Fundamenta Informaticae XX (2004) 1–31
IOS Press

Formal refinement-checking in a system-level design methodology

Jean-Pierre Talpin
INRIA-IRISA

Paul Le Guernic
INRIA-IRISA

Sandeep Kumar Shukla
Virginia Tech

Frédéric Doucet
University of California at San Diego

Rajesh Gupta
University of California at San Diego

Abstract. Rising complexity and performances, shortening time-to-market demands, stress high-level embedded system design as a prominent research topic. Ad-hoc design methodologies, that lifts modeling to higher levels of abstraction, the concept of intellectual property, that promotes reuse of existing components, are essential steps to manage design complexity, gain performance, accelerate design cycle. However, the issue of compositional correctness arises with these steps.

Given components from different manufacturers, designed at different levels of abstraction and with heterogeneous models of computation, combining them in a correct-by-construction manner is a difficult challenge. A gating factor for widespread adoption of the system-level design paradigm is a lack of formal models, method and tools to support refinement. In the absence of adequate behavioral synthesis techniques, the refinement of a system-level description toward its implementation is primarily a manual process. Furthermore, proving that the implementation preserves the properties of the higher system-level design-abstraction is an unsolved problem.

We address these issues and define a formal refinement-checking methodology for system-level design by considering the polychronous model of computation of the multi-clocked synchronous formalism signal. We demonstrate the effectiveness of our approach by the experimental case study of a SPECC programming example. It highlights the benefits of the formal models, methods and tools provided in the Polychrony workbench, in capturing the functional, architectural, communication and implementation abstractions of a SPECC design, in an automated manner, and in comparing and validating behavioral equivalence between the successive refinements.

It relies on an automated modeling technique that is conceptually minimal and supports a scalable notion and a flexible degree of abstraction. Our presentation targets SPECC, yet with a generic and language-independent method. Applications of our technique range from the detection of local design errors to the compositional design refinement and conformance checking.

Address for correspondence: INRIA-IRISA, Campus de Beaulieu, 35042 Rennes, France (`Jean-Pierre.Talpin@irisa.fr`)

1. Introduction

Rising complexity and performances, shortening time-to-market demands, stress high-level embedded system design as a prominent research topic Ad-hoc design methodologies, that lifts modeling to higher levels of abstraction, the concept of intellectual property, that promotes reuse of existing components, are essential steps to manage design complexity, gain performance, accelerate design cycle. However, the issue of compositional correctness arises with these steps. Given components from different manufacturers, designed at different levels of abstraction and with heterogeneous models of computation, combining them in a correct-by-construction manner is a difficult challenge.

A gating factor for widespread adoption of the system-level design paradigm is a lack of formal models, method and tools to support refinement. In the absence of adequate behavioral synthesis techniques, the refinement of a system-level description toward its implementation is primarily a manual process. Furthermore, proving that the implementation preserves the properties of the higher system-level design abstraction is an unsolved problem.

In this aim, system design based on the so-called "synchronous hypothesis" [5] consists of abstracting the non-functional implementation details of a system away and let one benefit from a focused reasoning on the logics behind the instants at which the system functionalities should be secured. From this point of view, synchronous design models and languages provide intuitive models for integrated circuits. This affinity explains the ease of generating synchronous circuits and verify their functionalities using compilers and related tools that implement this approach.

In the relational model of the POLYCHRONY workbench [28], this affinity goes beyond the domain of purely synchronous circuits to embrace the context of globally asynchronous locally synchronous (GALS) architectures. The unique features of this model are to provide a scalable capability to describe partially clocked specifications or multi-clocked architectures and to support a formal notion of design *refinement*, from the early stages of requirements specification, to the later stages of deployment and synthesis, using formal verification.

We address the issues of conformance checking system design by considering the polychronous model of computation of the POLYCHRONY workbench to define a formal refinement-checking methodology. Our approach builds upon previous work on the multi-clocked synchronous paradigm of SIGNAL [7] and verification using the related model-checking tool SIGALI [24] (the POLYCHRONY workbench). It consists of putting the polychronous model of computation [22] to work in the context of emerging high-level design languages such as SPECC [15] and study refinement relations between system design levels in SPECC.

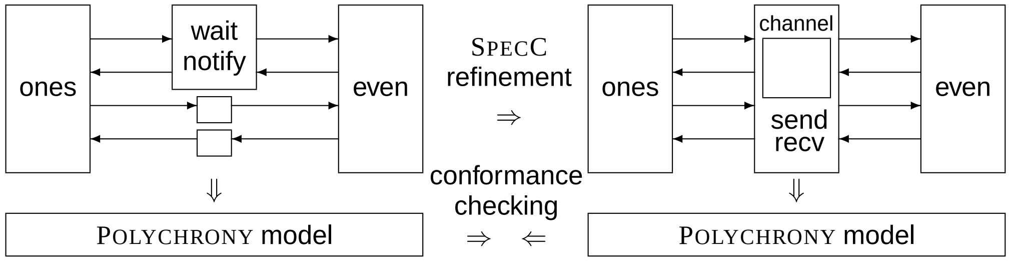


Figure 1. Checking conformance of a design refinement

We demonstrate the effectiveness of our approach by the experimental case study of a simple SPECC programming example. By giving a systematic model of the functional, architectural, communication and implementation abstractions of a SPECC programs in the multi-clocked synchronous formalism SIGNAL, this exercise highlights the benefits of the formal methods provided in the POLYCHRONY workbench in comparing and validating behavioral equivalence between the successive refinements or model transformations. Our presentation targets SPECC, yet with a generic, automated and language-independent method that is conceptually minimal and supports a scalable notion and a flexible degree of abstraction.

Outline

We start with an overview of the polychronous model of computation, Section 2, followed by an informal introduction to the SIGNAL data-flow notation of the POLYCHRONY workbench, Section 3. Starting from this presentation, Section 4 develops a methodology aimed at checking design refinement relations correct within the POLYCHRONY workbench by introducing a suitable notion of flow-preservation. This model and methodology are put to work, Section 5 in the context of system design using SPECC and a simple design example, the even-parity checker.

We outline a technique to automatically derive a model of SPECC specifications in the POLYCHRONY workbench and apply our methdology to checking the refinement of the EPC correct from its specification-level design to its RTL-level design. This exercise demonstrates the capability of POLYCHRONY to back high-level design transformations operated on SPECC programs with the validation services offered by the POLYCHRONY workbench.

2. A polychronous model of computation

We start with a brief overview of the polychronous model of computation, proposed in [22]. The polychronous model of computation consists of a *domain* of traces and of semi-lattice structures that render synchrony and asynchrony using timing equivalence relations: clock equivalence relates traces in the synchronous structure and flow equivalence relates traces in the asynchronous structure.

Domain of polychrony We consider a partially-ordered set $(\mathcal{T}, \leq, 0)$ of tags. A tag $t \in \mathcal{T}$ denotes a symbolic period in time. The relation $\leq$ denotes a partial order. Its minimum is noted 0. We note $C \in \mathcal{C}$ a (possibly infinite) *chain* of tags. Events, signals, behaviors and processes are defined as follows:

Definition 2.1. (polychrony)
- An *event* $e \in \mathcal{E} = \mathcal{T} \times \mathcal{V}$ is the pair of a value and a tag.
- A *signal* $s \in \mathcal{S} = C \to \mathcal{V}$ is a function from a *chain* of tags to a set of values.
- A *behavior* $b \in \mathcal{B}$ is a function from names $x \in \mathcal{X}$ to signals $s \in \mathcal{S}$.
- A *process* $p \in \mathcal{P}$ is a set of behaviors that have the same domain.

Figure 2 depicts a behavior b over three signals named x, y and z in the domain of polychrony. Two frames depict timing domains formalized by chains of tags. Signal x and y belong to the same timing domain: x is a down-sampling of y. Its events are synchronous to odd occurrences of events along y and share the same tags, e.g. t_1. Even tags of y, e.g. t_2, are ordered along its chain, e.g. $t_1 < t_2$, but absent from x. Signal z belongs to a different timing domain. Its tags, e.g. t_3 are not ordered with respect to the chain of y, e.g. $t_1 \not\leq t_3$ and $t_3 \not\leq t_1$.

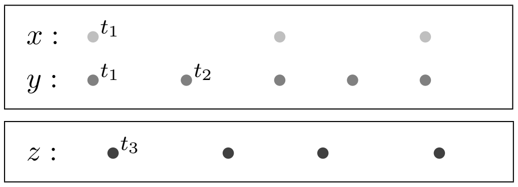


Figure 2. A behavior in the polychronous model of computation

In the remainder, we write $\mathrm{tags}(s)$ and $\mathrm{tags}(b) = \cup_{x \in \mathrm{vars}(b)} \mathrm{tags}\ (b(x))$ for the tags of a signal s and of a behavior b, $b|_X$ for the projection of a behavior b on $X \subset \mathcal{X}$ and $b/X = b|_{\mathrm{vars}(b) \setminus X}$ for its complementary, $\mathrm{vars}(b)$ and $\mathrm{vars}(p)$ for the domains of b and p. Synchronous composition is noted $p \,|\, q$ and defined by the union of all behaviors b (from p) and c (from q) which are synchronous: all signals they share, i.e. in $I = \mathrm{vars}(p) \cap \mathrm{vars}(q)$, are equal.

$$p \,|\, q = \{ b \cup c \mid (b, c) \in p \times q, I = \mathrm{vars}(p) \cap \mathrm{vars}(q), b|_I = c|_I \}$$

Figure 3 depicts the synchronous composition, right, of the behaviors b, left, and the behavior c, middle, of two processes p and q. Notice that the signal y, shared by p and q, must carry the same tags and the same values in both p and q in order for $b \cup c$, right, to belong to $p \,|\, q$.

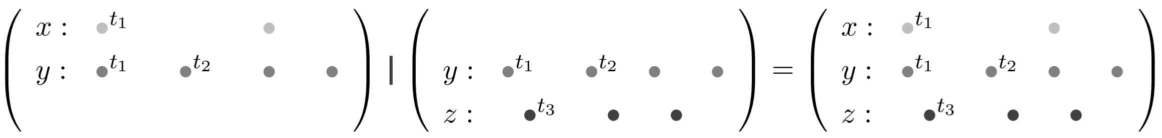


Figure 3. Synchronous composition of $b \in p$ and $c \in q$

Scheduling structure To render constraints between the occurrence of events during a period t, we refine the domain of polychrony with a scheduling relation. Figure 4 depicts a scheduling relation superimposed to the signals x and y of Figure 2. The relation $y_{t_1} \rightarrow x_{t_1}$ denotes a scheduling constraint: y should be calculated before x at the period t_1.

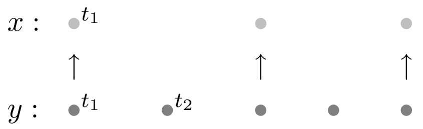


Figure 4. Scheduling relations between simultaneous events

The pair x_t of a time tag t and of a signal name x renders the date d of the event occurring at the symbolic period t along the signal x. The tag t represents the period during which multiple events take place to form a reaction. It corresponds to an equivalence class between dates d, as in the synchronous structures [27].

Definition 2.2. (scheduling relation)
The scheduling relation $\rightarrow^b$ is a pre-order defined on dates $\mathcal{D} = \mathcal{X} \times \mathcal{T}$ for a behavior b which satisfies:

$$\forall b \in \mathcal{B}, \forall x \in \mathrm{vars}(b), \forall t, t' \in \mathrm{tags}(b(x)),\ t < t' \Rightarrow x_t \rightarrow^b x_{t'} \wedge x_t \rightarrow^b x_{t'} \Rightarrow \neg(t' < t)$$

When no ambiguity is possible on the identity of b in $x \rightarrow^b y$, we write it $x \rightarrow y$. A scheduling relation is implicitly transitive ($x_t \rightarrow^b y_{t'} \rightarrow^b z_{t''}$ implies $x_t \rightarrow^b z_{t''}$) and its closure for restriction b/X is defined by $x_t \rightarrow^{b/X} y_{t'}$ iff $x_t \rightarrow^b y_{t'}$ and $x, y \notin X$.

Synchronous structure Building upon the domain of polychrony, we define the semi-lattice structure which relationally denotes synchronous behaviors. The intuition behind this relation is depicted figure 5. It is to consider a signal as an elastic with ordered marks on it (tags). If the elastic is stretched, marks remain in the same relative and partial order but have more space (time) between each other. The same holds for a set of elastics: a behavior. If elastics are equally stretched, the order between marks is unchanged. In the figure 5, the time scale of x and y change but the partial timing and scheduling relations are preserved. Stretching is a partial-order relation which defines clock equivalence (definition 2.3).

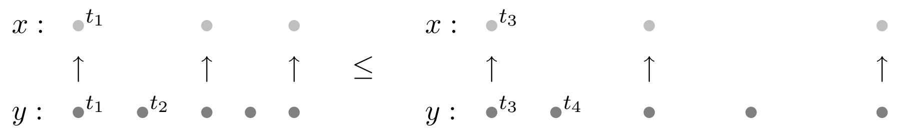


Figure 5. Relating synchronous behaviors by stretching.

Definition 2.3. (clock equivalence)
A behavior c is a *stretching* of b, written $b \leq c$, iff $\mathrm{vars}(b) = \mathrm{vars}(c)$ and there exists a bijection on tags f which satisfies

$$\left|\begin{array}{l} \forall t, t' \in \mathrm{tags}(b), t \leq f(t) \wedge (t < t' \Leftrightarrow f(t) < f(t')) \\ \forall x, y \in \mathrm{vars}(b), \forall t \in \mathrm{tags}(b(x)), \forall t' \in \mathrm{tags}(b(y)), t_x \rightarrow^b t'_y \Leftrightarrow f(t)_x \rightarrow^c f(t')_y \\ \forall x \in \mathrm{vars}(b), \mathrm{tags}(c(x)) = f(\mathrm{tags}(b(x))) \wedge \forall t \in \mathrm{tags}(b(x)), b(x)(t) = c(x)(f(t)) \end{array}\right.$$

b and c are *clock-equivalent*, written $b \sim c$, iff there exists a behavior d s.t. $d \leq b$ and $d \leq c$.

Asynchronous structure The asynchronous structure of polychrony is modeled by weakening the clock-equivalence relation to allow for comparing behaviors w.r.t. the sequences of values signals hold regardless of the time at which they hold these values. The *relaxation* relation allows to individually stretch the signals of a behavior in a way preserving scheduling constraints. Relaxation is a partial-order relation which defines flow-equivalence (definition 2.4). Two behaviors are flow-equivalent iff their signals hold the same values in the same order.

Definition 2.4. (flow equivalence)
A behavior c is a *relaxation* of b, written $b \sqsubseteq c$, iff $\mathrm{vars}(b) = \mathrm{vars}(c)$ and, for all $x \in \mathrm{vars}(b)$, $b|_{\{x\}} \leq c|_{\{x\}}$. b and c are *flow-equivalent*, written $b \approx c$, iff there exists a behavior d s.t. $d \sqsubseteq b$ and $d \sqsubseteq c$.

Figure 6 depicts two asynchronously equivalent behaviors related by relaxation. The first event along x has been shifted as the effect of delaying its transmission using e.g. a FIFO buffer.

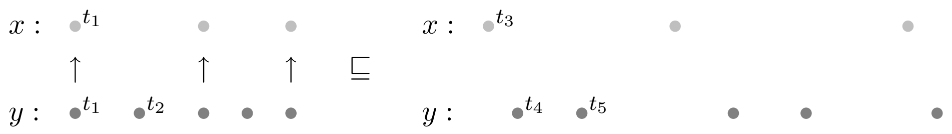


Figure 6. Relating asynchronous behaviors by relaxation.

Asynchronous composition is noted $p \parallel q$ and defined by considering the partial-order structure induced by the relaxation relation. The parallel composition of p and q consists of behaviors d that are relaxations of behaviors b and c from p and q along shared signals $I = \mathrm{vars}(p) \cap \mathrm{vars}(q)$ and that are stretching of b and c along the independent signals of p and q.

$$p \parallel q = \left\{ d \in \mathcal{B}|_{\mathrm{vars}(p) \cup \mathrm{vars}(q)} \mid \exists (b, c) \in p \times q, d/I \geq b/I \wedge d/I \geq c/I \wedge b|_I \sqsubseteq d|_I \wedge d|_I \sqsupseteq c|_I \right\}$$

Figure 7 depicts the asynchronous composition, right, of the behavior $b \in p$, left, and of the behavior $c \in q$, middle. Notice that the signal x and y are alternated in p, left, and synchronous in q, middle. Asynchronous composition allows for these signals to be independently stretched in both p and q in order to find a common flow in the asynchronously composed process, right.

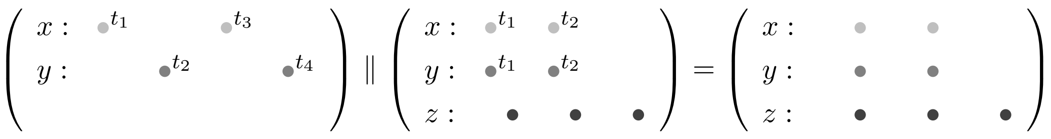


Figure 7. Asynchronous composition of $b \in p$ and $c \in q$

GALS structure An architecture consists of a globally asynchronous structure (a network) and of locally synchronous structures (processes). Among this structure, Definition 2.5 characterizes the class of processes and networks that are insensitive to internal and external delays. A process is said endochronous iff, given a set $I \subset \mathrm{vars}(p)$ of external input signals, it has the capability to reconstruct a unique synchronous behavior (up to clock-equivalence). A network is said endo-isochronous iff the interaction between processes in this network is endochronous (i.e. $(p|_I) | (q|_I)$ is endochronous for $I = \mathrm{vars}(p) \cap \mathrm{vars}(q)$ the signals shared by p and q).

Definition 2.5. (endo-isochrony)
p is endochronous iff there exists $I \subset \mathrm{vars}(p)$ such that $\forall b, c \in p$, $(b|_I) \approx (c|_I) \Rightarrow b \sim c$. If p, q and $(p|_I) | (q|_I)$ are endochronous with $I = \mathrm{vars}(p) \cap \mathrm{vars}(q)$ then $p \parallel q$ is endo-isochronous

The behavior of an endochronous process p is depicted in Figure 8. The process accepts flow-equivalent inputs x and y (left). Inputs are processed by p in clock equivalent ways (middle) so as to produce the same outputs in the same order at clock-equivalent rates (right).

The property of *flow-invariance*, depicted Figure 9, proposed in [22], is inspired from endochrony and endo-isochrony. It compositionally ensures that the refinement of an abstract specification p (e.g. a polychronous specification $p \,|\, q$) by its concrete implementation q (e.g. a distributed implementation $p \parallel q$) preserves flows.

Definition 2.6. (flow-invariance)
The refinement of $p \,|\, q$ by $p \parallel q$ is flow-invariant iff, for all $b \in p \,|\, q$, for all $c \in p \parallel q$, $(b|_I) \approx (c|_I)$ implies $b \approx c$ for I the input signals of $p \,|\, q$.

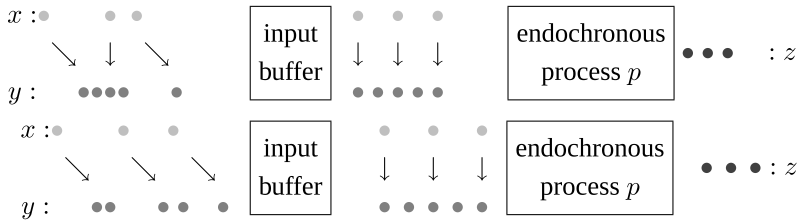


Figure 8. Endochrony: from flow-equivalent inputs to clock-equivalent outputs

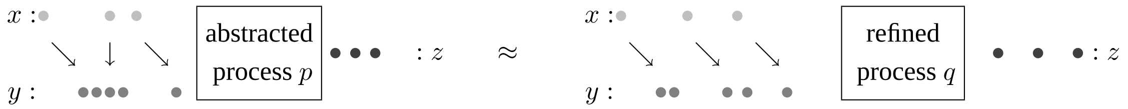


Figure 9. Flow-invariance: preservation of flow-equivalence during a design refinement

3. A polychronous design language

In the POLYCHRONY workbench, the polychronous model of computation is implemented by the multi-clocked synchronous data-flow notation SIGNAL [7]. It will serve as the specification formalism used for the case study of the present article.

Core syntax and semantics In SIGNAL, a process P consists of the composition of simultaneous equations $x := f(y, z)$ over signals x, y, z. A signal $x \in \mathcal{X}$ is a possibly infinite flow values $v \in \mathcal{V}$ sampled at a discrete clock noted $\hat{}x$.

$$P, Q ::= x := y \, f \, z \mid P/x \mid P \,|\, Q \qquad \text{(SIGNAL process)}$$

In the polychronous model of computation, Section 2, the denotation of a clock $\hat{}x$ is the domain of the signal associated to x: a chain of tags. We note $[\![P]\!]$ for the denotation of a process P. The synchronous composition of processes $P \,|\, Q$ consists of the simultaneous solution of the equations in P and in Q. The process P/x restricts the signal x to the lexical scope of P.

$$[\![P \,|\, Q]\!] = [\![P]\!] \,|\, [\![Q]\!] \text{ and } [\![P/x]\!] = [\![P]\!]/x = \{c \leq b/\{x\} \mid b \in [\![P]\!]\}$$

An equation $x := y\,f\,z$ denotes a relation between the input signals y and z and an output signal x by a combinator f. An equation is usually a ternary and infixed relation noted $x := y\,f\,z$ but it can in general be an $m+n$-ary relation noted $(x_1, \ldots x_m) := f(y_1, \ldots y_n)$.

Since signals denote possibly infinite flow of values an equation can be regarded as an invariant: a relation between its input and output signals that always holds. SIGNAL requires four primitive combinators to perform delay $x := y\$1\,\texttt{init}\,v$), sampling $x := y\,\texttt{when}\,z$, merge $x = y\,\texttt{default}\,z$ and specify scheduling constraints $x \to y\,\texttt{when}\,\hat{}z$.

The equation $x := y\$1\,\texttt{init}\,v$, also noted $x := \texttt{pre}\,v\,y$, initially defines the signal x by the value v and then by the previous value of the signal y. We write $\mathcal{B}|_X$ for the set of all behaviors defined on the set of variables X.

$$[\![x := y\$1\,\texttt{init}\,v]\!] = \left\{ b \in \mathcal{B}|_{x,y} \;\middle|\; \begin{array}{l} \text{tags}(b(x)) = \text{tags}(b(y)) = C \in \mathcal{C}, b(x)(\min(C)) = v \\ \forall t \in C \setminus \min(C),\, b(x)(t) = b(y)(\text{pred}_C(t)) \end{array} \right\}$$

We observe, Figure 10, that a signal y and its delayed copy $x := y\$1\,\texttt{init}\,v$ are synchronous: they share the same set of tags $t_1, t_2, \ldots$ Initially, at time t_1, the delayed signal x takes the declared value x and then, at tag t_n, the value of y at tag t_{n-1}, for all $n > 1$.

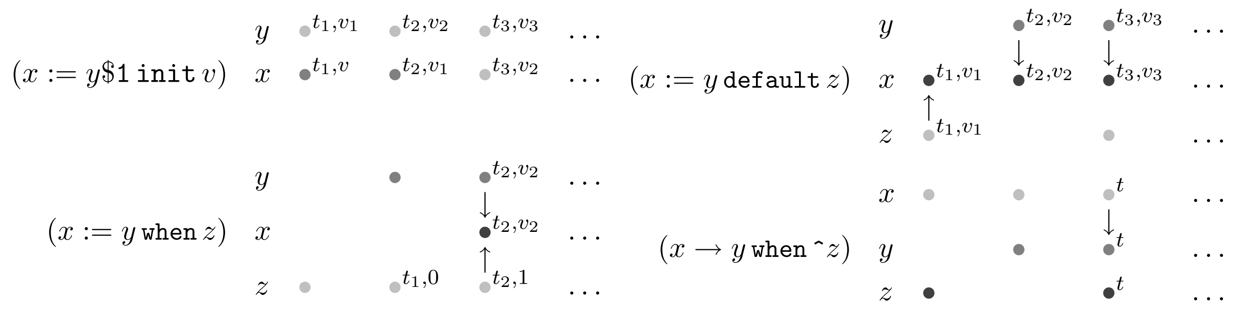

$(x := y\$1\,\texttt{init}\,v)$
y t_1,v_1 t_2,v_2 t_3,v_3 …
x t_1,v t_2,v_1 t_3,v_2 …
$(x := y\,\texttt{default}\,z)$
y t_2,v_2 t_3,v_3 …
x t_1,v_1 t_2,v_2 t_3,v_3 …
z t_1,v_1 …
$(x := y\,\texttt{when}\,z)$
y t_2,v_2 …
x t_2,v_2 …
z $t_1,0$ $t_2,1$ …
$(x \to y\,\texttt{when}\,\hat{}z)$
x t …
y t …
z t …

Figure 10. Behaviors of combinators

The equation $x \to y\,\texttt{when}\,\hat{}z$ forces x to occur before y when z is present. In Figure 10, for instance, there is no scheduling relation required from x to y unless both x, y, z are present, e.g. at tag t, or unless otherwise specified by another constraint.

$$[\![x \to y\,\texttt{when}\,\hat{}z]\!] = \{b \in \mathcal{B}|_{x,y,z} \mid \forall t \in \text{tags}(b(x)) \cap \text{tags}(b(y)) \cap \text{tags}(b(z)),\, x_t \to y_t\}$$

The equation $x := y\,\texttt{when}\,z$ defines x by y when z is true (and both y and z are present). Figure 10, x is present with the value v_2 at t_2 only if y is present with v_2 at t_2 and if z is present at t_2 with the value true. When this is the case, one needs to schedule the calculation of y and z before x, as depicted by $y_{t_2} \to x_{t_2} \leftarrow z_{t_2}$.

$$[\![x := y\,\texttt{when}\,z]\!] = \left\{ b \in \mathcal{B}|_{x,y,z} \;\middle|\; \begin{array}{l} \text{tags}(b(x)) = \{t \in \text{tags}(b(y)) \cap \text{tags}(b(z)) \mid b(z)(t) = \texttt{true}\,\} \\ \forall t \in \text{tags}(b(x)),\, b(x)(t) = b(y)(t) \wedge y_t \to x_t \wedge z_t \to x_t \end{array} \right\}$$

The equation $x := y\,\texttt{default}\,z$ defines x by y when y is present and by z otherwise. Three cases are depicted Figure 10. If y is absent and z present with v_1 at t_1 then x holds (t_1, v_1). If y is present (at t_2 or t_3) then x holds its value whether z is present (at t_2) or not (at t_3).

$$[\![x := y\,\texttt{default}\,z]\!] = \left\{ b \in \mathcal{B}|_{x,y,z} \left| \begin{array}{l} \text{tags}(b(y)) \cup \text{tags}(b(z)) = \text{tags}(b(x)) \in \mathcal{C} \\ \forall t \in \text{tags}(b(y)),\ b(x)(t) = b(y)(t) \wedge y_t \rightarrow x_t \\ \forall t \in \text{tags}(b(x)) \setminus \text{tags}(b(y)),\ b(x)(t) = b(z)(t) \wedge z_t \rightarrow x_t \end{array} \right. \right\}$$

Syntax and semantics of clocks The syntax of clock expressions e and clock relations E is a particular subset of SIGNAL that is defined by the induction grammar of Figure 11.

In SIGNAL, the presence of a value along a signal x is the proposition noted $\hat{}x$ that is true when x is present and that is absent otherwise. The clock expression $\hat{}x$ can be defined by the boolean operation $x = x$ (i.e. $y := \hat{}x =^{\text{def}} y := (x = x)$). Referring to the polychronous model of computation, it represents the set of tags at which the signal holds a value. Clock expression naturally represent control, the clock $\texttt{when}\,x$ represents the time tags at which the boolean signal x is present and true (i.e. $y := \texttt{when}\,x =^{\text{def}} y := \texttt{true when}\,x$). The clock $\texttt{when not}\,x$ represents the time tags at which the boolean signal x is present and false. We write 0 for the empty clock (the empty set of tags).

Clocks expressions e form a semi-lattice of union operator $e \,\hat{}\texttt{+}\, e'$ (i.e. $e\,\texttt{default}\,e'$) intersection $e \,\hat{}\texttt{*}\, e'$ (i.e. $e\,\texttt{when}\,e'$) and difference $e \,\hat{}\texttt{-}\, e'$ (i.e. $\texttt{when false when}\,e'\,\texttt{default true when}\,e$). A clock constraint E is regarded as a SIGNAL process. The constraint $e \hat{=} e'$ synchronizes the clocks e and e'. It corresponds to the process $(x := (e = e'))/x$. Composition $E \,|\, E'$ corresponds to the union of constraints and restriction E/x to the existential quantification of E by x. Finally, notice that scheduling constraints are transitive and distributive w.r.t. clocks: $x \rightarrow y\,\texttt{when}\,e \,|\, y \rightarrow z\,\texttt{when}\,e'$ implies $x \rightarrow z\,\texttt{when}\,e \,\hat{}\texttt{*}\, e'$ and $x \rightarrow y\,\texttt{when}\,e \,|\, x \rightarrow y\,\texttt{when}\,e'$ implies $x \rightarrow y\,\texttt{when}\,e \,\hat{}\texttt{+}\, e'$. We usually write $x \rightarrow y$ for $x \rightarrow y\,\texttt{when}\,\hat{}x$.

$$\begin{array}{lll} e & ::= & \hat{}x \mid \texttt{when}\,x \mid \texttt{when not}\,x \mid e \,\hat{}\texttt{+}\, e' \mid e \,\hat{}\texttt{-}\, e' \mid e \,\hat{}\texttt{*}\, e' \mid 0 \quad \text{(clock expressions)} \\ E & ::= & () \mid e \hat{=} e' \mid e \hat{<} e' \mid x \rightarrow y\,\texttt{when}\,e \mid E \,|\, E' \mid E/x \quad \text{(clock constraint)} \end{array}$$

Figure 11. Clock and scheduling constraints

Since clock constraints are expressible in the core syntax, they have a denotation noted $[\![E]\!]$. It is formally defined in [22]. Moreover, each process P corresponds to a clock constraint E satisfying $[\![P]\!] \subseteq [\![E]\!]$ by the inference system $P : E$ of figure 12.

$$\begin{array}{c} x := y\$1\,\texttt{init}\,v : \hat{}x \hat{=} \hat{}y \\ x := y\,\texttt{when}\,z : \hat{}x \hat{=} \hat{}y\texttt{when}\,z \,|\, y \rightarrow x\,\texttt{when}\,z \\ x := y\,\texttt{default}\,z : \hat{}x \hat{=} \hat{}y \,\hat{}\texttt{+}\, \hat{}z \,|\, y \rightarrow x \,|\, z \rightarrow x\,\texttt{when}\,(\hat{}z \,\hat{}\texttt{-}\, \hat{}y) \end{array} \qquad \frac{P : E \quad Q : E'}{P \,|\, Q : E \,|\, E'} \qquad \frac{P : E}{P/x : E/x}$$

Figure 12. Inference system

Hierarchization The clock and scheduling constraints E of a process P hold the necessary information to decide the property of endochrony [22]. Clock constraints determine the order $\preceq$ in which events are processed, definition 3.1. Rule 1 defines equivalence classes for signals of equivalent clocks. Rule 2 constructs elementary partial orders relations: the clock $\texttt{when}\,x$ is smaller than $\hat{}x$. Rule 3 defines the insertion of a partial order of maximum e_3 under a clock $e \succeq e_3$. The insertion algorithm, specified in [3], yields a canonical representation of the corresponding partial order by observing that there exists a unique minimum clock e' below e such that 3 holds. We write $E \Rightarrow E'$ iff E' is a proposition that is deductible from E in the semi-lattice of clocks.

E is *hierarchic* iff its clock relation $\preceq$ has a minimum, written $\min_{\preceq} E \in \mathrm{vars}(E)$, so that $\forall x \in \mathrm{vars}(E), \exists y \in \mathrm{vars}(E), y \preceq x$. H is *acyclic* iff $E \Rightarrow x \to x\,\texttt{when}\,e$ implies $E \Rightarrow e\,\hat{=}\,0$ (for all $x \in \mathrm{vars}(E)$). In [22], we show that if $P : E$ and if E is acyclic and hierarchic, then P is endochronous.

Definition 3.1. The partial order $\preceq$ of E is the largest relation satisfying

1. if $E \Rightarrow \hat{}x\,\hat{=}\,\hat{}y$ then $x \preceq y$.
2. if $E \Rightarrow \hat{}x\,\hat{=}\,\texttt{when}\,y$ or $E \Rightarrow \hat{}x\,\hat{=}\,\texttt{when}\,\texttt{not}\,y$ then $y \preceq x$.
3. if $y \preceq x \succeq w$ and $H \Rightarrow \hat{}z = \hat{}y f \hat{}w$ for any $f \in \{\,\hat{}\texttt{+}\,,\ \hat{}\texttt{*}\,,\ \hat{}\texttt{-}\,\}$ then $x \preceq z$.

x and y are equivalent, written $x \mathrel{-\!\!\diamond\!\!-} y$, iff $x \preceq y$ and $y \preceq x$.

Example The implications of definition 3.1 can be outlined by considering a simple SIGNAL program, Figure 13, left. Process `buffer` implements two functionalities. One is the process `current`. It defines a `cell` in which values are stored at the input clock `^i` and loaded at the output clock `^o`. `cell` is a predefined SIGNAL operation defined by:

$$x := y\;\texttt{cell}\;z\;\texttt{init}\;v \overset{\text{def}}{=} (m := x\$1\;\texttt{init}\;v \,|\, x := y\;\texttt{default}\;m \,|\, \hat{}x\,\hat{=}\,\hat{}y\;\hat{}\texttt{+}\;\hat{}z)\,/m$$

The other functionality is the process `alternate` which desynchronizes the signals `i` and `o` by synchronizing them to the true and false values of an alternating boolean signal `b`.

```
process buffer = (? i ! o)
  (| alternate (i, o)
   | o := current (i)
   |) where
process alternate = (? i, o ! )
  (| zb := b$1 init true
   | b := not zb
   | o ^= when not b
   | i ^= when b
   |) / b, zb;
process current = (? i ! o)
  (| zo := i cell ^o init false
   | o  := zo when ^o
   |) / zo;
```

```
(| c_b ^= b
 | b   ^= zb
 | zb  ^= zo
 | c_i := when b
 | c_i ^= i
 | c_o := when not b
 | c_o ^= o
 | i -> zo when ^i
 | zb -> b
 | zo -> o when ^o
 |) / zb, zo, c_b,
       c_o, c_i, b;
```

```
buffer_iterate () {
  b = !zb;
  c_o = !b;
  if (b) {
    if (!r_buffer_i(&i))
      return FALSE;
  }
  if (c_o) {
    o = i;
    w_buffer_o(o);
  }
  zb = b;
  return TRUE;
}
```

Figure 13. Specification, clock analysis and code generation

Clock inference (Figure 13, middle) applies the clock inference system of Figure 12 to the process `buffer` to determine three synchronization classes. We observe that `b, c_b, zb, zo` are synchronous and define the master clock `buffer`. Recalling Definition 3.1, we write

$$\texttt{b} \mathrel{-\!\!\diamond\!\!-} \texttt{c\_b} \mathrel{-\!\!\diamond\!\!-} \texttt{zb} \mathrel{-\!\!\diamond\!\!-} \texttt{zo}$$

There are two other synchronization classes, `c_i` and `c_o`, that corresponds to the true and false values of the boolean flip-flop variable `b`, respectively. Hence, we write

$$\texttt{b} \preceq \texttt{c\_i} \mathrel{-\!\!\diamond\!\!-} \texttt{i} \text{ and } \texttt{b} \preceq \texttt{c\_o} \mathrel{-\!\!\diamond\!\!-} \texttt{o}$$

This defines three nodes in the control-flow graph of the code to be generated Figure 13, right. At the main clock `c_b`, `b` and `c_o` are calculated from `zb`. At the sub-clock `b`, the input signal `i` is read. At the sub-clock `c_o` the output signal `o` is written. Finally, `zb` is determined. Notice that the sequence of instructions follows the scheduling constraints determined during clock inference.

Some more concrete syntax In addition to the core syntax of SIGNAL presented so far, we make extensive use of process declarations and partial equations for the purpose of modeling our case study. In SIGNAL, a partial equation $x ::= y\, f\, z\, \texttt{when}\, e$ is the partial definition of the variable x by the operation $y\, f\, z$ at the clock denoted by the expression e. The default equation $x\ ::=\ \texttt{defaultvalue}\, v$ defines the value of the variable x when it is present but no corresponding partial equation $x\ ::=\ y\, f\, z\, \texttt{when}\, e$ applies (because e is absent). Let x be a variable defined using n partial equations and a default value v:

$$x ::= x_1\, \texttt{when}\, e_1 \,\|\, \ldots \,|x ::= x_n\, \texttt{when}\, e_n \,|x ::=\ \texttt{defaultvalue}\, v$$

Once parsed, the SIGNAL compiler processes this definition by first checking the clock expressions $e_1, \ldots e_n$ mutually exclusive and then handling the definition as the equivalent equation:

$$x := (x_1\, \texttt{when}\, e_1)\, \texttt{default}\, \ldots\, \texttt{default}\, (x_n\, \texttt{when}\, e_n)\, \texttt{default}\, v$$

In SIGNAL, the declaration of a process P of name f, input signals $x_1, \ldots x_m$, output signals $x_{m+1}, \ldots x_n$ is noted

$$\texttt{process}\, f = (?\, x_1, \ldots x_m\, !\, x_{m+1}, \ldots x_n)\, (|\, P\, |);$$

Once declared, process f may be called $(y_{m+1}, \ldots y_n)\ :=\ f(y_1, \ldots y_m)$ with its actual parameters $y_1, \ldots y_n$ and behave as $P[y_{1,\ldots n}/x_{1,\ldots n}]$. A variant declaration is that of a foreign function f, accessible, e.g. from a separately compiled C library. Its call can be wrapped into SIGNAL by declaring its interface and by declaring an abstraction E of its behavior (consists of scheduling and clock constraints).

$$\texttt{process}\, f = (?\, x_1, \ldots x_m\, !\, x)\, \texttt{spec}\, (|\, E\, |)\, \texttt{pragmas}\, \texttt{C\_CODE}\text{"}\&\texttt{x} = \texttt{f}(\&\texttt{x}_1, \ldots \&\texttt{x}_\texttt{m})\text{"}\ \texttt{end}\, \texttt{pragmas};$$

4. A refinement-checking methodology

The definition of the polychronous model of computation [22] accurately renders the synchronous hypothesis implemented in the multi-clocked data-flow notation SIGNAL and relates it to architectures using communication with unbounded delay. In an embedded architecture, however, the flow of a signal usually slides from another as the result of finite delays incurred by resource-bounded protocols, e.g. `fifo` buffers. In this section, we seek towards a formulation of the formal properties implied by this practice to check correctness of a concrete design refinement methodology.

Finite relaxation We start from the model of a one-place `fifo` buffer in SIGNAL, Figure 14, which we will use to draw the spectrum of possible timing relations considered, modelled and checked in the context of the present case study. The processing of process `fifo` is decomposed into two functionalities. One is the process `access` which defines the necessary timing constraints on the input signal `i` and output signal `o` via the delayed value of the boolean signal `b`: `fifo` can accepts an input at the next time sample iff `b` is true.

```
process access = (? i, o ! )
  (| a := ^o default (not ^i) default b
   | b := a$1 init false
   | i ^= when b
   | o ^= when not b
   |) / a, b
```

```
process current = (? i ! o)
  (| o:= (i cell ^o init false) when ^o
   |)
process fifo = (? i ! o)
  (| access(i, o) | o := current (i)
   |)
```

Figure 14. A one-place first-in first-out buffer in SIGNAL

The other functionality of `fifo` is the process `current` of Figure 13. Figure 15 depicts the relation of the signals x and y and the cell m defined by the equation $y := \texttt{fifo}(x)$.

$y := \texttt{fifo}(x)$

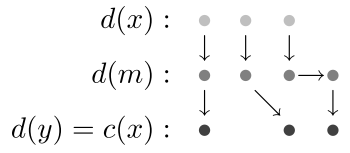


Figure 15. Relation between events through a one-place FIFO buffer

Definition 4.1 formalizes this relation and accounts for the behavior of `fifo` by implying a series of (reflexive-anti-symmetric) relations $\sqsubseteq_N$ (for $N > 0$) which yields the (series of) reflexive-symmetric flow relations $\approx_N$ to identify processes of same flows up to a flow-preserving first-in-first-out buffer of size N. In Definition 4.1, we write $\text{pred}_C(t)$ (resp. $\text{succ}_C(t)$) for the immediate predecessor (resp. successor) of the tag t in the chain C.

Definition 4.1. (finite relaxation)
The behavior c is a 1-relaxation of x in b, written $b \sqsubseteq_1^x c$ iff $\text{vars}(b) = \text{vars}(c)$ and there exists $d/m \geq b$ such that $d/x = c/x$ and a chain $C = \text{tags}(d(m)) = \text{tags}(d(x)) \cup \text{tags}(c(x))$ such that:

$$\begin{array}{rrcl} \forall t \in C, & t \in \text{tags}(d(x)) & \Rightarrow & d(m)(t) = d(x)(t) \\ & t \notin \text{tags}(d(x)) & \Rightarrow & d(m)(t) = d(m)(\text{pred}_C(t)) \\ & t \in \text{tags}(c(x)) & \Rightarrow & c(x)(t) = d(m)(t) \end{array}$$

and satisfying $\forall t \in \text{tags}(c(x)) \exists t' \in \{t, \text{succ}_C(t)\},\ c(x)(t') = d(x)(t)$. We write $b \sqsubseteq_1 c$ iff $b \sqsubseteq_1^x c$ for all $x \in \text{vars}(b)$, and, for all $n > 0$, $b \sqsubseteq_{n+1} c$ iff there exists d such that $b \sqsubseteq_1 d \sqsubseteq_n c$.

Desynchronization Now, recall the process `buffer` of Figure 13. It essentially differs from process `fifo` by the policy implemented by process `alternate`. Process `alternate` synchronizes `i` and `o` to the true and false values of an alternating signal `b`.

This guarantees the independence or exclusion between the clocks of `i` and `o`. Each tag t'_i of an event along `o` can only happen strictly between the tags t_i and t_{i+1} of two consecutive events along `i`, i.e. $t_i < t'_i < t_{i+1}$, and vice-versa.

```
process alternate = (? i, o ! )
  (| b := not (b$1 init true)
   | i ^= when b
   | o ^= when not b
   |) / b
```

```
process buffer = (? i ! o)
  (| alternate (i, o)
   | o := current (i)
   |)
```

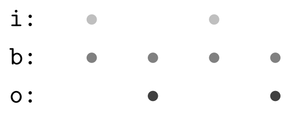


Figure 16. A desynchronization buffer in SIGNAL

Definition 4.2. (desynchronization)
The behavior c is a desynchronization of b, written $b \sqsubset c$ iff $\text{vars}(b) = \text{vars}(c)$ and there exists $d \geq b$ such that, for all $x \in \text{vars}(b)$, $d(x) = (t_i, v_i)_{i\geq 0}$, $c(x) = (t'_i, v_i)_{i\geq 0}$ and, for all $i \geq 0$, $t_i < t'_i < t_{i+1}$.

The relation between Definitions 4.1 and 4.2 and the implementation of `fifo` and `buffer` of Figure 14 and 16are brought together by the following proposition. The proof of proposition (1) consists formulating the four requirements of definition 4.1 as invariants or observers in SIGNAL and checking them against the model of `fifo`. Proposition (2) is proved by observing that all tags of `i` and `o` satisfy $t_i < t'_i$ by induction on i and that t_i and t'_i form two disjoint sub-chains of the domain of `b`.

Theorem 4.1.

$$\begin{array}{llllll} (1) & \forall b & \in & [\![x := \texttt{fifo}(y)]\!], & [z \mapsto b(y)] & \sqsubseteq_1^z & [z \mapsto b(x)] \\ (2) & \forall b & \in & [\![x := \texttt{buffer}(y)]\!], & [z \mapsto b(y)] & \sqsubset^z & [z \mapsto b(x)] \end{array}$$

Formal properties The series of relations $(\approx_n)_{n\geq 0}$ defines a spectrum between synchrony and asynchrony that can be modeled using the SIGNAL formalism. It is hence tempting to interpret the asynchronous partial-order $\sqsubseteq$ as the (inaccessible) limit or union $\cup_{N\geq 0} \sqsubseteq_N$ of this series.

Lemma 4.1.
- $b \sim b'$ implies $b \approx_1 b'$,
- $b \approx_n b'$ implies $b \approx b'$, for all $n > 0$
- $b \approx_m b' \approx_n b''$ implies $b \approx_{m+n} b''$, for all m and n

Instead, we focus on the largest equivalence relation that can be modeled using SIGNAL. It consists of behaviors equal up to a timing deformation performed by a finite FIFO protocol.

Definition 4.3. (finite flow-equivalence)
b and c are finitely flow-equivalent, written $b \approx^* c$, iff there exists $n > 0$ and d s.t. $d \sqsubseteq_n b$ and $d \sqsubseteq_n c$.

We say that a process P is finitely flow-preserving iff given finitely flow-equivalent inputs, it can only produce behaviors that are finitely flow equivalent. Example of finitely flow-preserving processes are endochronous processes. An endochronous process which receives finitely flow equivalent inputs produces clock-equivalent outputs.

Definition 4.4. (finite flow-preservation)
p is *finitely flow-preserving* with $I \subset \text{vars}(p)$ iff $\forall b, c \in p,\ (b|_I) \approx (c|_I) \Rightarrow b \approx^* c$.

A refinement-based design methodology based on the property of finite flow-preservation consists of characterizing sufficient invariants for a given model transformation to preserve flows.

Definition 4.5. (finite flow-invariance)
The refinement of p by q is *finitely flow-invariant*, written $p \ll^* q$, iff $I \subset \text{vars}(p) = \text{vars}(q)$ and $\forall (b, c) \in p \times q,\ (b|_I) \approx (c|_I) \Rightarrow b \approx^* c$.

The property of finite flow-invariance is a very general methodological criterion. It is reflexive ($p \ll^* p$) and transitive ($p \ll^* q \ll^* r \Rightarrow p \ll^* r$) for all flow-preserving processes (p, q, r). For instance, it can be applied to the characterization of correctness criteria for model transformations such as protocol insertion or desynchronization.

Verification methodology Theorem 4.1 provides all necessary elements to define an observer giving sufficient conditions for finite flow-preservation to hold and be provable by model checking. To this end, we consider the template SIGNAL process `observer` of Figure 17. It is parameterized by the notation `{P, Q}` over two processes named `P` and `Q` which we want to check finitely flow-equivalent.

The `observer` receives an input signal `i`. This input signal is used to generate two desynchronized signals (i.e. satisfying the hypothesis $b|_\mathtt{i} \approx c|_\mathtt{i}$) by using the process `buffer`. The flows $b|_\mathtt{i}$ and $c|_\mathtt{i}$ are injected to P and Q and the outputs collected by using `fifo` to avoid the synchronization of the outputs performed by the comparison `=`. If the output of the `observer` is always true then the equality is an invariant.

```
process observer = {P, Q} (? i ! o)
    (| o := fifo (P (buffer (i))) = fifo (Q (buffer (i)))
     |);
```

Figure 17. Observer function for the property of finite flow-equivalence

For the sake of simplicity, process`observer` is displayed Figure 17 for two processes `P` and `Q` that have only one input and one output signal and with a `fifo` buffer of lenght 1. Extending the `observer` to accept processes with m inputs, n outputs and a buffer of lenght k is obtained by structural induction starting from `fifo` and `buffer`.

Theorem 4.2 formalizes the implication of process`observer` for refinement checking by considering flow-preserving processes P and Q of same cardinality i.e. $\text{vars}(P) = \text{vars}(Q)$ and $\text{in}(P) = \text{in}(Q)$.

Theorem 4.2. (refinement checking)
Let P and Q be finitely flow-preserving processes of same cardinality $m = |\text{in}(P)| = |\text{in}(Q)|$. If, for all $b \in [\![x := \texttt{observer}\{P, Q\}(y_1, \ldots y_m)]\!]$ and, for all $t \in \text{tags}(b(x))$, $b(x)(t) = \texttt{true}$, then $P \ll^* Q$.

Let $p = [\![P]\!]$ and $q = [\![Q]\!]$ and $I = \text{vars}(p) = \text{vars}(q)$. By definition of process `observer` Figure 17 and by Theorem 4.1,

$$(1) : \forall b \in \mathcal{B}|_I,\ \forall (c, d) \in p \times q,\ b \sqsubset^I c|_I \wedge b \sqsubset^I d|_I \Rightarrow c \approx_1 d$$

Since p and q are both finitely flow-preserving, by definition 4.4,

$$(2) : \forall (c,d) \in p^2,\ c|_I \approx d|_I \Rightarrow c \approx^* d \qquad \forall (c,d) \in q^2,\ c|_I \approx d|_I \Rightarrow c \approx^* d$$

From (2), relaxing input delay in (1) yields flow-equivalent behaviors, hence

$$(3) : \forall (c,d) \in p \times q,\ c|_I \approx d|_I \Rightarrow c \approx_1 d$$

By lemma 4.1, Equations (1) and (2) yield the result expected in Theorem 4.2.

$$\forall (c,d) \in p \times q,\ c|_I \approx d|_I \Rightarrow c \approx^* d$$

5. Formal methods for refinement-based design in SpecC

The model and method presented in Sections 2 and 4 is applied to checking refinements between design abstraction-levels correct. Section 5.1 proposes a technique to automatically represent SpecC programs at various abstraction levels in the Polychrony workbench. Section 5.2 applies the methodology of Section 4 to formally establish the correctness of design refinements.

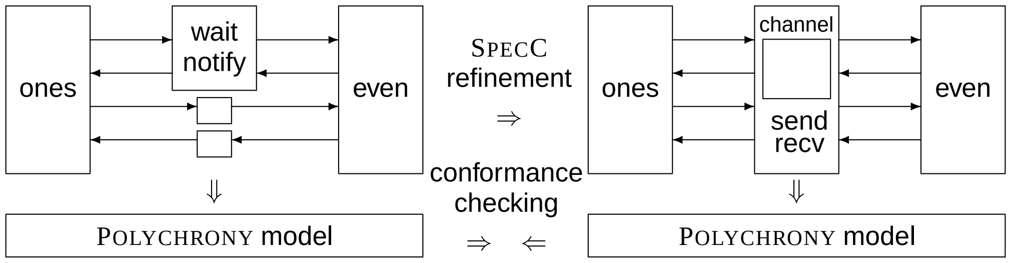


Figure 18. Checking conformance of a design refinement

This exercise is depicted by considering a simple SpecC programming example as case study to illustrate our methodology. It demonstrate the usability if the Polychrony workbench to provide the needed model, method and tool to automatically derive conditions on specifications, verifiable by static checking or model checking, and under which the refinement of a high-level specification, by its lower-level implementation can be formally checked, in a manner that is independent of a particular formalism (we consider SpecC in [36], Java in [35], SystemC in [37]).

Larger case studies applied to concrete examples (e.g. a finite input response filter and an ARM bus) are currently under way to demonstrate the capability of our technique to provide modular verification and an environmnent for co-simulation. In particular, modular verification is envisaged by considering the verification of a property in a system by checking it against the model of the very components that affects it while considering a static abstraction (in terms of clock and scheduling constraints) of all other components in the system under validation. Cosimulation is being investigated by considering the controller synthesis techniques provided in the Polychrony workbench and with the aim of applying them to the generation of optimized and control-sensitive simulators for large SystemC designs.

5.1. Modeling SpecC threads in Signal

The formal grammar of SpecC programs under consideration, Table 1, is represented in static single-assignment intermediate form akin to that of the Tree-SSA package of the GCC project [34]. SSA provides a language-independent, locally optimized intermediate representation (Tree-SSA currently accepts C, C++, Fortran 95, and Java inputs) in which language-specific syntactic sugar is absent. SSA transforms a given programming unit (a function, a method or a thread) into a structure in which all variables are read and written once and all native operations are represented by 3-address instructions.

An intermediate representation A program pgm consists of a sequence of labeled blocks $L{:}blk$. Each block consists of a label L and of a sequence of statements stm terminated by a return statement rtn. In the remainder, a block always starts with a label and finishes with a return statement: $stm_1; L{:}stm_2$ is rewritten as $stm_1;$ `goto` $L; L{:}stm_2$. A `wait` is always placed at the beginning of a block: $stm_1;$ `wait` v; stm_2 is rewritten as $stm_1;$ `goto` L; L:`wait` v; stm_2. Block instructions consist of native method invocations $x = f(y^*)$, lock monitoring and branches `if` x `goto` L. Blocks are returned from by either a `goto` L, a `return` or an exception `throw` x. The declaration `catch` x `from` L_1 `to` L_2 `using` L_3 that matches an exception x raised at block L_1 activates the exception handler L_3 and continues at block L_2.

Table 1. SSA intermediate representation for SpecC programs

(program)	$pgm ::= L{:}blk \mid pgm; pgm$		(block)	$blk ::= stm; blk \mid rtn$	
(instruction)	$stm ::= x = f(y^*)$	(invoke)	(return)	$rtn ::=$ `goto` L	(goto)
	$\mid$ `wait` x	(lock)		$\mid$ `return`	(return)
	$\mid$ `notify` x	(unlock)		$\mid$ `throw` x;	(throw)
	$\mid$ `if` x `goto` L	(test)		`catch` x `from` L `to` L `using` L	(catch)

To depict the structure of the Ssa for a typical SpecC program, consider the core of the Epc, Figure 19, and the behavior `ones`, that counts the number of bits set to 1 in a bit-array `data`. It consists of three blocks. The block labeled `L1` waits for the lock `istart` before initializing the local state variable `idata` to the value of the input signal `data` and `icount` to 0.

```
while true {
    wait (istart);
    idata = data;
    icount = 0;
    while (idata != 0) {
        icount += (idata & 1);
        idata >>= 1; }
    ocount = icount;
    notify (idone); }
```

```
L1: wait (istart);
    idata  = data;
    icount = 0;
    goto L2;

L3: ocount = icount;
    notify (idone);
    goto L1;
```

```
L2: T1 = idata;
    T2 = T1 == 0;
    if T2 goto L3;
    T3 = icount;
    T4 = T1 & 1;
    icount = T3 + T4;
    idata = T1 >> 1;
    goto L2;
```

Figure 19. From SpecC to static single assignment

Label `L2` corresponds to a loop that shifts `idata` right and adds its right-most bit to `icount` until termination (condition `T2`). Finally, in the block `L3`, `icount` is sent to the signal `ocount` and `idone` is unlocked before going back to `L1`.

Translation algorithm The function $\mathcal{I}[\![pgm]\!]$, Table 2, implements the translation from SSA to SIGNAL. It was first developed for the JIMPLE intermediate representation of JAVA [35], then redesigned and adapted to the wider spectrum of programming languages admitting the SSA intermediate representation [34] and its used exemplified for SYSTEMC in [37]. To each block of label $L \in \mathcal{L}_f$, the function $\mathcal{I}[\![pgm]\!]$ associates an *input clock* x_L and an *output clock* x_L^{exit}. The clock x_L is true iff L has been activated in the previous transition. The boolean signal x_L^{exit} is true iff execution of block L is terminates. The default activation condition of a block is hence $x_L\$1$ (equation (1) of Table 2).

Table 2. Modeling of SSA expressions into SIGNAL

$$
\begin{array}{lrl}
(1) & \mathcal{I}[\![L{:}blk;pgm]\!] = & \mathcal{I}[\![blk]\!]_L^{x_L\$1} \mid \mathcal{I}[\![pgm]\!] \\
(2) & \mathcal{I}[\![stm;blk]\!]_L^e = & \text{let}\ \langle P\rangle^{e_1} = \mathcal{I}[\![stm]\!]_L^e\ \text{in}\ P \mid \mathcal{I}[\![blk]\!]_L^{e_1} \\
(3) & \mathcal{I}[\![x = f(y_{1\ldots n})]\!]_L^e = & \langle \mathcal{E}(f)(x_{1\ldots n}e)\rangle^e \\
(4) & \mathcal{I}[\![\texttt{if}\ x\ \texttt{goto}\ L_1]\!]_L^e = & \langle y := x\ \texttt{when}\ e \mid x_{L_1} ::= \texttt{true when}\ y\rangle^{\texttt{not}\ y}\rangle \\
(5) & \mathcal{I}[\![\texttt{notify}\ x]\!]_L^e = & \langle x ::= \texttt{not}\ x\$1\ \texttt{when}\ e\rangle^e \\
(6) & \mathcal{I}[\![\texttt{wait}\ x]\!]_L^e = & \langle\ y := (x = x\$1)\ \texttt{when}\ e \mid x_L ::= \texttt{true when}\ y \\
 & & \mid z := \texttt{true when}\ y\ \texttt{default false}\ \rangle^{z\$1} \\
(7) & \mathcal{I}[\![\texttt{goto}\ L_1]\!]_L^e = & x_L^{exit} ::= \texttt{true when}\ e \mid x_{L_1} ::= \texttt{true when}\ e \\
(8) & \mathcal{I}[\![\texttt{return}]\!]_L^e = & x_L^{exit} ::= \texttt{true when}\ e \mid x_f^{exit} ::= \texttt{true when}\ e \\
(9) & \mathcal{I}[\![\texttt{throw}\ x]\!]_L^e = & x_L^{exit} ::= \texttt{true when}\ e \mid x ::= \texttt{true when}\ e \\
(10) & \mathcal{I}[\![\texttt{catch}\ x\ \texttt{from}\ L\ \texttt{to}\ L_1\ \texttt{using}\ L_2]\!]_L^e = & x_{L_2} ::= \texttt{true when}\ \hat{}x\ \texttt{when}\ x_L^{exit} \\
 & & \mid x_{L_1} ::= \texttt{true when}\ x_{L_2}^{exit}
\end{array}
$$

For a return instruction or for a block, function $\mathcal{I}$ returns a type P. For a block instruction stm, function $\mathcal{I}[\![stm]\!]_L^{e_1} = \langle P\rangle^{e_2}$ takes three arguments: an instruction stm, the label L of the block it belongs to, and an input clock e_1. It returns the type P of the instruction and its output clock e_2. The output clock of stm corresponds to the input clock of the instruction that immediately follows it in the execution sequence of the block.

```
L2: T1 = idata;          |    T1     ::= idata$1 when L2$1
    T2 = T1 == 0;        |  | T2     ::= T1 = 0 when L2$1
    if T2 goto L3;       |  | L3     ::= true when T2
    T3 = icount;         |  | T3     ::= icount$1 when not T2
    ...                  |  ...
```

Figure 20. From SSA to SIGNAL

For instance, consider block `L2` of behavior `ones`, Figure 20. The instruction `T1` $=$ `idata`, left, is associated with the partial equation `T1 ::= idata$1 when L2$1`, right. It means that, if the label `L2` is being executed, then `T1` is equal to `idata$1`. Next, consider instruction `if T2 goto L3`. It corresponds to the partial equation `L3 ::= true when T2`. It means that control is passed to `L3` when `T2` is true. Instructions that follow are conditioned by the negative `not T2` to means: ”in the block `L2` and not in its branch going to `L3`”.

Rules $(1-2)$ are concerned with the iterative decomposition of a program pgm into blocks blk and with the decomposition of a block into stm and rtn instructions. In rule (2), the input clock e of the block $stm; blk$ is passed to stm. The output clock e_1 of stm becomes the input clock of blk.

Rule (3) is concerned with the translation of native and external method invocations $x = f(y_{1...n})$. The generic type of f is taken from an environment $\mathcal{E}(f)$. It is given the name of the result x, of the actual parameters $y_{1...n}$ and of the input clock e to obtain the type of $x = f(y_{1...n})$. For instance, Figure 21 depicts the translation of native operations in block `L2` of behavior `ones`. The assignment of `icount` to the local variable `T3` is translated by the partial equation `T3 ::= icount$1 when not T2` which assigns the previous value of `icount` to the temporary `T3` at the clock `not T2` (i.e. when `T1` is not 0, Figure 20).

```
T3 = icount;          |   T3     ::= icount$1 when not T2
T4 = T1 & 1;          | | T4     ::= T1 & 1 when not T2
icount = T3 + T4;     | | icount ::= T3 + T4 when not T2
idata = T1 >> 1;      | | idata  ::= T1 >> 1 when not T2
```

Figure 21. Translating bative operations in SIGNAL

The input and output clocks of an instruction may differ. This is the case, rule (4), for an `if` x `goto` L_1 instruction in a block L. Let e be the input clock of the instruction and define the fresh signal name y by the equation $y := x$ `when` e. When y is false, then control is passed to the rest of the block: the output clock is `not` y. Otherwise, the control is passed to the block L_1 at the clock y.

The wait-notify protocol, rules $(5-6)$, is modeled using a boolean flip-flop variable x. Method `notify`, rule (5), defines the next value of the lock x by the negation of its current value at the input clock e. The `wait` method, rule (6), activates its output clock y iff the value of the lock x has changed at the input clock e. Otherwise, control goes back to L.

```
L1: wait (istart);        |   T1     ::= istart = istart$1 when L1$1
                          | | L1     ::= true when T1
                          | | L1b    ::= true when not T1
    ...                   |   ...
L3: ocount = icount;      | | ocount ::= icount$1 when L3$1
    notify (idone);       | | idone ::= not idone$1 when L3$1
    goto L1;              | | L1   ::= true when L3$1
```

Figure 22. Model of `wait-notify` in the EPC

For example, consider the wait-notify protocol at the blocks labeled `L1` and `L3` in the `ones` counter. The wait instruction receives control at the clock $x_{\mathtt{L1}}$. If the value of `istart` changes (i.e. `when not T0`) then `icount` and `idata` are initialized and the control is passed to the block `L2`. Otherwise, at the clock `when T0`, a transition back to `L1` is scheduled.

All return instructions, rules $(7-9)$, define the output clock x_L^{exit} of the current block L by their input clock e. This is the right place to do that: e defines the very condition upon which the block actually reaches its return statement. A `goto` L_1 instruction, rule (7), passes control to block L_1 unconditionally at the input clock e. A `return` instruction, rule (8), sets the exit clock x_f to true at clock e to inform the caller that f is terminated. A `throw` x statement in block L, rule (9), triggers the exception signal x at the input clock e by x ::= `true when` e. The matching `catch` statement, of the form `catch` x `from` L `to` L_1 `using` L_2 passes the control to the handler L_2 and then to the block L_1 upon termination of the handler. This requires, first, to activate L_2 from L when x is present and then to pass control to L_1 upon termination of the handler.

Completion Table 2 requires all entry clocks x_L and x_f to be simultaneously present when the f is being executed. Each signal x_L holds the value `true` iff the block L is active during a transition currently being executed. Otherwise, x_L is set to `false` by `L ::= defaultvalue false`. The same holds for local variables `T` with the default equation `T ::= defaultvalue T$1`. The SIGNAL compiler guarantees the completion of the next-state logic by aggregating partial equations.

```
  L1 := true when (T1 default L3$1) default false
| L2 := true when (L1b$1 default not T3) default false
| L3 := true when T3 default false
| L1 ^= L2 ^= L3
```

Figure 23. Completion of the next-state-logic for the EPC

The translation technique proposed in [35, 37] is modular (block per block), conceptually simple (one instruction corresponds to one equation) and language independent (it takes Gnu SSA as input formalism). The host formalism, SIGNAL, supports a scalable notion and a flexible degree of abstraction. Notice that the structure of the original program is represented by program labels L which play an essential role during modeling as they represent clocks, i.e. the data-structure used by the POLYCHRONY workbench to represent the control flow of programs. This information is propagated during modeling, verification and transformation. As a result, traceability is easily provided by this information to relate an error to its original block, in addition to the name of all variables it implies.

5.2. A case study: the even-parity checker

We focus on a simple SPECC programming example: an even-parity checker (EPC, figure 24), to illustrate our refinement-based methodology. We shows how the specification of the EPC can be refined toward a GALS implementation with the help of the tool POLYCHRONY, showing in what respects and at which critical design stages formal methods matter for engineering its architecture. This example demonstrates the capabilities of the polychronous model of computation of SIGNAL to provide formal modeling and verification support for the capture of behavioral abstractions of high-level system descriptions in, e.g., SPECC. The even-parity checked (EPC, figure 24) consists of three functional units: an interface thread `IO`, a master test thread `even` and a slave counting thread `ones` (gray elements are

SPECC-specific). A number is first sent from the thread `IO` to the thread `even` via the input port `In`. Then, `even` expects `start` to be notified by `IO` before to forward the input to the thread `ones` via the `data` port and notify `istart`. Upon this notification, thread `ones` counts the number of bits set to one in `data` and returns it to `even` via the `ocount` port and notifies `idone`. Thread `even` forwards the even status of that count to `IO` and notifies `done`.

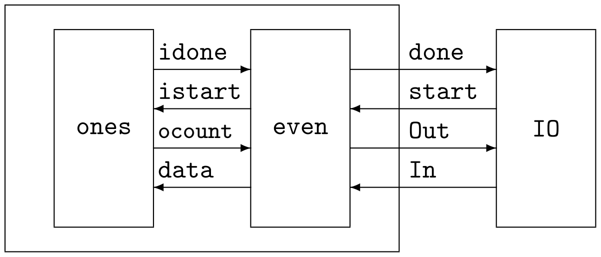


Figure 24. Functional architecture of an even-parity checker (EPC).

5.2.1. Specification-level design in SPECC

In SPECC, the design level of specification defines the functionalities and behavior of a system composed of hardware and software by means of parallel threads (called *behaviors*) of computations exchanging data via ports and synchronized by wait and notify events.

```
behavior ones (in unsigned int data,
               in event istart,
               out unsigned int ocount,
               out event idone) {
  void main (void) {
    unsigned int idata;
    unsigned int icount;
    while (true) {
      wait(istart);
      idata = data;
      icount = 0;
      while (idata != 0) {
        icount += idata & 1;
        idata >> 1; }
      ocount = icount;
      notify(idone);
}}}
```

```
behavior even (in unsigned int In,
               in unsigned int ocount,
               in event start,
               in event idone,
               out unsigned int Out,
               out unsigned int data,
               out event istart,
               out event done) {
 void main(void) {
   while (true) {
     wait(start);
     data = In;
     notify(istart);
     wait(idone);
     Out = ocount & 1;
     notify(done);
}}}
```

Figure 25. Specification-level design of the EPC in SPECC.

Behavior `ones`, Figure 25, first waits the event `istart` before to load data from the input port `data` into the variable `idata`. Then, it iteratively scans `idata` to count the number of bits set to one in it. This is done by shifting the bits in `idata` right and by comparing the right-most one to 1. When the data is processed (it equals 0 and there is no more bit to shift and count) the internal count `icount` is returned to the `ocount` port and the event `idone` is notified. The behavior `even` passes values from port `In` to port `data`, notifies `istart`, waits `idone`, and returns either 0 (if `ocount` is odd) or 1 (it is even) along the port `Out` before to notify `Done`.

Figure 26 gives a model of the SPECC behavior `ones` in SIGNAL. It displays partial equations obtained from the translation algorithm of Figure 2. The actual SIGNAL program which implements the specification model of the EPC is given Appendix A. It is obtained by completion of the next-state logics and by the synchronization of local variables. This yields a SIGNAL program that is endochronous. Hence, it is finitely flow-preserving and qualifies for our refinement-checking methodology.

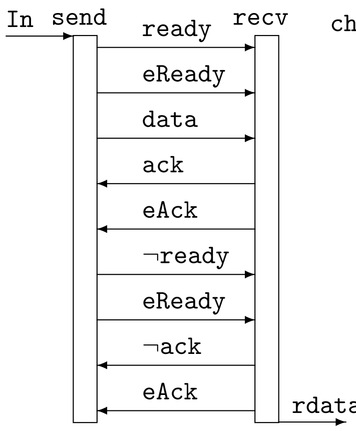
```
process ones = ()
(| T1     ::= istart = istart$1 when L1$1
 | L1     ::= true when T1
 | L2     ::= true when not T1
 | idata  ::= data$1 when L2$1
 | icount ::= 0 when L2$1
 | L3     ::= true when L2$1
 | T2     ::= idata$1 when L3$1
 | T3     ::= T2 = 0 when L3$1
 | L4     ::= true when T3
 | T4     ::= icount$1 when not T3
 | T5     ::= T2 & 1 when not T3
 | icount ::= T4 + T5 when not T3
 | idata  ::= T2 >> 1 when not T3
 | L3     ::= true when not T3
 | ocount ::= icount$1 when L4$1
 | idone  ::= not idone$1 when L4$1
 | L1     ::= true when L4$1
 |) / L1, L2, L3, L4, T1, T2,
     T3, T4, T5, idata, icount;
process even =  ()
(| T1     ::= start = start$1 when L1$1
 | L1     ::= true when T1
 | L2     ::= true when not T1
 | data   ::= In$1 when L2$1
 | istart ::= not istart$1 when L2$1
 | L3     ::= true when L2$1
 | T2     ::= idone = idone$1 when L3$1
 | L3     ::= true when T2
 | L4     ::= true when not T2
 | Out    ::= ocount$1 & 1 when L4$1
 | done   ::= not done$1 when L4$1
 | L1     ::= true when L4$1
 |) / L1, L2, L3, L4, T1, T2;
```

Figure 26. Specification-level design of the EPC in SIGNAL.

In initial state `L1`, the behavior `ones` waits for the event `istart`. If it receives it, at the clock `when not T0`, it initializes `idata` to `data`, `icount` to 0 and steps to state `L2`. While in state `L2`, at the clock `when not T2`, it adds the right-most bit of `idata` to the local count `icount`, shifts `idata` right and loops back to state `L2`. If `idata` is zero, at the clock `when T2`, the process `ones` steps to state `L2`, passes the value of `icount` to `ocount` and notifies `done` before going back to state `L1`. Process `even` implements a similar decomposition of the corresponding SPECC behavior in SSA form. We assume the notification events to be shared by the processes `ones` and `even` within a SIGNAL module.

5.2.2. Architecture-level design refinement

The translation of the even-parity checker of Section 30 demonstrates the capability of SIGNAL to model components for specification-level SPECC designs. The typical SPECC design-flow starts with the capture of IP-blocks represented as C functions and automatic partitioning according to an appropriate cost function. After partitioning, 2-way handshake protocols (message sequence below, or appropriate HW-SW protocols) are inserted between the functional units. In the specification model of the EPC, data exchange between the behaviors `ones` and `even` are performed by shared variables and arbitrated by a wait-notify protocol to ensure mutual exclusion of read and write operations. In the architecture model of the EPC, this specification is refined by the introduction of a double handshake protocol defined by the methods `send` and `recv` of the module `ChMP`, figure 27. The channel encapsulates the wait and notify operations performed in the specification model together with locally shared variables `ack` and `ready`.



```
channel ChMP() {
    bool ready = false, ack = false;
    event eReady, eAck;
    unsigned int data;
    void send (unsigned int in) {
        data = in;
        ready = true;
        notify (eReady);
        while (!ack) wait (eAck);
        ready = false;
        notify (eReady);
        while (ack) wait (eAck);
}
```

```
unsigned int recv () {
    unsigned int rtn;
    while (!ready)
     wait (eReady);
    rtn = data;
    ack = true;
    notify (eAck);
    while (ready)
     wait (eReady);
    ack = false;
    notify (eAck);
    return rtn;
}};
```

Figure 27. Implementation of a channel with double hand-shake in SPECC.

The SIGNAL model of the `send` method, Figure 28, of the `recv` method, of the architecture-layer refinement of the EPC, are obtained in the very same way as for the behaviors `even` and `ones` of the specification level. The `send` method of the `ChMP` module assigns the current value of the input `In` to the shared variable `data`, sets the shared `ready` flag to true and waits for the notification `eAck` from the `recv` process until the shared flag `ack` becomes true. The same transaction is repeated to wait until the transmission of its return data to ready a new transaction. Process `send` receives two parameters `Lsend` and `Lexit` in addition to its input data `In`. The boolean signal `Lsend` is true when `send` receives control from its caller. The boolean signal `Lexit` is true when `send` returns control to its caller. Appendix B gives the complete listing of the `send` and `recv` functionalities of the channel `ChMP` which were checked endochronous using the POLYCHRONY workbench to qualify for validation using our refinement checking methodology.

```
process send = (? In, L1 ! L7)
(| L2     ::= true when L1$1
 | data   ::= In$1 when L1$1
 | ready  ::= true when L1$1
 | eReady ::= not eReady$1 when L1$1
 | T2     :=  ack$1 when L2$1
 | L3     ::= true when not T2
 | L4     ::= true when T2
 | T1     :=  eAck = eAck$1 when L3$1
 | L2     ::= true when not T1
 | L3     ::= true when T1
```

```
| ready  ::= false when L4$1
| eReady ::= not eReady$1 when L4$1
| L5     ::= true when L4$1
| T4     :=  ack$1 when L5$1
| L6     ::= true when T4
| L7     ::= true when not T4
| T3     :=  eAck = eAck$1 when L6$1
| L5     ::= true when not T3
| L6     ::= true when T3
|) / L2, L3, L4, L5, L6,
    T1, T2, T3, T4;
```

Figure 28. Model of the architecture-level channel (send method) in SIGNAL.

5.2.3. Validation of the specification-to-architecture refinement

The installation of a channel between the processes `even` and `ones` incurs an additional desynchronization of the transmission between the `In` and `Out` signals, Figure 29. Showing that the refinement of the EPC from the specification level, process `epc1`, to the architecture level, process `epc2`, is correct requires checking that the refinement is finitely flow-invariant: `epc1` $\ll^*$ `epc2`.

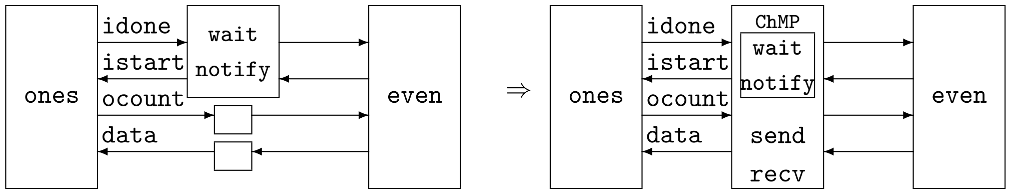


Figure 29. Refinement of the specification by an architecture layer.

The verification of this property amounts to proving that, for all behaviors b and c of `epc1` and `epc2`, flow equivalence of the input signal `In`, i.e. $b|_{\texttt{In}} \approx c|_{\texttt{In}}$ implies flow equivalence of the output signal `Out`, i.e. $b|_{\texttt{Out}} \approx c|_{\texttt{Out}}$.

$$(1) : \forall b \in [\![\texttt{epc1}]\!],\ \forall c \in [\![\texttt{epc2}]\!],\ b|_{\texttt{In}} \approx c|_{\texttt{In}} \Rightarrow b|_{\texttt{Out}} \approx c|_{\texttt{Out}}$$

Some preliminary observations are in order to facilitate and accelerate the proof of this equation. First, it is noticeable that the architecture model of the EPC only differs from the specification model by the introduction of a channel in place of a wait-notify protocol to synchronize concurrent accesses to shared variables. By contrast, the architecture model `epc2` uses the double handshake protocol of the `send` and `recv` functionality of the module `ChMP`.

Figure 30 isolates the pattern `specif` that characterizes the wait-notify protocol in the specification model. The variables of the EPC models `specif` and `archi` are interfaced to signals `In` and `Out` for the purpose of verification. The producer and consumer processes `prod1` and `cons1` are defined by infinite loops wrapping that wrap the patterns of interest.

```
process specif = (? boolean In
                  ! boolean Out)
(| Out := cons1() | prod1(In) |)
where boolean istart init false,
              data init false;
  process prod1 = (? In !)
  (| data   := In default data$1
   | istart := not istart$1 when ^In
               default istart$1 |);
```

```
process cons1 = (? ! Out)
(| T1  := istart = istart$1 when L1$1
 | L1  := true when (T1 default L2$1)
          default false
 | L2  := true when not T1 default false
 | Out := data$1 when L2$1
 | L1  ^= L2 ^= istart ^= data
 |) where boolean L1 init true, L2, T1;
end;
```

Figure 30. Isolation of the synchronization protocol in the specification model

Figure 31 isolates the corresponding communication pattern of the architecture model wrapped by the process `archi`. In Figure 30, the presence of an input `In` triggers a wait-notify protocol that synchronizes the transmission of `data` from `even` to `ones`. In Figure 31, this synchronization is embedded in the call to the `send` and `recv` methods. Control to and from `send` and `recv` is provided by the labels `L2` and `L3`. The additional label `L1` closes the infinite loop.

```
process archi = (? boolean In ! boolean Out)
  (| Out := cons2() | prod2(In) |)
where process cons2 = (? ! Out)
  (| L2  := L1$1 init true
   | Out := data$1 when L3$1
   | L1  := true when L3$1
            default false
   | L1  ^= L2 ^= L3 ^= data
   | (L3, data) := recv(L2)
   |) where boolean L1, L2, L3, data; end;
```

```
process prod2 = (? In !)
(| data := In when L1$1 default data$1
 | T1   := true when ^In when L1$1
 | L1   := true when (not T1 default L3$1)
           default false
 | L2   := true when T1 default false
 | L3   := send(data$1, L2)
 | L1   ^= L2 ^= L3 ^= data
 |) where boolean T1, L1 init true,
          L2, L3, data; end;
```

Figure 31. Isolation of the synchronization protocol in the architecture model

Under these observations, proving equation (1) reduces to showing that the desynchronization protocol introduced by module `ChMP` preserves the flow of the original wait-notify protocol of the specification model. This proof is done by checking that, given desynchronized input flows $b|_{\texttt{In}}$ and $c|_{\texttt{In}}$, the specification and architecture models `spec` and `arch` provide equivalent flows along the output `idata`

$$(2) : \forall b \in [\![\texttt{specif}]\!],\ \forall c \in [\![\texttt{archi}]\!],\ b|_{\texttt{In}} \approx c|_{\texttt{In}} \Rightarrow b|_{\texttt{Out}} \approx c|_{\texttt{Out}}$$

To formulate equation (2) in the model checker SIGALI, we consider a formulation of the protocol that manipulates boolean data `In` and `Out`. This approximation still implies the expected property (2), since neither `In` nor `Out` interfere with control in `specif` and `archi`. We obtain the formulation of the appropriate observer function (see appendix C). It is be checked invariant by SIGALI and yields the expected proof of conformance, by application of Theorem 4.2.

$$(3) : \texttt{cqfd := observer \{specif, archi\} (In)}$$

5.2.4. Refinement of the architecture model toward implementation

The communication layer of the EPC, Figure 32, consists of a refinement of the data structures manipulated in the `ChMP` channel and of modeling (in SPECC) the behavior of the actual bus of the architecture in place of the channel abstraction `ChMP`. The model of the bus consists of the decomposition of the methods `send` and `receive` into sub-processes that implement the bus `read` and `write` methods.

Showing this refinement correct reduces to proving that the model of the channel's `ChMP` methods `send` and `recv` are flow-equivalent to the methods `read` and `write` of the bus model. The control structure of the bus model in SIGNAL is identical to that of the channel, except for the implementation of the input/output integer signals as bit-vectors.

Compared to the communication model of Figure 32, the RTL or implementation model of the EPC in SPECC, Figure 33, consists of a cycle-accurate implementation of the EPC that is materialized the introduction of a master clock `clk` and of a reset signal `rst` together with the conversion of the EPC communication-layer specification into finite-state machine code. The structure of the `ones` thread itself

```
channel cBus() implements iBus {
    unsigned bit[31:0] data;
    cSignal ready, ack;
    void write (unsigned bit[31:0] wdata) {
        ready.assign(1);
        data = wdata;
        ack.waitval(1);
        ready.assign(0);
        ack.waitval(0);
    }
```

```
unsigned bit[31:0] read () {
    unsigned bit[31:0] rdata;

    ready.waitval(1);
    rdata = data;
    ack.assign(1);
    ready.waitval(0);
    ack.assign(0);
    return data;
}}
```

Figure 32. Communication-level model of a bus in SPECC.

is close to the original one in SSA intermediate form. Each transition from a value of the `state` variable is guarded by an explicit synchronization to the simulation clock `clk`.

```
behavior ones(in,event,clk, ...) {
  void main(void) {
    unsigned bit[31:0] idata, icount;
    enum state {S0, S1, S2, S3} state = S0;
    while (1) {
      wait(clk);
      if (rst == 1b) state = S0;
      switch (state) {
        case S0: done = 0b;
                 ack_istart = 0b;
                 if (start == 1b) state=S1
                 else state=S0;
                 break;
        case S1: ack_istart = 1b;
```

```
          idata = inport;
          icount = 0;
          state = S2;
          break;
case S2:  icount = icount + idata & 1;
          idata = idata >> 1;
          if (idata == 0) state=S3
          else state=S2;
          break;
case S3:  outport=icount;
          done = 1b;
          if (ack_idone == 1b) state=S0
          else state=S3;
          break;
```

Figure 33. RTL-level implementation of the EPC-core in SPECC.

Assessment By considering a simple SPECC programming example demonstrating all salient feature of the design language, we demonstrated the capability of the POLYCHRONY workbench to provide:

- a model of computation to formally model the functionalities and architectures of a system
- services to express model transformations check the correct.

The methodology of finite-flow invariance we employ is directly derived from previous work pertaining on the design of globally asynchronous locally synchronous architectures. The present case study departs from the very spectrum of GALS design by showing that imperative programs in the style of communicating sequential processes are equally covered. In other words, flow equivalence does not define the spectrum of architectures that are covered by our methodological principles but the very criterion for checking design refinements correct. Naturally, other criteria can be designed and other properties checked.

To this end, the POLYCHRONY workbench provides scalable abstraction of SPECC design for verification. Previous results and case studies span from the use theorem-proving [19, 25, 20] to prove properties on concrete models, model checking [6, 36] to prove properties on models abstracted by finite-state machines, static checking [26, 37] to prove properties over state-less boolean model abstractions. POLYCHRONY further provides means to optimally reuse, adapt, transform pre-defined system components and modules: hierarchization (combines several threads into one), distributed protocol synthesis (split synchronous threads into a network of communicating threads). Its current limitations are the absence of support for reasoning on dynamic resources management (memory or threads) and the lake of connexions to other models of computation (untimed, real-timed, continuous).

6. Related works

Synchronous programming being a computational model which is popular in hardware design, and desynchronization being a technique to convert that computational model into a more general, globally asynchronous and locally synchronous computational model, suitable for system-on-chip design, one may naturally consider investigating further the links between these two models understood as Ptolemy domains [9] and study the refinement-based design of GALS architectures starting from polychronous specifications captured from heterogeneous elementary components.

The aim of capturing both synchrony and asynchrony in a unifying model of computation is shared by several approaches: the interaction categories of Abramsky et al. in [1], the communicating sequential processes of Hoare [17] and Kahn networks [18] (communicating data-flow functions) that is one of the models supported by Ptolemy [21] and the methodology of latency insensitive protocols of Carloni et al. [10] and its extension to real-time [4]. All related models can be categorized as stratified, in the sense that they dissociate the semantics structure representing synchronous islands (the predefined *pearls* or IPS) from the one representing asynchrony. For instance, the heterogeneous model of [4] is layered into a model of tag-less asynchrony and of tagged synchrony (where tags model stuttering equivalence in a way akin to clock equivalence yet without scheduling). The polychronous model of computation does not feature such a decoupling between its synchronous and asynchronous structures.

Most of the existing formal frameworks to refinement-checking, such as B [2], Unity [11], CSP [17, 29], are essentially specification-based, in that modeling, transformation and verification are entirely envisaged within the framework of a formal notation and its companion methods and tools. By contrast, our data-flow oriented approach to refinement-checking is novel and allows us to combine a programatic approach (of SPECC) with the scalable modeling and verification techniques of the polychronous model of computation: theorem-proving [19, 25, 20], model checking [6, 36], static checking [26, 37]. Being combined to a language-independent translation technique enabling the capture of high-level system descriptions in general purpose languages such as C orJAVA, our workbench departs from specification-oriented approaches, by combining software model-checking capabilities with aggressive optimization services present in this workbench for the purpose of accelerated simulation and synthesis.

7. Conclusions

Until now, the refinement of a system-level description toward its implementation, in SPECC or SYSTEMC was primarily envisaged as a manual process and proving conformance from the system-level abstraction to the implementation an unsolved issue. To solve it, we proposed a formal refinement-checking methodology for system-level design formalized within the polychronous model of computation of the multi-clocked synchronous formalism SIGNAL.

We demonstrated the effectiveness of our approach by the experimental case study of a SPECC programming example, showing the benefits of our approach to give an accurate model of successive design abstractions of the system: functional, architecture, communication. We introduced, proved and applied a refinement-checking criterion allowing for comparing and validating behavioral equivalence relations between these successive refinements.

Our methodology relies on an automated modeling technique that is conceptually minimal and supports a scalable notion and a flexible degree of abstraction. Our presentation targets SPECC yet with a generic and language-independent method. Applications of our technique range from the detection of local design errors to the compositional design refinement and conformance checking.

References

[1] ABRAMSKY, S., GAY, S. J., NAGARAJAN, R. Interaction categories and the foundations of typed concurrent programming. In *Deductive Program Design: Proceedings of the 1994 Marktoberdorf International Summer School*. NATO ASI Series F, Springer-Verlag, 1996.

[2] ABRIAL, J.-R. *The B Book - Assigning Programs to Meanings*. Cambridge University Press, August 1996.

[3] AMAGBEGNON, T. P., BESNARD, L., LE GUERNIC, P. "Implementation of the data-flow synchronous language SIGNAL". In *Conference on Programming Language Design and Implementation*. ACM Press, 1995.

[4] BENVENISTE, A., CASPI, P., CARLONI, L. P., SANGIOVANNI-VINCENTELLI, A. L. "Heterogeneous Reactive Systems Modeling and Correct-by-Construction Deployment". In *Embedded Software Conference*. Springer Verlag, October 2003.

[5] BENVENISTE, A., CASPI, P., EDWARDS, S., HALBWACHS, N., LE GUERNIC, P., DE SIMONE, R. "The synchronous languages twelve years later". In *Proceedings of the IEEE, special issue on embedded systems*. IEEE Press, January 2003.

[6] BENVENISTE, A., CASPI, P., LE GUERNIC, P., MARCHAND, H., TALPIN, J.-P., TRIPAKIS, S. "A protocol for loosely time-triggered architectures". In *Embedded Software Conference*. Springer Verlag, October 2002.

[7] BENVENISTE, A., LE GUERNIC, P., JACQUEMOT, C. "Synchronous programming with events and relations: the SIGNAL language and its semantics". In *Science of Computer Programming*, v. 16. Elsevier, 1991.

[8] BERRY, G., GONTHIER, G. "The ESTEREL synchronous programming language: design, semantics, implementation". In *Science of Computer Programming*, v. 19, 1992.

[9] BUCK, J., HA, S., LEE, A., AND MESSERSCHMITT, D. Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems. In *International Journal of Computer Simulation, special issue on "Simulation Software Development"*, v. 4. Ablex, April 1994.

[10] CARLONI, L. P., MCMILLAN, K. L., SANGIOVANNI-VINCENTELLI, A. L. "Latency-Insensitive Protocols". In *International Conference on Computer-Aided Verification*. Lecture notes in computer science v. 1633. Springer Verlag, July 1999.

[11] CHANDY K. M., MISRA, J. *Parallel Program Design: A Foundation*. Addison Wesley, 1999.

[12] DE MICHELI, G., ERNST, E., AND WOLF, W. "Readings in Hardware/Software Co-Design". Morgan Kaufmann, 2001.

[13] F. DOUCET, F., OTSUKA,, M., SHUKLA, S., GUPTA, R. An environment for dynamic component composition for efficient co-design. In *Design Automation and Test in Europe*. IEEE Press, 2002.

[14] GAMATIÉ, A., GAUTIER, T. "Synchronous modeling of avionics applications using the SIGNAL language". In *Real-time embedded technology and applications symposium*. IEEE Press, 2002.

[15] GAJSKI, D., ZHU, J., DÖMER, R., GERSTLAUER, A., ZHAO, S. SPECC: Specification Language and Methodology. Kluwer Academic Publishers, March 2000.

[16] GROETKER, T., LIAO, S., MARTIN, G., SWAN, S. System Design with SYSTEMC. Kluwer Academic Publishers, May 2002.

[17] HOARE, C. *Communicating sequential processes*. Prentice Hall, 1985.

[18] KAHN, G. The semantics of a simple language for parallel programming. In IFIP *Congress*. North Holland, 1974.

[19] KERBOEUF, M., NOWAK, D., TALPIN, J.-P. "The steam-boiler problem in SIGNAL-COQ". *International Conference on Theorem Proving in Higher-Order Logics (TPHOLS)*. Springer Verlag Lectures Notes in Computer Science, Aout 2000.

[20] KERBOEUF, M., NOWAK, D., TALPIN, J.-P. "Formal proof of a polychronous protocol for loosely time-triggered architectures". *International conference on formal engineering methods*. Springer Verlag Lectures Notes in Computer Science, Novembre 2003.

[21] LEE, E., SANGIOVANNI-VINCENTELLI, A. "A framework for comparing models of computation". In IEEE *transactions on computer-aided design*, v. 17, n. 12. IEEE Press, December 1998.

[22] LE GUERNIC, P., TALPIN, J.-P., LE LANN, J.-L. Polychrony for system design. In *Journal of Circuits, Systems and Computers. Special Issue on Application Specific Hardware Design*. World Scientific, 2002.

[23] MARCHAND, H., BOURNAI, P., LE BORGNE, M., LE GUERNIC, P. Synthesis of Discrete-Event Controllers based on the Signal Environment. In *Discrete Event Dynamic System: Theory and Applications*, v. 10(4), pp. 325–346, 2000.

[24] H. MARCHAND, E. RUTTEN, M. LE BORGNE, M. SAMAAN. Formal Verification of SIGNAL programs: Application to a Power Transformer Station Controller. *Science of Computer Programming*, v. 41(1), pp. 85–104, 2001.

[25] NOWAK, D., BEAUVAIS, J.-R., TALPIN, J.-P. "Co-inductive axiomatization of a synchronous language". In *International Conference on Theorem Proving in Higher-Order Logics*. Springer Verlag, October 1998.

[26] NOWAK, D., TALPIN, J.-P., GAUTIER, T., LE GUERNIC, P. "An ML-like module system for the synchronous language Signal". *Proceedings of European Conference on Parallel Processing (EuroPAR)*. Springer Verlag Lectures Notes in Computer Science, Aout 1997.

[27] NOWAK, D., TALPIN, J.-P., LE GUERNIC, P. "Synchronous structures". In *International Conference on Concurrency Theory*. Springer Verlag, August 1999.

[28] The Polychrony workbench, available from `http://www.irisa.fr/espresso/Polychrony`.

[29] ROSCOE, A. W.. *The Theory and Practice of Concurrency*. Prentice-Hall, 1997.

[30] The SPECC website, `http://www.ics.uci.edu/~specc`.

[31] The SYSTEMC website, `http://www.systemc.org`.

[32] The SYSTEM VERILOG website, `http://www.systemverilog.org`.

[33] The Gnu Compiler Collection (GCC). `http://http://gcc.gnu.org`, 2004.

[34] The GCC Tree-SSA Branch. `http://gcc.gnu.org/projects/tree-ssa`, 2004.

[35] TALPIN, J.-P., GAMATIÉ, A., LE DEZ, B., BERNER, D., LE GUERNIC, P. Hard real-time implementation of embedded software in JAVA. In *Interntional workshop on scientific engineering of distributed JAVA applications*. Lectures Notes in Computer Science, Springer Verlag, November 2003.

[36] TALPIN, J.-P., LE GUERNIC, P., SHUKLA, S., GUPTA, R., DOUCET, F. "Polychrony for formal refinement-checking in a system-level design methodology". *Application of Concurrency to System Design*. IEEE Press, June 2003.

[37] TALPIN, J.-P., BERNER, D., SHUKLA, S. K., LE GUERNIC, P., GUPTA, R. "A behavioral type inference system for compositional system design". *Application of Concurrency to System Design*. IEEE Press, June 2004.

A. Listing of the specification model of the EPC

```
process epc = (? boolean start; integer In ! boolean done; integer Out)
(| (idone, ocount) := ones (istart, data)
 | (done, Out) := even(start, In)
 |) where integer ocount init 0, data init 0; boolean istart init false, idone init false;

process ones = (? boolean istart; integer data ! boolean idone; integer ocount)
(| T1     := istart = istart$1 when L1$1
 | T2     := idata$1 when L3$1 default T2$1
 | T3     := T2 = 0 when L3$1
 | T4     := icount$1 when not T3 default T4$1
 | T5     := X2(T2, 1) when not T3 default T5$1

 | L1     := true when (T1 default L4$1) default false
 | L2     := true when not T1 default false
 | L3     := true when (L2$1 default not T3) default false
 | L4     := true when T3 default false

 | idata  := data$1 when L2$1 default X1(T2, 1) when not T3 default idata$1
 | icount := 0 when L2$1 default T4 + T5 when not T3 default icount$1
 | ocount := icount$1 when L4$1 default ocount$1
 | idone  := not idone$1 when L4$1 default idone$1

 | idata ^= icount ^= ocount ^= idone ^= istart ^= data ^= L1 ^= L2 ^= L3 ^= L4 ^= T2 ^= T4 ^= T5
 |) where boolean L1 init true, L2 init false, L3 init false, L4 init false, T1, T3;
         integer idata init 0, icount init 0, T2, T4, T5; end;

process even =  (? boolean start; integer In ! boolean done; integer Out)
(| T1     := start = start$1 when L1$1
 | T2     := idone = idone$1 when L3$1

 | L1     := true when (T1 default L4$1) default false
 | L2     := true when not T1 default false
 | L3     := true when (L2$1 default T2) default false
 | L4     := true when not T2 default false

 | data   := In$1 when L2$1 default data$1
 | istart := not istart$1 when L2$1 default istart$1
 | Out    := X2(ocount$1, 1) when L4$1 default Out$1
 | done   := not done$1 when L4$1 default done$1

 | data   ^= istart ^= Out ^= done ^= L1 ^= L2 ^= L3 ^= L4
 |) where boolean L1 init true, L2 init false, L3 init false, L4 init false, T1, T2; end;

function X1 = (? i1, i2 ! i3)
  spec (| i1^=i2^=i3 | i1 --> i2 | i2 --> i3 |)
  pragmas C_CODE "&i3 = &i1 >> &i2" end pragmas;

function X2 = (? i1, i2 ! i3)
  spec (| i1^=i2^=i3 | i1 --> i2 | i2 --> i3 |)
  pragmas C_CODE "&i3 = &i1 & &i2" end pragmas;
end;
```

B. Listing of the channel model

```
module ChMP =
boolean ready init false, ack init false, eReady init false, eAck init false;
integer data init 0;

process send = (? In, L1 ! L7)
(| T1     := eAck = eAck$1 when L3$1
 | T2     := ack$1 when L2$1
 | T3     := eAck = eAck$1 when L6$1
 | T4     := ack$1 when L5$1
 | L2     := true when (L1$1 default not T1) default false
 | L3     := true when (not T2 default T1) default false
 | L4     := true when T2 default false
 | L5     := true when (L4$1 default not T3) default false
 | L6     := true when (T4 default T3) default false
 | L7     := true when not T4 default false
 | data   := In$1 when L1$1 default data$1
 | ready  := true when L1$1 default false when L4$1 default ready$1
 | eReady := not eReady$1 when (L1$1 default L4$1) default eReady$1
 | L1     ^= L2 ^= L3 ^= L4 ^= L5 ^= L6 ^= L7
 | L1     ^= ready ^= ack ^= eReady ^= eAck  ^=  data ^= In
 |) where boolean L2, L3, L4, L5, L6, T1, T2, T3, T4; end;

process recv = (? L1 ! rtn, L7)
(| T1   := eReady = eReady$1 when L3$1
 | T2   := not ready$1 when L2$1
 | T3   := eReady = eReady$1 when L6$1
 | T4   := not ready$1 when L5$1
 | L2   := true when (L1$1 default not T1) default false
 | L3   := true when (T2 default T1) default false
 | L4   := true when not T2 default false
 | L5   := true when (L4$1 default not T3) default false
 | L6   := true when (T4 default T3) default false
 | L7   := true when not T4 default false
 | rtn  := (data$1 when L4$1) default rtn$1
 | ack  := (true when L4$1) default (false when L7$1) default ack$1
 | eAck := (not eAck$1 when (L4$1 default L7$1)) default eAck$1
 | L1   ^= L2 ^= L3 ^= L4 ^= L5 ^= L6 ^= L7
 | L1   ^= rtn ^= ack ^= eAck
 |) where boolean L2, L3, L4, L5, L6, T1, T2, T3, T4; end;
end;
```

C. Listing of the observer function

```
process observer = (? boolean In ! boolean cqfd)
(| cqfd := fifo (specif (buffer (In))) = fifo (archi (buffer (In)))
 |) where

use ChMP;
use Fifo;

process specif = (? boolean In ! boolean Out)
(| Out := cons1() | prod1(In) |)
where boolean istart init false, data init false;
  process cons1 = (? ! Out)
  (| T1  := istart = istart$1 when L1$1
   | L1  := true when (T1 default L2$1) default false
   | L2  := true when not T1 default false
   | Out := data$1 when L2$1
   | L1  ^= L2 ^= istart ^= data
   |) where boolean L1 init true, L2, T1; end;
  process prod1 = (? In !)
  (| data   := In default data$1
   | istart := not istart$1 when ^In default istart$1
   |);
  end;

process archi = (? boolean In ! boolean Out)
(| Out := cons2() | prod2(In) |)
where process cons2 = (? ! Out)
  (| L2  := L1$1 init true
   | Out := data$1 when L3$1
   | L1  := true when L3$1 default false
   | L1  ^= L2 ^= L3 ^= data
   | (L3, data) := recv(L2)
   |) where boolean L1 init true, L2, L3, data; end;
  process prod2 = (? In !)
  (| data := In when L1$1 default data$1
   | T1   := true when ^In when L1$1
   | L1   := true when (not T1 default L3$1) default false
   | L2   := true when T1 default false
   | L3   := send(data$1, L2)
   | L1   ^= L2 ^= L3 ^= data
   |) where boolean T1, L1 init true, L2, L3, data; end;
  end;
end;
```