Distributed code generation of dataflow synchronous programs: the SACRES approach ♦

Albert Benveniste, Thierry Gautier, Paul Le Guernic, and Eric Rutten §

IRISA / INRIA
Campus de Beaulieu - 35042 Rennes Cedex - FRANCE

Abstract

The SACRES project is an Esprit R&D project, SACRES members are: Siemens (Lead partner), i-Logix, TNI (Techniques Nouvelles d'Informatique), OFFIS, INRIA, the Weizmann Institute of Science, British Aerospace, SNECMA. SACRES is dealing with the development of new design methodologies and associated tools for safety critical embedded systems. Main targeted users are aeronautics, automobile, process control and energy. Emphasis has been put on formal techniques for modular verification of the specifications, distributed code generation, and generated code validation against specifications. These techniques aim at making more flexible the exploration of the software life cycle. Verification of the specifications and generated code validation aim at helping for certification of the overall design. Distributed code generation aims at reducing the dependency of the design with respect to the target architecture. In all cases, modularity helps reuse of existing designs, and makes it possible to address much larger systems. A central item of SACRES is the DCM format for synchronous languages, which provides the common semantic framework for all tools as well as end user specification formalisms (STATEMATE and SILDEX/SIGNAL). In this paper, we mainly concentrate on the approach to distributed code generation.

Keywords: embedded systems, safety critical systems, distributed code generation, synchronous languages, SIGNAL.

♦This work is supported by the Esprit project R&D -SACRES (EP 20897).
†An extended version of this paper, "Safety Critical Embedded Systems Design: the SACRES approach" describing the whole SACRES approach, will appear as a tutorial on SACRES at PRRTFT’98 [4].
§All participants to the SACRES project are gratefully acknowledged. SACRES members are: Siemens (Lead partner), i-Logix, TNI (Techniques Nouvelles d'Informatique), OFFIS, INRIA, the Weizmann Institute of Science, British Aerospace, SNECMA.
1 Overall presentation: SACRES architecture

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. Here we briefly review SACRES architecture, and how this architecture helps meeting the objectives. We mainly discuss automatic code generation, with aiming at modularity and reuse. A more complete presentation of the whole project may be found in [4].

SACRES architecture is depicted in Figure 1. SACRES project is based on various tools. Some were available before the project started, and benefited from the project by various enhancements. Some have been developed entirely within the project. There has also been a lot of work integrating the various tools. Here is a description of the main groups of tools:

- The command-level interface permits the tools to be invoked. Some of the tool launch facilities are actually available directly through the menus from the specification tools STATEMATE and SILDEX.
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 from the STATEMATE tool, and SIGNAL based dataflow diagrams from the SILDEX tool\(^1\). Tight integration is offered thanks to the DC_+ pivot formalism. Finally, the dataflow diagram editor is also useful for describing target architectures for (possibly distributed) code generation. The SSL functionality allows to assemble models with components from different specification tools. SSL is available as a textual language, but it is also integrated as part of the graphical editors of some of the specification formalisms — e.g., SILDEX diagrams can be used to describe such assembly.

The DC_+ pivot formalism [16] is a global model format\(^2\) implementing the paradigm of synchronous programming in its full generality. DC_+ format is not expected to be read by the user, just to be passed between tools in the tool-set.

The code generation tool provides both target architecture independent and dependent automatic code generation. It includes 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 tool allows automatic checking of the models that are produced from the specification tools, and manages the results of the proof.

To help the designer in the design cycle (which has to be an iterative development process) SACRES offers the following, cf. Figure 1:

A **semantic based pivot format**: DC_. This format implements the paradigm of synchronous programming in its full generality. It plays a central role in maintaining the soundness all the way along the design cycle, and supports the semantic integration of subsystems at different stages of their design. The following is worth noticing. One main contribution of SACRES is the development of the pivot formalism DC_+.

The DC_+ formalism is a natural candidate for modelling the sampled time part of control engineering diagrams such as provided by, e.g., Simulink or Matrixx. Thus it is a natural candidate for linking control engineering tools to the SACRES tool-set, which would provide a smooth integration of control and software engineering together in the design process.

---

\(^1\)STATEMATE is a visual tool offering Statecharts, Activity Charts, and Module Charts for specifying systems, it is marketed by iLogix, Andover, MA, USA, http://wwwilogix.com. SILDEX is a SIGNAL-based graphical tool marketed by TNI, Brest, France, http://www.tni.fr.

\(^2\)DC_+ is originating from a previous version designed in the framework of the Eureka-Synchron project on synchronous languages.
Specification tools offering animation and simulation capabilities. These are the STATEMATE and SILDEX tools.

System verification. Each component or subsystem at any stage of its design can be checked against certain required properties. And properties of the system can be inferred from properties of their subdesigns.

Abstraction mechanisms. Each component or subsystem at any stage of its design can be abstracted to (an approximation of) its interface behaviour. This is made feasible thanks to the particular features of the DC₄ format, see section 2.

Architectural modelling and distributed code generation. Automatic code generation plays of course a central role in implementing the design cycle. More importantly, target dependent code generation is supported, including distributed asynchronous architectures.

Validating code generation. Official process development models such as DO178B and Def Stan 00-55 impose strong demand on how each move is performed throughout the design process. “Evidence” of correctness must be provided in support of the design. In support for this, SACRES offers code validation, a procedure in which object code can be checked against its associated source code. This involves checking for refinement correctness against compiler optimisations.

2 Symbolic Transition Systems (STS): concepts, objects, and notations

From now on, we review the fundamental bases for SACRES. As SACRES relies on a strong formal modelling basis, we spend significant effort discussing related issues in the section devoted to the model of Symbolic Transition Systems. This abstract model is used to introduce the basics of the DC₄ format. Then, using this general framework, we review code generation from a more theoretical standpoint. Fully theoretical aspects and results are not given here, however, but appropriate references are provided.

2.1 The essentials of the synchronous paradigm

There has been several attempts to characterize the essentials of the synchronous paradigm [5] [9]. With some experience and after attempts to address the issue of moving from synchrony to asynchrony (and back), we feel the following features are indeed essential for characterizing this paradigm:

1. Programs progress via an infinite sequence of reactions:

   \[ P = R^\omega \]
where $R$ denotes the family of possible reactions\textsuperscript{3}.

2. Within a reaction, decisions can be taken on the basis of the absence of some events, as exemplified by the following typical statements, taken from Esterel, Lustre, and Signal respectively:

```plaintext
present S else 'stat'
y = current x
y := u default v
```

The first statement is selfexplanatory. The "current" operator delivers the most recent value of $x$ at the clock of the considered node, it thus has to test for absence of $x$ before producing $y$. The "default" operator delivers its first argument when it is present, and otherwise its second argument.

3. When it is defined, parallel composition is always given by taking the conjunction of associated reactions:

$$P_1 \parallel P_2 = (R_1 \land R_2)$$

Typically, if specifying is the intention, then the above formula is a perfect definition of parallel composition. In contrast, if programming was the intention, then the need for this definition to be compatible with an operational semantics very much complicates the "when it is defined" prerequisite\textsuperscript{4}.

Of course, such a characterization of what the synchronous paradigm is makes the class of "synchrony-compliant" formalisms much larger than usually considered. But it has been our experience that these were the key features for the techniques we have developed so far to work.

Clearly, these remarks call for a common format implementing this paradigm, the DC/DCP format [16] has been proposed with this objective, see section 2.4. Also, this calls for a simplest possible formalism with the above features, on which fundamental questions should be investigated (the purpose of this basic synchronous formalism would not be to allow better specification or programming, however): the STS formalism we describe next has this in its objectives.

### 2.2 Specification: Symbolic Transition Systems (STS)

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

---

\textsuperscript{3}Reactive systems continuously observe their environment, they compute outputs and perform actions in reaction to the reception of stimuli from the environment.

\textsuperscript{4}For instance, most of the effort related to the semantics of Esterel has been directed toward solving this issue satisfactorily [8].

---

23
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. We denote by \( \mathcal{S} \) the set of all states. 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 \( s \).

Following A. Pnueli\(^5\), we define a **Symbolic Transition System (STS)** to be a system

\[
\Phi = (V, \Theta, \rho)
\]

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\(^6\) 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 \( \rho(s^-[V], s[V]) = T \), we say that state \( s^- \) is a \( \rho \)-predecessor of state \( s \).

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

\[
s_0 \models \Theta \land \forall i > 0, (s_{i-1}, s_i) \models \rho \tag{1}
\]

The composition of two STS \( \Phi = \Phi_1 \land \Phi_2 \) is defined as follows:

\[
\begin{align*}
V &= V_1 \cup V_2 \\
\Theta &= \Theta_1 \land \Theta_2 \\
\rho &= \rho_1 \land \rho_2
\end{align*}
\]

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:** we shall use the following generic notations in the sequel:

- \( c, v, w, \ldots \) denote STS variables, these are the declared variables of the STS; useful additional variables are the following.
- for \( v \) a variable, \( h_v \in \{T, \bot\} \) denotes its clock:

\[
[h_v \neq \bot] \iff [v \neq \bot]
\]

---

\(^5\)The talented reader will also notice some close relation to the TLA model of L. Lamport.

\(^6\)Usually, variables and primed variables are used to refer to current and next states. This is equivalent to our present notation. We have preferred to consider \( s^- \) and \( s \), just because the formulas we shall write mostly involve current variables, rather than past ones. Using the standard notation would have resulted in a burden of primed variables in the formulas.
• for \( v \) a variable, \( \xi_v \) denotes its associated state-variable, defined by:

\[
\text{if } h_v \text{ then } \xi_v = v \quad \text{else } \xi_v = \xi^{-}_v
\]

Values can be given to \( s_0[\xi_v] \) as part of the initial condition. Then, \( \xi_v \) is always present after the 1st occurrence of \( v \). By convention, \( \xi_v \) 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. The following set of predicates will be repeatedly used:

- a selector : if \( b \) then \( z = u \) else \( z = v \).
- a register : if \( h_z \) then \( v = \xi^+_z \) else \( v = \bot \).

For the selector, note that the "else" part corresponds to the property "\([b = f] \lor [b = \bot] \)". The more intuitive interpretation of the second statement is "\( v_n = v_{n-1} \)", 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: \( h_z \equiv h_u \). The following synchronization constraint is also a predicate of interest:

- a synchronization constraint : \((b = T) \equiv (h_u = T)\),

meaning that the clock of \( u \) is the set of instants at which boolean variable \( b \) is true.

Here is an example of STS that we shall study in the sequel:

![Diagram]

A run of this STS for the \( z \) variable is depicted on the figure above (considering \( s_0[\xi_z] = 0 \)). Each time \( u \) is received, \( z \) is reset and gets the value of \( u \). Then \( z \) is decremented by one (represented by \( v \)) at each activation cycle of the STS, until it reaches the value \( 0 \). Immediately after this latter instant, a fresh \( u \) can be read, and so on. Note the schizophrenic nature of the "inputs" of this STS. While the value carried by \( u \) is an input, the instant at which \( u \) is read is not: reading of the input is on demand-driven mode. This is reflected by the fact
that inputs of this STS are the pair \{activation clock \( h \), value of \( u \) when it is present\}.

Using these primitives, dataflow synchronous languages such as LUSTRE [9] and SIGNAL [13] are easily encoded. Statecharts [10] are encoded too [2, 3].

Open environments and 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, as will be revealed by our examples, clocks will always be variables, and we shall be able to relate clocks only using \( \land \) (intersection of instants of presence), \( \lor \) (union of instants of presence), and \( \setminus \) (set difference of instants of presence).

Stuttering. In the same vein, it should be permitted, for any STS, to do nothing while the environment is possibly working. This feature has been yet identified in the literature and is known as stuttering invariance or robustness [11, 12]. It is central to TLA, where it is understood that a transition with no event at all and no change of states is always legal.

For an STS \( \Phi \), stuttering invariance is defined as follows: if

\[
\sigma : s_0, s_1, s_2, \ldots
\]

is a run of \( \Phi \), so is

\[
\sigma' : s_0, \bot_0, \ldots, \bot_0, s_1, \bot_1, \ldots, \bot_1, s_2, \bot_2, \ldots, \bot_2, \ldots,
\]

where symbol \( \bot_i \) denotes a silent state in which all state variables keep the value they had at state \( s_i \), while other variables take the value \( \bot \). The number of inserted silent states is \( \geq 0 \) but finite. This models that the considered STS can do nothing for any arbitrary but finite “duration”.

Stuttering invariance is not hardwired into our STS formalism, but a quick inspection of all the statements we have introduced in our example reveals that any composition of them is stuttering invariant. More generally, any STS not involving the always present clock (be it directly or indirectly) is stuttering invariant.

2.3 Enhancing the STS model for the purpose of code generation

2.3.1 Two basic issues

Basically, the problem is twofold: 1/ brute force separate compilation can cause deadlocks, and 2/ generating distributed code is generally not compatible with
maintaining strict compliance with the synchronous model of computation. We illustrate briefly these two issues next.

**Naive separate compilation may be dangerous.** This is illustrated in the following picture:

![Diagram](image)

The first diagram depicts the "dependencies" associated with some STS specification: the 1st output needs the 1st input for its computation, and the 2nd output needs the 2nd input for its computation. The second diagram shows a possible scheduling, corresponding to the standard scheduling: 1/ read inputs, 2/ compute reaction, 3/ emit outputs. This gives a correct sequential execution of the STS. In the third diagram, an additional dependency is enforced by setting the considered STS in some environment which reacts with no delay to its inputs: a deadlock is created. In the last diagram, however, it is revealed that this additional dependency caused by the environment indeed was compatible with the original specification, and no deadlock resulted from applying it. Here, deadlock was caused by the actual implementation of the specification, not by the specification itself.

**Desynchronisation.** This is illustrated in the following picture:

![Diagram](image)

This figure depicts a communication scenario: two processors, modelled as sequential machines, exchange messages using an asynchronous medium for their communications. The natural structure of time is that of a *partial order*, as derived from the directed graph composed of 1/ linear time on each processor, and 2/ communications. This structure for time does not match the linear time corresponding to the infinite sequence of reactions which is the very basis of synchronous paradigm.
The need for reasoning about causality, schedulings, and communications. This need emerges from the above discussion. In the next subsection, we shall introduce a unique framework to handle these diverse aspects: the formalism of scheduling specifications.

2.3.2 Scheduling specifications

Preorders and partial orders to model causality relations, schedulings, and communications. We consider 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 $x \preceq z$). To $\preceq$ we associate the equivalence relation $\equiv$, defined by $x \equiv y$ iff $x \preceq y$ and $y \preceq x$. If equivalence classes of $\equiv$ are singletons, then $\preceq$ is a partial order. Preorders are naturally specified via (possibly cyclic) directed graphs:

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

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 $\text{dom}_V$ the domain of the set $V$ of variables. Denote by $\Lambda_V$ the set of all labelled preorders on $V$. A state $s$ is 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 $\text{dom}_{\Lambda_V}$ the domain in which states take their value.

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

$$\rho \subseteq \text{dom}_V \times \text{dom}_{\Lambda_V}, \quad (4)$$

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 (4).

STS involving such type of preorder relation shall be called in the sequel **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 $\text{bool} \cup \{\perp\}$, and $u, v$ variables of any type, the following generic conjunct will be used:

- $\text{if } b \text{ then } u \rightarrow v$, resp. $\text{if } b \text{ else } u \rightarrow v$

also written:

- $u \overset{b}{\rightarrow} v$ resp. $u \overset{\overline{b}}{\rightarrow} v$
In [6], it is shown that scheduling specifications have the following properties:

$$\begin{align*}
x \xrightarrow{b} y \land y \xrightarrow{c} z & \Rightarrow x \xrightarrow{b \land c} z \\
x \xrightarrow{b} y \land x \xrightarrow{c} y & \Rightarrow x \xrightarrow{b \lor c} y
\end{align*}$$

(5)

(6)

Properties (5,6) can be used to compute input/output abstractions of scheduling specifications:

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 (5,6).

**Inferring scheduling specifications from causality analysis.** We now provide a technique for inferring schedulings from causality analysis for STS specified as conjunctions of the particular set of generic conjuncts we have introduced so far. Considering this restricted set of generic conjuncts is justified by the fact that 1/ all known synchronous languages can be encoded using this set of basic conjuncts [15], and even more, 2/ these primitives allow to express the most general synchronisation mechanisms that are compatible with the paradigm of perfect synchrony [7]. We recall next this set of basic conjuncts for the sake of clarity:

$$\text{if } b \text{ then } w = u$$

$$\text{else } w = v$$

$$u \xrightarrow{b} w$$

(7)

In addition to the set (7) of primitives, state-variable $\xi_v$ 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. Later on, in the examples, we shall freely use nested expressions such as $\text{if } b \text{ then } w = \text{expr}$, where "expr" denotes an expression built on the same set of primitives. It is understood that such expressions need to be expanded prior to applying the rules of formulas (8) given next.
In formulas (8), each primitive statement has a scheduling specification associated with it, given on the corresponding right hand side of the table. Given an STS specified as the conjunction of a set of such statements, for each conjunct we add the corresponding scheduling specification to the considered STS. Since, in turn, scheduling specifications themselves have scheduling specifications associated with them, this mechanism of adding scheduling specifications must be applied until fixpoint is reached. Note that applying these rules until fixpoint is reached takes at most two successive passes. In formulas (8), labels of schedulings are expressions involving variables in the domain \{\bot, \top\} ordered by \{\bot < \top < \top\}; with this in mind, expressions involving the symbols "\land" (min) and "\lor" (max) have a clear meaning.

(R-1) \quad \forall u \quad h_u \rightarrow u

(R-2) \quad \begin{align*}
& \text{if} \quad b \quad \text{then} \quad w = u \\
& \text{else} \quad w = v
\end{align*} \Rightarrow

\begin{align*}
& b \quad h_k \land (b \land h_u) \rightarrow h_w \\
& h_u \quad b \land h_k \rightarrow h_w \\
& h_v \quad b \land h_k \rightarrow h_w \\
& u \quad b \land h_k \rightarrow w \\
& v \quad b \land h_k \rightarrow w
\end{align*} \quad (8)

(R-3) \quad u \xrightarrow{b} w \Rightarrow b \rightarrow h_w

(R-4) \quad w = f(u_1, \ldots, u_k) \quad h_{u_1} = h_{u_2} = \cdots = h_{u_k} \quad \Rightarrow \quad u_1 \rightarrow h_w \rightarrow w

Rules (R-1, \ldots, R-4) are formally justified in [6]. We briefly report the corresponding results. For \( P \) an STS, first apply Rules (R-1, \ldots, R-4) until fixpoint is reached: this yields an STS we call sched(P). Then, a sufficient condition for \( P \) to have a unique deterministic run is:

1. sched(P) is provably circuitfree at each instant, meaning that it is never true that

\[ x_1 \xrightarrow{b_1} x_2 \xrightarrow{b_2} x_1 \]

and

\[ (b_1 \land b_2 = \top) \]

unless \( x_1 = x_2 \) holds.
2. \textit{sched}(P) has provably no multiple definition of variables at any instant, meaning that, whenever

\[
\begin{align*}
\text{if } b_1 & \text{ then } z = \text{exp}_1 \\
\land & \text{ if } b_2 \text{ then } z = \text{exp}_2
\end{align*}
\]

holds in P and the exp\textsubscript{1} and exp\textsubscript{2} are different expressions, then

\[b_1 \land b_2 = \top\]

never holds in P.

Then P is said to be \textit{executable}, and \textit{sched}(P) provides (dynamic) scheduling specifications for this run.

\textbf{Examples.} We show here some STS statements and their associated scheduling as derived from causality analysis. In the following figures, vertices in boldface denote input clocks, vertices in bold-italic denote input data, and vertices in courier denote other variables. It is of interest to split between these two different types of inputs, as input reading for an STS can occur with any combination of data- and demand-driven mode. Note that, for each vertex of the graph, the labels sitting on the incoming branches are evaluated prior to the considered vertex. Thus, when this vertex is to be evaluated, it is already known which other variables are needed for its evaluation.

\textbf{A reactive statement:}

\[
\text{if } b \text{ then } z = u \text{ else } z = v
\]

In the above example, input data are pairwise associated with corresponding input clocks: this STS reads its inputs on a purely data-driven mode, input patterns \((u, v, b)\) are free to be present or absent, and, when they are present, their value is free also. We call it a "reactive" STS.

\textbf{The full example, a proactive STS:}
Applying the rules (R-1, ... , R-4) for inferring schedulings from causality, and then performing some straightforward simplifications, we get the result shown in Table 1. Note the change in control: \{ input clock, input data \} have been drastically modified from the "if b then \( z = u \) else \( z = v \)" statement to the complete STS: inputs now consist of the pair \( \{ h, v_u \} \), where \( v_u \) refers to the value carried by \( u \) when present. Reading of the \( u \) occurs on demand, when condition \( b \) is true. Thus we call such an STS "proactive".

### 2.4 DC\(_{+}\) and its STS Semantics

As the DC\(_{+}\) format actually used in SACRES is not just a model, we present here a sketch of this format. We refer the reader to [16] for a detailed description of the DC\(_{+}\). In the following, we describe how to construct an STS \( \Phi_P \) corresponding to a given DC\(_{+}\) program \( P \).

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 DC₄ 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 ⊥ (cf. section 2.2). Clocks are boolean flows, assuming the values \{T, ⊥\}. 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 \( \text{expr} \) in the language has its corresponding clock \( h_{\text{expr}} \) which indicates whether the value of the expression at the current instant is different from ⊥.

Besides external flows (input/output flows), which determine the interface of the DC₄ program with its environment, also internal flows are used and manipulated by the program.

A sketch of DC₄ primitives. We list in Table 2 the DC₄ primitives. Explanations and intuitive semantics is given now. In these primitives, \( u, v, w, b \)

| expression extended to sequences | \( u = f(u_1, \ldots, u_n) \) |
| assignment extended to flows | \( \text{equ} : w \leftarrow v \text{ at } c \text{ init } w_0 \) |
| memorization | \( \text{memo} : w \leftarrow v \text{ at } c \text{ init } w_0 \) |
| data dependent downsampling | \( v = \text{when} (u, b) \) |
| selector with priority | \( u = \text{pcond} (b, u, v) \) |
| \( \text{left}_1 \text{is true} \) | \( d = \text{left}_1 \text{tt} (b, c) \) |
| \( h \) is the clock of \( u \) | \( h = \text{event} (u) \) |
| scheduling specification | \( u \text{ depends } : u \text{ at } b \) |

Table 2: DC₄ primitives.

denote typed flows, i.e., sequences of values in the considered type extended with the special symbol ⊥. In writing these instructions, we should have written rather statements such as “ \( \text{equ} : w \text{ expr} \text{ at } c \text{ init } w_0 \)” where “\( \text{expr} \)” stands for any DC₄ expression, and then we should have defined “ \( \text{when} (u, b) \)” , etc., as DC₄ expressions. Our presentation will however make the writing of the STS semantics simpler. By convention, the \( b, c, d \) flows are assumed boolean, while the other ones are of any type. In the “ \( \text{at } c \)” statements, the boolean flow \( c \) is an activation condition as will explained later in the semantics. Finally, the last statement is the scheduling specification which states that \( v \) depends on \( u \) when \( b \) is present and true: it is thus exactly the corresponding STS statement. DC₄ program composition and closure are inherited from their respective STS counterparts. The intuitive as well as formal semantics of these instructions will be given subsequently. Next, we give the semantics of these instructions.

System Variables. The system variables of \( \Phi \) are given by \( V = U \), where \( U \) are the DC₄ flows explicitly declared and manipulated in \( P \). An auxiliary vari-
able by the name of $\zeta_v$ is included in an auxiliary set $X$ of variables whenever needed, this will be made explicit in giving the STS semantics of $\text{DC}_+$ subsequently. The value of $\zeta_v$ is intended to represent the value of $v$ at the previous instance (present excluded) that $v$ was different from $\bot$. The $\zeta_v$ is closely related to our $\xi_v$, but is not identical, hence the different notation for it.

**Initial Condition.** The initial condition for $\Phi$ is given by

$$\Theta: \bigwedge_{u \in U} u = \bot \land \bigwedge_{v \in U} \zeta_{\neg v} = v_0$$

As will result from our STS translation of $\text{DC}_+$ programs they are all stutteringly robust. Consequently, we can simplify things by assuming that the first state in every run of the system is a stuttering state.

**Transition Relation.** The transition relation $\rho$ will be a conjunction of assertions, where each $\text{DC}_+$ statement gives rise to a conjunct in $\rho$. We list the statements of $\text{DC}_+$ and, for each statement $S$, we present the conjunct contributed to $\rho$ by $S$.

- Consider the $\text{DC}_+$ expression $v = f(u_1, \ldots, u_n)$, where $f$ is a state-function. Its contribution to $\rho$ is given by:

$$h_v \equiv h_{u_1} \equiv \ldots \equiv h_{u_n} \land h_v \neq \bot \rightarrow v = f(u_1, \ldots, u_n))$$

This formula requires that the signals $v, u_1, \ldots, u_n$ are present at precisely the same time instants, and that at these instants $v = f(u_1, \ldots, u_n)$.

- The contribution of the statement

$$\text{equ} : w \; v \; \text{at} \; c \; \text{init} \; w_0$$

requires the introduction of an auxiliary variable $\zeta_v$ with initial value $w_0$, it is given by:

$$h_w \equiv h_v \equiv h_\zeta$$

$$\land \; \zeta_v = \begin{cases} \text{if } c = T \text{ then } v \\ \text{else } \zeta_{\neg v} \end{cases}$$

$$\land \; w = \begin{cases} \text{if } h_v \text{ then } \zeta_v \text{ else } \bot \end{cases}$$

Roughly speaking, this statement delivers the most recent value of $v$.

- The contribution of the statement

$$\text{memo} : w \; v \; \text{at} \; c \; \text{init} \; w_0$$
requires the introduction of an auxiliary variable $\zeta_v$ with initial value $w_0$, it is given by:

\[
\begin{align*}
  h_w & \equiv h_v \equiv h_c \\
  \wedge \zeta_v &= \left( \begin{array}{l}
  \text{if } c = T \text{ then } v \\
  \text{else } \zeta_v
  \end{array} \right) \\
  \wedge w &= \text{if } h_v \text{ then } \zeta_v \text{ else } \bot
\end{align*}
\]

This statement delivers the last value of $v$ (Compare with the semantics of \texttt{equ}).

- The contribution of the statement

\[
v = \text{when } (u, b)
\]

is given by:

\[
v = \text{if } (b = T) \text{ then } u \text{ else } \bot.
\]

- The contribution of the statement

\[
w = \text{pcond}(b, u, v)
\]

is given by:

\[
w = \left( \begin{array}{l}
  \text{if } h_v \text{ and } b = T \text{ then } u \\
  \text{else if } h_v \text{ and } b = F \text{ then } v \\
  \text{else } \bot
  \end{array} \right)
\]

- The contribution of the statement

\[
d = \text{left_tt}(b, c)
\]

is given by:

\[
d = \left( \begin{array}{l}
  \text{if } b = T \text{ then } T \\
  \text{else if } h_b \text{ or } h_c \text{ then } F \\
  \text{else } \bot
  \end{array} \right)
\]

- The contribution of the statement

\[
h = \text{event } (u)
\]

is given by:

\[
h = \text{if } h_u \text{ then } T \text{ else } \bot.
\]

- The contribution of the statement

\[
v \text{ depends } : u \text{ at } b
\]

is given by:

\[
u \xrightarrow{b} v.
\]
3 Code generation

3.1 Distributed code generation: SACRES method

The overall method is illustrated in the figures 2, 3, and 4.

Figure 2 shows what the designer has to do. The designer has on her/his screen (at least) three windows. 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 top 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 top) 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 asynchronous) communication media, they are discussed in section 3.4. Using these two windows, the designer builds, in the third window, her/his execution architecture: the source dataflow diagram is partitioned as shown in the figure, and corresponding sub-diagrams 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.

Figure 3 shows what the tool generates. From the specifications provided by the designer as in Figure 2, the tool generates, 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 (the first problem, cf. below); 2/ a structuring of the code into pieces of sequential code (the so-called “tasks”, cf. section 3.3) and a scheduler, aiming at guaranteeing separate compilation and reuse (the second problem, cf. below).

Figure 4 shows the use of the overall architecture model for simulation and profiling. The whole model of this figure (processors and channel models) can be used for architecture simulation. In addition, it can be enhanced for the purpose of profiling as follows. Duplicate the architecture model: the two models share their inputs. For $x$ an input variable of the architecture model, we associate
its sequence of dates of availability $\delta^x_n$: if $x_n$ is the $n$th present occurrence of $x$, then $\delta^x_n$ delivers the date of availability for $x_n$, measured in terms of some underlying physical quantitative clock (say, in microseconds, or in basic machine cycle when this makes sense). For each $x_n$ delivered to the original model, we deliver $\delta^x_n$ to its copy. Then, in the copy, computations or actions specified are replaced by their corresponding quantitative time subsumption (when can the outputs be delivered, given the date of availability of the inputs). Then, the copy of our architecture model would deliver, for each output $y$ of interest in the architecture model, a corresponding sequence $\delta^y$ of earliest dates of availability.

A particular case of this whole approach is implemented by TNI within the SILDEX tool, it is discussed in section 3.4, and shown in Figure 8.

As indicated before, two major issues need to be considered:

1. **First problem:** 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. **Second problem:** 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.

We consider these two issues next.

### 3.2 Relaxing synchrony

The major problem is that of testing for absence in an asynchronous environment! It is generally not possible, in an asynchronous environment, to decide upon presence/absence of a signal relatively to another one. The major problem is that of “testing for absence” as being part of the control. While this is perfectly sound in a synchronous paradigm, this is meaningless in an asynchronous one.

#### 3.2.1 Desynchronising while keeping the semantics of perfect synchrony

The solution consists in restricting ourselves to so-called *endochronous* STS. Endochronous STS are those for which the control depends only on 1/ the past state, and 2/ the values possibly carried by environment signals, but not on the presence/absence status of these signals. For an endochronous STS, loosing the synchronisation barriers that define the successive reactions will not result in changing its semantics; this is formalized in [6].

An example of an STS which is “exochronous” (i.e., not endochronous) is the “reactive” STS given on the left hand side of the following picture, whereas the “proactive” STS shown on the right hand side is an endochronous STS:
In the diagram on the left hand side, three different clocks are source nodes of the directed graph. This means that the primary decision in executing a reaction consists in deciding upon relative presence/absence of these clocks. In contrast, in the diagram on the right hand side, only one clock, the activation clock $h$, is a source node of the graph. Hence no test for relative presence/absence is needed, and the control only depends on the value of the boolean variable $b$, which is computed internally.

How endochrony allows us to desynchronize an STS is illustrated in an intuitive way on the following diagram, which depicts the scheduling specification associated with the (endochronous) pseudo-statement

"if $b$ then get.u":

In the left diagram, a history of this statement is depicted, showing the successive instants (or reactions) separated by thick dashed lines. In the middle, an instant has been twisted, and in the last one, thick dashed lines have been removed. Clearly, no information has been lost: we know that $u$ should be got exactly when $b = T$, and thus it is only needed to wait for $b$ in order to know whether $u$ is to be waited for also. A formal study of desynchronization and endochrony is presented in [6].

Moving from exochronous to endochronous is easily performed, we only show one typical but simple example:
The idea 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., $\{k = T\} \equiv \{b = T\}$, and similarly for $k', b'$. The resulting STS is endochronous, since boolean variables $b, b'$ are scrutinized at the pace of activation clock $h$. Other schemes are also possible, this is discussed in [6].

3.2.2 Maintaining synchronous semantics of composition while using asynchronous communication media

The second question is that of preserving the semantics of synchronous composition when an asynchronous communication medium is used. Requirement for such a medium is that: 1/ it should not loose messages, and, 2/ it should not change the order of messages. In [6], it is shown that the condition for this is that of the isochrony of the considered pair of STS. Roughly speaking, a pair of STS is isochronous if it is enough for two respective reactions of each STS to agree at each instant on present common variables for their carried values, in order to guarantee that these reactions agree on all common variables for their presence/absence status as well as carried values when they are present. Thus, again, common agreement for reaction composition can disregard absence.

3.3 Generating scheduling for separate modules

Relevant target architectures for embedded applications typically are 1/ purely sequential code (such as C-code), 2/ code using a threading or tasking mechanism provided by some kind of a real-time OS (here the threading mechanism offers some degree of concurrency), or 3/ DSP-type multiprocessor architectures with associated communication media.

On the other hand, the scheduling specifications we derive from rules (R-1,...,R-4) of 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, this was illustrated in subsection 2.3.1. The traditional answer to this problem by the synchronous programming school
has been to refuse considering separate compilation: modules for further reuse should be stored as source code, and combined as such before code generation.

We shall however see that this does not need to be the case, however. Instead, 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.

**The case of single-clocked STS.** We first discuss the case of single-clocked STS, in which all variables have the same clock. The issue is illustrated in the following picture, in which the directed graph defining the circuitfree scheduling specification of some single-clocked STS is depicted:

In the above picture, the gray zones group all variables which depend on the same subset of inputs, let us call them "tasks". Tasks are not subject to the risk of creating fake deadlocks from implementation. In fact, as all variables belonging to the same task depend on the same inputs, each task can be executed according to the following scheme: 1/ collect inputs, 2/ execute task. The actual way the task is executed is arbitrary, provided it conforms the scheduling specification.

In the next picture, we show how the actual implementation will be prepared:
The thick arrows inside the task depicted on the bottom show one possible fully sequential scheduling of this task. Then, what should be really stored as source code for further reuse is only the abstraction consisting of the task viewed as black-boxes, together with their associated interface scheduling specifications.

In particular, if the supporting execution architecture involves a real-time tasking system implementing some preemption mechanism in order to dynamically optimize scheduling for best response time, tasks can be freely suspended/resumed by the real-time kernel, without impairing conformity of the object code to its specification. The generalization to multiple-clocked STS is easy. The only new point is that all items discussed before are now clock-dependent. Thus the computations of the “tasks”, their internal scheduling, and their abstraction, must be performed using scheduling specifications labelled by booleans.

3.4 Using channel models for distributed code generation

In Figure 5, basic channel prototypes are depicted, together with their associated partial order models. In this figure we depict the different types of channel models we consider. From top to down: a bus, synchronous rendez-vous (figured by a token), an unbounded asynchronous fifo, a bounded fifo, and a shared memory. We do not mean that these are the only type of communication we can handle, we only say that these are different types of communication services that can be found in most real-time executives or communication layers. On
the right hand side, the behaviour of these channels is specified using scenarios. Horizontal lines figure physical time elapsing. Downgoing arrows (from \( \text{in} \) to \( \text{out} \)) depict the effective transfer of data. Upgoing arrows (from \( \text{out} \) to \( \text{in} \), dashed) show the blocking of the writing exercised by each considered type of channel. No such blocking is exercised neither by the unbounded asynchronous fifo, nor by the shared memory. Also, for the shared memory, the scenario reflects the fact that multiple writings and readings can occur.

In Figure 6, the partial order channel models of Figure 5 are redrawn according to the paradigm of perfect synchrony. This means that communication takes no time: the input-to-output arrows are now vertical. Of course, the upgoing arrows (dashed) are redrawn accordingly. Let us focus on the first case of the rendez-vous. Rename, for convenience, \( \text{in} : x \) and \( \text{out} : y \). Then, our scenario reads \( \forall n > 1 : x_n \rightarrow y_n \) and \( y_{n-1} \rightarrow x_n \). Setting \( z_n = y_{n-1} \) (formally: "if \( h_y \) then \( z = \xi_y \) else \( z = \bot \)"), we can translate this scenario using STS scheduling specifications as:

\[
\begin{align*}
  x \xrightarrow{h} y \\
  &\quad \land \ x \xrightarrow{h} z \\
  &\quad \land \ h \equiv h_x \equiv h_y \equiv h_z \\
  &\quad \land \ if \ h \ then \ z = \xi_y \ else \ z = \bot \\
  &\quad \land \ if \ h \ then \ y = x
\end{align*}
\]

For an unbounded fifo, we simply get:

\[
x \xrightarrow{h} y
\]
\[ h \equiv h_x \equiv h_y \\]
\[ \text{if } h \text{ then } y = x \]

For a 2-bounded fifo, we use the same specification as in (9), where \( x \) is now the output of a 2-register with input \( y \). The scenario for the shared memory reflects nondeterminism of this medium: clocks \( h_x \) and \( h_y \) are independent and the STS model is:

\[ \text{if } h_y \text{ then } y = \xi_x \text{ else } y = \bot \]

Thus some of the \( x \)'s are lost (when \( h_x = T \) but \( h_y = \bot \)), which models multiple writings. And multiple readings of the same value also occur. Regarded as an STS with \( x \) as input and \( y \) as output, this is a nondeterministic STS.

Clearly, user defined models of channels can be handled in the same way.

Then, assume we have been able to ensure the following for the architecture model of Figure 4:

(C\(_1\)) Each of the two processors \( P_1, P_2 \) is endochronous.

(C\(_2\)) The pair \((P_1, P_2)\) is isochronous.

(C\(_3\)) The compound model \( P_1 \land C \land P_2 \) (where \( C \) is the channel model according to Figure 6) is executable — in particular it is non blocking and deterministic.

Note that \((C_1, C_2)\) only involve the embedded code for each processor, not the channel model. Also, note that \((C_3)\) rules out the shared memory type of channel, due to its nondeterminism. Then:

1. As channel models of rendez-vous or fifo type guarantee message losslessness and order preserving, properties \((C_1, C_2)\) guarantee that the STS semantics is preserved even under asynchrony of communications.

2. For the case of rendez-vous or bounded fifo, bounded desynchronisation is guaranteed, by \((C_3)\). This guarantees use of bounded memory (including channel memory), as well as bounded response time, an important issue for real-time implementations.

The synchronisation of such an architecture is shown in Figure 7. In this figure, two runs are depicted, one for each processor of our two-processor architecture. Synchronisation occurs from time to time, and in between, each processor evolves freely, asynchronously from the other processor.

A particular case of this whole approach is implemented by TNI within the SLDSEX tool, it is shown in Figure 8. Requested channel properties are indicated. For each variable, say \( x \), involved in an asynchronous communication, the added protocol is indicated in gray. It consists of transmitting a boolean variable \( b_x \) whose true occurrences indicate when \( x \) is present. If the clock of \( b_x \) is not known to the receiver, then it is again transmitted in the form of the true occurrences of a boolean variable \( b_x \), and so on (this process terminates). Then,
Figure 7: *Structure of the synchronisation.* Thick vertical bars indicate synchronisation points, while thin vertical bars indicate reactions that are local to each processor: the latter occur freely and asynchronously.

Figure 8: *Implementation of the approach within the SILDEX tool.*

Optimizations can occur in order to remove unnecessary transmission of boolean variables. In doing so, clearly, isochrony of the pair of processes, with their associated protocol included, is guaranteed. Thus distributed implementation is correct by construction. Bounded memory and bounded reaction time is then a separate issue, which can be checked independently.

4 Conclusion

We have presented in this paper a framework for distributed code generation of synchronous programs, allowing for both relaxing synchrony and reuse of separately compiled programs. The heart of the study is the DCₚ format, for which it is given a formal semantics in terms of Symbolic Transition Systems. The format retains the full power of the synchronous paradigm through the handling of multi-clocked flows, but also with scheduling specifications that allow for reasoning about causality, schedulings, and communications. The conditions in which relaxing synchrony is possible and the characterization of components of an application that can be implemented onto one processor are expressed in this model. In addition, proper interfaces for subsequent reuse of components can be designed and it is shown how to generate a correct scheduling
and communication protocol for components. The method is implemented in
the SACRES project, through a number of software modules applicable to DC_+_
programs [14]. These are for example:

- clock calculus, which is the core of the DC_+ compiler and consists in
  computing the clock hierarchy [1] (it is used in particular to check en-
doehrony);

- root adjunction, and event conversion, implemented as a transfor-
mation of a DC_+ program, 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 macro-tasks, based on user directives of location mapping,
  which performs the extraction of DC_+ sub-programs and calculates the
  scheduler relating them;

- computing abstractions of DC_+ programs, which consists in com-
  puting 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 partitioning 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 structur-
ing of the code into tasks and macro-tasks, which uses the sequential code
generation for tasks and a specific code generation for the schedulers of
tasks and of macro-tasks; the generated code makes calls to communica-
tion 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 [2, 3], 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 SLEX tool commercialised by TNI.

Other faces of the whole SACRES project, that allows for improved design
methodologies and tools for safety critical embedded systems, were not presented
in this paper (see [4]); these are multi-formalism design, modular verification of
the applications, and generated code validation against specifications.
References


