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:
Polychrony for Formal Refinement-Checking in a System-Level Design Methodology

Jean-Pierre Taldrin¹, Paul Le Guernic¹, Sandeep Kumar Shukla²,
Rajesh Gupta³, Frédéric Doucet³
¹ INRIA/IRISA, ² Virginia Tech, 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) 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 GaLs hardware/software architectures. We demonstrate the effectiveness of our approach by the experimental, comparative, case study of an even-parity checker design in SPEC. 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 demands for electronic equipments, growing installed bases of intellectual property, requirements for adapting existing IPs with new services, all stress high-level design as a prominent research topic and call for the development of appropriate methodological solutions. In this aim, system design based on the so-called “synchronous hypothesis” 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 [13] and languages [5] 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 today’s multi-Giga-hertz SoC designs, the clock period is so small that clocking across the chip in a synchronous manner is a challenge. Hence newer SoC designs need to be globally asynchronous and locally synchronous (GaLs). The relational model of the POLYCHRONY¹ design platform [13] goes beyond the domain of purely synchronous circuits to embrace the context of architectures consisting of synchronous circuits and desynchronization protocols: GaLs architectures. The unique features of this model are to provide the notion of polychrony, the capability to describe multi-clocked (or partially clocked) circuits and systems; and to support formal design refinement, from the early stages of requirements specification, to the later stages of synthesis and deployment, and by using formal verification techniques.

In practice, a multi-clocked system description is often the representation or the abstraction of an asynchronous system or of a GaLs architecture. In system-level design, the asynchronous implementation of a system is obtained through the refinement of its description toward hardware-software co-design. However,

¹Available from http://www.irisa.fr/epresoso/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 polychronous 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 polychronous signals.

In the present article, we put the principles of polychronous 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 GALS 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 GALS 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 \( x \in \mathcal{X} \) describes a possibly infinite flow of discretely-timed values \( v \in \mathcal{V} \). 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 \( P \parallel Q \) consists of considering a simultaneous solution of the equations \( P \) and \( Q \) at any time. SIGNAL requires three primitive processes: \( \text{pre} \), 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 not, equality eq, etc).

\[
P := \text{pre } x = f y \mid P \parallel Q \mid P / x
f \in F \ni \{ \text{pre } v \mid v \in \mathcal{V} \} \cup \{ \text{when}, \text{default}, \ldots \}
\]

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

\[
y : (t_1, v_1) (t_2, v_2) (t_3, v_3) \ldots
\text{pre } y : (t_1, v) (t_2, v_1) (t_3, v_2) \ldots
\]

The equation \( x = y \text{ when } z \) defines \( x \) by \( y \) when \( z \) is true.

\[
y : (t_1, v_1) (t_2, v_2) (t_3, v_3) \ldots
z : (t_2, \#) (t_3, \#) (t_4, \#) \ldots
y \text{ when } z : (t_2, v_2) \ldots
\]

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

\[
y : (t_2, v_2) (t_3, v_3) \ldots
z : (t_1, v_1) (t_3, w_3) \ldots
y \text{ default } z : (t_1, v_1) (t_2, v_2) (t_3, v_3) \ldots
\]

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

```
process Count = (? event reset ! integer val)
  counter := pre 0 val
  \{ val := (0 when reset) default (counter + 1) \}
  where integer counter; end;
```

3 A model of polychronous signals

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

![Figure 2. A behavior b as a map from names to partially ordered tags and values](image-url)

\[
t_1 = t_3 \{ 0 \rightarrow 0, 1 \rightarrow 0, \# \rightarrow 0 \}
\]

\[
t_3 \not\leq t_4
\]
\[ \begin{align*}
[x := \text{pre } y] & \quad \begin{cases}
\{ b \in B \mid \forall \xi \in \text{var}(y), \text{tags}(b(x)) = \text{tags}(b(y)) = C \setminus \emptyset, b(x)(\text{min}(C)) = v \\
\forall t \in C \setminus \text{min}(C), b(x)(t) = b(y)(\text{pred}(t))
\end{cases} \\
[x := y\text{when } z] & \quad \begin{cases}
\{ b \in B \mid \forall \xi \in \text{var}(y, z), \text{tags}(b(x)) = t \in \text{tags}(b(y)) \cap \text{tags}(b(z)), b(x)(t) = \emptyset \\
\forall t \in \text{tags}(b(x)), b(x)(t) = b(y)(t)
\end{cases} \\
[x := y\text{default }] & \quad \begin{cases}
\{ b \in B \mid \forall \xi \in \text{var}(y, z), \text{tags}(b(x)) = t \in \text{tags}(b(y)) \cup \text{tags}(b(z)), b(x)(t) = \emptyset \\
\forall t \in C, b(x)(t) = \emptyset \text{ if } t \in \text{tags}(b(y)) \text{ then } b(y)(t) \text{ else } b(z)(t)
\end{cases}
\end{align*} \]

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

**Definition 1 (events, signals and behaviors)** An event \( e \in \mathcal{E} = \mathcal{T} \times \mathcal{V} \) relates a tag and a value. A signal \( s \in \mathcal{S} = \mathcal{T} \rightarrow \mathcal{V} \) 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 \( b \in B = \mathcal{X} \rightarrow \mathcal{S} \) is a partial function from signal names \( x \in \mathcal{X} \) to signals \( s \in \mathcal{S} \).

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

\[
p | q = \left\{ b \cup c \left| \begin{array}{l}
(b, c) \in p \times q, \\
\text{tags}(b|x)p) \cap \text{vars}(q) = \text{vars}(p) \cap \text{vars}(q)
\end{array} \right. \right\}
\]

**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 \( b \leq c \), iff \( \text{vars}(b) = \text{vars}(c) \) and there exists a bijection \( f \) on \( \mathcal{T} \) that is strictly monotonic (\( \forall t, t' < t \Rightarrow f(t) < f(t') \)), increasing (\( \forall t, t \leq f(t) \)) 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 \( b \leq c \), iff there exists a behavior \( d \) s.t. \( d \leq b \) and \( d \leq c \).

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

**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 \( b \equiv c \), iff \( \text{vars}(b) = \text{vars}(c) \) and for all \( x \in \text{vars}(b) \), \( b|_{x} \leq c|_{x} \). 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. \( d \equiv b \) and \( d \equiv c \).

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

\[
p \parallel q = \left\{ d \left| \exists b \in p, d|_{X \setminus Y} \leq b|_{X \setminus Y}, b|_{I} \equiv d|_{I} \right. \right\}
\]

**Denotation of Signal in the model of polychrony** The model of polychrony provides a purely relational denotation of SIGNAL (figure 1), consisting of the function \( \boxdot \) 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 $b \in \mathcal{P}$, then it admits any stretching $c \geq b$ (relaxation $c \leq b$) of $b$, i.e. $c \in \mathcal{P}$.

**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].

*Controllability 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 (controllability)** A process $p$ is endochronous on its input signals $I$ iff $\forall b, c \in p, (b \varphi)_I = (c \varphi)_I \Rightarrow b \leq c$.

*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 LITTA 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 $c \in p \parallel q$, $(b \varphi)_I = (c \varphi)_I$ implies $b \approx c$ for $I$ the input signals of $p|q$.

In SIGNAL, GALs architectures are modeled as endo-isochronous communicating endochronous components. We say that two endochronous processes $p$ and $q$ are endo-isochronous iff $(p \varphi)|((q \varphi))$ 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 [3], 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 SystemC, SpecC or System Verilog have been introduced as a way to raise the level of abstraction and there by 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, icount;
        while (1) {
            wait(idata);
            idata = data; icount = 0;
            while (idata != 0) {
                icount += data & 1;
                idata >>= 1;
            }
            ocount = icount;
            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/SPECc) 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 IO 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 0ed. The output count icount 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)
     | data := (data default nshift (pre InitData idata)) when c
     | icount := ((pre 0 icount) + xand(idata, 1)) when c
     | ocount := icount when data=0 when c
               default pre 0 ocout when tick
               notify(idone)(when c when idata=0)
   ) where integer idata, icount; event c :

process even = (? integer ln; 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(idone)(tick)
   | Out := xand (ocount when c2, 1)=1
   | notify(done)(when c2)
  ) where event c1, c2 end;

Figure 6. Corresponding model of the specification-layer in SIGNAL.

stant and is available at its own request. If so, the event acquired is present and the lock becomes false. The model of wait/notify makes use of partial equations. In SIGNAL, a partial equation $x := f(y)$ when $c$ partially defines $x$ by $f(y)$ at the clock $c$. Composed to $x := f(z)$ when $d$, it is equivalent to the equation $x := f(y)$ when $c$ default $f(z)$ when $d$ if the exclusion of the clocks $c$ and $d$, denoted by the constraint $c$ # $d$, can be checked satisfiable by the clock resolution engine of the compiler (meaning that the assignment to $x$ is deterministic). We note $x := f(c)(y)$ for a call to a SIGNAL process of module $f$ that takes the static parameters $c$.

process notify = {boolean lock} { ? event request ! }
  (| lock := true when request !
   )
process wait = {boolean lock} { ? event request ! event ack)
  (| lock := when request when pre false lock
     | lock := false when ack |)

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 $x = pre n y + 1$ where (when $y$ references a value computed during the previous transition in this block) or $x = y + 1$ where (if it has already been computed in the same transition), conditioned by an activation clock $c$. A conditional statement, e.g. if $x$ then $P$ 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
constant integer InitData = -32768;

\[
\text{process ones} = (\text{? integer data ! integer ocount }) \\
| \text{idata} := \text{data default rshift (pre InitData idata)} \\
| \text{iocount} := (\text{pre 0 iocount} + \text{and (data, 1)}) \\
| \text{ocount} := \text{iocount when idata=0} \\
| \text{where} \quad \text{integer idata, iocount end;} \\
\]

\[
\text{process even} = (\text{? integer ln, ocount ! boolean Out, data }) \\
| \text{data} := \text{ln} \\
| \text{Out} := (\text{xand (iocount, 1) = 1}) \\
| \;
\]

\[
\text{function rshift} = (\text{? i1 ! i2 }) \quad \text{spec} (\text{? i1' = i2'}) \quad \text{pragmas C\_CODE "&i2 = &i1 >> 1" end pragmas;} \\
\text{function xand} = (\text{? i1, i2 ! i3 }) \quad \text{spec} (\text{? i1' = i2' = i3'}) \quad \text{pragmas C\_CODE "&i3 = &i1 & &i2" end pragmas;}
\]

\[
\text{Figure 7. Polychronous model of the EPC-core}
\]

finished (i.e. when idata is 0), iocount is returned along the output signal ocount. Auxiliary notification signals of the original SPECC specification of the EPC, (e.g. start, done), appear behind comments as they are not necessary at this level of specification. Notice that the process ones is endochronous. The consumer process even simply reads the count sent along the ocount signal and checks whether it is even. The functions rshift and xand are available from an external C library. They are embedded in SIGNAL using interface specifications.

**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 introduction of the wait-notify protocol (figure 8, a box \( \square \) stands for a register).

\[
\text{Figure 8. Refinement of the polychronous model by the specification model}
\]

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

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

However, the polychronous model of the EPC only differs 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 specification layer consists of the insertion of delays in the transmission of data due to the wait-notify toggle.

\[
P \equiv (\text{data} = \text{start} | \text{idata} := \text{data} | \text{start'} = \text{ln} | \text{data} := \text{ln}) \\
S \equiv (\begin{align*}
& (c ::= \text{wait[\text{start}]}(\text{clock}) ) \\
& (\begin{align*}
& (\text{idata} := \text{data when c} \\
& \text{notify[\text{start}]}(\text{clock}) ) \\
& (\text{data} := \text{ln default pre InitData data when clock})
\end{align*})
\end{align*})
\]

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 equation (2). Indeed, notice that (2) implies (1).

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

In a similar manner as for loosely time-triggered architectures, studied in [4], this property is amenable to symbolic model checking using the tool SIGALI [15]. Verification is implemented by specifying the corresponding property in SIGNAL (figure 10), simulating the input \( \text{ln} \) and \( \text{idata} \) using, e.g. booleans (providing the corresponding implementations of the parameters \( \text{xand}, \text{rshift} \) and \( \text{InitData} \)) 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 encoding 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 (!ack) 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;
    }
}

Figure 9. Implementation of an architecture-level channel in SPEC-C

clock

process observer = (? boolean i ! boolean invariant)
    (i invariant := bufferP (buffer (i))) = bufferS(buffer (i))
    ) where process buffer = (? boolean i ! boolean o)
        (i o := Current (i) | Alternate (i, o))
    process Current = (? boolean i ! boolean o)
        (i o := (i cell "o init false" when "o")
    process Alternate = (? boolean i ! boolean o)
        (i ~ := when flop
            o ~ := when not flop
            flop := (pre true flop)
        ) where boolean flop:
end;

clock

process send = (? integer In; event clock !)
    (c1 := when (event In) when clock
        data := In when cl
        ready := true when cl
            default false when c2 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)
        rdata := pre InitData data when c1 when pre true ready
        ack := true when cl when pre true ready
            default false when c2 when not (pre true ready)
            default pre false ack 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))
    ) where event c1, c2 end;

Figure 10. Refinement-checking observer

capability of SIGNAL to give a polychronous model of components for specification-level SPEC-C designs. This level of abstraction (polychrony) allows for a better decoupling of the specificaiton 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 SPEC-C designflow starts with the capture of IP-blocks represented as C functions and then does an automatic partitioning according to 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 !)
    (c1 := when (event In) when clock
        data := In when cl
        ready := true when cl
            default false when c2 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)
        rdata := pre InitData data when c1 when pre true ready
        ack := true when cl when pre true ready
            default false when c2 when not (pre true ready)
            default pre false ack 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))
    ) where event c1, c2 end;

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.

![Diagram](image)

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

It is amenable to symbolic model checking in **S**i**G**a**L**i 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 $A_{\text{ones}}$.

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

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.

$$S \equiv (\text{data} := (\text{in when cdefault pre InitData data}) \text{ when clock})$$

$$A \equiv (\text{data} := \text{recv}(\text{clock}) \text{ when clock})$$

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

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

**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 $\text{send}$ and $\text{recv}$ into sub-processes, allowing for the isolation of the bus $\text{read}$ and $\text{write}$ methods.

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.waitval(1);  
  }  
  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 $\text{send}$ and $\text{recv}$ are flow-equivalent to the methods $\text{read}$ and $\text{write}$ of the bus model. The control structure of the bus model in **S**i**G**a**L**i is identical to that of the channel, except for the implementation of the input/output integer signals as bit-vectors.

**Figure 13. Communication-level bus in **S**pecC**

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

**RTL-layer design refinement** The RTL layer of the EPC (figure 15) consists of the introduction of a master clock $\text{clk}$ and of a reset signal $\text{rst}$ together with the conversion of the EPC communication-layer specification into finite-state machine code. This translation closely corresponds the **S**i**G**a**L**i’s encoding of the EPC into blocks (critical sections).

In **S**i**G**a**L**i, 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 == 1'b) state = S0;
      switch (state) {
        case S0: done = 0'b;
          ackＪstart = 0'b;
          if (start == 1'b) state = S1 else state = S0;
          break;
        case S1: ackＪstart = 1'b;
          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 = 1'b;
          if (ackＪdone == 1'b) 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 PolyClock 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 PolyClock 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-

semblly of existing IPs with pre-defined elementary protocol bricks.

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 [18] and study the refinement-based design of GALS architectures starting from polychronous specifications captured from heterogeneous elementary components.

6 Conclusion

We have put a polychronous design model to work for the refinement of a high-level even-parity checker in SpecC from the early stages of its functional specification to the late stages of its hardware/software GALS implementation. We have demonstrated the effectiveness of this approach by showing in what respects and at which critical design refinement stages formal verification and validation support was needed, highlighting the benefits of using the tool PolyClock in that design chain. The novelty of integrating PolyClock in a high-level design tool-chain lies in the formal support offered by the former to automate critical and complex design verification and validation stages yielding a correct-by-construction system design and refinement in the latter. Polychronous design allows for an early requirements capture and formal compositional and formally-checked transformational refinements, automating the most difficult design steps toward implementation using efficient clock resolution and synthesis.
techniques, implemented in the SIGNAL compiler.

References


