Code generation in the SACRES project *

Thierry Gautier and Paul Le Guernic **

IRISA / INRIA, Campus de Beaulieu – 35042 Rennes Cedex – FRANCE
e-mail: {Thierry.Gautier|Paul.LeGuernic}@irisa.fr
http://www.irisa.fr/EXTERNE/projet/ep-atr

Abstract. The SACRES project is dealing with the development of new design methodologies and associated tools for safety critical embedded systems. Emphasis is put on formal techniques for modular verification of the specifications, distributed code generation, and generated code validation against specifications. This is allowed by using a single formal model which is that of the DC_+ format, which provides a common semantic framework for all tools as well as end user specification formalisms. Modular and distributed code generation is the main subject of this paper. Distributed code generation aims at reducing the dependency of the design with respect to the target architecture. Modularity helps reuse of existing designs, and makes it possible to address much larger systems.

1 Introduction

The overall objective of the SACRES project is to provide designers of embedded control systems, in particular safety critical systems, with an enhanced design methodology supported by a toolset significantly reducing the risk for design errors and shortening overall design times. This is achieved through the use of the maximum degree of automation, especially in respect of code generation and verification [6]. SACRES architecture is depicted in figure 1. The SACRES toolset combines the following main groups of tools:

- The command-level interface permits the tools to be invoked. Some of the tool launch facilities are also available through the menus from the specification tools.
- The specification front-end tools are mostly self-contained graphical tools for building models. As software requirements for embedded systems typically involve a mixture of state oriented and dataflow oriented descriptions, SACRES offers the possibility of *mixed formalism* design, by combining expressive power of Statecharts [12] from the STATEMATE tool, and SIGNAL

* This work is supported by the Esprit project R&D SACRES (EP 20897). SACRES members are: Siemens (Lead partner), i-Logix, TNI (Techniques Nouvelles d'Informatique), OFFIS, INRIA, the Weizmann Institute of Science, British Aerospace, SNECMA.

** The following people have also participated to this work: Albert Benveniste, Loïc Besnard, Patricia Bournai, Sylvain Machard, Éric Rutten. They are gratefully acknowledged.

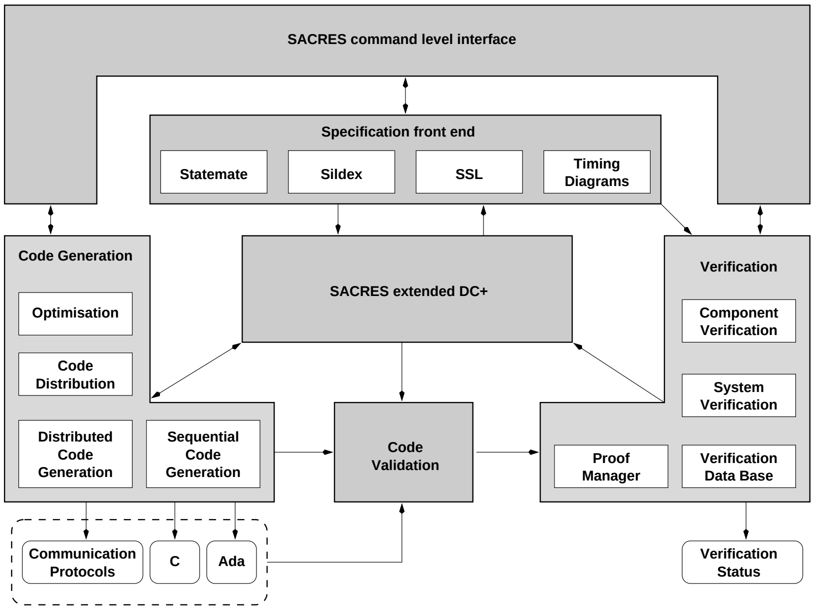

Fig. 1. SACRES *architecture*

[15] based dataflow diagrams from the Sildex tool. SSL is a textual language, used for assembling models with components from different specification tools. It is mainly available from the graphical editors of the specification tools.
- The DC_+ representation is a global model format [19]. These files are not expected to be read by the user, just to be passed between tools in the toolset.
- Code generation tools include an interface for defining target architecture and code distribution as well as the code generation engine itself.
- Code validation provides a formal correctness check for the code generation translation.
- The verification tools allow automatic checking of the models that are produced from the specification tools, and manage the results of the proof.

This paper is dedicated to a presentation of the methodology adopted in SACRES for *modular and distributed code generation.* The main add-on of the methodology is to allow *automatic generation* of efficient distributed code from the specification, entirely replacing the manual coding phase still employed in current industrial design flows. A requirement for that is, at the front-end level, the use of specification tools that are based on a formal semantical model. This is the case of both Statecharts and Signal formalisms. Further, since *mixed designs* are favoured in SACRES, using together both state-based and dataflow specification styles, a common representation was in some way mandatory. In

SACRES, this common representation is the $\text{D}\textsc{c}_+$ format. This format implements the paradigm of *synchronous programming* [8, 11] in its full generality. Although very close in its syntax of the *synchronized dataflow model* advocated by the SIGNAL language, it constitutes a model for the semantic integration of SIGNAL [10] and Statecharts specifications [13]. In particular, translations from Statecharts to $\text{D}\textsc{c}_+$ have been defined in SACRES [4, 5].

The semantical basis of the $\text{D}\textsc{c}_+$ format is that of *Symbolic Transition Systems* presented in the next section. This model includes in particular scheduling specifications, which are used to represent causality relations, schedulings, and communications. The compilation of programs expressed in $\text{D}\textsc{c}_+$ results in a hybrid structure which allows to represent both control and scheduling of the programs, and to study important properties such as *endochrony.* Part of this structure is a directed graph with boolean conditions on nodes and dependences. From section 3, we will use a more macroscopic view of this directed graph, with the help of diagrams, to present the methodology of code distribution. In section 3, we give an overview of the approach and in section 4, we present the main steps of the method.

2 The DC+ model

SACRES relies on a strong formal modelling basis, which is that of Symbolic Transition Systems with Scheduling specifications [17, 18]. This abstract model is used as the model of the $\text{D}\textsc{c}_+$ concrete format.

2.1 Symbolic Transition Systems with Scheduling specifications

Symbolic Transition Systems We assume a vocabulary $\mathcal{V}$ which is a set of typed variables. All types are implicitly extended with a special element $\perp$ to be interpreted as "absent". Some of the types we consider are the type of *pure flows* with domain $\{\text{T}\}$, and *booleans* with domain $\{\text{T}, \text{F}\}$ (recall both types are extended with the distinguished element $\perp$).

We define a *state* s to be a type-consistent interpretation of $\mathcal{V}$, assigning to the set of all variables, a value for it over its domain. For a subset of variables $V \subseteq \mathcal{V}$, we define a V-state to be a type-consistent interpretation of V. Thus a V-state assigns to the set V a value $s[V]$ for it over its domain; also, for $v \in V$ a variable, we denote by $s[v]$ its interpretation by state s.

We define a *Symbolic Transition System* (STS) to be a system

$$\Phi = \langle V, \Theta, \rho \rangle$$

consisting of the following components:

- V is a finite set of typed *variables,*
- $\Theta(V)$ is an assertion characterizing *initial states.*

- $\rho = \rho(V^-, V)$ is the *transition relation* relating past and current states s^- and s, by referring to both past and current versions of variables V^- and V. For example the assertion $x = x^- + 1$ states that the value of x in s is greater by 1 than its value in s^-. If $\rho(s^-[V], s[V]) = \mathrm{T}$, we say that state s^- is a *ρ-predecessor* of state s.

A *run* $\sigma : s_0, s_1, s_2, \ldots$ is a sequence of states such that

$$s_0 \models \Theta \quad \bigwedge \quad \forall i > 0 \ , \ (s_{i-1}, s_i) \models \rho$$

The *composition* of two STS $\Phi = \Phi_1 \bigwedge \Phi_2$ is defined as follows:

$$V = V_1 \cup V_2 \ , \Theta = \Theta_1 \wedge \Theta_2 \ , \rho = \rho_1 \wedge \rho_2 \ ,$$

the composition is thus the pairwise conjunction of initial and transition relations. Note that, in STS composition, interaction occurs through common variables only. Hence variables that are declared private to an STS will not directly contribute to any external interaction.

Notations for STS*:*

- $c, v, w, \ldots$ denote STS *variables,* these are the declared variables of the STS; useful additional variables are the following:
- for v a variable, $h_v \in \{\mathrm{T}, \perp\}$ denotes its *clock:*

$$[h_v \neq \perp] \Leftrightarrow [v \neq \perp]$$

- for v a variable, ξ_v denotes its associated *state-variable,* defined by:

$$\textbf{if } h_v \textbf{ then } \xi_v = v \ \textbf{ else } \xi_v = \xi_v^-$$

  Values can be given to $s_0[\xi_v^-]$ as part of the initial condition. Then, ξ_v is always present after the 1st occurrence of v. By convention, ξ_v is private to the STS in which it is used. Thus state-variables play no role in STS-composition. Also, note that $\xi_{\xi_v} = \xi_v$, thus "state-state-variables" need not to be considered.

Transition relations for STS are naturally specified using conjunction of predicates.

Modularity. As modularity is wanted, it is desirable that the pace of an STS is local to it rather than global. Since any STS is subject to further composition in some yet unknown environment, this makes the requirement of having a global pace quite inconvenient. This is why *we prohibit the use of clocks that are always present.* This has several consequences. First, it is not possible to consider the "complement of a clock" or the "negation of a clock": this would require referring to the always present clock. Thus, clocks will always be variables, and we shall be able to relate clocks only using $\wedge$ (intersection of instants of presence), $\vee$ (union of instants of presence), and $\setminus$ (set difference of instants of presence).

Scheduling specifications Modular code distribution, and in the same way, separate compilation, clearly require to be able to reason about causality, schedulings, and communications. This is why we enrich the STS model as follows.

Preorders to model causality relations, schedulings, and communications. We consider again a set V of variables. A *preorder* on the set V is a relation (generically denoted by $\preceq$) which is reflexive ($x \preceq x$) and transitive ($x \preceq y$ and $y \preceq z$ imply $x \preceq z$). Preorders are naturally specified via *(possibly cyclic)* directed graphs:

$$x \rightarrow y \quad \text{for} \quad x, y \in V \ .$$

The *conjunction* of two preorders is the minimal preorder which is an extension of the two considered conjuncts.

A *labelled preorder* on V is a preorder, together with a domain for each $v \in V$. Call $\mathbf{dom}_V$ the domain of the set V of variables. Denote by Λ_V the set of all labelled preorders on V. A *state* s is now a type consistent interpretation of the labelled preorder, i.e., a preorder on V together with a value $s[V]$ for the set of all variables belonging to V. Denote by $\mathbf{dom}_{\Lambda_V}$ the domain in which states take their value.

STS *with scheduling specifications.* Now we consider an STS $\Phi = \langle V, \Theta, \rho \rangle$ as before, but with the following modification for the transition relation $\rho = \rho(V^-, V)$:

$$\rho \subset \mathbf{dom}_V \times \mathbf{dom}_{\Lambda_V} \ , \tag{1}$$

i.e., transition relations relate the value for the t-uple of previous variables to the current state. As before, runs are sequences $s_0, s_1, s_2, \ldots$ that are consistent with transition relation (1).

STS involving such type of preorder relation will be called STS *with scheduling specifications.* STS with scheduling specifications are just like any other STS, hence they inherit their properties, in particular *they can be composed.*

Notations for scheduling specifications: for b a variable of type $bool \cup \{\bot\}$, and u, v variables of any type, the following generic conjunct will be used:

$$\mathbf{if}\ b\ \mathbf{then}\ u \longrightarrow v \quad \text{, resp.}\ \mathbf{if}\ b\ \mathbf{else}\ u \longrightarrow v$$

also written:

$$u \xrightarrow{\ b\ } v \text{ resp. } u \xrightarrow{\ \overline{b}\ } v$$

In [9], it is shown that scheduling specifications have the following properties:

$$x \xrightarrow{\ b\ } y \bigwedge y \xrightarrow{\ c\ } z \Rightarrow x \xrightarrow{\ b \wedge c\ } z \tag{2}$$

$$x \xrightarrow{\ b\ } y \bigwedge x \xrightarrow{\ c\ } y \Rightarrow x \xrightarrow{\ b \vee c\ } y \tag{3}$$

Properties (2,3) can be used to compute input/output abstractions of scheduling specifications. This is illustrated in figure 2. In this figure, the diagram on the left depicts a scheduling specification involving local variables. These are hidden in the diagram on the right, using rules (2,3).

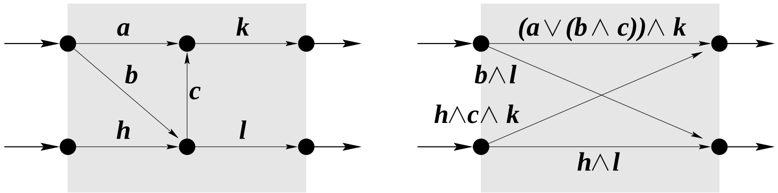

Fig. 2. *Input/output abstractions of scheduling specifications*

Syntax The following restricted set of generic basic conjuncts is sufficient to encode all known synchronous languages:

$$\begin{array}{c} \textbf{if } b \textbf{ then } w = u \textbf{ else } w = v \\ \\ u \xrightarrow{b} w \\ \\ \left.\begin{array}{rcl} w & = & f(u_1, \ldots, u_k) \\ h_w \equiv h_{u_1} & \equiv & \ldots \equiv h_{u_k} \end{array}\right\} \end{array} \tag{4}$$

In addition to the set (4) of primitives, state-variable ξ_v associated with variable v can be used on the right hand side of each of the above primitive statements. The third primitive involves a conjunction of statements that are considered jointly.

Examples

$$\begin{array}{rl} \text{a selector :} & \textbf{if } b \textbf{ then } z = u \textbf{ else } z = v \\ \text{a register :} & \textbf{if } h_z \textbf{ then } v = \xi_z^- \textbf{ else } v = \perp \end{array}$$

$$\text{a synchronization constraint : } (b = \text{T}) \equiv (h_u = \text{T})$$

For the selector, the "**else**" part corresponds to the property "$[b = \text{F}] \vee [b = \perp]$". The more intuitive interpretation of the second statement is "$v_n = z_{n-1}$", where index "n" denotes the instants at which both v and z are present (their clocks are equal). Clearly, this models a register. This statement implies the equality of clocks: $h_z \equiv h_v$. The synchronization constraint means that the clock of u is the set of instants at which boolean variable b is *true.*

Inferring schedulings from causality analysis The schedulings that can be inferred from an STS specification result from 1/ explicit scheduling specifications, and 2/ dataflow dependences that result from causality analysis. The idea supporting causality analysis of an STS specification is quite simple. On the one hand, a transition relation involving only the types "pure" and "boolean" can be solved by unification and thus made executable. On the other hand, a transition relation involving arbitrary types is abstracted as term rewriting, encoded via directed graphs. For instance, relation $y = 2uv^2$ (involving, say, real types) is abstracted as $(u, v) \longrightarrow y$, since y can be substituted by expression $2uv^2$.

Scheduling specifications associated with the primitive statements 4 are given in [9]. For example:

$$\left.\begin{array}{r} w \; = \; f(u_1, \ldots, u_k) \\ h_w \equiv h_{u_1} \equiv \ldots \equiv h_{u_k} \end{array}\right\} \Rightarrow u_i \xrightarrow{h_w} w \tag{5}$$

Given an STS specified as the conjunction of a set of basic statements, for each conjunct we add the corresponding scheduling specification to the considered STS. This yields a new STS **sched**(`P`), for which it is possible to give sufficient conditions so that `P` is *executable:* roughly, **sched**(`P`) is provably circuitfree at each instant, and **sched**(`P`) has provably no multiple definition of variables at any instant. Then **sched**(`P`) provides (dynamic) scheduling specifications for the run of `P`.

2.2 Endochrony

The STS model is the semantical model of DC_+ programs. We need to give some sketch of the compilation of these programs in order to complete the internal representation structure of programs: one face is a directed graph, the other one is a *clock hierarchy* allowing to represent the control of the program. This structure is the basis for studying *endochrony.*

A DC_+ program describes a reactive system whose behavior along time is an infinite sequence of *instants* which represent reactions, triggered by external or internal events. The main objects manipulated by a DC_+ program are *flows*, which are sequence of values synchronized with a *clock.* A flow is a typed object which holds a value at each instant of its clock. The fact that a flow is currently absent is represented by the bottom symbol $\perp$ (cf. section 2.1). Clocks are pure or boolean flows. A clock has the value `T` if and only if the flow associated with the clock holds a value at the present instant of time. Actually, any expression *exp* in the language has its corresponding clock h_{exp} which indicates whether the value of the expression at the current instant is different from $\perp$.

Clock hierarchy. Directed graphs obtained by causality analysis such as presented above are one of the objects used to represent programs and to calculate on them. The other very important object that has to be obtained is a representation of the *clock hierarchy.*

A first step of the DC_+ compilation is the construction of a system of equations on clocks. The system is solved, the result gives inclusion between clocks. The clocks are then organized in a *hierarchy* (i.e. a collection of trees) where if a clock h_1 is under a clock h_2 (called its father), then h_1 cannot be present if h_2 is not. In the clock hierarchy represented on figure 3, b, c denote boolean variables,

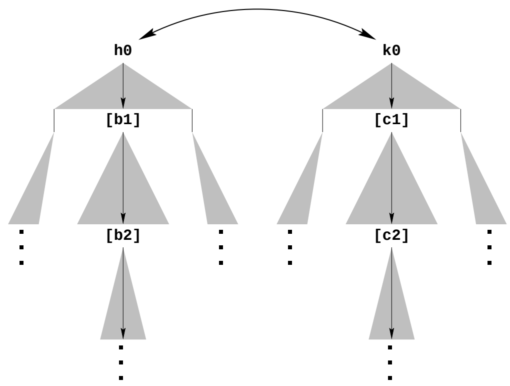

Fig. 3. *Clock hierarchy*

$[b], [c]$ denote corresponding clocks composed of the instants at which $b, c = \mathrm{T}$ holds, respectively. Finally, h, k are also clocks (i.e., variables of type `pure`). The down-arrows $h_0 \rightarrow [b_1]$, $[b_1] \rightarrow [b_2]$, etc., indicate that boolean variable b_1 has a clock equal to h_0 and only needs variables with clock h_0 for its evaluation, and so on. In doing so, a tree is built under each of the clocks $h_0, k_0, \ldots$, this yields the so-called *clock hierarchy* in the form of a "forest", i.e., a collection of trees. Roots of the trees are related by some clock equation, this is depicted as the bidirectional arrow relating $h_0, k_0, \ldots$ Then each flow of the program (and its definition) is attached to its clock in the hierarchy. This structure is detailed in [1] [2], where it is shown to be a canonical form for representing clock equations. The combination of clock equations and scheduling specifications of a program is represented by the combination of the clock hierarchy and of the directed graph.

Endochrony. An important property that will be determined on the clock hierarchy is *endochrony.* An STS is called *endochronous* if its control, i.e., the primary decision in executing a reaction, depends only on 1/ the previous state, and 2/ the values possibly carried by environment flows, but not on the presence/absence status of these flows. If an STS is not endochronous (it is *exochronous*), then the primary decision in executing a reaction consists in deciding upon relative presence/absence of clocks which are source nodes of the associated directed graph. In contrast, for an endochronous STS, only one *activation clock* is a source node of the graph. Hence no test for relative presence/absence of environment flows is needed.

It is shown in [9] that if a program P has a clock hierarchy consisting in a single tree, then it is endochronous.

Boolean $\mathrm{D}\textsc{c}_{+}$. In practice, the calculation of the clock hierarchy of a program is a key tool of the $\mathrm{D}\textsc{c}_{+}$ compiler; it is called the *clock calculus.* After the clock calculus, clocks can be defined as boolean flows in a sub-format of $\mathrm{D}\textsc{c}_{+}$ called $\textsc{b}\mathrm{D}\textsc{c}_{+}$: a boolean b represents the clock $[b]$ composed of the instants at which $b, c = \mathrm{T}$ holds, and is defined itself at the clock which is the father of $[b]$ in the clock hierarchy.

2.3 Issues for modular and distributed code generation

Two major issues need to be considered for modular and distributed code generation:

1. *Relaxing synchrony* is needed if distribution over possibly asynchronous media is desired without paying the price for maintaining the strong synchrony hypothesis via costly protocols.
2. *Designing modules equipped with proper interfaces for subsequent reuse,* and generating a correct scheduling and communication protocol for these modules, is the key to modularity.

It is shown in [9] that a solution to the first issue is to restrict ourselves to endochronous programs. Another aspect that has to be considered is that of maintaining synchronous semantics of composition while using asynchronous communication media. Requirement for such a medium is that: 1/ it should not loose messages, and, 2/ it should not change the order of messages. The condition for this is the *isochrony* of the considered couple of programs. This is not detailed here, see [9].

The scheduling specifications we derive from causality analysis still exhibit maximum concurrency. Actual implementations will have to conform to these scheduling specifications. In general, they will exhibit less (and even sometimes no) concurrency, meaning that further sequentialization has been performed to generate code. Of course, this additional sequentialization can be the source of potential, otherwise unjustified, deadlock when the considered module is reused in the form of object code in some environment. We shall see in 4 that a careful use of the scheduling specifications of an STS will allow us to decompose it into modules that can be stored as object code for further reuse, whatever the actual environment and implementation architecture will be.

Enforcing endochrony. Since endochrony is a key feature for programs we have to implement, we must consider the question of moving from exochronous to endochronous programs. As shown above, an answer is to make the roots of the clock hierarchy belonging to some single clock tree. The idea for a simple example such as that of the figure 4 is to add to the considered STS a monitor which delivers the information of presence/absence via the b, b' boolean variables with identical clock h, i.e., $\{k = \mathrm{T}\} \equiv \{b = \mathrm{T}\}$, and similarly for k', b'. The resulting STS is endochronous, since boolean variables b, b' are scrutinized at the pace of activation clock h. Intuitively, moving from exochrony to endochrony corresponds

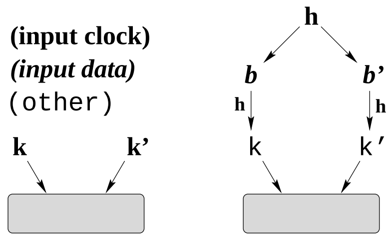

Fig. 4. *Enforcing endochrony*

to equipping the original P program with a suitable *communication protocol* Q in such a way that the compound program $P \bigwedge Q$ becomes endochronous. However, in the general case, this transformation is not unique [9].

3 Distributed code generation: overview of the SACRES approach

3.1 A first glance at the method

The overall method [6, 7, 16] is illustrated in the figures 5 and 6.

Figure 5 shows what the designer has to do. The designer has on her/his screen (at least) three windows (those of the top part of the figure). The first window—top left—is the (SIGNAL, or DC+, or Activity Charts in STATEMATE) program editor. In this window, a dataflow diagram is depicted. The arrows would typically depict flows of data, but also could correspond to scheduling requests. In the bottom right window, icons are shown which allow the designer to specify her/his target architecture. This architecture has two types of constitutive elements. The first one (on the left) is a *processor,* i.e. a machine that complies with the synchronous model of execution, in which a run is a sequence of atomic reactions. Thus processors can be pieces of sequential code (C/C_{++}-procedures, threads, etc.), or alternatively parallel machines running according to the model of perfect synchrony (e.g., synchronous hardware). Other icons refer to (generally *a*synchronous) communication media. Using these two windows, the designer builds, in the third window (top right), her/his execution architecture: the source dataflow diagram is partitioned as shown in the figure, and corresponding subdiagrams are mapped onto "processors" by click-and-point. Also, models of communication links are specified by the designer, by clicking-and-pointing to the appropriate icon. The result of this mapping is built automatically as illustrated in the bottom part of the figure.

Figure 6 shows what the tool generates for each processor. From the specifica-

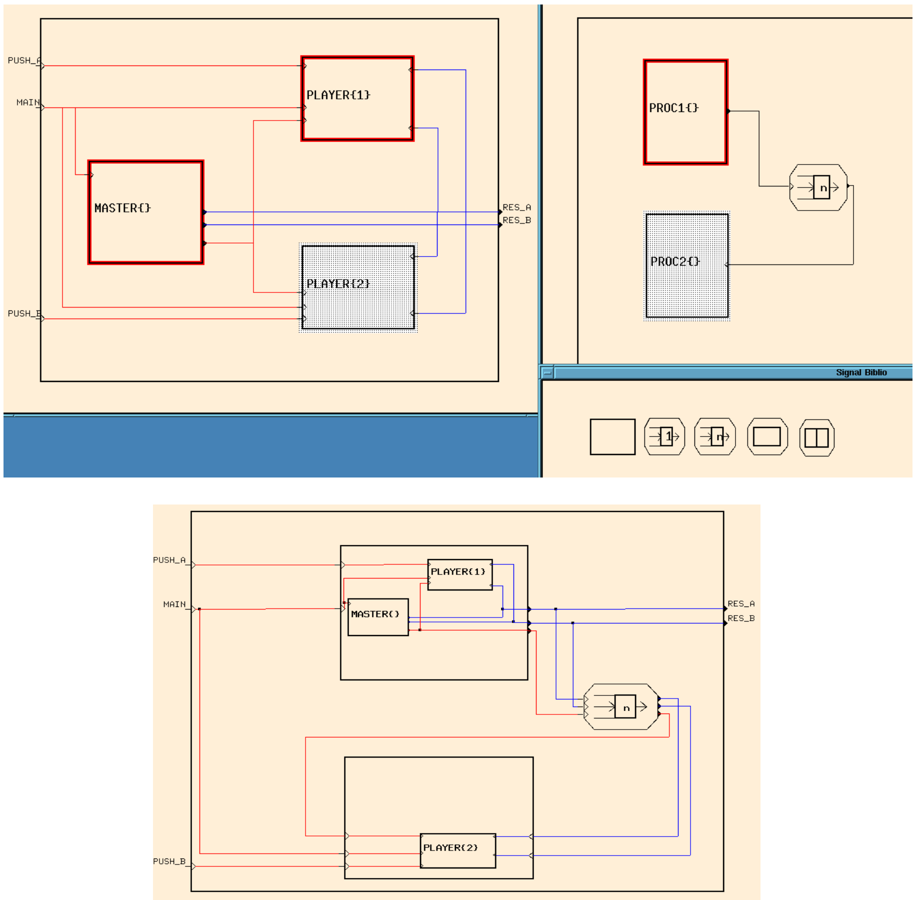

Fig. 5. *Code generation: what the designer does*

tions provided by the designer as in figure 5, the tool will generate, for embedding into each processor, the following: 1/ a suitable communication protocol which guarantees that the semantics of synchronous communication will be preserved even though an asynchronous communication medium is used; 2/ a structuring of the code into pieces of sequential code and a scheduler, aiming at guaranteing separate compilation and reuse.

Finally, the whole model (processors and channel models) can be used for architecture simulation and profiling.

3.2 Summary of the data needed to model the architecture

In the SACRES method, the path from the specification to the implementation goes through the DC$_+$ format. In this context, distributing an application consists in distributing a DC$_+$ program representing a functional graph of flows,

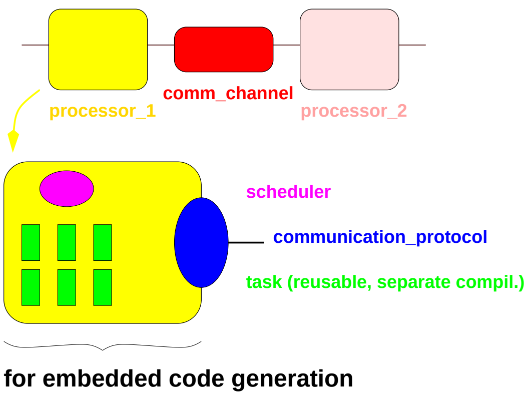

Fig. 6. *Code generation: what the tool generates*

operators and dependences. The target architecture is composed of a set of possibly heterogeneous set of execution components (processors, micro-controllers, ASIC...). At the level considered here, *processor* will be used as a generic name in the following.

We do not address the partitioning/scheduling problem against quantitative optimizations (as for traditional understanding in hardware/software partitioning). Conversely, we assume that an allocation function is given for nodes and flows to processors and links. Moreover, the user interface provided to describe this allocation function differs for specific environments (Sildex, StateMate)—in the figure 5, the Inria environment for Signal is used. These functions are ultimately given in pragma features of a Dc_+ description.

We summarize here which data have to be provided by the users of architecture-dependent implementation of applications in the SACRES environment. A general comment is that the level of detail at which the architecture needs to be known depends quite a lot on the refinement of the mapping to the architecture chosen. This means that in the simplest cases, the amount of data required is fairly small, and simple to assess:

- the set of processors or tasks, and the mapping from operations or subprocesses in the application specification to those processors or tasks. This information enables the partitioning of the Dc_+ graph into sub-graphs grouped according to the mapping.
- the topology of the network of processors, the set of connections between processors, and a mapping from inter-process communications to these communication links. This is useful in the case of flows exchanged between processes located on different processors or tasks, if several of them have to be routed through the same communication medium.
- a definition of the set of system-level primitives used e.g. for communications (readings and writings to the media). Roughly, this amounts to the profiles of the library of functions to which the code will have to be linked.

The concrete form to be given to this information is a question at two levels: the level of $\mathrm{D}\textsc{c}_+$ (which is where we actually perform the compilation work) and the user-level:

– the description of the location mapping can be made by having pragmas associated with nodes, defining their assigned location.
– the description of the architecture can be made using $\mathrm{D}\textsc{c}_+$, describing the graph of the network as a $\mathrm{D}\textsc{c}_+$ graph, with nodes for processors, nodes for communication links, edges for connections, and pragmas for attaching information to the nodes.
– at the user-level, tools and interfaces for entering this information are provided within the existing interfaces to SILDEX and STATEMATE.

The kind of information mentioned above supports the logical distribution of an application. In the perspective of having a more refined code generation, with further degrees of refinement of architecture-adaptation, more information has to be gathered on the architecture:

– concerning processors, taking into account the types of the different processors can lead to a code generation taking advantage of specific characteristics.
– concerning communications, the type and nature of the links (that could be implemented using shared variables, synchronous—blocking—or asynchronous communications...).

If the architecture targeted to features an OS, then in order to be able to generate code using its functionalities, the model needed consists basically in the profile of the corresponding functions, e.g., according to the degree of use of the OS, synchronization gates, communications (possibly including routing between processors) or tasking functions (in the case of un-interruptible tasks: starting and stopping; in the case of interruptible tasks: suspension and resuming, assignment and management of priority levels), etc.

4 Modular approach to the distributed code generation: main steps

When an application is executed on more than one processor, or in more than one task on a single processor, it is necessary to insure that the generated code is a correct implementation of the source specification. This correction has to be proved as far as possible. As long as the implementation process is some refinement in a single formally defined formalism, this proof results from theorems in this theory. Such an approach is adopted in the SACRES method.

We focus this presentation on synchronization and causality in the context of a structural decomposition of a synchronous program targeted on an optimized distributed code.

The code distribution is seen as a user activity allowed by providing a set of formally defined transformations using properties of the semantic model: commutativity, associativity, idempotence of the composition operator. The concept of static abstraction of a behavior (types of the flows plus their clocks and dependences) and parameterisation techniques make possible a modular approach to the distributed code generation.

Assumptions

- As mentioned above, we assume that a *locate* function gives the mapping from functional to physical architectures.
- We suppose the BDC$_+$ code of the program provided with a *well formed tree of clocks:* every flow x, including all the clocks but a single one (named *tick*) is associated with a boolean clock b_x. Each clock c is a boolean input or has an explicit circuitfree definition.
- Every flow which is not an input is functionally defined (no circular definition, the program is deterministic).

4.1 Virtual mapping

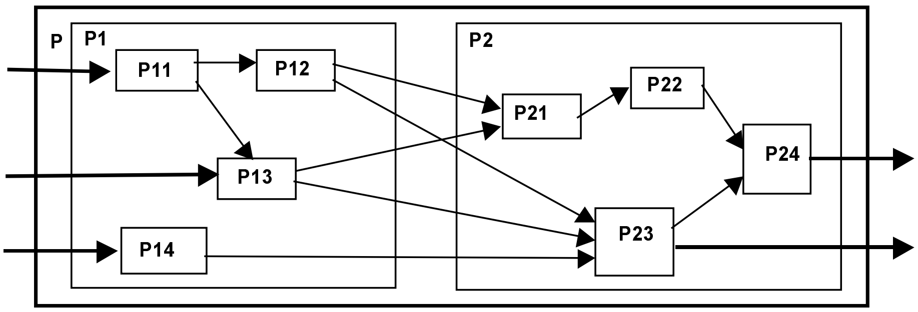

Fig. 7. $P = (P1 \bigwedge P2)$

Given:

- a source program $P = (P_1 \bigwedge P_2 \bigwedge \dots \bigwedge P_n)$, in which each of the P_i may recursively be composed of sub processes $(P_{i1} \bigwedge P_{i2} \bigwedge \dots \bigwedge P_{in})$, (in the example of figure 7, solid lines represent data-flow)
- a set of processors $q = \{q_1, q_2, ..., q_m\}$, and
- a function *locate:* $\{P_i\} \rightarrow \mathcal{P}(q)$ associating with each atomic process P_j a non empty subset of processors,

a process $Q = (Q_1 \bigwedge \dots \bigwedge Q_m)$ is built (cf. figure 8); each process Q_i (source code to be implemented on the processor q_i) is the parallel composition of the set of atomic processes P_k such that q_i belongs to *locate*(P_k). It is easy to prove that Q is equivalent to P. As shown by the profile of the *locate* function, Q may include redundant processes (this is allowed by idempotence). We call *s-task* such a Q_i process.

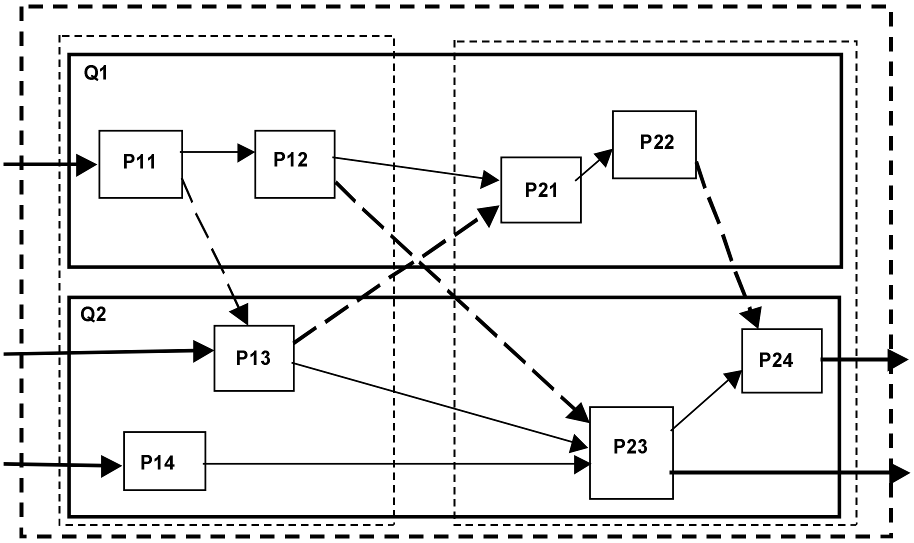

Fig. 8. *Virtual task allocation:* $P = (Q1 \bigwedge PQ2)$,
s-task $Q1 = (P11 \bigwedge P12 \bigwedge P21 \bigwedge P22)$, *s-task* $Q2 = (P13 \bigwedge P14 \bigwedge P23 \bigwedge P24)$

4.2 Traceable compilation

In the process of distributed code generation, we have to care about the structure of the graph after restructuring. This graph is built in such a way that each sub-graph will be executed on a location. The compilation process must preserve this structure. As usual (cf. section 2.2) the first step of the compilation is the construction of a system of clock equations. Clocks are from the overall program. We next solve this system and build a clock hierarchy. The compilation of the virtual mapping is done without splitting the Q_i s-tasks; this compilation builds a global clock system and a global multi-graph.

Instead of splitting the graph across this hierarchy, we will project the hierarchy of clocks onto each sub-graph. Boolean clock definitions generated by the compiler are composed with the Q_i (located on processor q_i) with respect to some heuristic such that each one of the resulting Q'_i has a local tree of boolean clocks (BDC$_+$ endochronous code); the root of this tree is the upperbound of the clocks in Q'_i. We prune the clock hierarchy on each sub-graph to obtain a minimal one, but preserving a BDC$_+$ structure.

We finally prune the clock hierarchy of the graph of sub-graphs conserving only the fastest clock of each sub-graph.

4.3 Local interface abstraction

Each Q_i is associated with an interface containing:

– as inputs, the input flows of P (thick solid arrows in figure 8) and the flow computed in another Q_j, used in Q_i, (thick dashed arrows in figure 8)

– as outputs, the output flows of P and the flows computed in Q_i, and used in another Q_j,
– the clock tree of the input/output flows,
– the clocked dependences between external flows (inputs and outputs of Q_i). For that purpose, a transitive closure has to be calculated.

Transitive closure. Inside a sub-graph, a transitive closure allows to know precedence relations between input flows and output flows (we know if an output flow is preceded by an input flow); however we do not have any information on the other way around (i.e. between an output and an input, through the environment). The transitive closure on the whole graph is the only way to take a global view of the program.

In the case of a dataflow graph, the transitive closure is a well known algorithm. In the case of the DC+ graph it is a little more complex one. We know that a dependence between flows is valid only at certain instants. All dependences of the transitive closure are also valid at certain instants and then associated with a clock. To obtain this clock, we have to apply rules (2,3) given in 2.1.

To avoid deadlocks at execution time, we have to add, in each sub-graph, precedence information resulting from the projection of the transitive closure on input and output flows. The code generation of a sub-graph must take into account these dependence relations to avoid making a dependence cycle when sub-graphs are executed together.

4.4 Local black box abstraction, sensitivity analysis

At the level of a s-task, we have to build a scheduling of the nodes of this s-task. The cost of dynamic execution leads us to reduce as much as possible dynamism. To do that, we will gather nodes is such a way that they can be considered in an atomic way. In this case, the scheduler only has to manage sets of nodes instead of nodes themselves.

Sensitivity equivalence

Definition *We say that two nodes N_1 and N_2 are sensitively equivalent if and only if for each input i:*
there is a causality path from i to N_1 $\Leftrightarrow$ there is a causality path from i to N_2.

Note that in this definition, we do not take clocks of dependences into account.

Two equivalent nodes wait for the same set of inputs: they can be executed in an atomic action depending upon this set of inputs. In this way, we build the classes of nodes transitively depending on subsets of input flows. For instance, the s-task Q_1 built above has two classes C_1 and C_2 as shown in figure 9.

Up to now, the applied transformations keep the program semantics unchanged.

Note that in the worst case, n input flows lead to 2^n sensitivity equivalence classes. In this case, the management of classes is almost as hard as the management of nodes themselves. Another way to do the partitioning into classes is to consider two or more input flows as atomic (considering two flows atomic is considering that they will be read or written at the same physical moment). Then the partitioning is done on subsets of atomic sets of input flows.

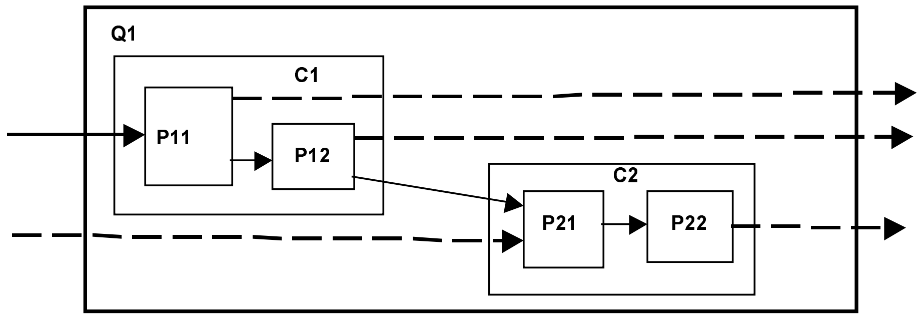

Fig. 9. *Sensitivity classes in $Q1$*

Local code generation All the nodes of a sensitivity class can be executed as soon as the subset of its input flows is available. We can generate any scheduling inside a class without modifying the observable behavior of the overall initial program. Thus, we will generate a sequential code, assuming that input values are available each time a new step starts for the class.

All the local flows, but state variables, can freely be implemented in registers or variables. State variables have to be implemented in remanent memory.

Black box abstraction We have said that inside a sensitivity class, we can generate any scheduling which respects dependences. This means that for any such class, the only dependences that we have to consider are that any input precedes any output at any time. For the sake of scheduling, a class is seen as a procedure call. The scheduling of classes in then done without taking into account what is inside, just the above property. For each class C_i, we just consider an interface in which all inputs precede all outputs. The abstraction of a class is a *black box* abstraction. For example, in the Q_1 sub-graph we obtain the result represented on figure 10, in which causality arcs have been substituted to the original dataflow sub-graphs C_1 and C_2.

4.5 Grey box abstraction for a s-task

If s-tasks are implemented as a single task on an execution entity, communication of internal data is done in the form of variable reads and writes in the same

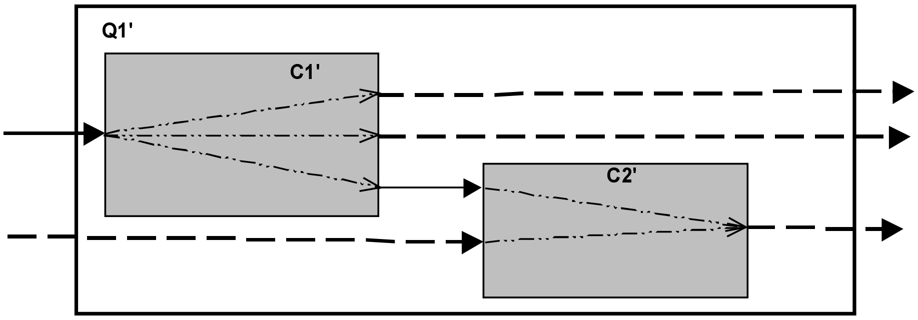

Fig. 10. *Sub-graph abstractions*

memory space. If s-tasks are implemented as a set of tasks, communication corresponds to inter-tasks communications as supported by the tasking system.

From the point of view of the environment, the kind of implementation chosen for a s-task does not matter. But to ensure the correct read-write sequencing, the internal communications have also to be abstracted as causality relations. Then we obtain what we call a *grey box abstraction,* in the sense that we know more than just its interface. For example, the grey box abstraction of the s-task Q_1 is depicted in figure 11.

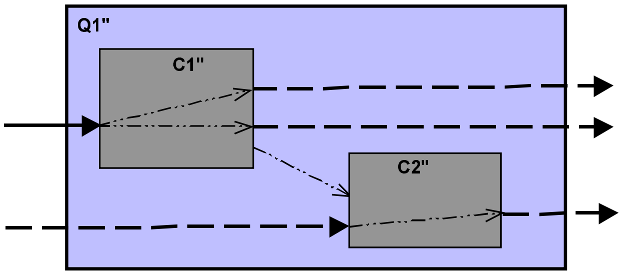

Fig. 11. *s-task abstraction*

4.6 Code generation for each s-task

Communications Inter s-task communication is communication between execution entities of an architecture. The communications are generated depending on the OS primitives. This generation is done by making calls to the right OS primitives given by the architecture description or by the user.

Communication features can be described as synchronous process abstractions (some *grey box*). We assume the mapping of links to these devices is given. Due to the large variety of communication features, multiplexing possibilities, synchronization relaxing policies, it is not possible to provide a general automatic mapping implementation; a specific library should be provided by the user (cf. figure 12).

To represent the new graph obtained by adding communications, we have to add causality links with the communication nodes.

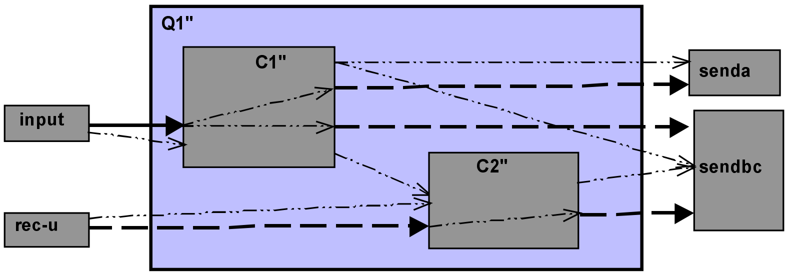

Fig. 12. *s-task abstraction with communications*

S-task scheduling A scheduling is computed for each s-task taking into account the global dependence graph and any other assertion on the environment.

In particular cases, for example obviously when the initial program is fully scheduled by construction, then we can build a static scheduling for its partition into s-tasks. In other cases, or when we want it, we have to build a dynamic scheduling, e.g. depending on the order of arrival of inputs of the different parts of the s-tasks. The order of the execution of nodes is given by scheduling clocks exchanged with the environment.

OS scheduling. For a task such as Q_1'' (cf. figure 12) it is possible to generate parallel processes scheduled and synchronized (synchronization gates) by OS primitives as shown for example in figure 13. In this case, nothing is known about the interleaving of elementary actions in T_1 and T_2.

Parameterized scheduling. Conversely, one can make more explicit the interactions between the s-task components and the OS; this is allowed by clocked causality arrows in $\mathrm{D{\small C}}_+$: considering two non ordered atomic actions A_1 and A_2 on the same processor, each time A_1 and A_2 are both executed, either A_1 is executed before A_2 or A_2 is executed before A_1; this variable scheduling can be represented by arrows from A_1 to A_2 labeled by S or conversely from A_2 to A_1 labeled by *not* S, where S is a boolean clock whose clock is the set of instants at which A_1 and A_2 are both executed. A single process is known by the OS. This is illustrated in figure 14.

The scheduling can then be described as a $\mathrm{D{\small C}}_+$ program depending on timing, values, etc., on which verification is possible; or also, S can be provided

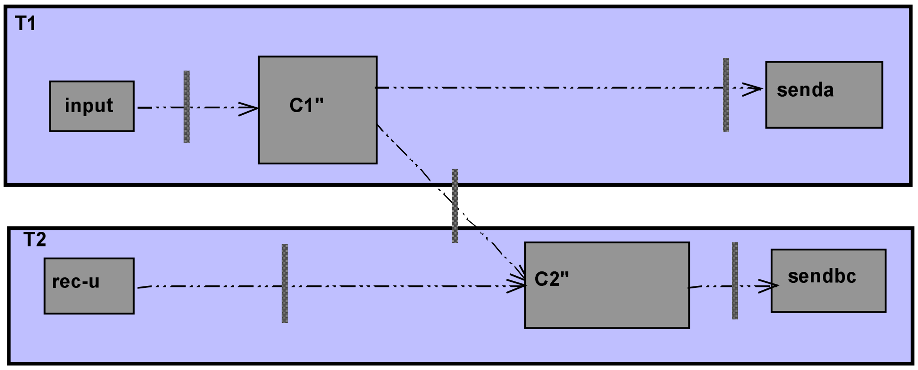

Fig. 13. *OS s-task scheduling*

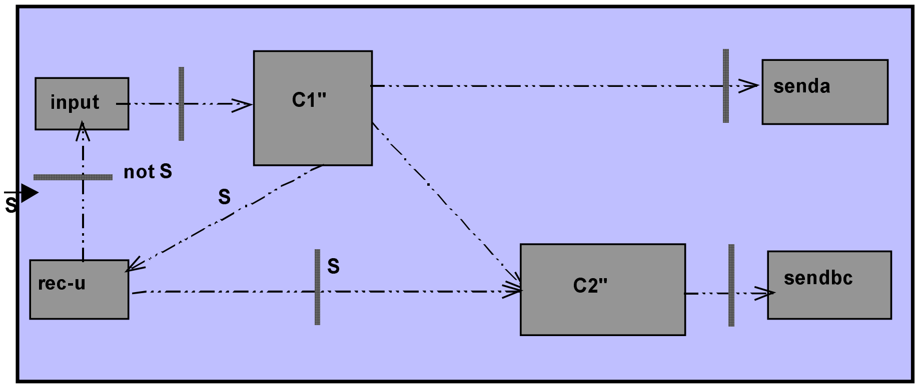

Fig. 14. *Parameterized s-task scheduling*

by the OS, or it can be a constant allowing then static scheduling as shown in figure 15.

Synchronization and communication The synchronization between tasks has to be defined by the user. Different schemes can be implemented.

Strong synchrony. The first implementation that can be done is to set a global gate to synchronize tasks. At each step, each activated task signals its completion to its predecessors in the graph when it has been completed and has received completion acknowledge from each of its successors. Thus at most one logical step is computed during the same instant.

Weak synchrony. A second scheme is to set local gates to synchronize tasks. At each step, each activated task signals to its predecessors in the graph that it is ready to proceed a new step when its previous step has been completed and a

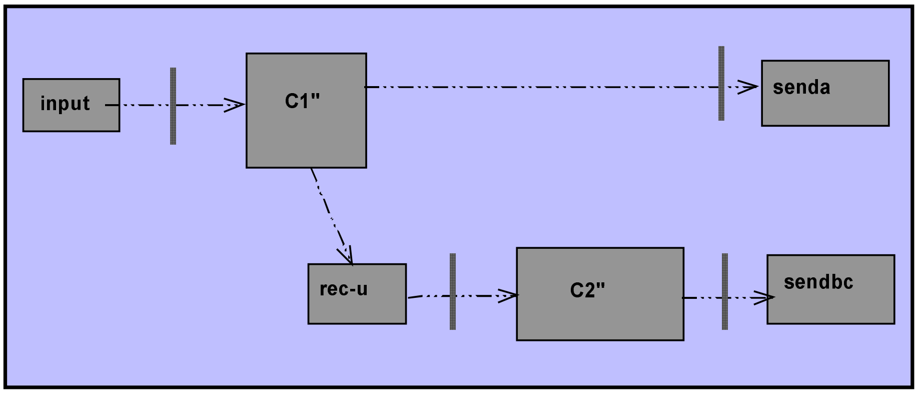

Fig. 15. *Static s-task scheduling with $S = true$*

"ready to proceed" has been received from each of its successors concerning this previous step. This implementation allows pipeline.

Bounded asynchrony. The most permissive scheme for embedded systems is to allow communication implementation as bounded FIFO; the *window* operator of $\mathrm{D}\textsc{c}_+$ makes possible the semantic description of this partially asynchronous scheme.

4.7 Code instrumentation

Finally, we briefly mention our approach to code instrumentation, to evaluate e.g. performance [14].

Given an implementation Q of a program and a model of time consumption for each of the atomic actions in Q, we automatically generate a program $T(Q)$ homomorphic to Q; $T(Q)$ is the parallel composition of the images $T(Q_i)$ of the components Q_i (including communications) of Q. $T(Q_i)$ are given by the user as $\mathrm{D}\textsc{c}_+$ components whose interfaces are composed of integer flows $T(x)$ instead of the original flows x. $T(x)$ represents the sequence of the availability dates for the occurrences of the original flow x.

$T(Q)$ is thus a model of real time consumption of the application (functional specification and architectural support). Some real time properties to be satisfied can be described as predicates in $\mathrm{D}\textsc{c}_+$. Then these properties can be checked by using the verification tools of SACRES for instance.

5 Conclusion

This paper presents a framework for distributed code generation of synchronous programs that allows relaxing synchrony, thanks to the property of endochrony. On the other hand, the definition of precise abstractions of the programs permits reuse of separately compiled programs. So the method is also a method for separate compilation.

It is implemented in the SACRES project, through a number of software modules applicable to DC_+ programs. These are for example:

- **clock calculus,** which is the core of the DC_+ compiler and consists in computing the *clock hierarchy* (it is used in particular to check endochrony);
- **root adjunction, and event conversion,** implemented as a transformation of a DC_+ program to a BDC_+ one, which consist in inserting a *master clock* and converting clocks into boolean flows (this is used in particular to move from exochronous to endochronous programs);
- **building s-tasks,** based on user directives of location mapping, which performs the extraction of DC_+ sub-programs;
- **computing abstractions of** DC_+ **programs,** which consists in computing the transitive closure of dependences and projecting it onto the input/output interface;
- **building tasks,** which performs the extraction of tasks according to an input driven partitionning and calculates the scheduler of these tasks;
- **sequentializing** DC_+ **programs,** which consists in preparing, for each executable program, the computing of a legal sequential scheduling;
- **distributed code generation,** performed on the result of the structuring of the code into tasks and s-tasks, which uses the sequential code generation for tasks and a specific code generation for the schedulers; the generated code makes calls to communication functions from a library to which it is linked.

The method can be applied to many possible targets, using different real-time kernels for instance (let us mention Posix, VxWorks, OSEK for automotive, ARINC for avionics...).

Also, thanks to the translation of StateMate to DC_+ developed in the SACRES project [4, 5], it is applicable not only to dataflow programs such as Signal ones, but also to StateMate designs. Finally, it is partly available in the industrial Sildex tool commercialised by TNI[1]. Application of the method has been carried out on case studies provided by industrial users [3].

References

1. T.P. Amagbegnon, L. Besnard, P. Le Guernic, *Arborescent canonical form of boolean expressions*, Inria Research Report RR-2290, June 1994, `http://www.inria.fr/RRRT/RR-2290.html`.
2. T.P. Amagbegnon, L. Besnard, P. Le Guernic, "Implementation of the dataflow language Signal", in *Programming Languages Design and Implementation,* ACM, 163–173, 1995.
3. P. Baufreton, X. Méhaut, E. Rutten, "Embedded Systems in Avionics and the SACRES Approach", in *Proceedings of The 16th International Conference on Computer Safety, Reliability and Security, SAFECOMP'97,* York, United Kingdom, Springer, September 1997.

1 http://www.tni.fr

4. J.-R. Beauvais, T. Gautier, P. Le Guernic, R. Houdebine, E. Rutten, "A translation of StateCharts into Signal", in *Proceedings of the International Conference on Application of Concurrency to System Design (CSD'98)*, IEEE Publ., Aizu-Wakamatsu, Japan, March 23–26, 1998, 52–62.
5. J.-R. Beauvais, R. Houdebine, P. Le Guernic, E. Rutten, T. Gautier, *A Translation of Statecharts and Activitycharts into Signal Equations,* Inria Research Report RR-3397, April 1998, `http://www.inria.fr/RRRT/RR-3397.html`.
6. A. Benveniste (& contributors), "Safety Critical Embedded Systems Design: the SACRES approach", in *Proceedings of Formal Techniques in Real-Time and Fault Tolerant systems, FTRTFT'98,* Lecture Notes in Computer Science, Springer Verlag, September 1998.
7. A. Benveniste, T. Gautier, P. Le Guernic, E. Rutten, "Distributed code generation of dataflow synchronous programs: the SACRES approach", in *Proceedings of The Eleventh International Symposium on Languages for Intensional Programming, ISLIP'98,* Sun Microsystems, Menlo Park, California (USA), May 1998.
8. A. Benveniste, G. Berry, "Real-Time systems design and programming", *Another look at real-time programming*, special section of *Proc. of the IEEE,* vol. 9 n^o 9, September 1991, 1270–1282.
9. A. Benveniste, P. Le Guernic, P. Aubry. *Compositionality in Dataflow Synchronous Languages: Specification & Code Generation,* Inria Research Report RR-3310, Nov. 1997, `http://www.inria.fr/RRRT/RR-3310.html`, see also a revised version co-authored with B. Caillaud, Mar. 1998.
10. A. Benveniste, P. Le Guernic, Y. Sorel, M. Sorine, "A denotational theory of reactive synchronous systems", *Information and Computation*, 99 n^{o}2:192–230, 1992.
11. N. Halbwachs, *Synchronous programming of reactive systems,* Kluwer Academic Pub., 1993.
12. D. Harel, "Statecharts: A visual formalism for complex systems", *Science of Computer Programming,* 8:231–274, 1987.
13. D. Harel, A. Naamad, "The STATEMATE Semantics of Statecharts", *ACM Transactions on software engineering and methodology,* vol. 5, n^o 4, 1996.
14. A.A. Kountouris, P. Le Guernic, "Profiling of SIGNAL Programs and its application in the timing evaluation of design implementations", in *Proceedings of the IEE Colloq. on HW-SW Cosynthesis for Reconfigurable Systems,* HP Labs, Bristol, UK, IEE, February 1996.
15. P. Le Guernic, T. Gautier, M. Le Borgne, C. Le Maire, "Programming real-time applications with Signal", *Another look at real-time programming*, special section of *Proc. of the IEEE,* vol. 9, n^o 9, September 1991, 1321–1336.
16. P. Le Guernic, S. Machard, E. Rutten, "Répartition de programmes Signal", in *Actes des Rencontres Francophones du Parallélisme des Architectures et des Systèmes, RenPar'10,* Strasbourg, 9–12 juin 1998 *(in French).*
17. A. Pnueli, N. Shankar, E. Singerman, "Fair Synchronous Transition Systems and their Liveness Proofs", in *Proceedings of Formal Techniques in Real-Time and Fault Tolerant systems, FTRTFT'98,* Lecture Notes in Computer Science, Springer Verlag, September 1998.
18. SACRES consortium, *EP 20897 Deliverable report: The semantic foundations of* SACRES, March 1997.
19. SACRES consortium, *The Declarative Code DC+, Version 1.4,* Esprit project EP 20897: Sacres, Nov. 1997.