Towards a Multi-Formalism Framework for Architectural Synthesis: the ASAR Project

P. Asar

Correspondence should be sent to Patrice Quinton
IRISA, Campus de Beaulieu, 35042 Rennes Cedex – email: quinton@irisa.fr

Abstract

This paper describes a research project – named ASAR – grouping together six French research teams, oriented towards architectural and system synthesis. A main concern of this project is hardware/software co-design and user-interface management.

1 Introduction

Most of the teams cooperating in the research project ASAR1 have been already involved in architectural synthesis for a long time, using distinct input formalisms and approaches. While independently developed, these approaches are complementary, and could be used during different stages of a design, or for designing different parts of a system.

The goal of the ASAR project is to build a generic architectural synthesis framework providing:

- the possibility of experimenting various approaches, corresponding to different target architectures and design description languages,
- multiple associated tools for transformations and synthesis.

This multi-formalism framework is currently under construction. It is built upon the CENTAUR system, already used by three teams as a common generic software environment for user-interfaces, formalisms and tools management.

A second objective of ASAR concerns hardware/software co-design. Recent advances in synthesis systems of ICs and the proliferation of advanced and inexpensive processors have prompted system architects to investigate CAD methods for systems that contain both application specific (hardware) and pre-designed reprogrammable components (software) [18, 11]. Although a reprogrammable processor can implement most system functionalities as a program, dedicated ASICs are needed for performance reasons. The ideal way to go about developing a new system design should then be to start with an abstract notation independent of a hardware or a software realization. System behavior is generally specified in a hardware description language (VHDL like HardwareC, Verilog) that supports timing constraints [25]. One of the key issues in such a co-design approach is partitioning. Indeed, we must identify which part of a system should be implemented in software and which part in hardware. One basic approach can be characterized by identifying software parts which consume high computing resources in hardware. The dual approach seeks to identify complex system parts which are good candidates to be implemented in software [28]. The application specific hardware is designed by using synthesis tools and needs to cooperate with application software running on the processor. At this time, there are only a few approaches dealing with an automatic hardware/software partitioning.

The Stanford approach is hardware oriented [12]. The input language is Hardware C, a subset of C defined for hardware description. The initial implementation of a system is hardware based, except program constructs with unbounded delays. Then, the design system, VULCAN, tries to gradually move hardware functions to software, checking timing constraints.

The COSYMA approach is software oriented [8]. The input language is Cx, a subset of the ANSI C standard with some extensions (timing, task intercommunications). The partitioning system identifies whether timing constraints are violated or not, and locates the faults. COSYMA implements those constructs in hardware in order to achieve a speedup of overall execution times.

The Virginia approach is a theoretical approach [18]. For each function of the system under development, the partitioning system examines a spectrum of Hardware-Software partitions. The goal of this exploration is to obtain a partition that satisfies some given criteria (performance, cost, reliability, etc.).

UNITY is used in [5] as a specification language. It supports an automatic Hardware/Software partitioning of the specification. UNITY is a theory for specifying parallel computations and a proof system. This method produces various implementation alternatives, and the major criteria for partitioning are the minimization of communication costs for synchronization, and performance improvement.

2 Framework structure

The overall structure of the ASAR framework is sketched in figure 1.

2.1 Basis

Three components of ASAR are of main importance:

CENTAUR. The CENTAUR generic environment is of course the first stone on which we can build this common
framework, allowing to manipulate different syntaxes (abstract, concrete) and user interfaces in an uniform way. CENTAUR is described in section 3.

**An Internal Common Formalism (ICF)** In order to federate our different approaches based on distinct user-formalisms, a sound semantic basis is needed, into which each formalism can be expressed. In other words, there is an underlying hidden common semantics to our multiple user-formalisms, to be conveyed by this ICF. This common formalism, a kind of common denominator, need not be human-readable, and could serve as a common exchange format between different tools.

As our different approaches are all synchronous, a synchronous data-flow semantics seems to be the most appropriate. It is the reason why the GC formalism is under investigation for the ICF. GC [17] is a common format for synchronous data-flow languages, developed in the french projet SYNCHRONIE. Its use, as a ICF for ASAR, is investigated in section 5.

**VHDL** The VHDL language, which is now a standard, cannot be ignored by any architecture oriented tool. Behavioral VHDL is of course one of the user-input language of ASAR, and structural VHDL will be one of its possible output format. VHDL is used by the Osys and GAUT synthesis tools, described in section 4.1 and 4.5 respectively.

**2.2 Multiple user-formalisms**

In addition to VHDL, three other-user-formalisms are present in ASAR, all belonging to the synchronous data-flow language family. SIGNAL favours a relational expression (section 4.2), ALPHA (section 4.3) and LUSTRE (section 4.4) are purely functional languages, ALPHA being suited for regular and systolic architectures.

Translators from concrete to abstract syntaxes are already available under CENTAUR, for VHDL, SIGNAL, ALPHA and LUSTRE. Pretty-printers (from abstract to concrete syntaxes) also exist. Other translators are planned or could be integrated in ASAR via CENTAUR:

- translators from user-formalisms abstract syntax, to the ICF;
- translators from the ICF to the hardware domain: structural or synthesizable VHDL, EDIF; and to the software domain: C.

**2.3 Multiple tools**

The ASAR framework will provide a set of tools oriented towards particular target architectures or domains. GAUT (section 4.5) is dedicated to signal processing designs. Osys (section 4.1) is a prototype of architectural synthesis system from behavioral VHDL descriptions. Alpha (section 4.3) and TRANSE (section 4.4) are tools based on a transformational interactive approach. The Signal system (section 4.2), oriented towards real-time systems design, provides proof checking tools and a Signal to C compiler.

As said before, it is expected that different tools and formalisms will be used or at least experimented for the same design, using ASAR. In order to facilitate this multi-formalism approach, the role of the ICF appears essential:

- as a target for synthesizing and transforming tools: input-L → ICF,
- as a source for translators into languages, like structural VHDL or C, which could be inputs for other external tools: ICF → output-L,
- as source and target for possible new transforming tools: ICF → ICF.

In the following, we give an overview of the main elements of ASAR.

**3 The generic environment CENTAUR**

The CENTAUR system [5] is a generic interactive environment parameterized by the syntax and the semantics of programming languages. When provided with the description of a particular programming language – including its syntax and semantics – it produces a language-specific environment. The system is actually a toolkit that helps in developing tools for manipulating programs or other structured objects.

With the help of CENTAUR one can develop tools needed for a programming environment: structure editors, type checkers, debuggers, interpreters, compilers, various translators and program transformations. These tools (generated from a formal definition of the language) have graphical user interfaces. For example, a type checker can generate a special editor with a list of error and warning messages; clicking upon any item will show the location in the source program that provoked the message.

The CENTAUR system has evolved in a set of cooperative tools, whereby we can couple CENTAUR with existing tools, e.g., an editor, a parser, or a compiler [6]. This has two benefits: the generated system is not monolithic (different generated tools may run in different processors) and, more importantly, one can integrate existing tools (developed independently of CENTAUR) in the generated environment. Several projects have experimented with CENTAUR and connecting it to external tools. These include work around scientific computing, symbolic computation, and theorem provers, to name a few. One of the major achievements is a bi-directional communication between the environment and the symbolic tools allowing developments of
interactive proof building, tactics designing, symbolic computing etc. In particular, CENTAUR can be used to provide a graphical user interface to a number of tools in the area of symbolic computation. CENTAUR communicates via a protocol with a number of computer algebra system (e.g., Maple, Sisyph, Ulysses), with plotting/surface drawing tools (e.g., Gnuplot, ZicVis), and with theorem provers (e.g., HOL, Isabelle, Coq) [27]. Here one profits from the user interface of CENTAUR, and, for example, in the case of theorem provers one can take advantage of many techniques from program environments: selecting the next subgoal to attack, saving an incomplete proof, browsing the various theories available, choosing tactics. Here the CENTAUR technology provides the Virtual Tree Processor, efficient pretty printing, and the generalized notion of location together with its tools for building a user interface.

4 Formalisms and tools

In this section, we describe successively the formalisms and tools which are being integrated during the ASAR project.

4.1 OSYS

OSYS has been designed under the Centaur environment, in order to generate a hardware high level synthesis tool. Nowadays where codesign seems to be a very important issue, the use of the same environment to generate both the architecture and the software of a composite system, will result in coherent hardware/software descriptions and will therefore simplify the design. OSYS will be a part of the codesign system environment.

OSYS is designed for the synthesis of synchronous ASICs. Its primary goal is to validate an approach in which we can take advantage of the existing specification environments in order to build high level design tools and, to end up with a generic synthesis system which could satisfy the criteria described in [10]: completeness, extensibility, controllability, interactivity, upgradeability [16].

4.1.1 Organisation of OSYS

OSYS is aimed to be an Open and Complete architectural Synthesis tool, which explores different architectures from a behavioral VHDL specification.

- Open: the standard VHDL is used as the common language through the different steps.
- Complete: all the steps of High Level Synthesis have been implemented in one environment.

The goal of High Level Synthesis (HLS) is to generate architecture at the structural level in VHDL to interface with the industrial RTL synthesis tools. In OSYS, figure 2, the designer finds all the HLS steps and interactively goes through the process, in choosing the technique to apply.

4.1.2 VHDL Formalism

In OSYS the designer can describe its architecture writing VHDL files if one is an expert; but also using what we called SDEV which is a "guided editor" to write VHDL without any knowledge of the syntax. The editor is guided by the VHDL syntax, described by IEEE Language Reference Manual. As the grammar, we have developed, is full VHDL, SDEV is a full VHDL editor. SDEV proposes the different statements available in VHDL, and the designer just has to click to build his description. This editor reduces programming errors and increases editing speed. Several menus are available for common tasks such as template editing and program transformations. The designer may toggle between "beginner" and "expert" modes, where menus present more or less detailed VHDL entries respectively.

4.1.3 Tools

Through OSYS, the designer can:

1. describe the system in VHDL with a Vhd1 Syntaxtic Driven Editor (SDEV), which allows to write full VHDL'87 algorithms without any knowledge of the precise VHDL syntax;
2. execute different kind of transformations;
3. execute compiler transformations which compute VHDL code to remove redundancies, unrool loops, propagate constants, etc.
- compilation of the input description into a flow graph (CDFG) [26]: flow graph transformations are used to change the degree of parallelism;
- hardware-specified transformations: to optimize and refine by his experience, the process synthesis;
4. schedule and allocate the resources: different heuristics have been implemented to give the opportunity to choose between different architectures; a new approach for allocation, CVFP (Clique Valued Fast Partitioning) is used [9].

4.2 SIGNAL

4.2.1 The language

The SIGNAL language [19] was developed at IRISA Rennes to design and implement real-time systems. The basic object handled by SIGNAL is a possibly infinite sequence (or a stream) of values called a signal.

A SIGNAL process is a set of relations (equations) between signals, specifying both constraints on their values and timing. SIGNAL is a formally defined language with a small set of basic operators. The kernel language is defined with the following operators:

- Functions: Usual functions AND, OR, +, *, etc. from instantaneous domains (boolean, integer...) are extended to signals.
- Delay: The delay operator gives access to past values of a signal. The SIGNAL statement corresponding to delay is: \( ZX := X \# k \), with \( ZX \) init \( X \). For every occurrence of the signal \( X \), \( ZX \) carries the previous value of \( X \). This operator is extended to a delay of \( k \) (\( k > 1 \)).
- When: This operator allows one to conditionally extract values from a given signal. The expression:
Y := X when C, where X and Y are signals of the same type and C is a boolean signal describes a downsampling of X. Y is present when both X and C are present and C is TRUE ("\( \perp \)" represents the absence of value).

\[
\begin{align*}
\text{f} & : x_1 \quad y_1 \\
\text{g} & : T \quad T \\
\text{h} & : T \quad T \\
\text{i} & : T \quad T \\
\end{align*}
\]

- **Default:** This operator allows the merge of signals of the same type: \( Z := X \text{ default } Y \).

One can build elementary processes as equations between signals \( X := \text{expr} \), where \( X \) is a signal and \( \text{expr} \) is any valid SIGNAL expression built using the kernel operators and other signals (e.g., \( X := (A \text{ when } B) \text{ default } (C \text{ or } D) \)). More complex processes can be built using the commutative and associative composition operator \( \circ \). The process \( (P_1 \mid P_2 \ldots \mid P_n) \) simply denotes the union of the equations expressed by \( P_1, P_2, \ldots, P_n \), where \( P_i \) is an elementary process or a composed process. Consider the following example:

\[
\begin{align*}
(1 \quad Y := (0 \text{ when } R) \text{ default } ((Y+1) \text{ when } C) \text{ default } Z \mid ZY := Y \& \bar{1})
\end{align*}
\]

This process behaves as follows: when the signal R (reset) is true Y is equal to 0, otherwise it is equal to the previous value of Y (noted ZY) plus one when the signal C (clock) is true, otherwise, it is equal to ZY. A possible execution is shown hereafter, ZY is initialized with 0:

\[
\begin{align*}
&X: \quad T \quad F \quad T \quad F \quad F \\
&Y: \quad 0 \quad 1 \quad 2 \quad 3 \quad 4 \quad 0 \\
&Z: \quad 0 \quad 0 \quad 0 \quad 1 \quad 2 \quad 3 \quad 4 \quad 4 \\
&\text{TMP}[i] := \text{TMP}[i-1] + (X[i] \cdot \overline{H[i]})
\end{align*}
\]

Other operators have been defined from the kernel that permit the reduction of programming effort, they can be found in [2].

A description of a FIR16 filter in SIGNAL is shown on figure 3, using the mixed schematic/textual program entry. It uses the sliding window operator (window) and the process array constructor (array). \( \text{X} \) and \( \text{Y} \) are respectively the input and the output of the filter; \( \text{N} \) and \( \text{H} \) (which is a vector of size \( \text{N} \)) are parameters. The window defines \( X \) as a vector of size \( \text{N} \) such that at any instant \( t \), \( X(t) = X_{n-\text{N}+1} \), the array expresses that at any instant \( t \), \( V_i = 1, \ldots, n \), \( \text{TMP}[i] := \text{TMP}[i-1] + (X[i] \cdot \overline{H[i]}) \).

![Figure 3: The FIR16 in SIGNAL](image)

### 4.3 ALPHA and its synthesis environment

ALPHA was designed for the synthesis of systolic arrays [20, 22]. It is a pure functional language based on recurrence equations.

```plaintext
system transverse_filter(M:N, H:O) parameter;
    x,d: \{ t : t >= 0 \} of integer; mu: integer
returns (y: \{ t : t >= M \} of integer);
var:
    x: \{ t : t >= 0 \} of integer;
    y: \{ t : t >= 0 \} of integer;
    x[t,k] = case;
        \{ k = 0 \} : 0;
        \{ k > 0 \} : H[t-1,k] * x[t-k];
    esac;
    y[t] = Y[t,M-1];
    H[t,k] = H[t-1,k] + mn[\circ] * e[t] * x[t-k];
tel;
```

Figure 4: An ALPHA program

To introduce the ALPHA language, we consider the transverse filter in figure 4. An ALPHA program is a system of recurrence equations. A system declaration consists of a name, parameters, input variable declarations and output variable declarations. All variables in ALPHA are strongly typed and based on polyhedral index domains. Variables and expressions denote mappings from an index domain (the set of all integral points within a specified polyhedron) to values in the value domain (integers, reals, or booleans). Polyhedra are set of integers bounded by a finite number of linear inequalities, \( \{ z \in Z^n \mid A z \geq b \} \). The syntax of a polyhedron in ALPHA is (for example):

\[
\{ i,j \mid 0 < i < n; 0 < a < j \}
\]

A system may also have local variables, which are declared after the system header using the keyword var. The system of equations that define the variables follow the declarations and are delimited by the pair of keywords let and tel.

The language is equational, and each equation \( \text{variable}_{HS} = \text{expression}_{RHS} \) names a variable on the LHS and has an expression on the RHS. In a first approximation, each equation may be thought of as a quantified statement, or as a pointwise operation. For example, \( e[t] = y[t] - d[t] \) can be read \( \forall t \in \text{domain}(e), e_t = y_t - d_t \), or as a pointwise operation between the semi-infinite "vectors" \( e, y \) and \( d \).

It is possible to specify affine dependencies, as in equation:
\( H(t,k) = H(t-1,k) + mu() \circ e(t) \circ x(t-k) \).

Here, \( H \) is defined by means of its shifted version \( H(t-1,k) \).

A case statement provides a means of selecting one expression or another, depending on the value of an index. This is shown in the definition of \( Y \) in the example.

The precise semantics of ALPHA is defined in [22]. It describes the effect of each operator - pointwise operators, case, and dependencies. As variables are functions from an index domain to a value domain, operators are function combinators. Their effect is not only to combine values, but to combine domains: for example, the domain of an expression \( X + Y \) is the intersection of the domains of \( X \) and of \( Y \), whereas the domain of \( case X : Y esac \) is the union of the domains of \( X \) and \( Y \).

ALPHA restricts the kinds of programs which can be represented to those where the index domains of variables are unions of convex polyhedra, loop bounds are piecewise affine functions of indices, data dependencies are affine index functions, and there are no data-dependent branches. These restrictions limit the expressive power of ALPHA, though not to the point of rendering it trivial and useless. A great number of linear algebra and digital signal processing algorithms fall within these limitations. A proper subset of the language (uniform ALPHA) is Turing complete, and there are analysis issues about an ALPHA program that are undecidable [23]. In return for this reduced expressive power, we can make use of a set of powerful static analysis tools that enable us to perform automatic compilation.

Synthesis of regular architectures from ALPHA specifications is done by a stepwise refinement approach, using basic transformations. First of all, ALPHA obeys the referential transparency rule of functional languages. One can thus substitute any variable by its definition, without altering the meaning of a program. Moreover, one can show that any equation can be reduced to a normal form, with a single level of case and of dependence functions. For example, the program of figure 4 is in normal form.

Of all the transformations which can be done on an ALPHA program, the change of basis is the most important. It is similar to the reindexing of loop variables and other loop transformations done in vectorizing and parallelizing compilers.

The synthesis transformations are implemented using Centaur and Mathematica. The typical synthesis trajectory includes data placement, operation scheduling, control signal synthesis, time and space dependence separation. The result is a description of an architecture at the Register Transfer Level (see [20] for more details).

4.4 Lustre: a purely functional formalism for describing reactive systems

Why Lustre? The VHDL language is becoming a standard in the domain of circuit design. But it is basically a simulation language. Some constructs are semantically intricate, particularly due to the problem of delta-delays, and can lead the user into traps. In order to canalize the multiple ways of using it, and for escaping traps, one is forced to define subsets and modeling styles. In many synchronous applications, the whole language is not necessary. So it may be useful to consider, on top of VHDL, other languages more suited to the task of specification, more focused on particular domains, and to investigate languages with the needed features: clarity, concision, power, abstraction, and referential transparency. In this respect, the synchronous language Lustre [15, 14] is a good candidate. Its expression is "natural", and may be graphical; this kind of formalism was first used by mathematicians. It is executable, and suited to synchronous systems, software and hardware as well [14, chap. 2]. Lustre is used by Merlin-Gérin for the control of its nuclear power stations and by Aérospatiale for the design of on board calculators for the Airbus aircrafts. The mathematical soundness of Lustre permitted the construction of verification tools, used to prove temporal properties [14, chap. 3]. The transformation tool TRANSE (see below) is also based on the mathematical properties of the language.

What is Lustre? It was designed at IMAG Intitute, Grenoble, France, ten years ago. Data-flow streams are essential objects of Lustre. They are potentially infinite lists of scalar values, representing successive signal values at each clock pulse. Boolean streams named clocks allow to manipulate sub-streams (sampling). Lustre programs are structured in "nodes", analogous to "black boxes" with input/output connections. The body of a node consists of a system of stream equation definitions. The Lustre

```plaintext
node scalprod (const n:int; x,y:int*n) returns (p:int)
var xy0 : int;
let p = with y for el then xy0
  else xy0 + scalprod(n-1,x[1...n-1],y[1...n-1]) ;
xy0 = x[0]*y[0] ;
end

tel
node delay (const n:int; x:int) returns (dx:int*n)
let dx = [x] | 0*(n-1) \rightarrow prod(dx[0...n-2]) ;
end

tel
const N = 16 ;
node FIR16 (N:int*H; x:int) returns (y:int) ;
var dx : int*N ;
let y = scalprod(N,dx,H[N-1...0]) ;
dx = delay(N,x) ;
endтел.
```

Figure 5: A Lustre program for the "FIR16".

Lustre program in figure 5 is structured in three nodes. The first one is a scalar product parameterized by the vector size n, expressed by static recursion. The second produces a pipeline of n delayed values of its input x, using the "polymorphic extension" of the prod operator, which returns its argument delayed by one unit of time. The construction "0*(n-1) \rightarrow x" is for initialisation. The "1" operator is an array concatenation. Lustre is in some respects close to the Ruby and Silage languages.

TRANSE: a tool for transforming Lustre descriptions. The aim of this tool is to support a top-down design method, providing a path between high level behavioral specifications and lower levels, for which automatic synthesis tools or compilers are available. With the TRANSE tool, Lustre specifications are interactively transformed through successive rewrites, until satisfactory (in terms of operational properties) programs are obtained. Program transformation techniques seem well suited to the hardware domain, since the strong constraints attached to this domain limit the number of potential solutions. Several examples have been successfully processed with TRANSE, belonging to different domains: signal processing, CPU architecture and arithmetic dedicated circuits [7].

Lustre and TRANSE in the ASAR project. Lustre (as well as SIGNAL) is a good candidate for a hardware/software co-specification language, mainly because
its abstraction level is good, and it can be accepted by hardware as well as software cultures. Moreover, mapping LUSTRE descriptions to hardware is rather straightforward, and translators from LUSTRE to C are available. Its well founded semantics should allow good estimations of quality measures (operational properties) under different partitionings and mappings. LUSTRE programs produced with TRANSE and destined to a hardware mapping should be synthesized by ASAR tools like OSYS or GAUT, via a VHDL or Internal Format translation.

4.5 GAUT

GAUT is a pipeline architectural synthesis tool, dedicated to real time Signal and Image Processing applications. The algorithm and the formal library of operators is described with the assistance of VHDL. The total determinism in the DFG to be executed leads to code transformations and description optimizations:

- Fixed iteration loops are unrolled;
- Variables and signals are propagated;
- Conditional assignments are resolved by creating multiplexed values [10].

These transformations allow to obtain the design optimization independent from the specification style. The history Flow Graph is synthesized according to a generic model core of Digital Signal Processors. Finally, the synthesis leads to the generation of a structural and functional VHDL description of the designed architectures.

4.5.1 Design Techniques and generic model of architecture

First, GAUT executes the module selection task, and then simultaneously executes the tasks of scheduling and assignment [21]. The aim of the module selection is to select the number and types of functional units from a given library, for given behavioral description and real time constraint. In GAUT this task is a time-constrained allocation. When several modules can execute the same operation with different time and cost, the problem is to select an optimal set of modules that respect the constraint for a minimal cost. Different technologies and different logical implementation lead to a real component library that may contain several types of Functional Units, each with different characteristics such as delay, size, functionality, power dissipation, etc.

GAUT allows multifunctional units and pipeline operators to be used. Furthermore, this enables a hierarchical design strategy by using operators resulting from a previous synthesis. The parameters of the library for functional units are area, delay or latency, number of pipe stage and inputs, bitwidth of each input/output.

The scheduling operation algorithm is performed by a list scheduling algorithm. This one is based on a mobility heuristic which depends upon the availability of allocated operators. Finally, a global optimization on the number of interconnection (multiplexers, tri-state, demultiplexers, busses) and storage (registers, number of memory banks) units is performed. This is done by time and space bus partitioning and by a reassignment of the operations on operators with branch and bound techniques using heuristic cost calculation.

The topology of the Processing Unit model is based on an elementary cell which includes several multiplexers, registers, and demultiplexers interconnected for the requirements of the application around an operator (figure 6). Each cell is dedicated to the processing to be executed.

The various cells communicate via a parallel multi-bus network. The control model integrates two features: the use of the pipeline in order to control the operators and the registers; and the use of a multi-phase clock to control the events.

![Figure 6: Structural model of the generic architecture](image)

4.5.2 Output Interface

The output interface generates the structural description of the processing part and the functional description of the control part of the synthesized architecture. This file is in the VHDL format which is the input of logic synthesis tool as COMPASS. An example of architectural synthesis for the FIR16 is shown in figure 7. GAUT generates some graphical and textual performance evaluation measures, that can lead to modify the specification of the considered algorithm [24]. These estimations may be used during the Hardware/Software partitioning.

5 Towards an internal formalism

5.1 The common formats of synchronous languages

GC was defined as one of the common formats of synchronous languages (such as SIGNAL, LUSTRE and ESTEREL) and of the commercially available environments of these languages (such as SILDEX, developed by the TNI company for SIGNAL) [17]. GC is a data-flow graph parallel format. It is a hierarchical format: a node can be a graph. GC is used as a target format for the synchronous languages SIGNAL and LUSTRE. It is, moreover, a natural candidate for separate compilation, for performance evaluation and placement and routing on a distributed architecture, and for hardware synthesis. This is indeed of special interest for the ASAR project. It can be used also by tools issued from imperative languages: a GC code can contain imperative structures allowing to describe actions linked to the other nodes of the graph with dependencies which are sequencing constraints.

The other common formats of synchronous languages are an imperative parallel format, IC, and a sequential automaton format, OC. These formats (see figure 8), which...
are based on the same semantic model than synchronous languages [1], have been defined as being a candidate public standard for synchronous programming technology. They are a major component of the SYNCHRON Eureka project.

![Diagram of the Common Base for Synchronous Programming]

Figure 8: The Common Base for Synchronous Programming

5.2 The format GC

The vertices of a GC graph are:

- flow definitions which are explicit equations linking input flows to one output flow;
- instances of GC "nodes" representing sub-graphs;
- procedure calls.

The edges of the graph are implicitly represented by the identity of names (or, more exactly, of indexes, since all objects of a GC code are designated by an index in some table of objects) of connected flows. The dependencies between the input flows and the output flows of a vertex are conditioned by the clock (a clock represents a sequence of instants) at which they are effective. For the vertices which are flow definitions, these dependencies are induced from the operators used in the expression of definition; they are implicit, but they can be explicited in a table of dependencies. For the vertices which are instances of GC nodes, the dependencies are generally specified in the interface of the node. Moreover, explicit dependencies can be added.

In addition to data dependencies (which are dependencies between flows), control dependencies can be specified between the vertices of the graph.

The general organization of the format is not detailed here. It allows a modular description of programs, which are structured in entities (data blocks—where for example constants, types and associated functions are declared—, GC nodes and interfaces of these nodes, IC or OC modules). Each entity is composed of tables of objects. Specific informations (required execution times, for instance) can be added as pragmas.

The basic entity of the GC format is the node (a GC node represents a description of graph, as we said above). A node consists of an interface, a description of data, and a body. The interface of the node describes its properties which are visible from the outside: external flows, assertions, synchronizations and dependencies that concern these external flows. The body of a node describes the definitions of the values of flows (vertices of the graph); it includes also assertions, synchronizations and dependencies.

The main operators that can be used in the expressions of definition are the following:

- usual arithmetic and logic functions, predefined or not;
- functions $\text{pre}$ (delay: the value of the output is equal to the previous value of the input) and $\$\text{by}$ (initialization);
- functions $\text{window}$ (sliding window) and $\text{select}$ (access to an element in a window);
- extraction function $\text{when}$ (the value of the output is that of the first input when this first input is present, and when the second input, which is a "pure" flow—without associated value—is also present);
- deterministic merge $\text{default}$ (the value of the output is that of the first input when this first input is present, and that of the second input when this second input is present but the first one is not);
- function $\text{clock}$ (the output is the pure flow present when the input is present);
- function $\text{tt}$ (the output is the pure flow present when the input, which is a boolean, is present and true);
- different functions on pure flows (or clocks)...

As an example, a microscopic representation of the flow graph associated with

\[
\text{if } c \text{ then } x := y + 1; \text{ else } x := z;
\]

is given by the following piece of GC code, for which a graphic view is shown on figure 9:

```gc
... definitions: 7 -- comments
0: define: 0 $g0(0.1); -- c1=#tt(c)
1: define: 1 $g0(0.1); -- cc=#not(c)
2: define: 2 $g0(0.1); -- c2=#tt(cc)
3: define: 3 $g10(0.0,0); -- y1=#when(y,c1)
4: define: 4 $g14(3,#1); -- x1=#plus(y1,1)
5: define: 5 $g10(0.2,1); -- z1=#when(z,c2)
6: define: 0.3 $g9(4,5); -- x=#default(x1,z1)
end:
```

![GC Graph]

Figure 9: A GC graph

5.3 The Internal Common Formalism

The current version of the format GC does not include a complete set of operators to manipulate arrays, or more generally, polyhedral domains such as those that can be used in ALPHA. The ICF should be an extension of GC including operations on such domains. This extension is still in progress.

6 Conclusion and future work

Three main objectives have been defined in this project:

- integrate the different architectural synthesis tools in a generic environment,
• define an internal common formalism (ICF),
• study the actual state of art in cosimulation in order to extract the different characteristics needed to be automated.

The first objective has been obtained with the implementation of Asar.0 which includes the system manager of the different tools. The designer disposes of a coherent framework, where different architectural synthesis tools co-exist. The inter-communication between formalisms being based on the ICF.

For the second point, we have studied different benchmarks in order to compare the internal formalisms used by each tool. The last two candidates were the CDFG (Control Data Flow Graph) [10] and a subset of GC [17]. The subset of GC seems well suited for our purposes. Figure 10 shows the communication between the different tools and the interface with other languages like C, Fortran, etc.

Figure 10: Communication between tools

The third point presents the different techniques now used in Hardware/Software codesign [18, 13]. Three main approaches are emerging: software oriented [8], hardware oriented [12], and a theoretical approach [18]. The next step is to apply these different techniques on benchmarks like FIR16, MICDA, in order to give quantitative results.

References