Polychrony for Formal Refinement-Checking in a System-Level Design Methodology
Jean-Pierre Talpin, Paul Le Guernic, Sandeep Shukla, R.K. Gupta, Frédéric Doucet

To cite this version:

HAL Id: hal-00542156
https://hal.archives-ouvertes.fr/hal-00542156
Submitted on 1 Dec 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Polychrony for Formal Refinement-Checking in a System-Level Design Methodology

Jean-Pierre Talpin\textsuperscript{1}, Paul Le Guernic\textsuperscript{1}, Sandeep Kumar Shukla\textsuperscript{2}, Rajesh Gupta\textsuperscript{3}, Frédéric Doucet\textsuperscript{3} \\
\textsuperscript{1}INRIA/IRISA, \textsuperscript{2}Virgina Tech, \textsuperscript{3}UC San Diego

Abstract

The productivity gap incurred by the rising complexity of system-on-chip design have necessitated newer design paradigms to be introduced based on system-level design languages. A gating factors for widespread adoption of these new paradigms is a lack of formal tool support of refinement based design. A system level representation may be refined manually (in absence of adequate behavioral synthesis algorithms and tools) to obtain an implementation, but proving that the lower level representation preserves the correctness proved at higher level models is still an unsolved problem. We address the issue of formal refinement proofs between design abstraction levels using the concepts of polychronous design. Refinement of synchronous high-level designs into globally asynchronous and locally synchronous architectures is formally supported in this methodology. The polychronous (i.e. multi-clocked) model of the SIGNAL design language offers formal support for the capture of behavioral abstractions for both very high-level system descriptions (e.g. SYSTEMC/SPEC\textsuperscript{C}) and behavioral-level IP components (e.g. VHDL). Its platform, POLYCHRONY, provides models and methods for a rapid, refinement-based, integration and a formal conformance-checking of GAL\textsuperscript{S} hardware/software architectures. We demonstrate the effectiveness of our approach by the experimental, comparative, case study of an even-parity checker design in SP\textsuperscript{EC\textsuperscript{C}}. It highlights the benefits of the formal models, methods and tools provided in POLYCHRONY, in representing functional, architectural, communication and implementation abstractions of the design, and the successive refinements.

1 Introduction

Rising complexities and performances of integrated circuits and systems, shortening time-to-market de-

\footnote{Available from \url{http://www.irisa.fr/expresso/Polychrony}}
clocks are often left unspecified at the functional level, and no choice on a master clock is made at the architectural level. As communication and implementation layers are reached, however, multiple clocks might be a way of life. In the polyynchronous model of computation (MoC), one can actually design a system with partially ordered clocks and refine it to obtain master-clocked components integrated within a multi-clocked architecture, while preserving the functional properties of the original high-level design, thanks to the formal verification methodology provided by the formal theory (model and theorems) of polyynchronous signals.

In the present article, we put the principles of polyynchronous design to work in the context of the emerging high-level languages such as SystemC/SpeCC [11, 19, 20] by studying the refinement of a high-level specification, the even-parity checker (EPC) toward its implementation. Our goal is to derive automatically verifiable conditions on specifications under which refinement-based design principles work. In other words, we seek toward tools and methodologies to allow to take a high-level SystemC/SpeCC specification and to refine it in a semantic-preserving manner into a GaiaS implementation. We focus on a simple case study to illustrate our methodology and we show how the specification of the EPC in SpeCC can be refined toward a GaiaS implementation with the help of POLYCHRONY.

2 An informal introduction to Signal

In Signal, a process $P$ consists of the composition of simultaneous equations over signals. A signal $x \in X$ describes a possibly infinite flow of discretely-timed values $v \in V$. An equation $x = f y$ denotes a relation between a sequence of operands $y$ and a sequence of results $x$ by a process $f \in F$. Synchronous composition $P \mid Q$ consists of considering a simultaneous solution of the equations $P$ and $Q$ at any time. Signal requires three primitive processes: $\text{prev}$, to reference the previous value of a signal in time; $\text{when}$, to sample a signal; and $\text{default}$, to deterministically merge two signals (and provides, e.g., negation $\neg$, equality $\equiv$, etc).

$$P := x = f y \mid P \mid Q \mid P / x$$

$$f \in F \supseteq \{ \text{prev} \} \cup \{ \text{when}, \text{default}, \ldots \}$$

The equation $x = \text{prev} y$ initially defines $x$ by $y$ and then by the previous value of $y$ in time (tags $t_1, t_2, t_3$ denote instants).

$$y : (t_1, v_1) (t_2, v_2) (t_3, v_3) \ldots$$
$$\text{prev}y : (t_1, v) (t_2, v_1) (t_3, v_2) \ldots$$

The equation $x = y$ when $z$ defines $x$ by $y$ when $z$ is true.

$$y : (t_1, v_1) (t_2, v_2) (t_3, v_3) \ldots$$
$$z : (t_2, \#) \ (t_3, \#) \ldots$$
$$y \text{ when } z : (t_2, v_2) \ldots$$

The equation $x = y \text{ default } z$ defines $x$ by $y$ when $y$ is present and by $z$ otherwise.

$$y : (t_2, v_2) (t_3, v_3) \ldots$$
$$z : (t_1, v_1) \ (t_3, v_3) \ldots$$
$$y \text{ default } z : (t_1, v_1) (t_2, v_2) (t_3, v_3) \ldots$$

We exemplify the equational/relation design model of Signal by considering the definition of a counting process: Count. It accepts an input event reset and delivers the integer output val. A local counter, initialized to 0, stores the previous value of val (equation counter := $\text{prev} 0 \text{ val}$). When the event reset occurs, val is reset to 0 (i.e., $0 \text{ when } \text{reset}$). Otherwise, counter is incremented (i.e., $(\text{counter} + 1)$). The activity of Count is governed by the clock of its output val, which differs from that of its input reset: Count is multi-clocked.

```plaintext
process Count = (? \text{event reset} \mid \text{integer val})
| counter := \text{prev} 0 \text{ val}
| val := (0 \text{ when } \text{reset}) \text{ default } (\text{counter} + 1)
where integer counter; end;
```

3 A model of polyynchronous signals

Starting from the model of tagged signals of Lee et al. [12, 6], we give the tagged model of polyynchronous signals [13] for the formal study of protocol properties. We consider a set of boolean and integer values $v \in V$ to represent the operands and results of computations. A tag $t \in T$, denotes an instant. The dense set $T$ is equipped with a partial order relation $\leq$ to denote synchronization and causal relations. The subset $\mathcal{T} \subset T$ of a given process is chosen to be a semi-lattice $(\mathcal{T}, \leq, 0)$. A chain $C \in \mathcal{C}$ is a totally ordered subset of $\mathcal{T}$.

$$t_1 = t_3 \begin{cases} # & \text{if } t_2 = 0 & 1 \\ 0 & 1 & 0 \end{cases}$$

Figure 2. A behavior $b$ as a map from names to partially ordered tags and values
\[
\begin{align*}
[x := \text{pre } y] &= \begin{cases} \{ b \in B \mid \text{pre } y \} \cup \{ b \in B \mid \text{pre } y \} & \text{if } x \in \text{pre } y \\cup \{ 0 \} \\
[x := \text{ywhen } z] &= \begin{cases} \{ b \in B \mid \text{ywhen } z \} \cup \{ b \in B \mid \text{ywhen } z \} & \text{if } x \in \text{ywhen } z \\
[x := \text{ydefault } z] &= \begin{cases} \{ b \in B \mid \text{ydefault } z \} \cup \{ b \in B \mid \text{ydefault } z \} & \text{if } x \in \text{ydefault } z
\end{cases}
\end{cases}
\end{align*}
\]

**Figure 1. Denotation of elementary Signal equations**

**Definition 1 (events, signals and behaviors)** An event \( e \in \mathcal{E} = T \times V \) relates a tag and a value. A signal \( s \in \mathcal{S} = T \to V \) is a partial function relating a chain of tags to a set of values. We write \( \text{tags}(s) \) for the domain of \( s \). A behavior \( b \in B = \lambda \to \mathcal{S} \) is a partial function from signal names \( x \in \lambda \) to signals \( s \in \mathcal{S} \).

We write \( \text{vars}(b) \) for the domain of \( b \) and \( \text{tags}(b) = \cup_{x \in \text{vars}(b)} \text{tags}(b(x)) \) for its tags. Hence, the informal sentence "\( x \) is present at \( t \) in \( b \)" is formally defined by \( t \in \text{tags}(b(x)) \). We write \( b \mid_X \) for the projection of a behavior \( b \) on a set \( X \subseteq \lambda \) of names (i.e. \( \text{vars}(b \mid_X) = X \) and \( \forall x \in X, b(x) = b(x) \) and \( b_X \) for its complementary of \( b \mid_{\text{vars}(b) \setminus X} \). A process \( p \in \mathcal{P} = \mathcal{P}(B) \) is a set of behaviors that have the same domain \( \lambda \) (written \( \text{vars}(p) \)). Synchronous composition \( p \mid q \) is defined by the set of behaviors that extend a behavior \( b \in p \) by the restriction \( c \mid_{\text{vars}(q)} \) of a behavior \( c \in q \) if the projections of \( b \) and \( c \) on \( \text{vars}(p) \cap \text{vars}(q) \) are equal.

\[
p \mid q = \left\{ b \cup c \mid (b,c) \in p \times q, \text{ and } b \mid_{\text{vars}(p) \cap \text{vars}(q)} = c \mid_{\text{vars}(p) \cap \text{vars}(q)} \right\}
\]

**Scalability** is a key concept for engineering systems and reusing components in a smooth design process. A formal support for allowing time scalability in design is given in our model by the so-called stretch-closure property. The intuition behind this relation is to consider a signal as an elastic with ordered marks on it (tags). If it 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 partial order between marks is unchanged. Stretching is a partial-order relation which gives rise to an equivalence relation between behaviors: clock equivalence.

**Definition 2 (clock equivalence)** Formally, a behavior \( c \) is a stretching of \( b \), written \( b \preceq c \), iff \( \text{vars}(b) = \text{vars}(c) \) and there exists a bijection \( f \) on \( T \) that is strictly monotonic (\( \forall t,t' \equiv f(t) < f(t') \)), increasing (\( \forall t \equiv f(t) \)) and satisfies \( \text{tags}(c(x)) = f(\text{tags}(b(x))) \) for all \( x \in \text{vars}(b) \) and \( b(x)(t) = c(x)(f(t)) \) for all \( x \in \text{vars}(b) \) and all \( t \in \text{tags}(b(x)) \). The behaviors \( b \) and \( c \) are stretch-equivalent, written \( b \preceq c \), iff there exists a behavior \( d \) s.t. \( d \sqsubset b \) and \( d \sqsubset c \).

Both relations extend to processes. A process \( p \) is stretch-closed iff for all \( b \in p \), \( c \preceq b \Rightarrow c \in p \). A non-empty, stretch-closed process \( p \) admits a set of strict behaviors (a strict behavior is the \( \preceq \)-minimum of a \( \preceq \)-equivalence class), written \( (p) \preceq_s \), s.t. \( (p) \preceq_s \subseteq p \) (for all \( b \in p \), there is a unique \( c \in (p) \preceq_s \subseteq p \).

**Distribution** To model asynchrony, we consider a weaker relation which discards synchronization relations and allows for comparing behaviors w.r.t. the sequences of values signals hold. The relaxation relation allows to individually stretch the signals of a behavior. Relaxation is a partial-order relation that defines the flow-equivalence relation.

**Definition 3 (flow equivalence)** A behavior \( c \) is a relaxation of \( b \), written \( b \preceq c \), iff \( \text{vars}(b) = \text{vars}(c) \) and for all \( x \in \text{vars}(b) \), \( b_x \preceq c_x \). Two behaviors are flow-equivalent iff their signals hold the same values in the same order. The behaviors \( b \) and \( c \) are flow-equivalent, written \( b \approx c \), iff there exists a behavior \( d \) s.t. \( d \sqsubset b \) and \( d \sqsubset c \).

The \( \approx \)-equivalence classes of a process \( p \) admit strict behaviors, written \( (p) \approx \). We use relaxation to define the meaning of asynchronous composition \( p \parallel q \) (we note \( X = \text{vars}(P), Y = \text{vars}(Q) \) and \( I = X \cap Y \)).

\[
p \parallel q = \left\{ d \mid \exists b \in p, d_x \mid_Y \sqsubseteq b_x \mid_Y, b_x \sqsubset d_x \right\}
\]

**Denotation of Signal in the model of synchrony** The model of polychrony provides a purely relational denotation of SIGNAL (figure 1), consisting of the function \( [] \) that associates a SIGNAL process to the set of its possible behaviors. Notice that the semantics of SIGNAL is closed in the structure of polychronous signals, in that, whenever a process \( P \) (network \( Q \))
has a behavior $b$, written $b \in [P]$, then it admits any stretching $c \geq b$ (relaxation $c \supseteq b$) of $b$, i.e. $c \in [P]$.

**Polychronous design properties** The model of polychronous signals allows to define formal properties that are essential for the component-based design of GALS architectures [13].

*Controlability or input-endochrony* is a key design property. A process is input-endochronous iff, given an external (asynchronous) stimulation of its inputs $I$, it reconstructs a unique synchronous behavior (up to stretch-equivalence). Endochrony denotes the class of processes that are insensitive to (internal and) external propagation delays.

**Definition 4 (controlability)** A process $p$ is endochronous on its input signals $I$ iff $\forall b, c \in p, (b|I)_= = (c|I)_= \Rightarrow b \leq c$.

*Flow-equivalence* offers the right criterion for checking the refinement of a high-level system specification with distributed communication protocols correct. For instance, it is considered in [4] for the refinement-based design of the LTTA protocol in SIGNAL. *Flow-invariance* is the property that ensures that the refinement of a functional specification $p|q$ by an asynchronous implementation $p \parallel q$ preserves flow-equivalence. Formally,

**Definition 5 (flow-invariance)** $p$ and $q$ are flow-invariant iff, for all $b \in p|q$, for all $c \in p \parallel q$, $(b|I)_= = (c|I)_= \Rightarrow b \approx c$ for $I$ the input signals of $p|q$.

In SIGNAL, GALS architectures are modeled as endo-isochronously communicating endochronous components. We say that two endochronous processes $p$ and $q$ are endo-isochronous iff $(p|I) \approx (q|I)$ is endochronous (with $I = \text{vars}(p) \cap \text{vars}(q)$). Endo-isochrony implies flow-invariance and is directly amenable to static verification by the SIGNAL compiler using its clock resolution and control synthesis engine [1]. Automated techniques of distribution using protocol synthesis techniques are implemented in the POLYCHRONY platform [2]: endo-isochronous distribution consists of a causality-aware exchange (duplication) of boolean clocks among interacting components.

Notice that the properties of controllability and flow-invariance introduced in [13], considered in the present study, imply the previously studied properties of IO-endochrony and isochrony of [3] (a process is IO-endochronous iff $\forall b, c \in p, b \approx c \Rightarrow b \leq c$ and two processes are isochronous iff their synchronous and asynchronous compositions have the same traces). Whereas IO-endochrony and isochrony allow non-deterministic (in the aim of modeling distributed reactive systems), input-endochrony and flow-invariance imply determinism (embedded systems and SoC architectures are the target).

Hence, controllability and flow-invariance offers precise, accurate, behavioral-level refinement checking conditions to characterize protocol synthesis, while IO-endochrony and isochrony state global, process-level, relation between synchrony and asynchrony.

**Capturing high-level design using polychrony** Although system-level design languages such as SPEC/C or SYSTEM VERILOG have been introduced as a way to raise the level of abstraction and thereby handling design correctness at a higher level, there is not much research literature that can prove refinement between abstraction levels to be correctness preserving. We propose a program analysis-based representation of system-level models at various abstraction levels in SIGNAL, and then apply the analysis on these SIGNAL models. This will provide us with a technique to formally establish correctness of refinements of higher level representation of designs to lower level implementation.
behavior ones(in unsigned int data, out unsigned int ocount, in event istart, out event idone) {
    void main(void) {
        unsigned int idata, iocount;
        while (1) { wait(istart);
            idata = data; iocount = 0;
            while (idata != 0) { iocount += data & 1;
                idata >>= 1;
            }
            ocount = iocount;
            notify(idone); }}}};

Figure 4. Specification-level design of the EPC in SpecC

In this paper, we do not discuss the compilation of these system level languages in SIGNAL, because for compilation, we need to fix the semantics of the language, which is not properly done yet. However, that is a part of our on-going effort. However, here we assume a semantics, and manually translate the SpecC code into SIGNAL code, and apply our methodology.

In the polychronous design paradigm, one can give a functional-level specification of a system in terms of relations and partially-ordered clocks. A refinement, at the architecture-level, consists of isolating the master-clock of components and of integrating them within multi-clocked architectures, while preserving the functional properties of the original design, thanks to the formal verification of flow-invariance. The main benefit of considering the model of polychronous signals for high-level C-like design languages lies in the formal semantics backbone/platform it provides, on which verification and optimization techniques can then be plugged in.

Our approach to applying the Polychrony model to high-level GALS architectures modeling in C-like design languages (figure 3) consists of automatically synthesizing or capturing the behavioral abstraction or model of a SpecC design as a SIGNAL process. Other formalisms, such as interface automata or algebra could be used. What matters is to choose a formalism in which deciding properties about models (equivalence, bisimulation, etc.) is decidable.

4 A case study: the even parity checker

The polychronous model of the SIGNAL design language offers formal support for the capture of behavioral abstractions for both very high-level system descriptions (e.g. SystemC/SPEC C) and behavioral-level IP components (e.g. VHDL). Its platform, Polychrony, provides formal methods for a rapid, refinement-based, integration and a formal conformance-checking of GALS hardware/software architectures. We focus on a case study that illustrates our methodology by showing how the specification of the EPC in SpecC 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 such architectures. The EPC consists of three functional units (figure 5): an I/O interface process, an even test process and a main ones counting process (gray elements are SpecC-specific).

Figure 5. Functional architecture of the EPC

Specification-level design in SpecC The behavior ones in SpecC (figure 4) determines the parity of an input data received along data. Upon receipt of the start notification, it repeatedly shifts the data until it is 0xed. The output count iocount is sent along ocount and done notified. The behavior even performs the mirror notifications and outputs the final parity check along the Out.

Synchronization mechanisms between threads can easily be modeled in SIGNAL. Suppose we have N elementary threads (i.e. critical sections) communicating via locks. Let us identify each of them by a symbolic datum. A notification consists of setting a lock to true upon request of the notifier. A waiting process checks whether the lock has been notified at the previous in-
process ones = (? integer data; event tick
   ! integer Out; boolean istart, idone)
   (| c := wait[istart](tick)
     | idata := (data default nshift (pre InitData idata)) when c
     | icount := ((pre 0 icount) + xand(idata, 1)) when c
     | ocount := icount when idata=0 when c
       default pre 0 ocount when tick
       notify[done](when c when idata=0)
   ) where integer idata, icount; event c :

process even = (? integer ln, ocount; event tick ! boolean Out,
   data; boolean start, istart, done, idone)
   (| c1 := wait[start](tick)
     | data := ln when c1 default pre InitData data when tick
     | notify[istart](when c1)
     | c2 := wait[done](tick)
     | Out := xand(ocount when c2, 1)=1
       notify[done](when c2)
   ) where event c1, c2 end:

A systematic translation of a specification-level behavior in SIGNAL (for instance that of the thread ones, figure 6) consists, first, of decomposing the syntactic structure of the SPECC program into an intermediate representation that renders the imperative structure of the original program together with its most characteristic features (use of locks, interrupts, etc). In this structure, each thread consists of a sequence of blocks (critical sections) delimited by wait and notify synchronization statements.

A related work, reported in [16], consists of a POLYCHRONY plugin which translates multi-threaded real-time JAVA programs in SIGNAL. In this tool, the JVM real-time runtime system is modeled using the ARINC library of SIGNAL [9]. This library gives a generic model of real-time operating systems APIs in SIGNAL. The translator allows for entirely modeling the behavior of a multi-threaded real-time JAVA component and to reuse and reconfigure its package of real-time thread classes according to a given target architecture.

Within such blocks, basic control structures are then encoded. A method call or a basic operation, e.g. $x = y + 1$ with $y$ declared as int $y = n$, is encoded by an equation, e.g. either $x = \text{pre} n y + 1 \text{wherec}$ (when $y$ references a value computed during the previous transition in this block) or $x = y + 1 \text{wherec}$ (if it has already been computed in the same transition), conditioned by an activation clock $c$. A conditional statement, e.g. if $x \text{then} P \text{else} Q$, is encoded by constraining the clock of $P$ by $x$ and that of $Q$ by not $x$. While loops are encoded by over-sampling. Interrupts are rendered by events. An interrupt conditions the activation clock of subsequent equations in the control flow graph; if it escapes the scope of the method in which it is raised, it becomes an output signal of the process that encodes the method in order to propagate in the context of use of that method.

In the specification-layer of the behavior ones, there is only one critical section, delimited by a wait and a notify. It is encoded much like the polychronous specification of the previous section, with the noticeable addition of the wait-notify protocol and the simulation scheduling tick. The process is activated when it obtains the lock on istart. Then, at its own rate (now conditioned by the clock c), it determines the count. When it is finished, it sends the notification.

A polychronous model of the EPC  By contrast, the polychronous design-layer of SIGNAL could start at a much higher design abstraction-level, without making any implicit (simulation) architecture choices. By contrast, at the SPECC specification-level, the system is already distributed into a set of behaviors (i.e. threads) which interact via shared variables and wait and notify synchronization mechanisms.

At the polychronous design layer of the EPC in SIGNAL, we put these implementation details off until later refinement stages and focus on its most characteristic threads, ones and even (figure 7). The process ones consists of an iterative computation of the parity, implemented using over-sampling: a local signal idata, reset upon receipt of an inputdata, is iteratively scanned to count the number of bits set to 1 (signal icount). When
Validation of the polychrony-to-specification
design refinement. Checking that the specification-
level design of the EPC is a correct refinement of the
polychronous SIGNAL specification amounts to checking
that these two designs are flow-invariant to the intro-
duction of the wait-notify protocol (figure 8, a box
stands for a register).

Figure 7. Polychronous model of the EPC-core

Figure 8. Refinement of the polychronous
model by the specification model

The validation of this design refinement amounts to
proving that, for all behaviors \( b \) and \( c \) of the poly-
chronous and specification layers of the EPC, noted
\( P_{\text{ones}} \) and \( S_{\text{ones}} \), flow equivalence of the input signal
In, i.e. \( b|_{\text{in}} \approx c|_{\text{in}} \) implies flow equivalence of the signal

\[
\text{constant integer InitData} = -32768;
\]

\[
\text{process ones} = (? \text{ integer data ! integer ocount }) \% \text{ boolean istart, idone} \%
\]

\[
\begin{align*}
& ( \mid \text{idata} := \text{data default rhsift (pre InitData idata)} \% \text{ istart} \leftarrow \text{data} \% \\
& \mid \text{icount} := \text{(pre 0 ocount) + xand (idata, 1)} \% \text{idone} \leftarrow \text{ocount} \%
\end{align*}
\]

\[
\text{process even} = (? \text{ integer ln, ocount ! boolean Out, data}) \% \text{ boolean start, istart, done, idone} \%
\]

\[
\begin{align*}
& ( \mid \text{data} := \text{ln} \% \text{ istart} \leftarrow \text{ln} \leftarrow \text{start} \% \\
& \mid \text{Out} := \text{xand (ocount, 1) = 1}) \% \text{ ocount} \leftarrow \text{done} \leftarrow \text{idone} \%
\end{align*}
\]

\[
\text{function rhsift} = ( ? \text{ i1 ! i2 }) \% \text{ spec (i1 \leftarrow i2 )} \%
\]

\[
\text{function xand} = ( ? \text{ i1, i2 ! i3 }) \% \text{ spec (i1 \leftarrow i2 \leftarrow i3)} \%
\]

\[
\text{pragma C_{CODE} "&i2 = &i1 >> 1" end pragmas;}
\]

\[
(1) : \forall b \in [P_{\text{ones}}], \forall c \in [S_{\text{ones}}], b|_{\text{in}} \approx c|_{\text{in}} \Rightarrow b|_{\text{out}} \approx c|_{\text{out}}
\]

However, the polychronous model of the EPC only dif-
fers from the specification layer by the introduction
of a wait-notify protocol, which implements a generic
synchronization scheme \( P \) of the polychronous model.
The matching pattern \( S \) of the protocol in the spec-
ification layer consists of the insertion of delays in the
transmission of data due to the wait-notify toggle.

\[
P \equiv (\text{data} : = \text{start} \mid \text{idata} : = \text{data} \mid \text{start} \leftarrow \text{ln} \mid \text{data} : = \text{ln})
\]

\[
S \equiv \left( \begin{array}{c}
\text{c} : = \text{wait}\{\text{start}\}(\text{clock}) \\
\mid \text{idata} : = \text{data when c}
\mid \text{notify}\{\text{start}\}(\text{clock}) \\
\mid \text{data} : = \text{ln default pre InitData data when clock}
\end{array} \right)
\]

Hence, proving equation (1) reduces to showing that
the refinement of the polychronous synchronization
scheme \( P \) by the wait-notify synchronization protocol
\( S \) preserves flow-equivalence, as specified by equa-
tion (2). Indeed, notice that (2) implies (1).

\[
(2) : \forall b \in [P], \forall c \in [S], b|_{\text{in}} \approx c|_{\text{in}} \Rightarrow b|_{\text{data}} \approx c|_{\text{data}}
\]

In a similar manner as for loosely time-triggered archi-
tectures, studied in [4], this property is amenable to
symbolic model checking using the tool SIGNAL [15].
Verification is implemented by specifying the corre-
sponding property in SIGNAL (figure 10), simulating
the input In and idata using, e.g. booleans (providing
the corresponding implementations of the parameters
xand, rhsift and InitData) and by calculating that its
output (the invariant) never becomes false. A buffer
is used to avoid altering synchronizing signals between
the models \( P \) and \( S \). Flow-invariance modulo buffer
implies flow-invariance.

Architecture-layer design refinement. The en-
coding of the even-parity checker demonstrates the
channel ChMP () {
    unsigned int data; event eReady, eAck;
    bool ready = false, ack = false;
    void send (unsigned int In) {
        data = In;
        ready = true;
        notify (eReady);
        while (lack) wait (eAck);
        ready = false;
        notify (eReady);
        while (ack) wait (eAck);
    }
    unsigned int recv () {
        unsigned int rdata;
        while (ready) wait (eReady);
        rdata = data;
        ack = true;
        notify (eAck);
        while (ready) wait (eReady);
        ack = false;
        notify (eAck);
        return rdata;
    }
}

process observer = (? boolean i! boolean invariant)
| (i invariant := buffer[P(buffer(i))] = buffer[S(buffer(i))])
| where process buffer = (? boolean i! boolean o)
| (o := Current (i) | Alternate (i, o))
| process Current = (? boolean i! boolean o)
| (o := (i cell "o init false) when "o))
| process Alternate = (? boolean i! boolean o)
| (i " := when flop
| o « := when not flop
| flop := not (pre true flop)
| j) where boolean flop;
end;

clock | . . . . . clock | . . . .
| i 1 0 1 ⇒ i 1 0 0 1

Figure 9. Implementation of an architecture-level channel in SpecC

capability of SIGNAL to give a polychronous model of components for specification-level SpecC designs. This level of abstraction (polychrony) allows for a better decoupling of the specification of the system under design from early architecture mapping choices. It additionally allows for an optimized recombination of behaviors. For instance, the SIGNAL compiler could merge the behaviors IO and even using its clock resolution engine. In comparison, the typical SpecC designflow starts with the capture of IP-blocks represented as C functions and then does an automatic partitioning according to an appropriate cost function. After partitioning, 2-way handshake protocols (or appropriate HW-SW protocols) are inserted between the functional units.

Consider the architecture layer of the EPC (figure 9). We now have two behaviors, ones and even that communicate asynchronously via the CHMP channel. Modeling the architecture-layer refinement of the EPC in SIGNAL consists of modeling the double handshake protocol implemented by the methods send and recv of the CHMP channel, which obey the message sequence depicted on the right. The model of send and recv in SIGNAL (figure 11) is obtained in the very same way as for the behaviors even and ones of the specification level, except that the ready and ack flags correspond to state variables (declared at the same lexical level as send in the CHMP module). By installing the channel process between producer and consumer, we obtain a desynchronization of the transmission between the In and Out processes (in addition to a desynchronization of locks, obtained in the specification-layer).

process send = (? integer In; event clock !)
| (i cl := when (event In) when clock
| data := In when cl
| ready := true when cl
| default false when cl2 when not (pre false ack)
| default pre false ready when clock
| notify (eReady) | cl)
| (c2 := wait (eAck) | (when pre false ack when clock)
| notify (eReady) | (when c2 when not (pre false ack))
| (c3 := wait (eAck) | (when not (pre false ack) when clock)
| where event c1, c2, c3 end;

process recv = (? event clock ! integer rdata)
| (c1 := wait (eReady) | (when not (pre true ready) when clock)
| rddata := preInitData data when c1 when pre true ready
| ack := true when c1 when pre true ready
| default false when c2 when not (pre true ready)
| default pre true ready when clock
| notify (eAck) | (when c1 when pre true ready)
| (c2 := wait (eReady) | (when pre true ready when clock)
| notify (eAck) | (when c2 when (not pre true ready)))
| (c3 := wait (eReady) | (when pre true ready when clock)
| notify (eAck) | (when c2 when (not pre true ready)))
| where event c1, c2 end;

Figure 10. Refinement-checking observer

Figure 11. Model of the architecture-level channels in SIGNAL

Validation of the specification-to-architecture refinement. Showing that the refinement of the EPC
from the specification level \( S_{\text{ones}} \) to the architecture level \( A_{\text{ones}} \) (figure 12) is correct amounts to checking flow-invariance between the two designs.

**Figure 12. Refinement of the specification by an architecture layer**

It is amenable to symbolic model checking in SIGNAL using the criterion (3) that, in a similar manner as (1), states the flow-equivalence of the specification and architecture models \( S_{\text{ones}} \) and \( A_{\text{ones}} \).

\[ (3): \forall b \in [S_{\text{ones}}], \forall c \in [A_{\text{ones}}], b|_{\text{in}} \approx c|_{\text{in}} \Rightarrow b|_{\text{out}} \approx c|_{\text{out}} \]

In the same manner as for the polychrony-to-specification refinement, proving (3) reduces to showing that the desynchronization protocol introduced by the channel module ChMP preserves flow equivalence between the original specification layer and the final architecture layer. This amounts to showing that the specification model \( S \) is flow-equivalent to the process \( A \) in the architecture model.

\[
S \equiv (\text{data} := (\text{ln when cdefault pre InitData data}) \text{ when clock}) \\
A \equiv (\text{data} := \text{recv}(\text{clock}) \mid \text{send}(\text{ln, clock}))
\]

Showing that \( A \) is flow-equivalent to \( S \) is amenable to symbolic model checking by specifying the property (4) in SIGNAL (simulating the input \( \text{ln} \) and output \( \text{data} \) using booleans). The tool SIGNAL allows to prove that the corresponding invariant never becomes false. Notice again that (4) implies (3).

\[ (4): \forall b \in [S], \forall c \in [A], b|_{\text{in}} \approx c|_{\text{in}} \Rightarrow b|_{\text{data}} \approx c|_{\text{data}} \]

**Communication-layer design refinement** The communication layer of the EPC (figure 13) consists of a data-type refinement of the ChMP channel and of the implementation of the ChMP as a bus. It consists of the decomposition of the methods send and receive into sub-processes, allowing for the isolation of the bus read and write methods.

**Figure 13. Communication-level bus in SPECC**

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;
    }
}

Showing this refinement correct (figure 14) 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.

**Figure 14. Refinement of an architecture-level channel by a communication-level bus**

R TL-layer design refinement  The RTL layer of the EPC (figure 15) consists of 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. This translation closely corresponds the SIGNAL's encoding of the EPC into blocks (critical sections).

In SIGNAL, this refinement (figure 16) corresponds to an implementation-clock accurate, endochronous,
behavior ones(invent clk, in unsigned bit[0:0] rst, in unsigned bit[31:0] inport, out unsigned bit[31:0] outport, . . . ) {
    void main(void) {
        unsigned bit[31:0] data, ocount;
        enum state {S0, S1, S2, S3} state = S0;
        while (1) {
            wait(clk);
            if (rst == 1b) state = S0;
            switch (state) {
                case S0: done = 0b;
                    ack_i_start = 0b;
                    if (start == 1b) state = S1 else state = S0;
                    break;
                case S1: ack_i_start = 1b;
                    data = inport;
                    ocount = 0;
                    state = S2;
                    break;
                case S2: ocount = ocount + data & 1;
                    data = data >> 1;
                    if (data == 0) state = S3 else state = S2;
                    break;
                case S3: outport = ocount;
                    done = 1b;
                    if (ack_i_done == 1b) state = S0 else state = S3;
                    break;
            }
        }
    }
}

Figure 15. RTL-level implementation of the EPC-core in SpecC

model of the EPC. The RTL model can be regarded as a temporal refinement of the SIGNAL model, in which the master clock is stretched in such a way as to allow for a single sentence of the SpecC design to be simulated at a time.

**Toward an integration platform** In the aim of automating the above process within a versatile component integration platform, the use of POLYCHRONY as a refinement-checking tool provides the required support by using controller synthesis techniques [14]. Whereas model-checking consists of proving a property correct w.r.t. the specification of a system, control synthesis consists of using this property as a control objective and to automatically generate a coercive process that wraps the initial specification so as to guarantee that the objective is an invariant. To this end, we aim at using POLYCHRONY as a semantic platform for the SYSTEMC design tool BALBOA [17, 8], by using SIGNAL as an internal representation of behavioral type descriptions for SYSTEMC components, allowing for a correct by construction component-based design of high-level, system-on-chip SYSTEMC designs, and the systematic synthesis of interface protocols between components.

5 Related works

The (multi-clocked) notion of flow-equivalence relates to the (single-clocked) notion of latency-equivalence of Carloni et al. [6]. Two signals are latency-equivalent iff they present the same values in the same order. Flow-invariance casts the property of flow-equivalence to the general context of design refinement checking, whereas Carloni et al. concentrate with latency-equivalence on the correct-by-construction as-
techniques, implemented in the SIGNAL compiler.

References


