body}} \mathtt{q}'$$

Where in both cases:

- `g` is a predicate representing the *guard* of the transition which may depend on variables visible in the process (including timers, clocks and and buffers, where buffers are accessed through a set of primitives).

- `body` is a sequence of the following types of atomic actions:
  – *outputs* of the form "**output** *sig*(*par_list*) **to** *buf*" have as effect to append a *signal* of the form "*sig*(*par_list*)" at the end of the buffer *buf*.
  – usual *assignments*.
  – *settings* of timers of the form "**set** *timer* := `exp`". This has the effect to activate `timer` and to set it to the value of `exp`. An active timer decreases with progress of time. SDL timers expire when they reach the value 0, but in IF any timer tests are allowed. Clocks are always active and they increase with progress of time.
  – *resettings* of timers and clocks, which have the effect to inactivate timers and to assign the value 0 to clocks.

- The attribute `u`∈{**eager, delayable, lazy**} defines the urgency type of each transition. **eager** transitions have absolute priority over progress of time, **delayable** transitions may let time progress, but only as long as they remain enabled, whereas **lazy** transitions cannot prevent progress of time. These urgency types are introduced in [20] and lead to a more flexible notion of urgency than in timed automata, where enabled transitions become urgent when an *invariant* — depending on values of timers and ordinary variables — associated with the starting state $q$ becomes `false`. For compatibility with timed automata we allow also the definition of an attribute *tpc*(q) representing such an invariant (defining a necessary condition for time progress). Notice also that
  – the concept of urgency defines a priority between ordinary transitions and time progress which moreover may vary with time. Thus the concept of urgency is orthogonal to the usual concept of priority between ordinary transitions.
  – urgency and *tpc* attributes are only relevant in *stable* states, as only *stable* states are visible at the semantic level, and therefore only in *stable* states interleaving of transitions (ordinary ones as well as time progress) is possible.

- `input` is of the form "**input** `sig`(*reference_list*) **from** `buf` **if** `cond`" where
  – `sig` is a signal,
  – *reference_list* the list of references[2] in which received parameters are stored,

[2]that is an "assignable" expression such as a variable or an element of an array

– `buf` is the name of the buffer from which the signal should be read
– `cond` is a "postguard" defining the condition under which the received signal is accepted; `cond` may depend on received parameters.

Intuitively, an input transition is enabled if its guard is `true`, the first signal in `buf` — that can be consumed according to the *save* and *discard* attributes — is of the form `sig`$(v_1, ...v_n)$ and the postguard `cond` holds *after* assigning to the variables of the *reference_list* the corresponding values $v_1, ...v_n$.

That means that the input primitive is — as in SDL — rather complicated, but if one wants to translate the SDL input primitive by means of a simpler primitives, one needs to allow access the elements of the buffer in any order. This means that the input buffer becomes an ordinary array and no specific analysis methods can be applied.

### 3.2. Semantics of IF

We show how with a process can be associated a labeled transition system, and then, how these process models can be composed to obtain a system model.

#### 3.2.1. Association of a model with a process

Let `P`= (*var-def*, `Q`, cTRANS) be a process definition in the system `Sys` and:

- Let TIME be a set of environments for timers and clocks (for simplicity of the presentation, we suppose that these environments are global, that is, applicable to all timers and clocks occurring in `Sys`). An environment $\mathcal{T}\in$TIME defines for each clock a value in a time domain T (positive integers or reals), and for each timer either a value in T or the value *"inac"* (represented by a negative value) meaning that the timer is not active. Setting or resetting a timer or a clock affects a valuation $\mathcal{T}$ in an obvious manner. Progress of time by an amount $\delta$ transforms the valuation $\mathcal{T}$ into the valuation $\mathcal{T} \boxplus \delta$ in which the values of all clocks are increased by $\delta$, and the values of all timers are decreased by $\delta$ (where the minimal reachable value is zero).
- Let BUF be a set of buffer environments $\mathcal{B}$, representing possible contents of the buffers of the system, on which all necessary primitives are defined: usual buffer access primitives, such as "*get the first signal of a given buffer, taking into account the save and the discard attribute of a given control state*", "*append a signal at the end of a buffer*",... and also "*time progress by amount* $\delta$", denoted by $\mathcal{B} \boxplus \delta$, is necessary for buffers with delay.
- Let ENV be a set of environments $\mathcal{E}$ defining the set of valuations of all other variables defined in the system `Sys`.

The semantics of `P` is the labeled transition system $[\texttt{P}] = (\texttt{Q}\times\text{VAL}, \text{TRANS}, \text{TTRANS})$ where

- `Q`×VAL is the set of states and VAL= ENV×TIME×BUF is the set of data states.
- TRANS is the set of untimed transitions obtained from control transitions by the following rule: for any $(\mathcal{E},\mathcal{T},\mathcal{B}),(\mathcal{E}',\mathcal{T}',\mathcal{B}')\in$VAL and input transition (and simpler for an internal transition)

$$\mathtt{q} \xrightarrow[(\mathtt{u})]{\mathtt{g} \mapsto (\mathtt{sig}(x_1...x_n),\mathtt{buf},\mathtt{cond})\ ;\ \mathtt{body}} \mathtt{q}' \ \in \text{cTRANS} \qquad \text{implies}$$

$$(\mathtt{q},(\mathcal{E},\mathcal{T},\mathcal{B})) \xrightarrow{\ell} (\mathtt{q}',(\mathcal{E}',\mathcal{T}',\mathcal{B}')) \in \text{TRANS}, \qquad \text{if}$$

- the guard g evaluates to `true` in the environment ($\mathcal{E}$,$\mathcal{T}$,$\mathcal{B}$)
- the first element of `buf` in the environment $\mathcal{B}$— after elimination of appropriate signals of the discard attribute and saving of the signals of the save attribute — is a signal $\mathtt{sig}(v_1...v_n)$, and the updated buffer environment, after consumption of $\mathtt{sig}(v_1...v_n)$, is $\mathcal{B}''$
- $\mathcal{E}''=\mathcal{E}[v_1...v_n/x_1...x_n]$ and $\mathcal{T}''=\mathcal{T}[v_1...v_n/x_1...x_n]$ are obtained by assigning to $x_i$ the value $v_i$ of the received parameters,
- the postguard `cond` evaluates to `true` in the environment ($\mathcal{E}''$,$\mathcal{T}''$,$\mathcal{B}''$)
- $\mathcal{E}'$ is obtained from $\mathcal{E}''$ by executing all the assignments of the body,
- $\mathcal{T}'$ is obtained from $\mathcal{T}''$ by executing all the settings and resettings occurring in the body, without letting time progress,
- $\mathcal{B}'$ is obtained from $\mathcal{B}''$ by appending all signals required by outputs in the body,
- $\ell$ is an appropriate labeling function used for tracing.

- TTRANS is the set of *time progress transitions*, which are obtained by the following rule which is consistent with the intuitively introduced notion of urgency: in any state (q,($\mathcal{E}$,$\mathcal{T}$,$\mathcal{B}$)), time can progress by the amount $\delta$, that is

$$(\mathtt{q},(\mathcal{E},\mathcal{T},\mathcal{B})) \xrightarrow{\delta} (\mathtt{q},(\mathcal{E},\mathcal{T}\boxplus\delta,\mathcal{B}\boxplus\delta)) \ \in \text{TTRANS} \qquad \text{if}$$

  1. q is *stable* and
  2. time can progress in the state (q,($\mathcal{E}$,$\mathcal{T}$,$\mathcal{B}$)), and
  3. time can progress by steps until $\delta$: whenever time has progressed by an amount $\delta'$ where $0 \leq \delta' < \delta$, time can still progress in the reached state which is (q,($\mathcal{E}$,$\mathcal{T} \boxplus \delta'$,$\mathcal{B} \boxplus \delta'$)).

  Time can progress in state (q,($\mathcal{E}$,$\mathcal{T}$,$\mathcal{B}$)) if and only if the following conditions hold:

  - the time progress attribute *tpc*(q) holds in ($\mathcal{E}$,$\mathcal{T}$,$\mathcal{B}$)
  - *no* transition with urgency attribute **eager** is enabled in (q,($\mathcal{E}$,$\mathcal{T}$,$\mathcal{B}$))
  - for each **delayable** transition tr enabled in (q,($\mathcal{E}$,$\mathcal{T}$,$\mathcal{B}$)), there exists a positive amount of time $\epsilon$, such that tr cannot be disabled while time progresses by $\epsilon$.

### 3.2.2. Composition of models

The semantics of a system *Sys* = (*glob-def*,PROCS) is obtained by composing the models of processes by means of an associative and commutative parallel operator $\|$.
Let $[\mathtt{P}_i] = (\mathtt{Q}_i \times \text{VAL}, \text{TRANS}_i, \text{TTRANS}_i)$ be the models associated with processes (or subsystems) of *Sys*. Then, $[\mathtt{P}_1] \parallel [\mathtt{P}_2] = (\mathtt{Q} \times \text{VAL}, \text{TRANS}, \text{TTRANS})$ where

- $\mathtt{Q} = \mathtt{Q}_1 \times \mathtt{Q}_2$ and $\begin{cases} \mathit{init}((\mathtt{q}_1,\mathtt{q}_2)) & = & \mathit{init}(\mathtt{q}_1) \wedge \mathit{init}(\mathtt{q}_2) \\ \mathit{stable}((\mathtt{q}_1,\mathtt{q}_2)) & = & \mathit{stable}(\mathtt{q}_1) \wedge \mathit{stable}(\mathtt{q}_2) \end{cases}$

- TRANS is the smallest set of transitions obtained by the following rule and its symmetrical rule:

$$\frac{(q_1,\mathcal{V}) \xrightarrow{\ell} (q_1',\mathcal{V}') \in \textsc{Trans}_1 \text{ and } \neg stable(q_1) \vee stable(q_2)}{((q_1,q_2),\mathcal{V}) \xrightarrow{\ell} ((q_1',q_2),\mathcal{V}') \in \textsc{Trans}}$$

- TTRANS is the smallest set of transitions obtained by the following rule:

$$\frac{(q_1,\mathcal{V}) \xrightarrow{\delta} (q_1,\mathcal{V}') \in \textsc{Ttrans}_1 \text{ and } (q_2,\mathcal{V}) \xrightarrow{\delta} (q_2,\mathcal{V}') \in \textsc{Ttrans}_2}{((q_1,q_2),\mathcal{V}) \xrightarrow{\delta} ((q_1,q_2),\mathcal{V}') \in \textsc{Ttrans}}$$

### 3.3. Translation from SDL to IF

We present the principles of the translation from SDL to IF considering the structural and the behavioral aspects. We do not present the translation of data aspects, and in particular the object oriented features, as they do not interfere with our framework.

#### 3.3.1. Structure

SDL provides a complex structuring mechanism using blocks, substructures, processes, services, etc, whereas IF systems are *flat*, that is consisting of a single level of processes, communicating directly through buffers. Therefore, a structured SDL system is flattened by the translation into IF. Also, the structured communication mechanism of SDL using channels, signal routes, connection points, etc is transformed into point to point communication through buffers by computing for every output a statically defined unique receiver process (respectively its associated buffer).

All predefined SDL data types, arrays, records and enumerated types can be translated. For abstract data types, only the signatures are translated, and for simulation, the user must provide an appropriate implementation.

In SDL all signals are implicitly parameterized with the pid of the sender process, therefore in IF all signals have an additional first parameter of type `pid`.

#### 3.3.2. Processes

Basically, for each instance of an SDL process, we generate an equivalent IF process and associate with it a default input queue. If the number of instances can vary in some interval, the maximal number of instances is created.

**Variables:** Each local variable/timer of an SDL process becomes a local variable/timer of the corresponding IF process. We define also variables `sender`, `offspring` and `parent` which are implicitly defined in SDL. Remote exported/imported variables declared inside an SDL processes become global variables, declared at IF system level.

**States:** All SDL states (including *start* and *stop*) are translated into *stable* IF control states. As IF transitions have a simpler structure than SDL transitions, we introduce also systematically auxiliary non stable states for each *decision* and each *label* (corresponding to a "join") within an SDL transition. For each *stable* IF state we define the *save* and *discard sets* to be the same as for the corresponding SDL state.

**Transitions:** For each *minimal path* between two IF control states, an IF transition is generated. It contains the triggers and actions defined on that path in the same order.

All the generated transitions are by default *eager* i.e. they have higher priority than the progress of time; this allows to be conform with the notion of time progress of the tool *Object*GEODE; more liberal notions of time progress can be obtained by using different translations from SDL to IF (see the example below).

- inputs: SDL signal inputs are translated directly into IF inputs, where the sender parameter must be handled explicitly: each signal receives the first parameter in the local variable `sender`.
  Spontaneous input none is translated by an assignment of the `sender` to the pid of the current process. No input part is generated in this case.
- timeouts expirations are *not* notified via timeout signals in IF: each timeout signal consumption in an SDL process is translated into a transition without input, which tests if the corresponding timer evaluates to *zero*, followed by the reset of that timer. The reset is needed to avoid multiple consumption of the same timeout expiration.
- priority inputs: are translated into normal inputs by enforcing the guards of all low priority inputs and the save set of the source state. The guard of each low priority input is conjuncted with a term saying that "*there is no higher priority signal in the buffer*". All low priority signals are explicitly saved if "*at least one input with higher priority exists in the buffer*". Such tests can effectively be expressed by predefined predicates on buffers.
- continuous signal: SDL transitions triggered by a continuous signal test, are translated into an IF transition without input, whose guard is equivalent to the SDL continuous signal.
- enabling condition: an enabling condition following an SDL input signal is translated directly into a post guarded input where the received parameters can be tested.
- task: all SDL formal tasks are translated into IF assignments. Informal tasks become comments in the IF specification.
- set and reset: SDL timer sets become IF timer sets, where an *absolute* value "$now + T$" becomes a *relative* value "$T$". SDL timer resets become IF timer resets.
- output: SDL outputs become IF outputs: if the *to pid-expression* clause is present in the SDL output, the same pid-expression is taken as destination for the IF output. Otherwise, according to *signal routes signature*, *via restrictions*, *connections*, etc. we compute **statically** the set of all possible destinations. If this set contains exactly one process instance, it become the IF destination, otherwise, this output is not translated (and a warning is produced). Any output contains as first parameter the `pid` of the sending process.
- decision: each alternative of an SDL formal decision is translated into a guard starting an IF-transition from the corresponding non stable state.
- create: the dynamic creation of processes is not yet handled. But we will translate this construction by using the rendez-vous mechanism of IF: a new instance is created (an "inactive" instance is activated) by synchronizing its first action with

the process creating (activating) it. During this synchronization, parameters can be passed between the "creating" and the "created" process, such as the the values of the `parent` and the `offspring` variables, etc.

- procedures: IF does not directly support procedures. But we handle a relatively large class of SDL programs containing procedures by *procedure inlining*, which consists in directly inserting the procedure graph, instead of its call, in the process graph.

**Example: translation of the token ring to IF**

To illustrate IF, we present the translation of the token ring introduced in Section 2. The translation of the structure is completely straightforward in this example. Figure 3 contains the IF version of the process $S_1$, where the additional non stable states are dotted.

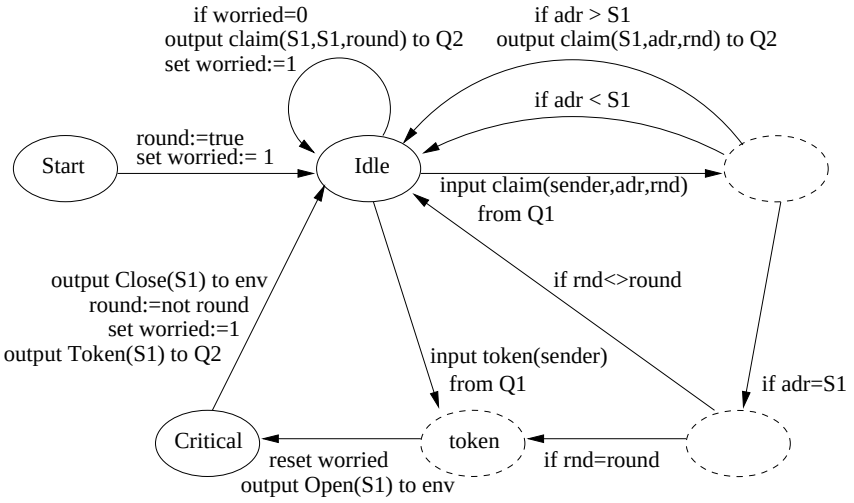


Figure 3. The "graphical" IF description of station $S_1$

By default, all transitions are **eager**, which leads to the same behavior as in *Object*GEODE. Thus, time can only progress, and the timeout occur, if the token is really lost (that is, no transition is enabled), and therefore a leader election algorithm is only initiated if necessary. In IF, a different notion of time, closer to reality, can be modeled, e.g. by considering the transition from the `critical` state as **lazy**, thus allowing time to pass in this state by an arbitrary amount. In order to limit the time a process can remain in the `critical` state, one can consider this transition as **delayable**, introduce a clock `cl_crit` which is reset when entering `critical` and add to the outgoing transition the guard `cl_crit`$\leq$*some_limit*.

## 4. AN OPEN VALIDATION ENVIRONMENT BASED ON IF

One of the main motivations for developing IF is to provide an intermediate representation between several tools in an "open" validation environment for SDL. Indeed, none of the existing tools provides all the validation facilities a user may expect. Therefore, we want to allow them to cooperate, as much as possible using program level connections. An important feature is the ability of the environment to be open: in particular connections with KRONOS [21] (a model checker for timed automata) and INVEST [22,23] (a tool computing abstractions) are envisaged.

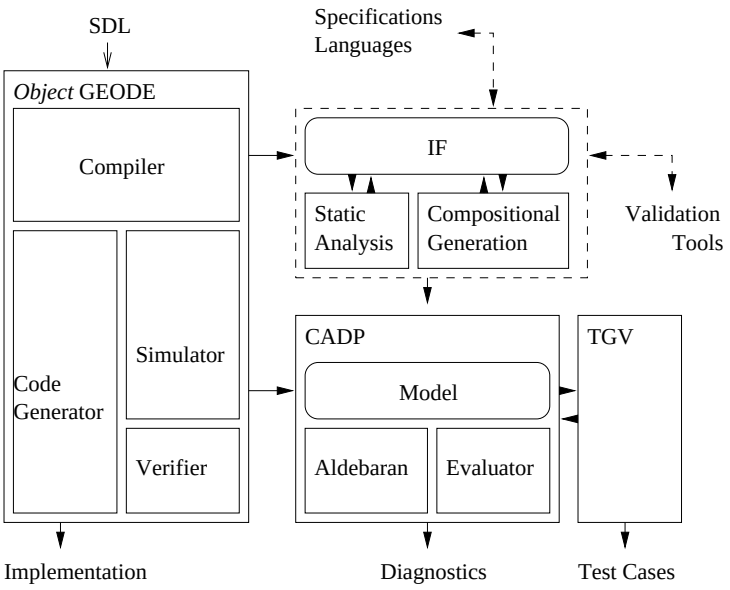


Figure 4. An open validation environment for SDL

In this section, we first present the architecture of this environment and its main components. Then, we describe in a more detailed manner two more recent modules concerning static analysis (section 4.2) and compositional generation (section 4.3) which are based on IF.

### 4.1. Architecture

The environment is based on two validation toolsets, *Object*GEODE and CADP, connected through the intermediate representation IF. There exists already a connection between these toolsets at the simulator level [24], however using IF offers two main advantages:

- The architecture still allows connections with many other specification languages or tools. Thus, even specifications combining several formalisms could be translated into a single IF intermediate code and globally verified.

- The use of an intermediate program representation where all the variables, timers, buffers and the communication structure are still explicit, allows to apply methods such as static analysis, abstraction, compositional generation. These methods are crucial for the applicability of the model checking algorithms.

#### *Object*GEODE

*Object*GEODE is a toolset developed by VERILOG supporting the use of SDL, MSC and OMT. It includes graphical editors and compilers for each of these formalisms. It also provides a C code generator and a simulator to help the user to interactively debug an SDL specification. The *Object*GEODE simulator also offers some verification facilities since it allows to perform automatic simulation (either randomly or exhaustively), and behavioral comparison of the specification with special state machines called observers [25].

**CADP and TGV**

We have been developing for more than ten years a set of tools dedicated to the design and verification of critical systems. Some of them are distributed in collaboration with the VASY team of INRIA Rhône-Alpes as part of the CADP toolset [2,26]. We briefly present here two verifiers integrated in CADP (ALDEBARAN and EVALUATOR) and the test sequence generator TGV [3] built upon CADP jointly with the PAMPA project of IRISA. These tools apply model-checking on behavioral models of the system in the form of labeled transition systems (LTS). ALDEBARAN allows to compare and to minimize finite LTS with respect to various *simulation* or *bisimulation* relations. This allows the comparison between the observable behavior of a given specification with its expected one, expressed at a more abstract level. EVALUATOR is a model-checker for temporal logic formulas expressed on finite LTS. The temporal logic considered is the alternating-free $\mu$-calculus. TGV aims to automatically generate test cases for conformance testing of distributed systems. Test cases are computed during the exploration of the model and they are selected by means of *test purposes*. Test purposes characterize some abstract properties that the system should have and one wants to test. They are formalized in terms of LTS, labeled with some interactions of the specification. Finally, an important feature of CADP is to offer several representations of LTS, enumerative and symbolic ones based on BDD, each of them being handled using well-defined interfaces such as OPEN-CAESAR [27] and SMI [28].

**SDL2IF and IF2C**

To implement the language level connection through the IF intermediate representation we take advantage of a well-defined API provided by the *Object*GEODE compiler. This API offers a set of functions and data structures to access the abstract tree generated from an SDL specification. SDL2IF uses this abstract tree to generates an IF specification operationally equivalent to the SDL one.

IF is currently connected to CADP via the *implicit model representation* feature supported by CADP. IF programs are compiled using IF2C into a set of C primitives providing a full basis to simulate their execution. An exhaustive simulator built upon these primitives is also implemented to obtain the explicit LTS representation on which all CADP verifiers can be applied.

### 4.2. Static analysis

The purpose of static analysis is to provide global informations about how a program manipulates data without executing it. Generally, static analysis is used to perform global optimizations on programs [29–31]. Our goal is quite different: we use static analysis in order to perform model reductions before or during its generation or validation. The expected results are the reduction of the state space of the model or of the state vector.

We want to perform two types of static analysis: *property independent* and *property dependent* analysis. In the first case, we use classic analysis methods such as live variable analysis or constant propagation, without regarding any particular property or test purpose we are interesting to validate. In the second case, we take into account informations on data involved in the property and propagate them over the static control structure of the program. Presently, only analysis of the first type is implemented but we are also investigating constraint propagation and more general abstraction techniques. For instance, through the connection with INVEST we will be able to compute abstract IF

programs using general and powerful abstraction techniques.

**Live variables analysis**

A variable is *live* in a control state if there is a path from this state along which its value can be used before it is redefined. An important reduction of the state space of the model can be obtained by taking into account in each state only the values of the live variables.

More formally, the reduction considered is based on the relation $\sim_{live}$ defined over model states: two states are related if and only if they have the same values for all the live variables. It can be easily proved that $\sim_{live}$ is an equivalence relation and furthermore, that it is a bisimulation over the model states. This result can be exploited in several ways. Due to the local nature of $\sim_{live}$ it is possible to directly generate the quotient model w.r.t. $\sim_{live}$ instead of the whole model without any extra computation. Exactly the same reduction is obtained when one modifies the initial program by introducing systematic assignments of non-live variables to some particular value. This second approach is presently implemented for IF programs.

Consider now the token ring protocol example. In the `idle` state the live variables are `round` and `worried`, in the `critical` state only `round` is live, while variables `sender`, `adr` and `rnd` are never live. The reduction obtained by the live reduction is shown in Table 1 (line 3).

**Constant propagation**

A variable is *constant* in a control state if its value can be statically determined in the state. Two reductions are possible. The first one consists in modifying the source program by replacing constant variables with their value. Thus, it is possible to identify and then to eliminate parts of dead code of the program e.g. guarded by expressions which always evaluates to `false`, therefore to increase the overall efficiency of the program. The second reduction concerns the size of the state vector: for a control state we store only the values of the non-constant variables. The constant values do not need to be stored, they can always be retrieved by looking at the control state.

Note that, both of the proposed reductions do not concern the number of states of the model, they only allow to improve the state space exploration (time and space). However, this kind of analysis may be particularly useful when considering extra information about the values assigned to variables, extracted from the property to be checked.

### 4.3. Compositional generation

As shown in the previous section, efficient reductions are obtained by replacing a model $M$ by its quotient w.r.t an equivalence relation like $\sim_{live}$. However, stronger reductions can be obtained by taking into account the properties under verification. In particular, it is interesting to consider a weaker equivalence $R$ — which should be a congruence for parallel composition —, able to abstract away non observable actions. The main difficulty is to obtain the quotient $M/R$ without generating $M$ first.

A possible approach is based on the "divide and conquer" paradigm: it consists in splitting the program description into several pieces (i.e., processes or process sets), generating the model $M_i$ associated with each of them, and then composing the quotients $M_i/R$. Thus, the initial program is never considered as a whole and the hope is that the

generated intermediate models can be kept small.

This compositional generation method has already been applied for specification formalisms based on *rendez-vous* communication between processes, and has been shown efficient in practice [32–34]. To our knowledge it has not been investigated within an SDL framework, may be, because buffers raise several difficulties or due to lack of suitable tools.

To illustrate the benefit of a compositional approach we briefly describe here its application to the token ring protocol:

1. We split the IF description into two parts, the first one contains processes $S_1$ and $S_2$ and the second one contains processes $S_3$ and $S_4$. For each of these descriptions the internal buffer between the two processes is *a priori* bounded to two places. Note that, when a bounded buffer overflows during simulation, a special *overflow* transition occurs in the corresponding execution sequence.

2. The LTS associated with each of these two descriptions are generated considering the "most general" environment, able to provide any potential input. Therefore, the *overflow* transitions appear in these LTS (`claim` and `token` can be transmitted at any time).

3. In each LTS the input and output transitions relative to the internal buffers ($Q_2$ and $Q_4$) are hidden (i.e., renamed to the special $\tau$ action); then these LTS are reduced w.r.t an equivalence relation preserving the properties under verification. For the sake of efficiency we have chosen the branching bisimulation [35], also preserving all the safety properties (e.g. mutual exclusion).

4. Each reduced LTS is translated back into an IF process, and these two processes are combined into a single IF description, including the two remaining buffers ($Q_1$ and $Q_3$). It turns out that the LTS generated from this new description contains no *overflow* transitions (they have been cut off during this last composition, which confirms the hypothesis on the maximal size of the internal buffers).

The final LTS is branching bisimilar to the one obtained from the initial IF description. The gain, obtained by using compositional generation in addition to static analysis, can be found in Table 1 (line 4).

**Results**

We summarize in the table below the size of the LTS obtained from the token-ring protocol using several generation strategies.

Table 1. LTS obtained for the token ring example

| | *Generation method* | *Number of states* | *Number of transitions* |
|---|---|---|---|
| 1 | *Object*GEODE | 3018145 | 7119043 |
| 2 | IF | 537891 | 2298348 |
| 3 | IF + live reduction | 4943 | 19664 |
| 4 | IF + compositional generation | 1184 | 4788 |

The difference between the model generated by *Object*GEODE (line 1) and the one obtained from IF (line 2) are due to the following reasons:

- the handling of timer expirations in *Object*GEODE involves two steps: *first* the time-out signal is appended to the input buffer of the process, and *later* it is consumed, whereas in IF these two steps are collapsed into a single one, bypassing the buffer.
- *Object*GEODE introduces "visible" states for each informal decision, whereas these states do not appear in the model obtained from IF.

The most spectacular reduction is obtained by the live-reduction: the reduced model is about 100 times smaller than the one obtained by direct generation, preserving all properties (models 2 and 3 are strongly bisimilar).

Finally, when considering as visible only the `open` and `close` signals all four LTS are branching bisimilar to the one shown in Figure 5, which proves, in particular, the mutual exclusion property of the protocol.

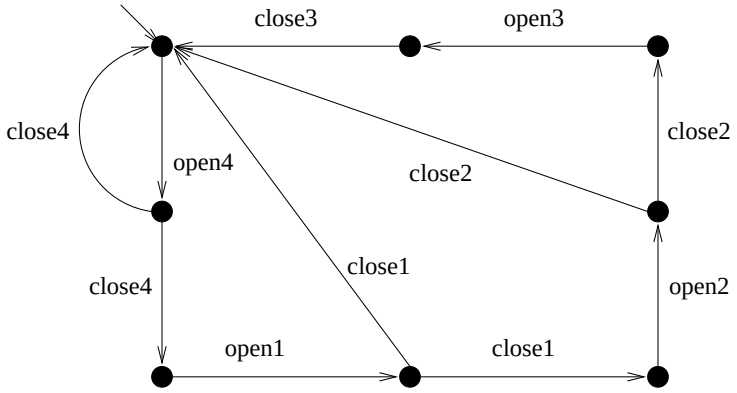


Figure 5. The reduced behavior of the token ring.

## 5. CONCLUSION AND PERSPECTIVES

We have presented the formalism IF which has been designed as an intermediate representation for SDL, but it can be used as a target language for other FDT as it contains most of the concepts used in these formalisms. The use of IF offers several advantages:

- IF has a formal semantics based on the framework of communicating timed automata. It has powerful concepts interesting for specification purposes, such as different urgency types of transitions, synchronous communication, asynchronous communication through various buffer types (bounded, unbounded, lossy, ... ).
- IF programs can be accessed at different levels through a set of well defined API. These include not only several low-level model representations (symbolic, enumerative, ...) but also higher level program representation, where data and communication structures are still explicit. Using these API several tools have been already interconnected within an open environment able to cover a wide spectrum of validation methods.

The IF package is available at `http://www-verimag.imag.fr/DIST_SYS/IF`. In particular, a translation tool from SDL to IF has been implemented and allows both to experiment different semantics of time for SDL and to analyze real-life SDL specifications with CADP.

A concept which is not provided in IF is dynamic creation of new process instances of processes and parameterization of processes; this is due to the fact that in the framework of algorithmic verification, we consider only static (or dynamic bounded) configurations. However, it is foreseen in the future to handle some kinds of parameterized specifications.

The results obtained using the currently implemented static analysis and abstractions methods are very encouraging. For each type of analysis, it was possible to build a module which takes an IF specification as input and which generates an *reduced* one. This architecture allows to chain several modules to benefit from multiple reductions applied to the same initial specification. We envisage to experiment more sophisticated analysis, such as constraints propagation, and more general abstraction techniques. This will be achieved either by developing dedicated components or through the connections with tools like INVEST.

## REFERENCES


1. Verilog. Object*GEODE SDL Simulator - Reference Manual*. http://www.verilogusa.com/solution/pages/ogeode.htm, 1996.
2. J.-C. Fernandez, H. Garavel, A. Kerbrat, R. Mateescu, L. Mounier, and M. Sighireanu. CADP: A Protocol Validation and Verification Toolbox. In *Proceedings of CAV'96 (New Brunswick, USA)*, volume 1102 of *LNCS*, August 1996.
3. J.-C. Fernandez, C. Jard, T. Jéron, and C. Viho. An Experiment in Automatic Generation of Test Suites for Protocols with Verification Technology. *SCP*, 29, 1997.
4. ITU-T. *Annex F.2 to Recommendation Z-100. Specification and Description Language (SDL) - SDL Formal Definition: Static Semantics*. 1994.
5. ITU-T. *Annex F.3 to Recommendation Z-100. Specification and Description Language (SDL) - SDL Formal Definition: Dynamic Semantics*. 1994.
6. C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall International, 1984.
7. D. Bjørner and C.B. Jones. *Formal Specification and Software Development*. Prentice Hall Publications, 1982.
8. M. Broy. Towards a Formal Foundation of the Specification and Description Language SDL. *Formal Aspects on Computing*, 1991.
9. J.C. Godskesen. An Operational Semantic Model for Basic SDL. Technical Report TFL RR 1991-2, Tele Danmark Research, 1991.
10. J.A. Bergstra and C.A. Middelburg. Process Algebra Semantics of $\varphi$SDL. In *2nd Workshop on ACP*, 1995.
11. J.A. Bergstra, C.A. Middelburg, and Y.S. Usenko. Discrete Time Process Algebra and the Semantics of SDL. Technical Report SEN-R9809, CWI, June 1998.
12. S. Mork, J.C. Godskesen, M.R. Hansen, and R. Sharp. A Timed Semantics for SDL. In *FORTE IX: Theory, Applications and Tools*, 1997.
13. U. Gläser and R. Karges. Abstract State Machine Semantics of SDL. *Journal of Universal Computer Science*, 3(12), 1997.
14. G. Le Lann. Distributed Systems – Towards a Formal Approach. In *Information Processing 77*. IFIP, North Holland, 1977.
15. E. Chang and R. Roberts. An Improved Algorithm for Decentralized Extrema-Finding in

Circular Configurations of Processes. *Communications of ACM*, 22(5), 1979.
16. H. Garavel and L. Mounier. Specification and Verification of Distributed Leader Election Algorithms for Unidirectional Ring Networks. *SCP*, 29, 1997.
17. M. Bozga, J-C. Fernandez, L. Ghirvu, S. Graf, L. Mounier, J.P. Krimm, and J. Sifakis. The Intermediate Representation IF. Technical report, Verimag, 1998.
18. R. Alur, C. Courcoubetis, and D.L. Dill. Model Checking in Dense Real Time. *Information and Computation*, 104(1), 1993.
19. T.A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic Model Checking for Real-Time Systems. *Information and Computation*, 111(2), 1994.
20. S. Bornot, J. Sifakis, and S. Tripakis. Modeling Urgency in Timed Systems. In *International Symposium: Compositionality - The Significant Difference, Malente (Holstein, Germany)*, 1998. to appear in LNCS.
21. S. Yovine. KRONOS: A Verification Tool for Real-Time Systems. *Software Tools for Technology Transfer*, 1(1-2), December 1997.
22. S. Graf and H. Saidi. Construction of Abstract State Graphs with PVS. In *Proceedings of CAV'97, Haifa*, volume 1254 of *LNCS*, June 1997.
23. S. Bensalem, Y. Lakhnech, and S. Owre. Computing Abstractions of Infinite State Systems Compositionally and Automatically. In *Proceedings of CAV'98, Vancouver, Canada*, volume 1427 of *LNCS*, June 1998.
24. A. Kerbrat, C. Rodriguez, and Y. Lejeune. Interconnecting the *Object*GEODE and CADP Toolsets. In *Proceedings of SDL Forum'97*. Elsevier Science, 1997.
25. B. Algayres, Y. Lejeune, and F. Hugonnet. GOAL: Observing SDL Behaviors with GEODE. In *Proceedings of SDL Forum'95*. Elsevier Science, 1995.
26. M. Bozga, J.-C. Fernandez, A. Kerbrat, and L. Mounier. Protocol Verification with the ALDEBARAN Toolset. *Software Tools for Technology Transfer*, 1, 1997.
27. H. Garavel. OPEN/CÆSAR: An Open Software Architecture for Verification, Simulation, and Testing. In *Proceedings of TACAS'98*, volume 1384 of *LNCS*, March 1998.
28. M. Bozga. SMI: An Open Toolbox for Symbolic Protocol Verification. Technical Report 97-10, Verimag, September 1997.
29. A. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, Readings, MA, 1986.
30. M.N. Wegman and F.K. Zadeck. Constant Propagation with Conditional Branches. *ACM Transactions on Programming Languages and Systems*, 13(2), April 1991.
31. S. Muchnick. *Advanced Compiler Design Implementation*. Morgan Kaufmann Publishers, San Francisco, CA, 1997.
32. S. Graf, G. Lüttgen, and B. Steffen. Compositional Minimisation of Finite State Systems using Interface Specifications. *Formal Aspects of Computation*, 3, 1996.
33. A. Valmari. Compositionality in State Space Verification. In *Application and Theory of Petri Nets*, volume 1091 of *LNCS*, 1996.
34. J.P. Krimm and L. Mounier. Compositional State Space Generation from LOTOS Programs. In *Proceedings of TACAS'97*, volume 1217 of *LNCS*, Enschede, The Netherlands, 1997.
35. R.J. van Glabbeek and W.P. Weijland. Branching-Time and Abstraction in Bisimulation Semantics. CS R8911, CWI, 1989.