pp 83–93, San Francisco, California, January 1995.

GC: the Data-Flow Graph Format of Synchronous Programming

Pascal Aubry   Thierry Gautier
IRISA/INRIA, EP-ATR team
Campus de Beaulieu, 35042 Rennes Cedex, FRANCE
{Pascal.Aubry, Thierry.Gautier}@irisa.fr

Abstract

Based on an abstraction of the time as a discrete logical time, the synchronous languages, armed with a strong semantics, enable the design of safe real-time applications. Some of them are of imperative style, while others are declarative. Academic and industrial teams involved in synchronous programming defined together three intermediate representations, on the way to standardization:

- **IC,** a parallel format of imperative style,
- **GC,** a parallel format of data-flow style,
- **OC,** a sequential format to describe automata.

In this paper, we describe more specifically the format GC, and its links with the synchronous data-flow language SIGNAL. Thanks to the first experimentations, GC reveals itself as a powerful representation for graph transformations, code production, optimizations, hardware synthesis, etc.

1 Introduction

As real-time applications become more and more demanding in terms of safety, complexity, flexibility, modularity, integrability, etc., a new concept for real-time programming has emerged these last few years: the *synchronous programming.* The so-called *synchronous languages* ESTEREL [7], LUSTRE [12] and SIGNAL [14] are the pillars of synchronous programming, but other formalisms such as the STATECHARTS [13] are also closely related. From the user's point of view, the main characteristics of these languages are the following:

- a precise mathematical semantics,
- precise notions of modularity and encapsulation,
- formal proof and verification techniques concerning both "control" (logic, synchronization) and "architecture" (data paths and data dependencies),
- automatic generation of (possibly distributed) executable code,
- various target machines, ranging from purely software ones (C, ADA), dedicated architectures, up to hardware.

Relying on the same principles, the synchronous languages adopt different styles of programming: while ESTEREL has an imperative style, LUSTRE and SIGNAL are data-flow languages (LUSTRE is purely functional and SIGNAL is relational). It was quickly apparent that a cooperation between the approaches proposed by these languages should be highly profitable. This is why it was decided to develop common intermediate formats that would be the base of the synchronous software technology. The development of the 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 [18]. 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.

This paper mainly concentrates on one format, the data-flow graph format called GC, and its relations to the language SIGNAL. However, in section 2, a general view of the formats is given and their common structures are presented. The core of the format GC is described in section 3 and its current situation in the INRIA SIGNAL environment is detailed in section 4. Finally, section 5 mentions some perspectives.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

IR'95–1/95 San Francisco, California USA
© 1995 ACM

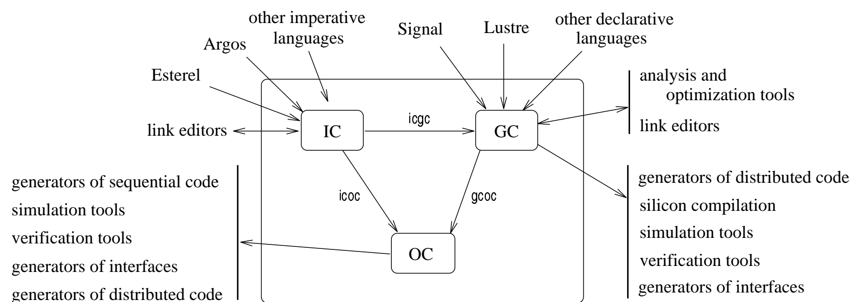

Figure 1: The Common Base for Synchronous Programming

2 The Common Formats of Synchronous Languages

First of all, 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 (nevertheless, a readable decompilation such as that used in this paper can be very useful...). The common formats consist of three formats (see 2.1) relying on the same structures (see 2.2).

2.1 The three formats

The three formats are the following:

IC, a parallel format of imperative style, targeted in particular to ESTEREL.

GC, a parallel format of data-flow style, targeted in particular to LUSTRE and SIGNAL. This format will be the main tool for generating distributed executions.

OC, a sequential format to describe automata. OC is interesting for sequential code generation (FORTRAN, C, ADA, PLC languages...) and for specific tools dedicated to automata.

A general view of the formats, expected translators between the formats, and planned relations with different tools is depicted in figure 1. A Translation from IC to GC-like formats is described in [16].

2.2 Common structures

This subsection presents the common structures of the three formats.

2.2.1 Main common features

Main structures: As IC, OC and GC are formats (compared to languages), their 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 on figure 2.

Packages are used as libraries of entities. In this sense, the package structure does not have any synchronous semantics. Gathering entities into packages allows modularity in the formats and only affects the addressing mechanism.

Packages are referred to by their identifier; all other objects are referred to by indices, which avoids complex syntactical search of identifiers.

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 syntax. Another 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. For example, two pragmas accepted by any of the three formats are `%main:%` and `%comment:%`. The first one applies to an entity, meaning that this one is to be considered as the main entity of a program. The second one can be found anywhere in a program.

Some pragmas are specific to some formats, especially IC and OC, because they are intermediate code for ESTEREL and LUSTRE for quite a long time. To be official, a pragma and its contextual meanings must be submitted to all the participants of the SYNCHRON project, and accepted by them (at least the ones who have to deal with the pragma if it is specific to a format).

Pragmas appear as the easiest way to make the three formats 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 (see 2.2.3), IC or GC (see 3.5) **predefinitions**. The prefixes are respectively `$`, `$I` 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. The 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.

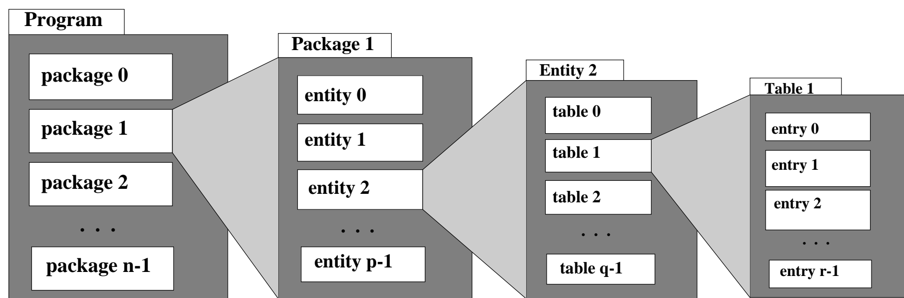

Figure 2: The main structures of the three formats

2.2.2 The entities

Entities 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.

Entities can be interfaces (in GC programs), local or external nodes (also in GC programs), IC modules (in IC programs), OC modules (in OC programs) or data blocks.

The only entity that can be used by the three formats is the **data block**. 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 another way).

2.2.3 The common predefinitions

Many applications use simple types, built-in in most languages. Those types are declared in a special data block called **predefined data block**. They are `$pure` (clocks) (see 3.1), `$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 (addition `$plus`, multiplication `$times`, logical conjunction `$and`, equality `$eq`, etc.).

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.

3 The Data-Flow Graph Format: 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 [6, 18].

An example of GC program is shown on the right side of figure 5; it is the translation of the data-flow graph obtained by the compilation of the SIGNAL program `CHRONO` (shown in figure 4).

3.1 Flows and clocks

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 values are present.

3.2 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 instantiated or activated (see 3.4) 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 available. In particular, they can be the GC perspective of an imperative module. This is to be the link between GC and IC codes.

3.3 Interfaces

The interface of a node is the summary required for use of the node as a "black box" in its instantiation 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.

3.4 Local nodes

Such a node described in the GC format has an interface, some data and a body.

- As we have seen, 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;

  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 principle (their lefthand side can be substituted for the righthand side in all cases).

  The definitions can be **equations**, **node instantiations**, **procedure calls** or **activations of nodes**[1]. We give below a few examples of definitions[2].

  Equation: `define:X $plus(Y,Z)`
  specifies that `X`, `Y` and `Z` are synchronous, and that $X_t = Y_t + Z_t$ at each instant t `X` is present.

  Node instantiation: `set:CLK N($or(X,Y),S)`
  instantiates the node `N` at each instant the clock (pure flow) `CLK` is present, with the given parameters (the instantiation of a node corresponds to a macro-expansion of the node).

  Procedure call: `call:CLK P($uminus(S),T)`
  calls the procedure `P` at each instant the clock (pure flow) `CLK` is present, with the given parameters.

  Activation of node: `do:CLK N($eq(X,Y),Z)`
  activates the node `N` at each instant the clock (pure flow) `CLK` is present, with the given parameters.

  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 instantiations or activations, the dependencies can be specified in the interface of the node.

  Moreover, explicit dependencies can be added between definitions: this is a way to specify control dependencies.

  Synchronizations: They specify **constraints** governing the semantics of the program. For instance, the following expression
  `$clkeq(C1,$clkadd(C2,C3))`
  as an entry of a table of synchronizations specifies that the clock `C1` is the upper bound of the clocks `C2` and `C3`.

  Assertions: They are properties expressed on the flows of the current node. They are not supposed to be computed by all tools, and can represent properties that have to be verified to insure the correctness of an execution. For instance, the following expression
  `$le(X,#0)`
  as an entry of a table of assertions, means that the correctness of the program assumes that `X` must always be positive or null. Some code generators could use assertions to generate *suspicious* code (with exception management).

  Dependencies: They allow the specification of a partial execution order between communication flows in any instant. For instance, the following dependency
  `X --> Y at C`
  specifies that at each instant of clock `C`, `Y` can not occur before `X`.

[1] Several syntactic occurrences of instantiation 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.

[2] To be more explanatory, indices have been replaced by identifiers.

3.5 The GC predefinitions

GC provides, for any of its codes, a predefined data block, holding the **GC predefinitions**, that we describe in this subsection.

3.5.1 The GC types

GC defines, for each type μ of the common predefinitions, a new type `$win`(μ), consisting in a sliding window of values of type μ. It also defines the type `$any`, used to declare polymorphic functions.

3.5.2 The GC functions

The first category of GC functions is the set of polymorphic functions. They are generically declared, using the type `$any`. The main ones are given below:

- **delay:**

  `$pre(X)` is a flow of the type of `X`, present if and only if the flow `X` is present; its value is the previous value of `X`.

- **initialization:**

  `$fby(I,X)` is a flow of the type of `X`, present if and only if the flow `X` is present; it takes the first value of `I`, followed by the values of `X`.

- **deterministic merge:**

  `$default(X,Y)` is a flow of the same type as the flows `X` and `Y` which is present if and only if the flow `X` or the flow `Y` is present; its value is equal to the value of `X` when `X` is present, and to that of `Y` in the other case.

- **clock sampling:**

  `$when(X,Y)` is a flow of the same type as the flow `X`, present if and only if the flow `X` and the flow `Y` (`Y` must have the type `$pure`) are present; its value is then equal to the value of `X`.

The second category of functions offered by the GC predefinitions are the functions operating on pure flows (clocks), defining the control part of a program:

- **boolean sampling:**

  `$tt(X)` is the flow of type `$pure` present if and only if the boolean flow `X` is present and true.

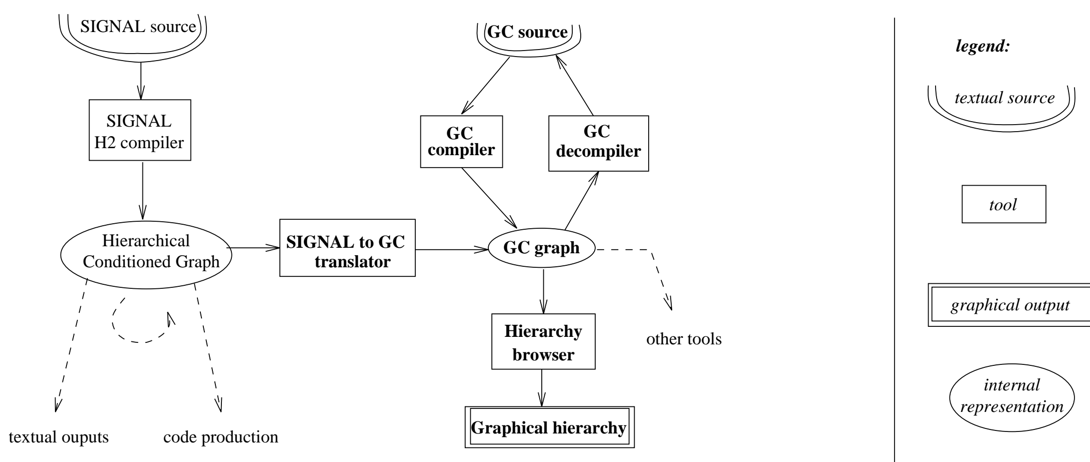

Figure 3: GC tools at IRISA

- **clock union:**

  `$clkadd(X,Y)` is the flow of type `$pure` present if and only if the flow `X` of type `$pure` or the flow `Y` of type `$pure` is present.

- **clock difference:**

  `$clkdiff(X,Y)` is the flow of type `$pure` present if and only if the flow `X` of type `$pure` is present and the flow `Y` of type `$pure` is not present.

- **clock intersection:**

  `$clkmult(X,Y)` is the flow of type `$pure` present if and only if the flow `X` of type `$pure` and the flow `Y` of type `$pure` are present.

- **clock value:**

  `$present(X)` is the boolean flow on the clock `X`, the value of which is always true.

- **special clock operations:**

  - `$clkeq(X,Y)` is the boolean flow present if and only if the flow `X` of type `$pure` or the flow `Y` of type `$pure` is present; its value is true if and only if the flows `X` and `Y` are both present.
  - `$clkimplies(X,Y)` is the boolean flow present if and only if the flow `X` of type `$pure` or the flow `Y` of type `$pure` is present; its value is true if and only if the flow `X` is present more often than the flow `Y`.
  - `$clkmutex(X,Y)` is the boolean flow present if and only if the flow `X` of type `$pure` or the flow `Y` of type `$pure` is present; its value is true if and only if at most one of the flows `X` or `Y` are present.

- **clock of a flow:**

  `$clock(X)` is the flow of type `$pure` present if and only if the flow `X` is present.

3.6 Discussion

Data-flow graph formats are widely used, either for optimization purposes of classical languages [1], or as intermediate formats of data-flow languages [17]. However, to our sense, GC is not just another data-flow graph format: it relies on the sound semantical model of synchronous languages [6, 18] (in particular, this appears in the format through the explicit notion of *clock*). Significant consequences of the synchronous semantics are the following:

- GC programs contain all relevant information to perform formal verification;
- in particular, GC programs contain all relevant information concerning the control, data dependencies, and the scheduling of computations and communications, thus facilitating provably correct and optimized code generation;
- GC programs are hierarchically structured as modules, called nodes, each module has an interface and a body, the composition of modules is associative and commutative, so the architecture of a GC program can easily be modified while preserving semantic equivalence;
- a GC interface carries the necessary information to know how the module should communicate with its environment, and how the internal scheduling of its body is reflected onto its interface, and thus provides an adequate notion of abstraction;
- thus GC provides a formal description of all relevant information to perform provably correct (possibly asynchronous) distributed code generation and separate compilation.

Another crucial use of flowgraph representations is in architectural synthesis [11]. Such representations constitute intermediate forms, through different transformations, between descriptions in Hardware Description Languages and synthesized target architectures. Two main classes of flowgraphs are distinguished:

- the graphs with disjoint representation of control and data flows (or CDFG);

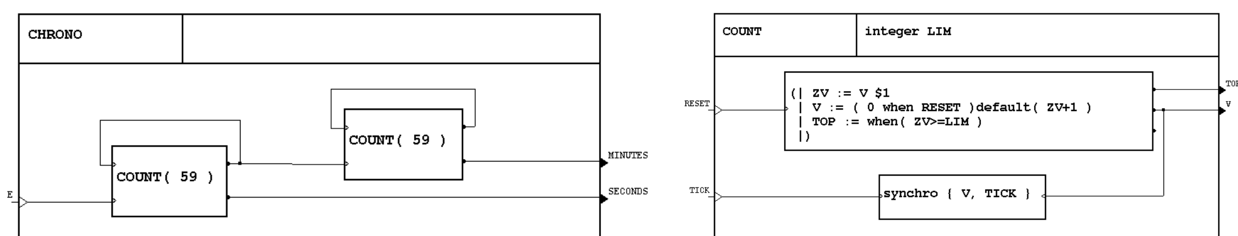

Figure 4: The program CHRONO

- the data-flow graphs with hybrid representation of control and data transfers (the format ASCIS [19] for instance).

Thanks to its ability to handle both data and control dependencies, the format GC can join this second category. For example, GC has already been chosen as the common format of the ASAR[3] project [2, 3], the purpose of which is to build a multi-formalism framework oriented towards architectural synthesis. In this project, GC will be the common denominator of the different formalisms (including VHDL, the synchronous languages Lustre and Signal, the Alpha language for the design of regular architectures) and high level synthesis tools available in the framework. Some extensions to the current version of GC are necessary for that purpose, to handle arrays for example. Moreover, in the hardware domain, other significant experiences are conducted to design architectures using both synchronous languages and VHDL [5].

4 GC as a link to Signal

In this section, we describe the implementation of the GC format in the IRISA/INRIA software environment of the synchronous language Signal.

The first achievement was the design of a good internal representation, and its associated decompiler. We were then able to write a translator from Signal to GC, and a GC compiler. Another tool makes GC readable by browsing programs and permits quick and easy views of them. The figure 3 describes these tools.

4.1 The internal representation

A first phase in the development of the formats consisted in the improvement of their definition. So the internal representation had to support modifications easily. Moreover, since the description capabilities of GC are large, this internal representation can be quite complex.

We have chosen **O.O.P.**[4] and **C++** to help us to encapsulate the possibly evolving concepts of the formats; as a matter of fact, this implementation is quite natural, because of the structure of GC. Some GC objects share properties, which fits well with inheritance, while some others (tables for example) are generically defined, thus claim template definitions.

The package (in the sense of modularity) representing GC is composed of some forty modules and about one hundred classes.

[3] ASAR stands for Atelier d'accueil générique pour la Synthèse ARchitecturale.

[4] Object Oriented Programming.

As GC (as well as the other formats, IC and OC) 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 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. Appendices A and B respectively show such representations.

We have also implemented a graphical browser, with which the figure 5 showing a GC code in this paper has been produced.

4.2 The GC browser

Useful for debugging and producing documents, the browser [4] becomes rapidly quite indispensable to GC users.

It is based on a template kernel that allows programmers to build applications showing hierarchies. Two applications are in use at the moment; they provide facilities for browsing GC programs and Conditioned Hierarchical Graphs (which will be called CHGs thereafter) produced by the Signal compiler (see 4.3).

The principle of the browser is the creation of data structures pointing to the graph to be shown. The user can, by this way, travel through the graph, inspecting every detail (the graph can be displayed down to syntactic level).

The base classes are generic so that nodes can be configured to be attached to local menus, responding to actions on the graph.

In the future, new features will be added to the existing ones: multi-windowing, editing (copy/cut/paste) operations, PostScript output, etc.

4.3 A brief introduction to Signal

Signal [14] is a synchronous real-time language, data-flow oriented and built around a minimal kernel.

It manipulates signals, which are unbounded series of typed values, with an implicit associated clock denoting the instants when values are present. For instance, a signal `X` denotes the sequence $(\mathtt{x}_t)_{t\in\mathbb{N}}$ of data indexed by time-index t. Signals of a special kind called `event` are characterized only by their clock i.e., their presence (they are given the boolean value `true` at each occurrence). Given a signal `X`, its clock is obtained by the expression `event X`, resulting in the event that is present simultaneously with `X`. Different signals can have different clocks: one signal can be absent relatively to the clock of another signal.

The constructs of the language are used to specify the behavior of systems in an equational style: each equation states a relation between signals i.e., between their values

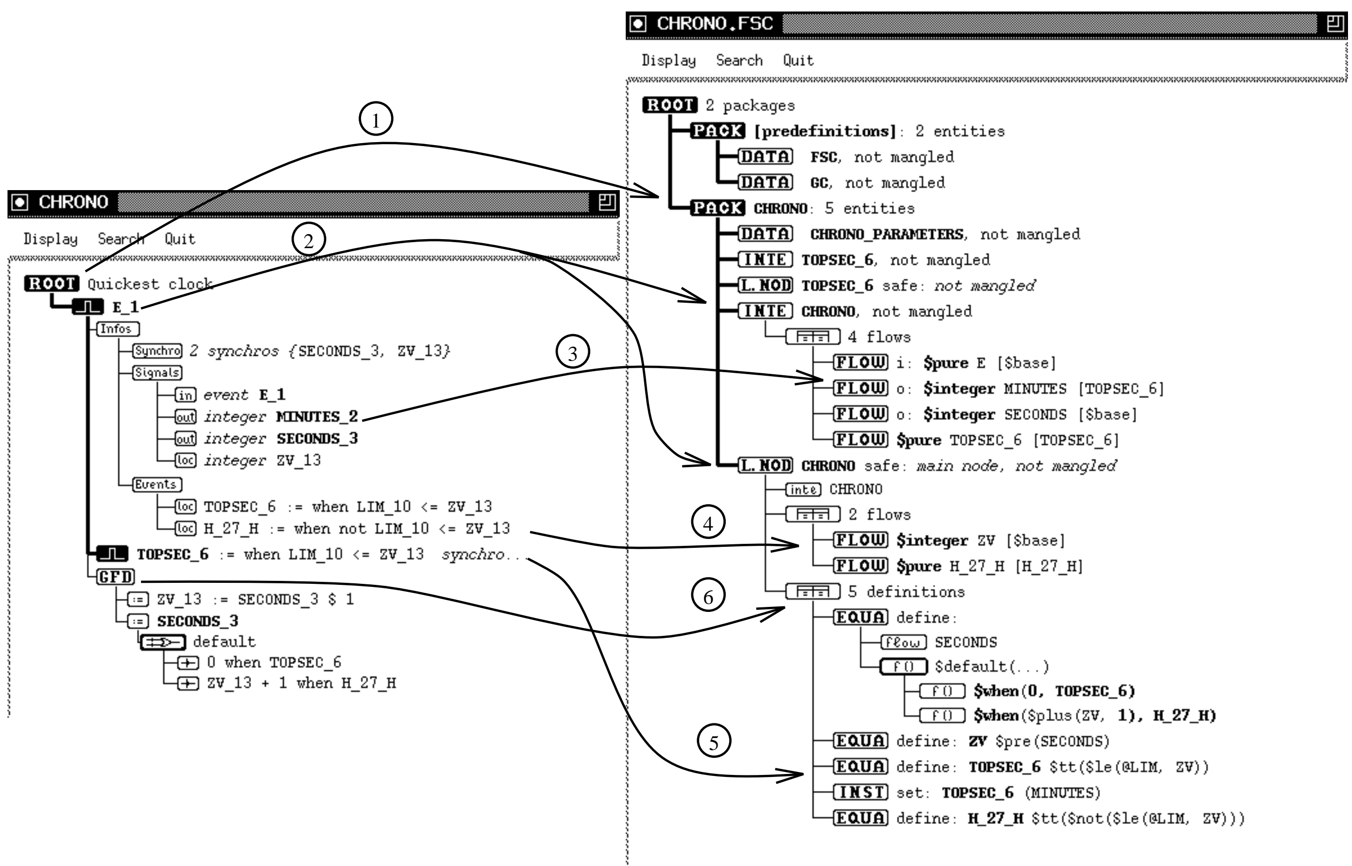

Figure 5: SIGNAL to GC translation

and between their clocks. Systems of equations on signals are built using the composition construct. When equations have a deterministic solution depending on inputs, the resulting program is reactive (i.e., input-driven); in other cases, correct programs can be demand-driven or control-driven.

4.3.1 The kernel of SIGNAL

The kernel comprises the following features:

`Y := f{X1,...,Xn}`	functions extended to sequences: $\forall k, \mathtt{Y}_k = \mathtt{f}(\mathtt{X1}_k, ..., \mathtt{Xn}_k)$
`Y := X $1`	delay (shift register) $\forall k, \mathtt{Y}_k = \mathtt{X}_{k-1}$
`Y := X when B`	boolean dependent downsampling `Y` = `X` when `B` is present and true
`Y := U default V`	merge with priority `Y` = `U` when `U` is present, otherwise `Y = V`
`P | Q`	composition of processes
`P / X`	assigning local scope to signal `X` in process `P`

Derived processes have been defined from the primitive operators, providing programming comfort. E.g., `synchro{X,Y}` which constrains signals `X` and `Y` to be synchronous, i.e. to have the same clock; `when C` giving the clock of the occurrences of `C` with the value `true`. A structuring mechanism is proposed in the form of process models, defined by a name, typed parameters, input and output signals, a body, and local declarations.

4.3.2 An example

The process `CHRONO` represented in the Figure 4 (obtained with the graphical interface for SIGNAL) is a simple chronometer that counts the `SECONDS`, at the clock delivered by the `event`-type input signal `E`, and the `MINUTES` (the outputs `SECONDS` and `MINUTES` are `integer`-type signals). It uses two instances of the process model `COUNT`, the body of which is briefly detailed now:

- `synchro { V, TICK }`; the occurrences of the `integer` signal `V` and that of the `event` signal `TICK` are synchronous: `V` counts the occurrences of `TICK`;
- `ZV := V $1`; the `integer` signal `ZV` carries, at each one of its occurrences, the previous value of `V`; both signals are implicitly synchronous;
- `V := (0 when RESET) default (ZV+1)`; the signal `V` is reset at each occurrence of the `event` signal `RESET`; otherwise, it takes its previous value (carried by `ZV`) incremented by 1;
- `TOP := when (ZV>=LIM)`; the `event` signal `TOP` and the occurrences of `ZV` with the value of the parameter `LIM` are synchronous: a `TOP` is produced every `(LIM+1)` `TICK`.

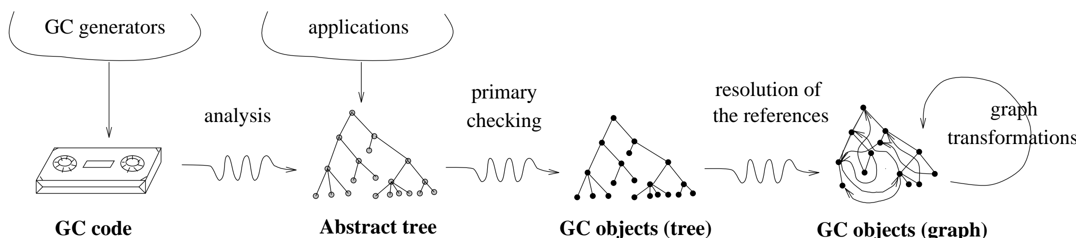

Figure 6: The compilation process

4.3.3 Compilation

The compiler performs the analysis of the consistency of the system of equations, and determines whether the synchronization constraints among the signals are verified or not. For that purpose, it synthesizes the hierarchy of clocks of the program. This *clock calculus* [8] relies on the partial order of clocks which corresponds to the inclusion of instants (a clock being more frequent than another one). To understand this calculus, consider the following situation: `H` is the clock of some, say, `integer` valued signal `ZV`, and `K` is the set of instants where signal `ZV` exceeds some threshold: `K := when (ZV>=LIM)`. Then 1/ each instant of `K` is an instant of `H`, and 2/ to compute the status of `K` we must first know the status of `H`. Thus we have both that `K` is less frequent than `H` and that there is a causality constraint from `H` to `K`. This is denoted by `H` $\rightarrow$ `K`. Successive such downsamplings organize clocks into several *trees,* the collection of these trees builds the *forest* of the clocks of the considered program. If a single tree is obtained, the synchronization of the considered program easily results and execution is straightforward.

Finally, since every computation is performed at some particular clock, computations are attached to their corresponding clock within this forest, hence building a CHG. The resulting structure is the intermediate level form of the SIGNAL program and is the basis for all formal manipulations (including optimizations) and code generations [15].

The CHG of the process CHRONO is shown on the left side of figure 5, using the generic browser. For reasons of legibility, on this figure, the clock-node `TOPSEC_6`, producing the signal `MINUTES_2` has been iconified, as well as its corresponding GC-node (right hand side of the figure). The overall GC code can be read in appendix A.

4.4 From SIGNAL to GC

The generation of GC code from a SIGNAL program applies on the CHG created by the compilation. It is a recursive traversal of the clock hierarchy, as described next.

Each step of the translation is shown by an arrow on figure 5.

A package corresponds to a program (arrow ①).

A clock is a member of the main hierarchy if it has associated computations or sub-clocks. For each such clock, an interface and a local node are created in the package (arrow ②). This node is further instantiated in the table of definitions of its upper clock in the clock hierarchy.

The signals and clocks are transcribed in the tables of flows of the interface (interface flows and clocks of interface flows, arrow ③) and of the node (other local flows, arrow ④).

The sub-clocks (if there are some) are defined in the table of definitions of the node (arrow ⑥).

The synchronous data-flow graph associated with the current clock is represented by equations defining the flows, and node activations (or calls of procedures) for the SIGNAL external calls (arrow ⑤). Dependencies can be implicit or explicit (in the example, they are let implicit).

Finally, the other definitions produced at the current clock are instantiations of the nodes corresponding to the sub-clocks, which are created recursively. The external calls of the CHG are, depending on the will of the user, transcribed into external GC nodes in the package, or procedures in a special data block (created in the same package).

4.5 GC compilation

The global process of the compilation of GC is depicted in figure 6.

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 analyser.

The main problem of the compilation is what we call 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: importations are attached to their corresponding entity.

resolution of the types: windowed types are attached to the corresponding type.

resolution of the type-references: typed objects (constants, flows, functions, arguments of functions, parameters of procedures...) are attached to their type.

other resolutions: they 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. Then, a complete internal representation of the data-flow graph can be produced. The SPEAK[5] project, consisting in the development of a public instanciation of the formats, will allow the implementation of gates between those graphs and particular representations; by this way, different tools will apply to GC.

[5] SPEAK stands for Synchronous Programming Environment Access Kernel.

Finally, the data-flow graph produced by the compilation can be used for transformations, code production, optimizations, hardware synthesis, etc.

Let us consider optimization aspects for instance. Traditionally, optimizers create a dependence graph between the variables of the program to be optimized. However, the dependencies are not conditioned by *clocks:* for example, in the program

```
if (X < 0) then C := A else C := B
```

the dependencies from `A` to `C` and from `B` to `C` are considered in a same way, without taking into account the condition `(X < 0)` which, according to its value, inhibits one of them. On the contrary, in the GC-like equation

```
define:  C $default(A,B)
```

the dependencies from `A` to `C` and from `B` to `C` are considered as effective ones at exclusive and perfectly defined clocks. Thus, finer optimizations can be made: for example, if `A` and `B` have exclusive *computation* and *availability clocks*[6], `A`, `B` and `C` can share the same memory space.

So, very deep optimizations are possible on GC programs representing CHGs. They concern as well the reduction of the computation frequency of the flows, as the reduction of the memory space and the removal of assignments. For instance, if the graph is rewritten with a sub-sampling of all the flows by their *utilization clock*[7], the resulting program is equivalent to the initial one with respect to its external behavior, but the availability clocks of internal flows, and by the way, the computation clocks, have been reduced.

The browser, the translator and the compiler presented here belong to the first generation of tools applying to the common formats. One of the objectives was to improve GC, as defined in [18]; 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.

This may allow GC to be chosen in the future as a new intermediate code for the different phases of the SIGNAL compiler.

5 Conclusion

We have presented the data-flow graph format GC and its current implementation in the INRIA software environment of the synchronous language SIGNAL. Recall that GC is one of the three vertices of the triangle composed by the formats of synchronous programming. A lot of tools will be progressively available around these formats: translators, code generators, formal verification tools, etc. IC, OC and GC programs can be exchanged by the participants of the SYNCHRON project via an ftp account; this speeds up the finalization of the formats and maintains a coherence between all the teams working around GC.

As a data-flow graph format, GC is also a natural candidate to be an intermediate representation for VHDL descriptions and link to hardware synthesis tools.

The formats are being implemented in both industrial and academic environments of synchronous languages. Moreover, a public instance of the formats will be developed at INRIA. In parallel, the standardization process will be pursued.

[6] The *availability clock* is the clock of the flow: it defines the instants at which the clock is present; the *computation clock* defines the instants at which the value of the flow is computed (if a flow is partially defined using its previous value—operator $pre—, it is more frequently present than it is computed).

[7] The *utilization clock* of a flow defines the instants at which the value of the flow is necessary.

References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers–Principles, Techniques and Tools.* Wesley–Addison, 1986.

[2] P. ASAR. Framework and Multi-Formalism: the ASAR Project. In F.J. Ramming and F.R. Wagner, editors, *Proc. of the 4th International IFIP WG 10.5 Working Conference on Electronic Design Automation Frameworks, Gramado, Brazil*, to be published by Chapman & Hall, 1995, November 1994.

[3] P. ASAR. Towards a multi-formalism framework for architectural synthesis: the ASAR project. In *International Workshop on Hardware-Software Codesign, Codes/CASHE'94*, September 1994.

[4] Pascal Aubry and Sylvain Machard. Représentation graphique d'arbres sous X11R5 : Implémentation générique orientée objet, Applications. IRISA, Rennes. 1994. To appear.

[5] Mohammed Belhadj. *Conception d'architectures en utilisant* SIGNAL *et VHDL.* PhD thesis, Université de Rennes 1, December 1994.

[6] Albert Benveniste, Paul Caspi, Paul Le Guernic, and Nicolas Halbwachs. Data-Flow Synchronous Languages. In J.W. de Bakker, W.P. de Roever, and G. Rozenberg, editors, *Lecture Notes in Computer Science 803, Proc. of the REX School/Symposium, Noordwijkerhout, Netherlands*, pages 1–45, Springer–Verlag, June 1993.

[7] Gérard Berry and Georges Gonthier. The ESTEREL synchronous programming language: design, semantics, implementation. *Science of computer programming*, 19(2):87–152, 1992.

[8] Loïc Besnard. *Compilation de* SIGNAL *: horloges, dépendances et environnement.* PhD thesis, Université de Rennes 1, September 1992.

[9] Patricia Bournai, Bruno Chéron, Thierry Gautier, Bernard Houssais, and Paul Le Guernic. *SIGNAL manual.* Research report 1969, INRIA Rennes, September 1993.

[10] Bruno Chéron. *Transformations syntaxiques de programmes* SIGNAL. PhD thesis, Universit de Rennes 1, September 1991.

[11] Daniel D. Gajski, Nikil D. Dutt, Allen C-H. Wu, and Steve Y-L. Lin. *High-Level Synthesis – Introduction to Chip and System Design.* Kluwer Academic Publishers, 1992.

[12] Nicolas Halbwachs, Paul Caspi, Paul Raymond, and Daniel Pilaud. The synchronous data flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1321, September 1991.

[13] David Harel. STATECHARTS: a visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[14] Paul Le Guernic, Thierry Gautier, Michel Le Borgne, and Claude Le Maire. Programming real-time applications with SIGNAL. *Proceedings of the IEEE*, 79(9):1321–1336, septembre 1991.

[15] Olivier Maffeïs and Paul Le Guernic. From SIGNAL to fine-grain parallel implementations. In *Int. Conference on Parallel Architectures and Compilation Techniques, IFIP A-50, North-Holland*, pages 237–246, August 1994.

[16] Frédéric Mignard. *Compilation du langage Esterel en systèmes d'équations booléennes.* PhD thesis, École des Mines de Paris, October 1994.

[17] S.K. Skedzielewski and M.L. Welcome. Data Flow Graph Optimization in IF1. In G. Goos and J. Hartmanis, editors, *Lecture Notes in Computer Science 201, Proc. of Functional Programming Languages and Computer Architectures, Nancy, France*, pages 17–34, Springer–Verlag, September 1985.

[18] Projet SYNCHRON. *Les formats communs des langages synchrones.* Technical report 157, INRIA, June 1993. version 0.

[19] J.T.J von Eijndhoven, G.G. de Jong, and L. Stok. *The ASCIS Data-Flow Graph: Semantics and Textual Format.* EUT Report 91-E-251, Eindhoven University of Technology, June 1991.

A An example of GC code

```
package: 5 CHRONO
  0: data: dc:0 CHRONO_PARAMETERS
     % nomangling: %
    constants: 2
      0: LIM_1 $2 value: #59;
      1: LIM_2 $2 value: #59;
    end: -- constants:
  enddata: -- CHRONO_PARAMETERS
  ----------
  1: interface: gc:0 DEPSEC_10
     % nomangling: %
    flows: 1
      0: o: MINUTES $2 $g0;
    end: -- flows:
  endinterface: -- DEPSEC_10
  ----------
  2: node: gc:0 DEPSEC_10 safe:
     % nomangling: %
    import: 2
      0:  1;
      1:  0;
    end: -- import:
    flows: 3
      0:    ZCPT $2 value: $19(#1) $g0;
      1:    DEPMIN_11 $0 1;
      2:    H_35_H $0 2;
    end: -- flows:
    definitions: 4
      0: define: 0.0 $g9($g10(#0, 1),
                         $g10($14(0, #1),
                         2));
      1: define: 0 $g11(0.0);
      2: define: 1 $g0($11(@1.0, 0));
      3: define: 2 $g0($6($11(@1.0, 0)));
    end: -- definitions:
  endnode: -- node: DEPSEC_10
  ----------
  3: interface: gc:0 CHRONO
     % nomangling: %
    flows: 4
      0: i: E $0 $g0;
      1: o: MINUTES $2 3;
      2: o: SECONDES $2 $g0;
      3:    DEPSEC_10 $0 3;
    end: -- flows:
  endinterface: -- CHRONO
  ----------
  4: node: gc:0 CHRONO safe: % main: %
     % nomangling: %
    import: 3
      0:  3;
      1:  0;
      2:  2;
    end: -- import:
    flows: 2
      0:    ZCPT $2 value: $19(#1) $g0;
      1:    H_39_H $0 1;
    end: -- flows:
    definitions: 5
      0: define: 0.2 $g9($g10(#0, 0.3),
                     $g10($14(0, #1),
                         1));
      1: define: 0 $g11(0.2);
      2: define: 0.3 $g0($11(@1.1, 0));
      3: set: 0.3 2(flows: 0.1);
      4: define: 1 $g0($6($11(@1.1, 0)));
    end: -- definitions:
  endnode: -- node: CHRONO
endpackage: -- CHRONO
```

B The same example, made readable

```
package: 5 CHRONO
  0: data: dc:0 CHRONO_PARAMETERS
     % nomangling: %
    constants: 2
      0: LIM_1 $integer value: #59;
      1: LIM_2 $integer value: #59;
    end: -- constants:
  enddata: -- CHRONO_PARAMETERS
  ----------
  1: interface: gc:0 DEPSEC_10
     % nomangling: %
    flows: 1
      0: o: MINUTES $integer $base;
    end: -- flows:
  endinterface: -- DEPSEC_10
  ----------
  2: node: gc:0 DEPSEC_10 safe:
     % nomangling: %
    import: 2
      0:  DEPSEC_10;
      1:  CHRONO_PARAMETERS;
    end: -- import:
    flows: 3
      0:    ZCPT $integer value: $uminus(#1) $base;
      1:    DEPMIN_11 $pure DEPMIN_11;
      2:    H_35_H $pure H_35_H;
    end: -- flows:
    definitions: 4
      0: define: MINUTES $default($when(#0, DEPMIN_11),
                                  $when($plus(ZCPT, #1),
                                        H_35_H));
      1: define: ZCPT $pre(MINUTES);
      2: define: DEPMIN_11 $tt($le(@LIM_1, ZCPT));
      3: define: H_35_H $tt($not($le(@LIM_1, ZCPT)));
    end: -- definitions:
  endnode: -- node: DEPSEC_10
  ----------
  3: interface: gc:0 CHRONO
     % nomangling: %
    flows: 4
      0: i: E $pure $base;
      1: o: MINUTES $integer DEPSEC_10;
      2: o: SECONDES $integer $base;
      3:    DEPSEC_10 $pure DEPSEC_10;
    end: -- flows:
  endinterface: -- CHRONO
  ----------
  4: node: gc:0 CHRONO safe: % main: %
     % nomangling: %
    import: 3
      0:  CHRONO;
      1:  CHRONO_PARAMETERS;
      2:  DEPSEC_10;
    end: -- import:
    flows: 2
      0:    ZCPT $integer value: $uminus(#1) $base;
      1:    H_39_H $pure H_39_H;
    end: -- flows:
    definitions: 5
      0: define: SECONDES $default($when(#0, DEPSEC_10),
                                   $when($plus(ZCPT, #1),
                                         H_39_H));
      1: define: ZCPT $pre(SECONDES);
      2: define: DEPSEC_10 $tt($le(@LIM_2, ZCPT));
      3: set: DEPSEC_10 DEPSEC_10(flows: MINUTES);
      4: define: H_39_H $tt($not($le(@LIM_2, ZCPT)));
    end: -- definitions:
  endnode: -- node: CHRONO
endpackage: -- CHRONO
```