Application of Interface Theories to the Separate Compilation of Synchronous Programs

Albert Benveniste, Benoît Caillaud, and Jean-Baptiste Raclet

Abstract*— We study the problem of *separate compilation,* i.e., the generation of modular code, for the discrete time part of block-diagrams formalisms such as Simulink, Modelica, or Scade. Code is modular in that it is generated for a given composite block independently from context (i.e., without knowing in which diagrams the block is to be used) and using minimal information about the internals of the block. Just using off-the-shelf C code generation (e.g., as available in Simulink) does not provide modular code. Separate compilation was solved by Lublinerman et al. for the special case of *single-clocked* diagrams, in which all signals are updated at a same unique clock. For the same case, Pouzet and Raymond proposed algorithms that scale-up properly to real-size applications. The technique of Lublinerman et al. was extended to some classes of multi-clocked and timed diagrams. We study this problem in its full generality and we show that it can be cast to a special class of controller synthesis problems by relying on recently proposed *modal interface theories.

I. Introduction

Modeling languages for embedded control systems, such as Simulink/Stateflow, Modelica, Scade, RT-Builder, and Ptolemy II, have become instrumental in embedded control systems design. Most of the above formalisms use the paradigm of *synchronous programming.* Synchronous programs [2] (Esterel, Lustre, Signal, Lucid Synchrone) progress by a non-terminating loop of successive *reactions* consisting of a number of (possibly concurrent) atomic actions. Synchronous languages support high quality code generation [1]. However, complex control applications run on distributed computing architectures, not obeying the synchronous programming paradigm. Preparing for code distribution requires modular code generation, in which each module is compiled to a grey box equipped with the minimal interface needed for subsequent reuse of the module [3], [4].

To allow for subsequent reuse, the basic approach in Synchronous Programming is to store modules in source format. It has been indeed observed that storing modules as sequentially compiled object code (e.g., C code) may cause problems, as the following simple example shows [5]:

$$P \quad : \quad \forall n \in \mathbb{N} \left\{ \begin{array}{lcl} x_n & = & f(u_n) \\ y_n & = & g(v_n) \end{array} \right. \tag{1}$$

(From now on, logical time index n is always universally quantified and we do not mention this any further.) In (1) successive *reactions* are indexed by n. A typical sequential code to execute one generic step of (1) is

```
void step ( /* input */ Cell *cu, Cell *cv,
            /* output */ Cell *cx, Cell *cy)
{ int u,v,x,y;
  u = cu->get();
  v = cv->get();
  x = f(u);
  y = g(v);
  cx->set(x);
  cy->set(y);};
```

Now, suppose the programmer subsequently reuses P in combination with $Q : v_n = h(x_n)$. The parallel composition of P and Q causes no problem, as each of its reactions can be executed as the following three micro-steps in sequence

$$x_n = f(u_n);\ v_n = h(x_n);\ y_n = g(v_n)$$

However, if P is stored, not as the source specification (1) but as its object code `P.step` and similarly for Q (stored as `Q.step`), then a problem arises: any interleaving of `P.step` and `Q.step` will deadlock, since `P.step` will get blocked waiting for `v.get()` to complete and `Q.step` will get blocked waiting for `x.get()` to complete.

Of course, this deadlock would not have happen if a different scheduling had been generated for P while generating code. Generating the proper scheduling for P indeed requires knowing the context in which P will have to execute. *Separate compilation,* however, aims at generating object code reusable in *any* context.

For example (1), only the source specification could be stored for reuse in any context. On the other hand,

$$P' \quad : \quad \left\{ \begin{array}{lcl} x_n & = & f(u_n) \\ y_n & = & g(x_n) \end{array} \right. \tag{2}$$

can only be scheduled as the sequence of two micro-steps

$$x_n = f(u_n);\ y_n = g(x_n)$$

due to the causality constraint between the two equations constituting (2). This scheduling to the source specification is intrinsic and was not arbitrarily generated by the compiler. By involving a combination of hierarchy and parallel composition, real-size applications exhibit, for their compiled form, a complex intricacy of intrinsic and arbitrary scheduling.

Modular code generation for real-size *single-clocked* programs[2] is indeed a difficult graphical problem that was ad-

An extended version with proofs is is available as a research report [1].

A. Benveniste and B. Caillaud are with Inria, Rennes, France (`Surname.Name@inria.fr`). J-B. Raclet is with IRIT-CNRS, Toulouse, France (`raclet@irit.fr`).

[1]For example, Scade comes with a certified code generator http://www.esterel-technologies.com/products/scade-suite/do-178b-code-generation

[2]By this we mean programs implementing an arbitrary system of recurrent difference equations involving a single discrete time index n, as in our previous simple examples.

dressed by several authors [6], [7], [3], [8], through heuristics and then formally.

However, real-size applications generally involve a number of different clocks. Time-triggered periodic clocks occur in sampled time feedback control. Event triggered clocks arise in the handling of alarms and in mode management. Real-size applications typically involve thousands of different such clocks. This was the motivation for developing rich theoretical support for the compilation of general, multi-clocked, synchronous programs [2], [5]. Regarding separate compilation, only heuristics are implemented in existing tools, however [6], [7].[3] [9] proposes an extension of the approach of [3] for two classes of synchronous programs: programs in which blocks have boolean triggers, and timed block-diagrams in which blocks possess statically defined triggers depending on physical time. The causality analysis used in [9] can be, however, refined.

Altogether, the problem of separate compilation and modular code generation for general programs has not received a comprehensive mathematical answer. We propose such one in this paper. Our approach is by combining the following two theories:

Constructive Semantics: The Constructive Semantics (CS) [10], [5] of a synchronous program is a representation of its set of reactions as a data structure exhibiting program states, the different atomic actions involved in the execution of a reaction of the program, and the causality constraints relating these atomic actions in each different program state. CS is the basis to generate proper schedulings for these atomic actions while generating code.

Interface Theories: Component based design of software was developed since the 80's and resulted in the widespread of Object Oriented programming (OOP) [11]. Modularizing crosscutting concerns in a large system was solved in the 90's by the introduction of Aspect Oriented programming (AOP) [12]. Aspects are now available in Java and other languages [13]. So far OOP and AOP focus on the syntax, leaving aside the underlying mathematical models of programs (their semantics). It is not until 2000 that de Alfaro and Henzinger [14], [15] proposed a mathematically rich theory of *Interface Automata* supporting "separate mathematical analysis" of Nancy Lynch' input-output automata [16]. Based on knowing an Interface Automaton for a given component, it is possible to simulate it, analyze it, and implement it regardless of its future context of use. More recently, Raclet et al. [17], [18] have proposed *Modal Interfaces* and *Acceptance Interfaces*, offering semantic support for both component based and aspect based modular development of input-output automata. These are interface models supporting the composition of interfaces, the conjunction of crosscutting aspects, and a residuation for interface composition—residuation, also called quotient, turns out to be a powerful tool for supervisory controller synthesis [17].

By applying interface techniques to CS in the form of micro-step automata, we are able to solve the issue of separate compilation in its fully general form. The combination CS + interfaces provides an elegant framework to formulate the problem and derive effective algorithms for solving it.

[3]See, e.g., http://www.irisa.fr/espresso/Polychrony/index.php

II. Background on Constructive Semantics

In this section we recall the Constructive Semantics of synchronous programs. Rather than developing it for a concrete programming language, we present it for the abstract model of *Synchronous Transition Systems*, introduced next.

A. Synchronous Transition Systems

We assume a finite alphabet $\mathcal{V}$ of typed *variables*. To capture the multiplicity of clocks, all domains of values are implicitly extended with a special value $\perp$ to be interpreted as "absent." Some of the domains we consider are the domain of pure signals with domain $\{\mathrm{T}\}$, and Booleans with domain $\{\mathrm{T},\mathrm{F}\}$ (both domains are extended with the distinguished element $\perp$). A *state* q is a map, assigning to each variable x a value $x(q)$ over its domain. For a subset of variables $V \subset \mathcal{V}$, we define a V-state to be the restriction of a state to V. We denote by Q_V (or simply Q when no confusion can occur) the set of all V-states.

Definition 1: We define a synchronous transition system *(*STS*) to be a triple $\Sigma = (V, I_0, \rightarrow)$, where: $V \subseteq \mathcal{V}$ is a finite set of typed variables, $I_0 \subseteq Q_V$ is the* initial condition*, and $\rightarrow \subseteq Q_V \times Q_V$ is the transition relation relating past and current states denoted by ${}^{\bullet}q$ and q, respectively.*

Write ${}^{\bullet}q \rightarrow q$ to mean $({}^{\bullet}q, q) \in \rightarrow$. For convenience, we associate to each variable $x \in V$, the following auxiliary variables:

- The *clock* of x, defined by
$$h_x = \textbf{if } x \neq \perp \textbf{ then } \mathrm{T} \textbf{ else } \perp$$
is a pure signal that is present exactly when x is present.
- A *past variable* ${}^{\bullet}x$ defined by ${}^{\bullet}x(q) = x({}^{\bullet}q)$;[4] an initial value for ${}^{\bullet}x$ must be specified. For example, the assertion $x = {}^{\bullet}x + 1$ states that the value of variable x in current state equals its value in previous state plus 1.
- A *memory variable* ξ_x defined by the transition relation
$$\xi_x = \textbf{if } x \neq \perp \textbf{ then } x \textbf{ else } {}^{\bullet}\xi_x$$
Thus, ξ_x is present in any reaction and holds the current or last present occurrence of variable x.

A *run* of Σ is a sequence $\sigma = q_0, q_1, q_2, \ldots$ of states such that $q_0 \in I_0$ and, for every $n > 0$, $q_{n-1} \rightarrow q_n$. Like systems of equations, STS *compose* by conjunction:

$$\Sigma_1 \parallel \Sigma_2 \;=\; (\, V_1 \cup V_2 \,,\, I_{1,0} \wedge I_{2,0} \,,\, \rightarrow_1 \wedge \rightarrow_2 \,)$$

B. Constructive Semantics, an informal introduction

The following example models the behaviour of the first author's venerable watch shown on Figure 1 (initial condi-

[4]${}^{\bullet}x$ roughly corresponds to the "$z^{-1}x$" in control notations.

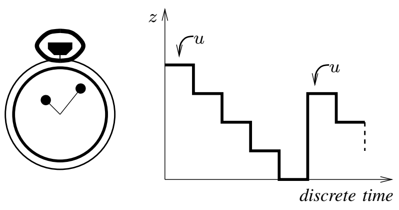


Fig. 1. Albert's venerable watch and a run of it

tions are omitted for simplicity):

$$\begin{array}{rl} & z = \mathtt{if}\ b\ \mathtt{then}\ u\ \mathtt{else}\ v \\ \wedge & v = \mathtt{if}\ h_z\ \mathtt{then}\ {}^\bullet\xi_z - 1\ \mathtt{else}\ \perp \\ \wedge & b = \mathtt{if}\ h_v\ \mathtt{then}\ (v{\leq}0)\ \mathtt{else}\ \perp \\ \wedge & h_u = h_{[b=\mathrm{T}]} \\ \wedge & h_v = h_z = h_b = h \end{array} \tag{3}$$

A capacity u is input to the spring, from which the watch runs for a certain number of seconds, figured by the decreasing counter z. The second equation says that v takes the previous value of z decremented by 1. In the third equation, a boolean test b checks when the capacity for delivering seconds reaches zero. As long as b remains false, then $z = v$ holds, where v is the current value of the decreasing counter—this is the "else" branch of the first equation. When b switches to true, no more capacity is left and thus input capacity must be delivered at the next reaction. The 4th equation expresses that u and the true occurrences of b must have identical clocks, i.e., $b = \mathrm{T}$ triggers the reading of u. The value for z is then provided by the "if" branch of the first equation. The last equation states that signals v, z, and b must have the same clock; we call h this clock, which is the activation clock of the program. A run of Example (3) is depicted on Figure 1. This run only shows the reactions in which this STS is active.

Generating code for Example (3) is by no means trivial. The value carried by u when it is delivered is certainly an input to this STS. On the other hand, "when" u should be delivered is not an input. It is instead decided by the watch itself, upon reaching of capacity zero. Thus u possesses a schizophrenic status, making the causality analysis of this example—and of STS in general—subtle.

In Example (I) below, we show the *causality analysis* of Example (3). The following simple principle is applied in deriving it. Example (3) was specified as a system of equations. We like to see an equation $z = exp(b, u, v)$ defining a variable z as a mean to replace, everywhere in the program, the variable z by the expression $exp(b, u, v)$ defining it. This possibility of rewriting z as $exp(b, u, v)$ is simply encoded using directed graphs: $z{\leftarrow}(b, u, v)$. This may be sometimes too crude an abstraction, however. For example, referring to the first equation of Example (3), dependency $z{\leftarrow}(b, u, v)$ is valid but too coarse. In particular, knowing that $b = \mathrm{T}$ we can infer that v can be discarded from the tuple. Thus, a more accurate abstraction of this first equation is

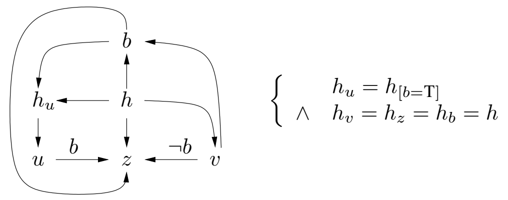
$$(z \leftarrow b) \wedge (z \leftarrow \mathtt{if}\ b\ \mathtt{then}\ u\ \mathtt{else}\ v)$$

Using systematically the above principles, the causality analysis of Example (3) is derived, structurally, for the five

$$\begin{array}{lll} & \overbrace{z \leftarrow b}^{\text{c-dep}} \quad \wedge & \overbrace{z \leftarrow \mathtt{if}\ b\ \mathtt{then}\ u\ \mathtt{else}\ v}^{\text{data dependencies}} \\ \wedge & v \leftarrow h & \\ \wedge & b \leftarrow h \quad \wedge & b \leftarrow \mathtt{if}\ h_v\ \mathtt{then}\ v \\ \wedge & h_u = h_{[b=\mathrm{T}]} & \\ \wedge & \underbrace{h_v = h_z = h_b = h}_{\text{clock synchro}} & \end{array}$$

TABLE I
CAUSALITY ANALYSIS OF EXAMPLE (3)

successive equations shown in Table I. Expression "c-dep" stands for "clock dependency" or "control dependency". The clock synchronizations (4th and 5th equations of Table I) are copied from the original program. The data flow equations (1st to 3rd equations) are abstracted as state dependent graphs. Statement $z \leftarrow \mathtt{if}\ b\ \mathtt{then}\ u\ \mathtt{else}\ v$ means that causality $z{\leftarrow}u$ holds when $b = \mathrm{T}$ and causality $z{\leftarrow}v$ holds when $b = \mathrm{F}$. Observe that no data dependency is associated to the expression $v = \mathtt{if}\ h_z\ \mathtt{then}\ {}^\bullet\xi_z - 1$; the reason is that ${}^\bullet\xi_z$ is a memory whose store is known before starting the reaction.



b
h_u
h
u
b
z
$\neg b$
v
$h_u = h_{[b=\mathrm{T}]}$
$\wedge \quad h_v = h_z = h_b = h$

Fig. 2. This graphical display of Table I yields the due abstraction of Example (3) to generate executable code

Figure 2 is an attempt to display Table I as a graph whose branches are labeled by a predicate on program states—the dependency is active if its associated predicate holds true. The absence of label means a h, the activation clock of the program. The diagram of Figure 2 is a direct specification of an interpreter to execute a reaction of the considered program. Observe that the graph of Figure 2 is acyclic, hence nodes of the graph can be evaluated starting from the source node. Observe that labels sitting on branches are evaluated before entering this branch, so it is known before entering a branch whether it is active or not.

C. Constructive Semantics of an STS

In this section we formalize this notion and briefly recall how to derive the Constructive Semantics of an STS. Due to lack of space we do not provide a complete development but only a sketch of it. The interested reader is referred to [5].

Causality constraints of the form $z{\leftarrow}b$ are turned to another kind of equation by augmenting each domain of values with a special value **?** meaning *not evaluated yet.* Thus, any STS variable possesses the following status while a reaction is being executed: **?** (not evaluated yet), $\perp$ (absent), or present and evaluated to some value of the domain. To simplify we assume a universal domain D of values for all variables, such that $D \ni \perp$. Using the extended domain $D \cup \{\mathbf{?}\}$, $z \leftarrow b$ simply writes as the logical equation $[b = \mathbf{?}] \Rightarrow [z = \mathbf{?}]$. More generally:

$$(z \leftarrow b) \wedge (z \leftarrow \texttt{if}\ b\ \texttt{then}\ u\ \texttt{else}\ v)$$

rewrites as the following equations in the extended domain:

$$\begin{aligned}(b = \mathbf{?} \Rightarrow z = \mathbf{?}) \wedge (b = \mathrm{T} \Rightarrow (u = \mathbf{?} \Rightarrow z = \mathbf{?}))\\ \wedge (b = \mathrm{F} \Rightarrow (v = \mathbf{?} \Rightarrow z = \mathbf{?}))\end{aligned}$$

Focus now on the following causality equation:

$$(b = \mathbf{?} \Rightarrow z = \mathbf{?}) \wedge (b = \mathrm{T} \Rightarrow (u = \mathbf{?} \Rightarrow z = \mathbf{?})) \qquad (4)$$

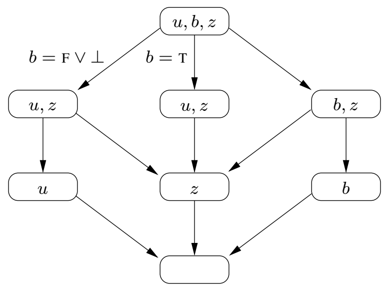


Fig. 3. Micro-step automaton encoding causality equation (4).

In Figure 3, a loop-free automaton is displayed, which models all possible ways of executing a reaction complying with causality constraint (4), in the form of a path starting at the source node and terminating at the sink node. Each state of this automaton lists the variables that are not evaluated yet while traversing it. Call this a Micro-Step Automaton.

A transition evaluates one among the variables listed in its source state. The value assigned is shown as a label of the transition and this variable is thus removed from the sink state of the transition. Attaching the corresponding evaluation action to that transition yields the intermediate format that is the basis for separate compilation.

Micro-step automata (MS-automata) are formalized next:

Definition 2 (MS-automaton and its schedulings): Call MS-automaton *a tuple $A = (V, \mathcal{G}, \varepsilon)$, where*

- *$V \subset \mathcal{V}$ is a finite set of variables;*
- *$\mathcal{G} = (N, E)$ is a finite circuit-free directed graph having a unique minimal vertex n_0, where minimal refers to the order induced by the directed graph; vertices are called* states *and edges are called* transitions*;*
- *$\varepsilon : N \cup E \mapsto 2^V \cup 2^D$ is a* labeling function *such that $\varepsilon(N) \subseteq 2^V$, $\varepsilon(E) \subseteq 2^D$, and*

$$\varepsilon(n_0) = V \qquad (5)$$

$$\forall (n, n') \in E, \exists x \in V : \quad \varepsilon(n) = \varepsilon(n') \cup \{x\} \qquad (6)$$

Formula (6) defines an auxiliary transition labeling *$\lambda : E \mapsto V$ by setting $\lambda(n, n') = x$.*

MS-automaton A is called safe *if any maximal state is labeled by $\emptyset$. For A a safe MS-automaton, call* scheduling *any maximal path of $\mathcal{G}$.*

The state label indicates the set of variables that remain to be evaluated. Condition (6) states that every transition $e = (n, n')$ of the graph specifies the evaluation of exactly one variable. The label $\varepsilon(e)$ of that transition is a predicate over the set of possible values for that variable. With no loss of generality, we can assume that no two different transitions can share both their initial and final states.

Figure 3 shows an MS-automaton. The auxiliary edge labeling λ is redundant and not shown. Two labels "T" ("true") and "F $\vee \perp$" ("false or absent") are assigned to the two alternative evaluations of boolean variable b. The absence of a label for an transition indicates a trivial predicate $\varepsilon(e) = D$.

Observe that, for a safe MS-automaton, we can always assume the existence of a unique maximal state—just superimpose all maximal states since they possess identical labels.

For A_1 and A_2 two MS-automata, their *pre-composition* $A = A_1 \times_{\mathrm{MS}} A_2$ has $V = V_1 \cup V_2$ as its set of variables, and its set of states and transitions is such that $e \in E$ if and only if the two MS-automata agree on which variable to evaluate:

$$\lambda(e) \in V_1 \cap V_2 \quad \Rightarrow \quad \exists e_i \in E_i : \left\{ \begin{array}{l} \lambda(e) = \lambda_1(e_1) = \lambda_2(e_2) \\ \varepsilon(e) = \varepsilon_1(e_1) \cap \varepsilon_2(e_2) \end{array} \right.$$

$$\lambda(e) \in V_i \setminus V_j \quad \Rightarrow \quad \exists e_i \in E_i : \left\{ \begin{array}{l} \lambda(e) = \lambda_i(e_i) \\ \varepsilon(e) = \varepsilon_i(e_i) \end{array} \right.$$

where $i, j \in \{1, 2\}$ and $j \neq i$. The *parallel composition*

$$A \quad = \quad A_1 \parallel_{\mathrm{MS}} A_2 \qquad (7)$$

is obtained by keeping, in $A_1 \times_{\mathrm{MS}} A_2$, the part that is reachable from the minimal state. The composition of MS-automata is associative and commutative.

Using MS-automata we can now formalize what a compilation of an STS Σ is. By following the same technique we used for causality equation (4), we can represent any primitive STS Σ by the MS-automaton $[\![\Sigma]\!]$ collecting all valid schedulings for the evaluation of its variables in any transition of Σ.[5] This and the formula

$$[\![\Sigma_1 \parallel \Sigma_2]\!] \quad = \quad [\![\Sigma_1]\!] \parallel_{\mathrm{MS}} [\![\Sigma_2]\!], \qquad (8)$$

allows defining $[\![\Sigma]\!]$ for any STS. Attaching the corresponding evaluation action to each transition of $[\![\Sigma]\!]$ yields $\{\{\Sigma\}\}$, the *Constructive Semantics* (CS) of Σ. Intuitively, $\{\{\Sigma\}\}$ is a program that implements Σ as its "most permissive" interpreter. Provided that the evaluation of each variable is owned by at most one STS, (8) extends to the CS:

$$\{\{\Sigma_1 \parallel \Sigma_2\}\} \quad = \quad \{\{\Sigma_1\}\} \parallel_{\mathrm{MS}} \{\{\Sigma_2\}\}, \qquad (9)$$

Formula (9) expresses that the CS offers *separate compilation*: to compile $\Sigma_1 \parallel \Sigma_2$, one can safely compile each Σ_i regardless of any context of use and then compose the two compiled forms for each component using the composition $\parallel_{\mathrm{MS}}$. In the following, we consider and solve:

Problem 1: Find a map $\Sigma \mapsto \{\Sigma\}$, mapping any STS *$\Sigma$ to an MS-automaton $\{\Sigma\}$ such that*

1) *For any transition of Σ, $\{\Sigma\}$ contains a non-empty subset of all the schedulings of the CS $\{\{\Sigma\}\}$;*

[5]Deriving the CS for primitive equations requires having a modeling language for STS, see the introduction for examples of such languages. How CS is derived from the syntax of the language is developed in [5].

2) *Separate compilation holds:*

$$\{\Sigma_1 \parallel \Sigma_2\} \;=\; \{\Sigma_1\} \;\parallel_{\text{MS}}\; \{\Sigma_2\}$$

3) $\Sigma \mapsto \{\Sigma\}$ *is minimal (for the inclusion of sets of schedulings) satisfying 1 and 2.*

III. Proposed method

So far our model of STS is not rich enough to distinguish what is under the control of a component from what is under the control of its environment. To encompass this, we partition the set of variables of STS $\Sigma = (V, I_0, \rightarrow)$ into $V = V^{\text{in}} \cup V^{\text{out}}$, where V^{out} and V^{in} collect the variables under control by the component and by the environment, respectively. This partitioning is reflected by the auxiliary labeling function λ of MS-automaton $[\![\Sigma]\!] = (V, \mathcal{G}, \varepsilon)$:

- $\lambda(e) \in V^{\text{out}}$ indicates an evaluation that is under the control of the component;
- $\lambda(e) \in V^{\text{in}}$ indicates an evaluation that is under the control of the environment;
- to account for the abstraction inherent to the Constructive Semantics, we take the convention that the predicate label $\varepsilon(e)$ for $e \in E$ is not under the control of the component (the actual evaluation is not within the scope of A; only the result of the evaluation is visible).

We illustrate in Figure 4 the use of MS-automata for solving the problem of separate compilation.

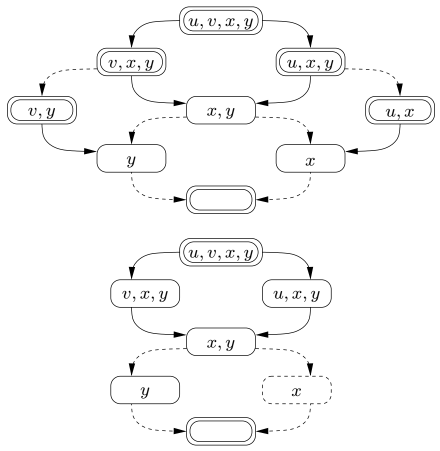


Fig. 4. Micro-step automata encoding causality equations $x \leftarrow u \wedge y \leftarrow v$ (top) and $(x, y) \leftarrow (u, v)$ (bottom). For both STS, u, v are declared inputs and x, y are declared outputs. With the partitioning of transitions as solid/dashed and of states as marked/unmarked, we get Convex Acceptance Interfaces introduced below.

This figure displays the MS-automata encoding the following STS and associated causality equations:

	STS Σ	causality equations
top:	$x = f(u) \wedge y = g(v)$	$x \leftarrow u \wedge y \leftarrow v$
bottom:	$(x, y) = h(u, v)$	$(x, y) \leftarrow (u, v)$

In Figure 4, some transitions are dashed and some states are *marked* (depicted as double rounded boxes). In performing this, the following first set of rules was applied:

Rules 1:

- every transition of the MS-automaton that is under the control of the environment is solid;
- every transition of the MS-automaton that is under the control of the component is dashed;
- every state that is the source of at least one solid transition is marked;
- the final state is marked. □

Applying Rules 1 yields, on Figure 4-top, the MS-automaton associated to $x \leftarrow u \wedge y \leftarrow v$, and on Figure 4-bottom, the MS-automaton associated to $(x, y) \leftarrow (u, v)$.

Finally, we must take into account the coupling between transitions originating from the same state and evaluating the same variable, see Figure 3 for such a situation. We mark this relation as shown on Figure 5. Corresponding rule is:

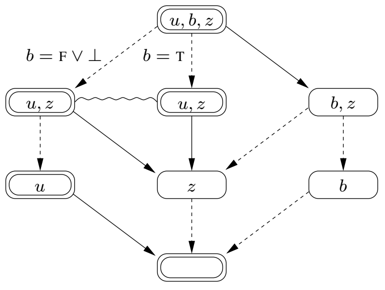


Fig. 5. The MS-automaton of Figure 3 with all the rules applied, thus yielding a Convex Acceptance Interface. For this example, we assume that u is an input and b, z are outputs.

Rules 2: For any three states n_1, n_2 and n_2' such that $\lambda(n_1, n_2) = \lambda(n_1, n_2')$, add a zigzag nondirected arc between n_2 and n_2'. □

The entity denoted $\mathcal{S}$ that is obtained after applying Rules 1 and 2 is called a *Convex Acceptance Interface*. By construction, we have:

Theorem 1: Pruning away $\mathcal{S}$ from dashed transitions preserves separate compilation as long as the following conditions are met: 1) marked states must remain reachable, and 2) zig-zag related transitions must be either removed or kept as a whole.

IV. Convex Acceptance Interfaces (CAI)

We now introduce formally Convex Acceptance Interfaces (CAI) and their pruning. First, we show how all possible separate compilations can be cheaply deduced from a CAI. Second, we equip CAI with a parallel composition reflecting the parallel composition of separate compilations for each module. One can then, either compose several CAI and then extract a separate compilation, or, equivalently, extract a separate compilation from each CAI, and then compose the corresponding separate compilations.

A. Definition and properties

Definition 3: Call Convex Acceptance Interface *a tuple $\mathcal{S} = (V, \mathcal{G}, \varepsilon, E^{\dashrightarrow}, E^{\rightarrow})$ where*

- *$(V, \mathcal{G}, \varepsilon)$ is an MS-automaton as in Def. 2 over the finite circuit-free directed graph $\mathcal{G} = (N, E)$;*

- *$E^{-\to}$ and $E^{\to}$ form a partition of E.*

We denote by $E(n)$, $E^{-\to}(n)$ and $E^{\to}(n)$ the sets of transitions of E, $E^{-\to}$, and $E^{\to}$ stemming from a state n. Next, define $Must(n) \in 2^{2^V}$ and $May(n) \in 2^V$ as follows:

$$Must(n) = \begin{cases} \{\{x_1 \ldots x_k\} \mid x_i \in \lambda(E^{\to}(n))\} & \text{if } E^{\to}(n) \neq \emptyset \\ \{\{x_1\} \ldots \{x_k\} \mid x_i \in \lambda(E^{-\to}(n))\} & \text{otherwise} \end{cases}$$

$$May(n) = \lambda\left(E^{-\to}(n) \cup E^{\to}(n)\right)$$

Now, we claim that pruning $\mathcal{S}$ according to Theorem 1 amounts to choosing, for every state n, a set $X \subseteq May(n)$ of variables such that $Y \subseteq X$ for some $Y \in Must(n)$. Let us denote by $\rho : N \to 2^V$ such a total mapping. The pruning induced by $\mathcal{S}$ and ρ is the following MS-automaton:

*Definition 4: Given $\mathcal{S} = (V, \mathcal{G}, \varepsilon, E^{-\to}, E^{\to})$ and ρ, a ρ-*pruning *of $\mathcal{S}$ consists of an MS-automaton $A^\star = (V^\star, \mathcal{G}^\star, \varepsilon^\star)$ and a simulation relation $\pi \subseteq V^\star \times V$ such that:*

- *the initial states are related: $(n_0^\star, n_0) \in \pi$;*
- *for any pair $(n^\star, n) \in \pi$ and any $x \in \rho(n)$, there exist a transition $e^\star = (n^\star, n^{\star\prime})$ of $A^\star$ and a transition $e = (n, n') \in E^{-\to}(n) \cup E^{\to}(n)$, such that:*
  1) *$\lambda^\star(e^\star) = \lambda(e) = x$ and $\varepsilon^\star(e^\star) = \varepsilon(e)$,*
  2) *and $(n^{\star\prime}, n') \in \pi$.*

Write $A^\star \models \mathcal{S}$ for any so obtained MS-automaton and say that $A^\star$ is an implementation *of $\mathcal{S}$.*

The next step in our construction consists in associating, to every MS-automaton $A = (V, \mathcal{G}, \varepsilon)$ with a decomposition of V into $V = V^{\text{in}} \cup V^{\text{out}}$, a family of CAI as follows:

1) Let $\mathcal{S}_A = (V, \mathcal{G}, \varepsilon, E^{-\to}, E^{\to})$ where $(V, \mathcal{G}, \varepsilon) = A$, $E^{-\to}$ collects the transitions whose λ-label belongs to V^{out} and $E^{\to}$ collects the transitions whose λ-label belongs to V^{in}.
2) A state $m \in N$ is *marked* if it is the source of a transition belonging to $E^{\to}$. Denote by M the set of marked states. Call $\mathcal{G}_m = (N_m, E_m)$ be the sub-graph of $\mathcal{G}$ that is co-reachable from m.
3) to each marked state m, we associate a CAI
$$\mathcal{S}_{A,m} = (V, \mathcal{G}, \varepsilon, E^{-\to}, E^{\to}_m)$$
where $E^{\to}_m$ is obtained from $E^{\to}$ as follows:
$$\begin{aligned} E^{\to}_m(n) \;=\; & \left\{ X \cup \{e\} \;\middle|\; \begin{array}{l} X \in E^{\to}(n) \\ X \cap E_m = \emptyset \\ e \in E^{-\to}(n) \cap E_m \end{array} \right\} \\ & \bigcup \left\{ X \;\middle|\; \begin{array}{l} X \in E^{\to}(n) \\ X \cap E_m \neq \emptyset \end{array} \right\} \end{aligned}$$

Define the *conjunction* of the family $\{\mathcal{S}_{A,m} \mid m \in M\}$ as follows:

$$\bigwedge_{m \in M} \mathcal{S}_{A,m} = (V, \mathcal{G}, \varepsilon, E^{-\to}, E^{\to}_{\wedge}), \text{ where} \tag{10}$$

$$E^{\to}_{\wedge}(n) = \min \left\{ \bigcup_{m \in M} X_m \;\middle|\; X_m \in E^{\to}_m(n) \right\}$$

This definition is justified by the following result:

Theorem 2: Every $A^\star \models \bigwedge_{m \in M} \mathcal{S}_{A,m}$ according to Definition 4 meets the conditions of Theorem 1.

Theorem 2 is important as it expresses that implementations of CAI are the desired intermediate formats for separate compilation. Figure 6 illustrates the pruning.

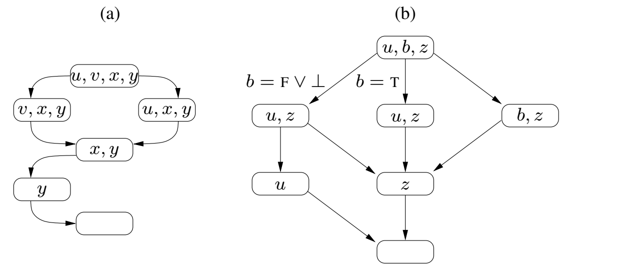


Fig. 6. (a) Showing an MS-automaton implementing the CAI of Figure 4-bottom-right; four implementations exist for this CAI; (b): Showing the unique MS-automaton implementing the CAI of Figure 5.

B. Parallel Composition

Two CAI $\mathcal{S}_i = (V_i^{out} \cup V_i^{in}, \mathcal{G}_i, \varepsilon_i, E_i^{-\to}, E_i^{\to}), i = 1, 2$, are called *composable* if $V_1^{out} \cap V_2^{out} = \emptyset$.

Definition 5: The composition *$\mathcal{S}_1 \otimes \mathcal{S}_2$ of two composable CAI $\mathcal{S}_1$ and $\mathcal{S}_2$ has as sets of input and output variables*

$$\begin{aligned} V^{out} &= V_1^{out} \cup V_2^{out} \\ V^{in} &= (V_1^{in} \cup V_2^{in}) \setminus V^{out} \end{aligned}$$

It is first defined as the composition of the underlying MS-automata as defined in Equation (7). Then,

- *a transition e is in $E^{\to}$ if $\lambda(e) \in V_1 \cap V_2$ and it stems from $e_1 \in E_1^{\to}$ and $e_2 \in E_2^{\to}$ or if $\lambda(e) \in V_i \setminus V_j$ and it stems from $e \in E_i^{\to}$ for $i \in \{1, 2\}$;*
- *e is in $E^{-\to}$ otherwise.*

The composition is commutative and associative, reflecting that components can be assembled in any order without affecting the result. CAI support separate compilation of components as stated in the following theorem:

Theorem 3: The following holds:

$$\left.\begin{array}{l} A_1 \models \mathcal{S}_1 \\ A_2 \models \mathcal{S}_2 \end{array}\right\} \Rightarrow A_1 \parallel_{\text{MS}} A_2 \models \mathcal{S}_1 \otimes \mathcal{S}_2$$

The converse implication is, however, not true. This should not come as a surprise since 1) composing two synchronous programs, and then 2) compiling the code for separate compilation yields in general a code with less concurrency than 1) compiling each module for separate compilation, and then 2) composing the so obtained intermediate codes.

Theorem 3 is in particular useful if we want to restrict the composition of CAI so that the composition of any two safe MS-automata obtained by pruning is also safe. This requires introducing *consistent* pairs of states:

Definition 6: Given n_1 and n_2 two states from $\mathcal{S}_1$ and $\mathcal{S}_2$ respectively, the pair (n_1, m_2) is consistent *if for any $X_1 \in Must(n_1)$ and $X_2 \in Must(n_2)$, we have:*

$$\left(X_1 \cup (V_2 \setminus V_1)\right) \cap \left(X_2 \cup (V_1 \setminus V_2)\right) \neq \emptyset$$

Any pair of two maximal states is, by convention, consistent.

Theorem 4: If every pair of states of $\mathcal{S}_1 \otimes \mathcal{S}_2$ is consistent then the product $A_1 \|_{\mathrm{MS}} A_2$ of any safe A_1 with any safe A_2 such that $A_1 \models \mathcal{S}_1$ and $A_2 \models \mathcal{S}_2$ is also safe.

C. Procedure for Separate Compilation and Complexity

Our algorithms to generate modular code suited to separate compilation consists of the following steps, to be performed for each module:

1) Deriving, from a synchronous program, a Constructive Semantics (CS) in the format of Table I or Figure 2;
2) Translating each equation of the CS into a basic MS-automaton, e.g., moving from (4) to Figure 3; this yields a set of MS-automata;
3) Computing the corresponding composition yields the CS of the given module in the form of a single MS-automaton;
4) Identifying the responsibilities, for the evaluation of each variable, e.g., moving from Figure 3 to Figure 5;
5) Applying the pruning.

For each step we provide the corresponding complexity:

1) This is syntax-dependent, so we do not include it in our evaluation;
2) Linear;
3) This step yields an MS-automaton whose number of states and transitions is exponential in the number of equations and variables of the given module;
4) Linear;
5) Due to the conjunction operation, this step is exponential in the number of states of the MS-automaton.

In practice, steps 1)–3) are not performed this way. Efficient compilers use highly optimized procedures as this is the key in obtaining rapidly good and compact sequential code. Still, the overall complexity should not differ.

V. Conclusion

Separate compilation for multi-clocked synchronous programs was solved in its full generality. Our solution relies on two tools: Micro-Step Automata representing the Constructive Semantics of synchronous programs, and Convex Acceptance Interfaces, a framework belonging to the family of interface theories.

It makes sense that separate compilation relies on interface theories, since the latter aim at providing formal bases to component based design of systems. More generally, interface theories provide a novel and efficient tool to solve supervisory control problems in a compositional way. Indeed, a CAI can be seen as a plant enriched with controllability information: solid transitions are labeled with *uncontrollable* actions while dashed transitions are labeled with *controllable* actions, a basic distinction introduced in Ramadge and Wonham's control theory [19]. Moreover, zig-zag related transitions are similar to the notion of indistinguishability from [20] where the occurrence of an action among a set of actions is detected. Last, the reachability constraints from marked states in CAI corresponds to a control objective and the pruning operation is similar to controlling a plant.

Convex Acceptance Interfaces possess more operators than presented here. They were not developed as they do not appear to be relevant to separate compilation. We envision further use of CAI in the debugging of synchronous programs, particularly for languages such as Esterel or Signal in which causality circuits can occur.

References

[1] A. Benveniste, B. Caillaud, and J.-B. Raclet, "Application of interface theories to the separate compilation of synchronous programs," INRIA 8030, Tech. Rep., 2012. [Online]. Available: http://hal.inria.fr/hal-00721049

[2] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. L. Guernic, and R. de Simone, "The synchronous languages 12 years later," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 64–83, 2003.

[3] R. Lublinerman, C. Szegedy, and S. Tripakis, "Modular code generation from synchronous block diagrams: modularity vs. code size," in *POPL*, Z. Shao and B. C. Pierce, Eds. ACM, 2009, pp. 78–89.

[4] D. Biernacki, J.-L. Colaço, G. Hamon, and M. Pouzet, "Clock-directed modular code generation for synchronous data-flow languages," in *LCTES*, K. Flautner and J. Regehr, Eds. ACM, 2008, pp. 121–130.

[5] A. Benveniste, B. Caillaud, and P. L. Guernic, "Compositionality in Dataflow Synchronous Languages: Specification and Distributed Code Generation," *Inf. Comput.*, vol. 163, no. 1, pp. 125–171, 2000.

[6] O. Maffeïs and P. L. Guernic, "Distributed Implementation of SIGNAL: Scheduling & Graph Clustering," in *FTRTFT*, ser. Lecture Notes in Computer Science, H. Langmaack, W. P. de Roever, and J. Vytopil, Eds., vol. 863. Springer, 1994, pp. 547–566.

[7] P. Aubry, P. L. Guernic, and S. Machard, "Synchronous Distribution of Signal Programs," in *HICSS (1)*, 1996, pp. 656–665.

[8] M. Pouzet and P. Raymond, "Modular static scheduling of synchronous data-flow networks: an efficient symbolic representation," in *EMSOFT*, S. Chakraborty and N. Halbwachs, Eds. ACM, 2009, pp. 215–224.

[9] R. Lublinerman and S. Tripakis, "Modular code generation from triggered and timed block diagrams," in *IEEE Real-Time and Embedded Technology and Applications Symposium*. IEEE Computer Society, 2008, pp. 147–158.

[10] G. Berry, "The foundations of Esterel," in *Proof, Language, and Interaction*, G. D. Plotkin, C. Stirling, and M. Tofte, Eds. The MIT Press, 2000, pp. 425–454.

[11] M. Abadi and L. Cardelli, *A Theory of Objects*. New York: Springer Verlag, 1996.

[12] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. marc Loingtier, and J. Irwin, "Aspect-oriented programming," in *ECOOP*. SpringerVerlag, 1997.

[13] I. Kiselev, *Aspect-Oriented Programming with AspectJ*. Indianapolis, IN, USA: Sams, 2002.

[14] L. de Alfaro and T. A. Henzinger, "Interface automata," in *ESEC / SIGSOFT FSE*, 2001, pp. 109–120.

[15] ——, "Interface theories for component-based design," in *EMSOFT*, ser. Lecture Notes in Computer Science, T. A. Henzinger and C. M. Kirsch, Eds., vol. 2211. Springer, 2001, pp. 148–165.

[16] N. A. Lynch and E. W. Stark, "A proof of the kahn principle for input/output automata," *Inf. Comput.*, vol. 82, no. 1, pp. 81–92, 1989.

[17] J.-B. Raclet, "Residual for component specifications," *Electr. Notes Theor. Comput. Sci.*, vol. 215, pp. 93–110, 2008.

[18] J.-B. Raclet, E. Badouel, A. Benveniste, B. Caillaud, A. Legay, and R. Passerone, "A modal interface theory for component-based design," *Fundam. Inform.*, vol. 108, no. 1-2, pp. 119–149, 2011.

[19] P. Ramadge and W. Wonham, "The control of discrete event systems," *Proceedings of the IEEE; Special issue on Dynamics of Discrete Event Systems*, vol. 27, no. 1, pp. 81–98, 1989.

[20] A. Arnold, X. Briand, and I. Walukiewicz, "Synthesis of decentralized controllers : decidable and undecidable cases," in *ATPN - Workshop on DES control*, 2003, 24th Int. conf. on Application Theory of Petri Nets (ATPN 2003), Eindhoven.