# Framework and Multi-Formalism: the ASAR Project

P. Asar*
Correspondence should be sent to Michel Israël
LaMI Université d'Evry, Bld. des Coquibus, 91025 Evry Cedex
email: israel@univ-evry.fr

**Abstract**

The main concern of the research project ASAR is to build a multi-formalism framework oriented towards Architectural Synthesis. This paper describes the CENTAUR system, used to build this framework, and the common intermediate data-flow format GC, that will be the common denominator of the different formalisms and tools available in that framework.

**Keywords:** Framework Technologies, Exchange Formats, Multiple CAD tools integration, Intermediate Data Flow Graph.

## 1 Introduction

This paper describes a research project—named ASAR[1]—grouping together six french research teams. A main concern of this project is Architectural Synthesis Framework and user-interface management.

Most of these teams 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 [4] system, already used by three teams as a common generic software environment for user-interfaces, formalisms and tools management.

## 2 ASAR structure

The overall structure of the ASAR framework is sketched in figure 1. The ASAR framework provides a set of languages and tools oriented towards particular target architectures or domains.

SIGNAL [12] and LUSTRE [8] are data-flow oriented synchronous languages especially designed for real-time applications. They can be used to specify hardware/software systems at different levels: system level

*P. Asar is a generic name for the ASAR project members, namely, P. Aubry (IRISA), M. Auguin (I3S), M. Belhadj (IRISA), J. Benzakki (LaMI), T. Bouguerba (LaMI), C. Carrière (I3S), G. Durrieu (CERT-Onera), Th. Gautier (IRISA-INRIA), M. Israël (LaMI), P. Le Guernic (IRISA-INRIA), M. Lemaître (CERT-Onera), E. Martin (LASTI), P. Quinton (IRISA-CNRS), L. Rideau (INRIA-Sophia), F. Rousseau (LaMI), O. Sentieys (LASTI).

[1] ASAR stands for "**A**telier d'accueil générique pour la **S**ynthèse **AR**chitecturale".

and behavioral level for instance. They provide SIGNAL/LUSTRE to C compilers and proof checking tools. The SIGNAL environment, which includes a mixed graphic/textual program entry, provides a multi-level VHDL generator prototype (behavioral—that allows the use of synthesis tools such as Synopsys, RTL, gate level).

TRANSE [7] is an interactive transformational tool based on the LUSTRE language: a circuit is obtained through successive semantic preserving rewritings of LUSTRE programs, until good operational properties are obtained.

ALPHA [13] is a functional language based on recurrence equations, for the design of parallel hardware accelerators; it allows the expression of regular algorithms and/or architectures and the synthesis of regular architectures by stepwise refinement.

GAUT [14] and OSYS [9] are high level synthesis tools from behavioral VHDL descriptions. GAUT is dedicated to signal processing designs; OSYS is a general purpose prototype tool.

A more complete presentation of these tools can be found in [2].

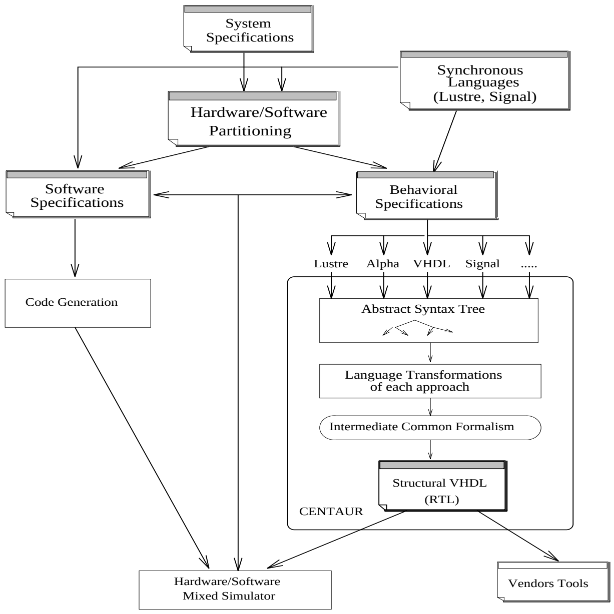


Figure 1: Overall structure of ASAR

Three components of ASAR are of main importance:

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

**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 semantic 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 semantic seems to be the most appropriate. It is the reason why the *GC* formalism is under investigation for the ICF. GC [15] is a common format for synchronous data-flow languages, developed in the french *projet SYNCHRONE*.

**VHDL** The VHDL language, which is now a standard [1], 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.

The main concern of this paper is to describe the framework and the common formalism: the CENTAUR system is presented in section 3 and the data-flow graph format GC is described in section 4.

# 3 Centaur: a Distributed Programming Environment

## 3.1 Centaur

The CENTAUR system [4] 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. 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 recently evolved in a set of cooperative tools, whereby we can couple CENTAUR with existing tools, e.g., an editor, a parser, or a compiler. This has two benefits: the generated system is no longer monolithic (different generated tools may run on different processors) and, more importantly, one can integrate existing tools (developed independently of CENTAUR) in the generated environment. Actually the system can be seen as a toolkit that helps in developing tools for manipulating programs or other structured objects, as well as a multi-formalism **framework** providing communication facilities and an homogenous user-interface to various independent tools.

The cooperation of tools implies communication and the sharing and exchange of data between these tools. In the following section, we present SOPHTALK [10, 11], an implementation of such a communication platform.

## 3.2 Sophtalk

Recent progress in software engineering techniques has consolidated the idea that any tool that performs a task (editor, button, typechecker, etc.) should be able to operate in isolation, without having to worry about sources of information it requires or recipients of information it generates [5, 6]. Rather than converse directly with its comrades, a tool announces an important event by broadcasting an appropriate message "into the world." The term *multicast* is more appropriate since a message only reaches the ears of those tools that have already declared their interest in such a message. Listening tools may react to a message, perform some activity, and emit their own messages. If no tools listen to a given message, it falls on deaf ears.

This event model of communication has been widely used to coordinate communication between several processes, but the principle of separating a tool's task from its communication needs also applies to single process tool interaction. A tool emits a message whenever it requires information, or when it accomplishes (or fails to accomplish) an important task. "I have finished compiling," "Someone has modified this file," or "The current selection has changed" are examples of significant events. Events may occur at any moment, so this model of communication is naturally asynchronous, but one may impose synchronization through design choices.

With SOPHTALK, one may design a system as a network of event-conscious tools. This kind of design allows developers to add, modify, substitute, or remove tools from the system without disturbing global behavior.

SOPHTALK implements an event model of communication. System objects, called *stnodes,* emit messages when significant events occur, such as the termination of a computation, a request for a service from another object, error conditions, etc. Messages circulate asynchronously in a network of stnodes. An stnode's type determines which messages it will receive; upon reception, an action corresponding to the message and stnode instance is triggered. An example of *stnodes* is illustrated by the figure 2.

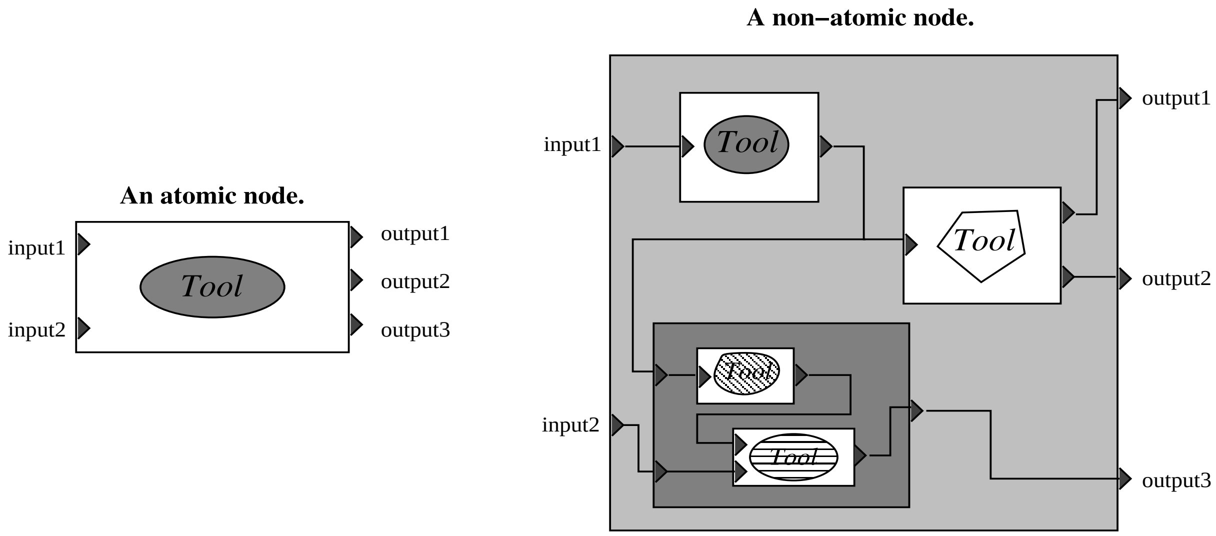


Figure 2: *Atomic and non-atomic stnodes*

## 3.3 Connecting Centaur with external tools

### Some examples

Several projects have experimented with Centaur and connecting it to external tools using Sophtalk. 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, Sisyphe, Ulysses), with plotting/surface drawing tools (e.g., Gnuplot, ZicVis), and with theorem provers (e.g., HOL, Isabelle, Coq). 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 programming environments: selecting the next subgoal to attack, saving an incomplete proof, browsing the various theories available, choosing tactics; 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.

## 3.4 Connecting tools within ASAR

In the ASAR project, some tools have already been developed under Centaur:

- Osys: an environment for **VHDL**
- Transe: for Lustre program tranformations
- Alpha du Centaur: an environment for the Alpha language.

A structure editor (parser and pretty-printer) has also been developed using Centaur for the Signal language. The ASAR common framework built with Centaur allows these various tools to easily communicate, and to compare their results obviously. Moreover this common framework facilitates the design of the Internal Common Formalism **ICF**.

Furthermore Centaur's new architecture allowing integration of existing external tools is also useful in the ASAR context:

- communication between closely related tools as Osys and Gaut is facilitated;

- communication with the external SIGNAL compiler makes the existing CENTAUR environment up (since a syntax editor already exists);
- communication with MATHEMATICA makes the cooperation between the two ALPHA DU CENTAUR's implementations (Centaur and Mathematica) easier;
- communication with a graph server allows computed graphs visualization.

Note that the two first communication (between Osys and Gaut, and between the CENTAUR framework and the SIGNAL compiler) will use the ICF format to transfer data. This Internal Common Format is also used to translate the framework results into the external tools input languages (structural or synthesisable VHDL, EDIF, C, etc.).

# 4 An Internal Common Formalism for ASAR: GC

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.

GC, as a synchronous data-flow format, is presumed to be the ICF for all the participants to the ASAR project. In this section, we present the place of GC in the common formats of synchronous languages, its main features and functionalities, and the tools already implemented applying to it.

## 4.1 The common formats of synchronous languages

GC was defined as one of the common intermediate formats of synchronous languages. The development of these formats has several major objectives:

- Establishing a standard for synchronous programming, the formats would support the largest part of the compilation process of all synchronous languages.
- Developing targeted user interfaces to the formats should be easy and of low cost.
- Developing targeted code generators for particular architectures should also be easy and of low cost.
- Providing tools and services within the synchronous technology (proofs, simulations, performance evaluation...) could be performed by different companies.

A first public version of the definition of the formats has been issued in june 1993 [15]. The formats are now a major component of the European SYNCHRON Eureka project, grouping together european companies and research centers involved in real-time studies and developments. A crucial issue of this project will be the standardization of the formats.

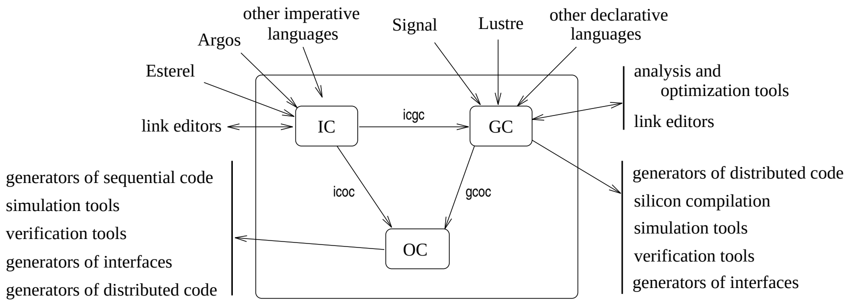


Figure 3: The Common Base for Synchronous Programming

The common formats consist of three formats relying on the same structures :
**IC,** a parallel format of imperative style.
**GC,** a parallel format of data-flow style, targeted in particular to LUSTRE and SIGNAL.
**OC,** a sequential format to describe automata.

A general view of the formats, expected translators between the formats, and planned relations with different tools is depicted in figure 3.

It should be noted that the word "format", used in contrast with "language", means that this object will not be accessible to end users but is rather designed to be handled easily by computers (neverthless, a readable decompilation such as that used in this paper can be very useful. . . ).

## 4.2 Main common features

**Main structures:** As GC is a format (compared to a language), its syntax is strongly structured: it has been designed to be well accepted by automatic analysers. A **program** is a list of **packages**, which are lists of **entities**, themselves made of **tables** of **objects** (various entries of the tables). These structures are shown in figure 4. Packages do not have any semantics: they are used to put together entities linked by contextual meanings. They are referred to by their identifier; all other objects are referred to by indices, which avoids complex syntactical research of identifiers.

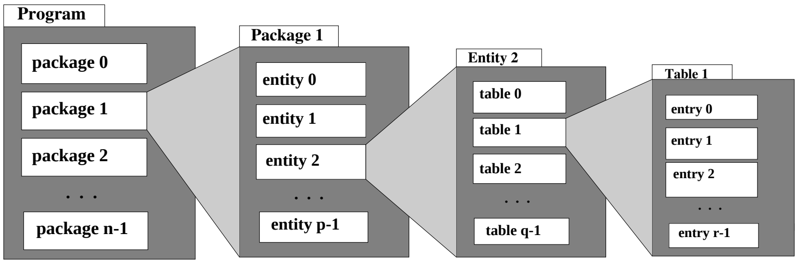


Figure 4: The main structures of the three formats

**Comments:** Though the formats are not designed to be read directly by users, the presence of comments, that may be useful for debugging, is allowed by the grammar. An other way to put comments in programs is the use of the pragma `%comment:%`.

**Pragmas:** Pragmas are "executable comments", kept by tools like translators between formats, and adding informations to objects to which they refer. At the moment, two pragmas are accepted by GC tools (especially the compiler): `%main:%` and `%comment:%`. The first one applies to a node (see 4.3), meaning that this one is to be considered as the principal node of a program. The second one can be found anywhere in a program. Pragmas appear as the easiest way to make GC evolve, once the main structures will be improved by the implementation of the first tools.

**Addressing mechanism:** As said before, the formats, designed for easy automatic analysis, own a very simple addressing mechanism. There are three ways to refer to an object:

- with a prefix, if the object belongs to common or GC **predefinitions** (see below). The prefixes are respectively `$` and `$g`.
- directly, by the index of the object, if it belongs to the current entity.
- indirectly, if the object belongs to an other entity. This other entity has to be imported by the current entity, by appearing as one entry of the table of **importations**. The index, called then *coumpound-index*, is made of the index of the corresponding importation, followed by the local index of the objet in the imported entity.

With this mechanism, all the objects can be reached by simple translations in packages and entities.

**Entities:** They are made of tables (distinguished one from the other by a keyword), and each of them can hold a table of importations. Importations are used in an entity to refer indirectly to objects declared in other entities. GC entities can be data blocks, interfaces, local or external nodes. Data blocks can hold tables of *types*, *constants*, *functions* and *procedures*. Those objects are only declared, which allows their use (their definitions are external and specified in an other way).

**Predefinitions:** Many applications use simple types, built-in in most languages. Those types are declared in a special data block called *common predefined data block*. They are `$pure` (clocks), `$boolean` (logicals), `$integer`, `$string` (strings of characters), `$float` and `$double` (extended floats).

The predefined data block also declares constants: `$top` (always-present clock), `$true` and `$false` (always-present logicals). It also provides the most used functions on predefined types. This data block does not need to be imported by entities: common predefinitions are supposed to be known in all the programs written in any of the three formats. Any entity can refer to predefined objects without importing the predefined data block.

## 4.3 A brief overview of GC

GC is a hierarchical representation in the data-flow or block diagram style of synchronous programs, augmented with explicit control. The declarative part of a GC code has a synchronous semantics, presented in [15].

Since GC is a data-flow format, the basic object it deals with is the **flow**. A flow is a possibly infinite sequence of typed values with an explicit associated **clock** denoting the instants at which values are present.

**Structure of a GC code** A GC code is made up of a list of declarations of three kinds of entities: data blocks, **interfaces** and **nodes**. A node represents a declared sub-graph. It can be instanciated or activated (see below) with the only knowledge of its interface. There are two kinds of nodes: *local* ones and *external* ones. External nodes are nodes for which the GC description is not avaible. In particular, they can be the GC perspective of an imperative module. This is to be the link between GC and IC codes.

**Interfaces** The interface of a node is the summary required for use of the node as a "black box" in its instanciation context. It is made up of:

- **a data part** which describes mainly in form of data block imports, the types of flows entering and exiting the node (called *interface flows*), and, in the form of flow declarations, the interface flows and their clocks;
- **dependencies** which enable the specification of a partial communication order among the interface flows within the same instant;
- **assertions and synchronizations** which are the properties expressed on the interface flows.

**Local nodes** A GC local node has an interface, data and a body.

- As seen before, the interface of a node describes the properties which are visible from the outside.
- The data describe, in the form of **data block imports**, the types, constants, functions and procedures used by the node, and in the form of **flow declarations**, the local flows (the knowledge of which is not necessary outside of the node).
- The body of a node is made up of the following items:

  **Definitions of values of flows:** They express the computation of values of output (and local) flows in terms of functions of input (and local) flows. They are deterministic and obey the referential transparency principal (their lefthand side can substituted for the righthand side in all cases). The definitions can be *equations*, *node instanciations*, *procedure calls* or *activations of nodes*[2].

  A set of definitions can be seen as a data-flow graph. The vertices of the graph are the definitions. The edges are implicitly represented by the identity of indices of connected flows. The dependencies between the input flows and the output flows of a vertex are conditioned by the clock at which they are effective. For the vertices that are equations, 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 (see below). For the vertices which are node instanciations or activations, the dependencies can be specified in the interface of the node. Moreover, explicit dependencies can be added between definitions.

[2] Several syntactic occurrences of instanciation of the same node produce distinct objects. At the opposite, several syntactic occurrences of activation of the same node make reference to the same object.

**Synchronizations:** They specify constraints governing the semantics of the program.

**Assertions:** They are properties expressed on the flows of the current node. They are not supposed to be computed by all tools.

**Dependencies:** They allow the specification of a partial execution order between communication flows in any instant.

An example of GC program is shown on the left side of figure 5; it is the translation of the data-flow graph obtained by the compilation of a SIGNAL program describing a FIR. An equivalent representation of the same program expressed in VHDL is shown on the right part of the figure. In this example, loops have been unfold, as it is done in GAUT for instance.

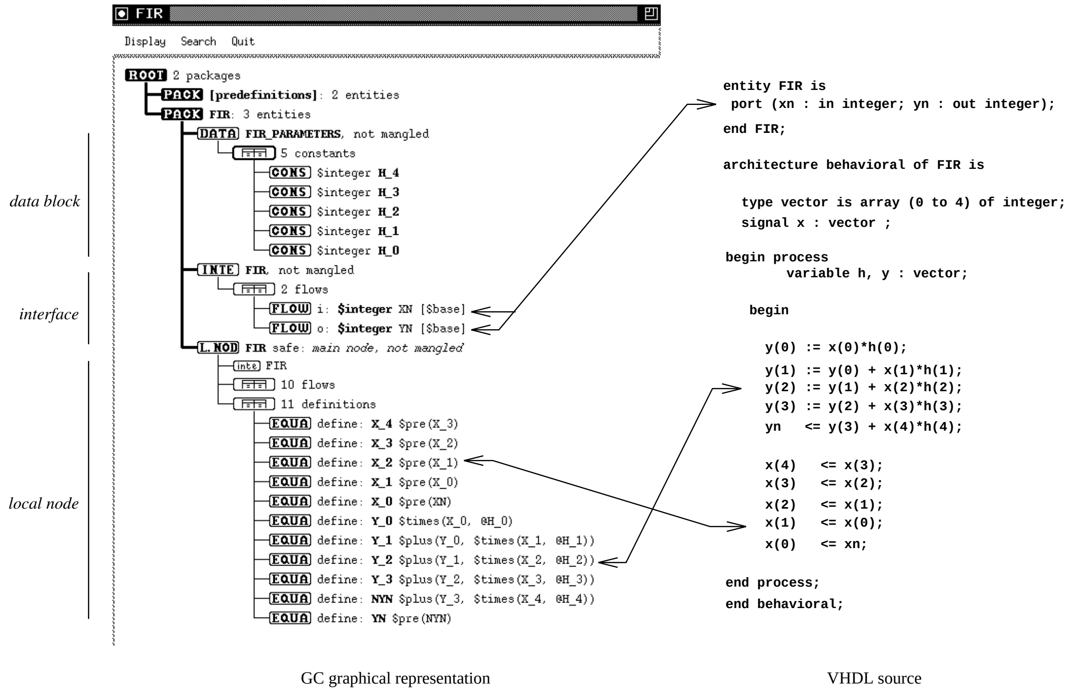


Figure 5: GC and VHDL equivalent programs

## 4.4 GC existing tools

In this subsection, we describe the existing tools applying to GC. They have been implemented in the IRISA/INRIA software environment of the synchronous language SIGNAL (see figure 6).

**An internal representation:** All the tools manipulating GC graphs use the same internal object-oriented representation. As GC (and the other formats) becomes nearly unreadable by a normal human brain as soon as the programs size more than a few pages (in particular because of the addressing mechanism), the associated decompiler produces two different files: the first one is written in GC as defined by the grammar, and the second one is a readable version where all the indices have been replaced by the corresponding identifiers.

**A graphical browser:** Useful for debugging and producing documents, the browser [3] becomes rapidly quite indispensible to GC users. In the future, new features will be added to the existing ones: multi-windowing, editing (copy/cut/paste) operations, PostScript output...

**A SIGNAL to GC translator:** An essential work of the SIGNAL compiler consists of a clock and dependency calculus; the result is a conditioned hierarchical graph (CHG). The CHG is a clock hierarchy organized as a forest of trees, where each clock owns a synchronous data-flow graph. The translator, integrated in the SIGNAL environment as an option of the SIGNAL compiler, builds a GC graph representing the clock hierarchy. The GC program presented in figure 5 has been produced by the SIGNAL compiler. The translator is unidirectionnal (from SIGNAL to GC), but the first experimentations reveal GC as a powerful representation and the format could become in a near future the intermediate code of the different phases of the SIGNAL compiler.

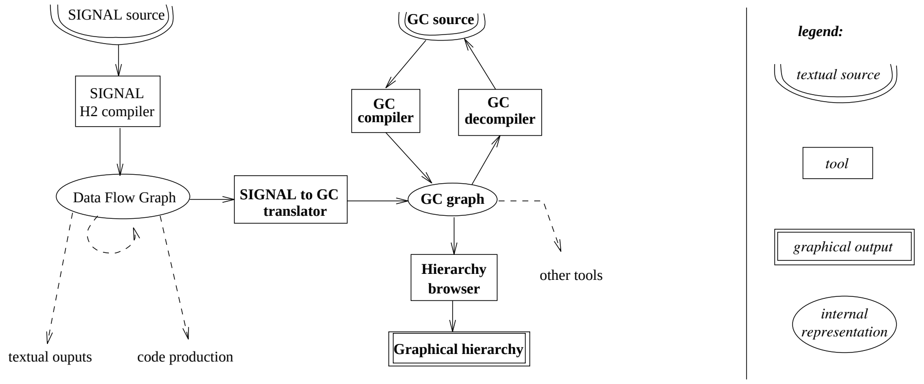


Figure 6: GC tools in the SIGNAL environment

**A compiler:** The global process of the compilation of GC is depicted in figure 7.

An intermediate phase of creation of an abstract tree from the source code has been introduced to enable the connection with other applications, without running the lexical and syntactical analysers.

The main problem of the compilation is the resolution of references, due to the addressing mechanism. This resolution is necessary to increase the security of tools applied to GC. This phase consists in attaching to each object the objects it may need in operations on the graph. The main steps are the following: *resolution of the importations,* where importations are attached to their corresponding entity, *resolution of the type-references,* where typed objects (constants, flows, functions, arguments of functions, parameters of procedures...) are attached to their type, and *last resolutions,* which consist essentially in the verification of all the objects referred to, and in type-matching.

The result of all the resolutions is an internal graph where the importations, only needed by the addressing mechanism, can be suppressed. The data-flow graph produced can be used for transformations, code production, optimizations, hardware synthesis, etc.

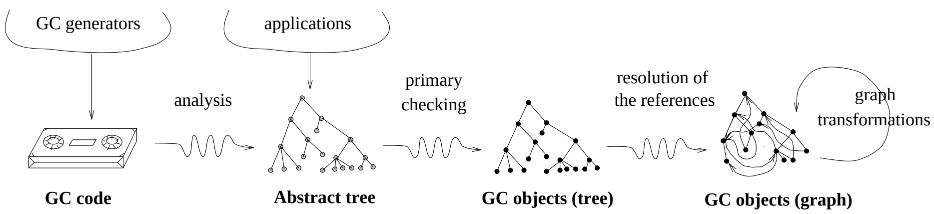


Figure 7: Compilation of GC

The browser, the translator and the compiler presented here belong to the first generation of tools applying to the common formats. The implementation of these tools has allowed the achievement of a second release of the definition of GC and revealed GC as a sure and practical format.

# 5 Conclusion and future work

Two main objectives have been defined in this project:

- integrate the different architectural synthesis tools in a generic environment,
- define an internal common formalism (ICF).

The first objective has been obtained with the implementation of Asar.0 which includes the system manager of the different tools in the CENTAUR environment. The designer disposes of a coherent framework, where different architectural synthesis tools co-exist.

For the second point, the data-flow graph format GC, with possible evolutions, seems well suited to play the important role of inter-communication between formalisms in our framework. Moreover, it will be used as a link to external tools such as generators of distributed code, formal verification tools, etc. It has now to be integrated as a possible input or output format in each one of the tools available in the ASAR project.

# References


[1] *IEEE Standard VHDL Language Reference Manual.* IEEE, 1987. IEEE Std 1076-1987.

[2] P. ASAR. Towards a Multi-formalism Framework for Architectural Synthesis: the ASAR Project. In *Third International Workshop on Hardware/Software Codesign,* Grenoble, France, IEEE Computer Society Press, September 1994, 25–32.

[3] P. Aubry and S. Machard. *Représentation graphique d'arbres sous X11R5 : Implémentation générique orientée objet, Applications.* IRISA, Rennes, to appear.

[4] P. Borras, D. Clément, Th. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. Centaur: the system. In *Proc. of ACM SIGSOFT '88: Third Symposium on Software Development Environments,* Boston, November 1988. (Also as INRIA research report number 777).

[5] D. Clément, A distributed architecture for programming environments. In *Proceedings of the Fourth ACM SIGSOFT Symposium on Software Development Environments,* Edited by R. N. Taylor, Irvine, California, December 1990, 11–21.

[6] D. Clément, F. Montagnac, and V. Prunet, Integrated Software Components: a Paradigm for Control Integration. In *Proceedings of the European Symposium on Software Development Environments and Case Technology,* Konigswinter, Germany, June 1991.

[7] G. Durrieu and M. Lemaître. Design by transformation of synchronous descriptions. In *3rd International Workshop on Algorithms and Parallel VLSI Architectures,* Leuven, Belgium, August 1994.

[8] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language LUSTRE. *Proceedings of the IEEE,* 79(9):1305–1320, September 1991.

[9] M. Israël and J. Benzakki. Osys: synthesis tool and hardware/software codesign. In *IFIP Codes/cashes93,* IFIP, Austria, May 1993.

[10] I. Jacobs and J. Bertot, *Sophtalk tutorials.* INRIA Rapport Technique no 149, Feb. 1993.

[11] I. Jacobs, F. Montagnac, J. Bertot, D. Clément, and V. Prunet, *The Sophtalk Reference Manual.* INRIA Rapport Technique no 150, Feb. 1993.

[12] P. Le Guernic, Th. Gautier, M. Le Borgne, and C. Le Maire. Programming real-time applications with SIGNAL. *Proceedings of the IEEE,* 79(9):1321–1336, September 1991.

[13] H. Le Verge, C. Mauras, and P. Quinton. The ALPHA language and its use for the design of systolic arrays. In *Journal of VLSI Signal Processing,* 3:173–182, 1991.

[14] E. Martin, O. Sentieys, H. Dubois, and J.L. Philippe. An architectural synthesis tool for dedicated signal processors. In *EURO-DAC 93,* 1993.

[15] J.-P. Paris, G. Berry, F. Mignard, Ph. Couronné, P. Caspi, N. Halbwachs, Y. Sorel, A. Benveniste, Th. Gautier, P. Le Guernic, F. Dupont, and C. Le Maire. *Projet SYNCHRONE – Les formats communs des langages synchrones.* Technical Report 157, INRIA, June 1993.