HAL
open science

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:

Jean-Pierre Talpin, Paul Le Guernic, Sandeep Shukla, R.K. Gupta, Frédéric Doucet. Polychrony for Formal Refinement-Checking in a System-Level Design Methodology. Third International Conference on Application of Concurrency to System Design (ACSD '03), Jun 2003, Guimarães, Portugal. pp.9-19. hal-00542156

HAL Id: hal-00542156

https://hal.science/hal-00542156v1

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[1], Paul Le Guernic[1], Sandeep Kumar Shukla[2], Rajesh Gupta[3], Frédéric Doucet[3]
[1] Inria/Irisa, [2] Virgina 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/SpecC*) 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 demonstrates the effectiveness of our approach by the experimental, comparative, case study of an even-parity checker design in* SpecC*. 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 Polychrony[1] 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,

[1] Available from `http://www.irisa.fr/espresso/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 $\boldsymbol{x} = f\boldsymbol{y}$ denotes a relation between a sequence of operands $\boldsymbol{y}$ and a sequence of results $\boldsymbol{x}$ 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: pre, to reference the previous value of a signal in time; when, to sample a signal; and default, to deterministically merge two signals (and provides, e.g. negation not, equality eq, etc).

$$P ::= \boldsymbol{x} = f\,\boldsymbol{y} \mid P \parallel Q \mid P \,/\, x$$
$$f \in F \supseteq \{\,\mathsf{pre}\,v \mid v \in \mathcal{V}\} \cup \{\,\mathsf{when}\,,\ \mathsf{default}\,,\ \ldots\}$$

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

$$\begin{array}{rl} y: & (t_1, v_1)\ (t_2, v_2)\ (t_3, v_3) \ldots \\ \mathsf{pre}\ v\,y: & (t_1, v)\ \ (t_2, v_1)\ (t_3, v_2) \ldots \end{array}$$

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

$$\begin{array}{rlll} y: & (t_1, v_1)\ (t_2, v_2) & (t_3, v_3) & \ldots \\ z: & \quad\ \ (t_2, tt) & (t_3, ff)\ (t_4, tt) & \ldots \\ y\ \mathsf{when}\ z: & \quad\ \ (t_2, v_2) & & \ldots \end{array}$$

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

$$\begin{array}{rlll} y: & & (t_2, v_2)\ (t_3, v_3) & \ldots \\ z: & (t_1, v_1) & \quad\ \ (t_3, w_3) & \ldots \\ y\ \mathsf{default}\ z: & (t_1, v_1)\ (t_2, v_2) & (t_3, v_3) & \ldots \end{array}$$

We exemplify the equational/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 val. A local counter, initialized to 0, stores the previous value of val (equation counter := pre 0 val). When the event reset occurs, 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 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 $t \in \mathbb{T}$, denotes an instant. The dense set $\mathbb{T}$ is equipped with a *partial order* relation $\leq$ to denote synchronization and causal relations. The subset $\mathcal{T} \subset \mathbb{T}$ 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 $\mathbb{T}$.

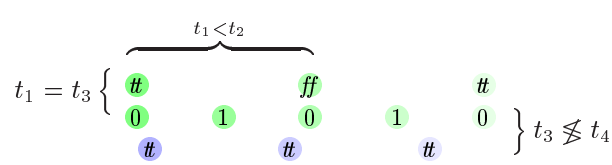

Figure 2. A behavior b as a map from names to partially ordered tags and values

$$
\begin{array}{rl}
[\![x := \mathsf{pre}\, v\, y]\!] = & \left\{ b \in \mathcal{B}|_{x,y} \;\middle|\; \begin{array}{l} \mathrm{tags}(b(x)) = \mathrm{tags}(b(y)) = C \in \mathcal{C} \setminus \emptyset,\ b(x)(\min(C)) = v \\ \forall t \in C \setminus \min(C),\ b(x)(t) = b(y)(\mathrm{pred}_C(t)) \end{array} \right\} \cup \{0|_{x,y}\} \\
[\![x := y\, \mathsf{when}\, z]\!] = & \left\{ b \in \mathcal{B}|_{x,y,z} \;\middle|\; \begin{array}{l} \mathrm{tags}(b(x)) = \{t \in \mathrm{tags}(b(y)) \cap \mathrm{tags}(b(z)) \,|\, b(z)(t) = \mathsf{tt}\} \\ \forall t \in \mathrm{tags}(b(x)),\ b(x)(t) = b(y)(t) \end{array} \right\} \\
[\![x := y\, \mathsf{default}\, z]\!] = & \left\{ \{b \in \mathcal{B}|_{x,y,z} \;\middle|\; \begin{array}{l} \mathrm{tags}(b(y)) \cup \mathrm{tags}(b(y)) = \mathrm{tags}(b(x)) = C \in \mathcal{C} \\ \forall t \in C,\ b(x)(t) = \mathrm{if}\ t \in \mathrm{tags}(b(y))\ \mathrm{then}\ b(y)(t)\ \mathrm{else}\ b(z)(t) \end{array} \right\}
\end{array}
$$

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 $s \in \mathcal{S} = \mathcal{T} \rightharpoonup \mathcal{V}$ *is a partial function relating a* chain *of tags to a set of values. We write* $\mathrm{tags}(s)$ *for the domain of* s*. A* behavior $b \in \mathcal{B} = \mathcal{X} \rightharpoonup \mathcal{S}$ *is a partial function from signal names* $x \in \mathcal{X}$ *to signals* $s \in \mathcal{S}$*.*

We write $\mathrm{vars}(b)$ for the domain of b and $\mathrm{tags}(b) = \cup_{x \in \mathrm{vars}(b)} \mathrm{tags}(b(x))$ for its tags. Hence, the informal sentence "x is present at t in b" is formally defined by $t \in \mathrm{tags}(b(x))$. We write $b|_X$ for the projection of a behavior b on a set $X \subset \mathcal{X}$ of names (i.e. $\mathrm{vars}(b|_X) = X$ and $\forall x \in X, b|_X(x) = b(x)$) and $b_{/X}$ for its complementary of $b|_{\mathrm{vars}(b) \setminus X}$. A *process* $p \in \mathcal{P} = \mathcal{P}(\mathcal{B})$ is a set of behaviors that have the same domain X (written $\mathrm{vars}(p)$). Synchronous composition $p \| q$ is defined by the set of behaviors that extend a behavior $b \in p$ by the restriction $c_{/\mathrm{vars}(p)}$ of a behavior $c \in q$ if the projections of b and c on $\mathrm{vars}(p) \cap \mathrm{vars}(q)$ are equal.

$$
p \,|\, q = \left\{ b \cup c \;\middle|\; \begin{array}{l} (b, c) \in p \times q, \\ b|_{\mathrm{vars}(p) \cap \mathrm{vars}(q)} = c|_{\mathrm{vars}(p) \cap \mathrm{vars}(q)} \end{array} \right\}
$$

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* $\mathrm{vars}(b) = \mathrm{vars}(c)$ *and there exists a bijection* f *on* $\mathcal{T}$ *that is strictly monotonic* ($\forall tt', t < t' \Leftrightarrow f(t) < f(t')$)*, increasing* ($\forall t, t \leq f(t)$) *and satisfies* $\mathrm{tags}(c(x)) = f(\mathrm{tags}(b(x)))$ *for all* $x \in \mathrm{vars}(b)$ *and* $b(x)(t) = c(x)(f(t))$ *for all* $x \in \mathrm{vars}(b)$ *and all* $t \in \mathrm{tags}(b(x))$*. The behaviors* b *and* c *are* stretch-equivalent*, written* $b \lessgtr c$*, 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 $b \in p$, $c \lessgtr b \Rightarrow c \in p$. A non-empty, stretch-closed process p admits a set of strict behaviors (a strict behavior is the $\leq$-minimum of a $\lessgtr$-equivalence class), written $(p)_{\lessgtr}$, s.t. $(p)_{\lessgtr} \subset p$ (for all $b \in p$, there is a unique $c \in (p)_{\lessgtr}$ s.t. $c \lessgtr b$).

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* $b \sqsubseteq c$*, iff* $\mathrm{vars}(b) = \mathrm{vars}(c)$ *and for all* $x \in \mathrm{vars}(b)$*,* $b_{|x} \leq c_{|x}$*. 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.* $d \sqsubseteq b$ *and* $d \sqsubseteq c$*.*

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 $X = \mathrm{vars}(P)$, $Y = \mathrm{vars}(Q)$ and $I = X \cap Y$).

$$
p \parallel q = \left\{ d \;\middle|\; \begin{array}{l} \exists b \in p, d_{|X \setminus Y} \lessgtr b_{|X \setminus Y},\ b|_I \sqsubseteq d|_I \\ \exists c \in q, d_{|Y \setminus X} \lessgtr c_{|Y \setminus X},\ c|_I \sqsubseteq d|_I \end{array} \right\}
$$

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 $[\![\,]\!]$ 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)

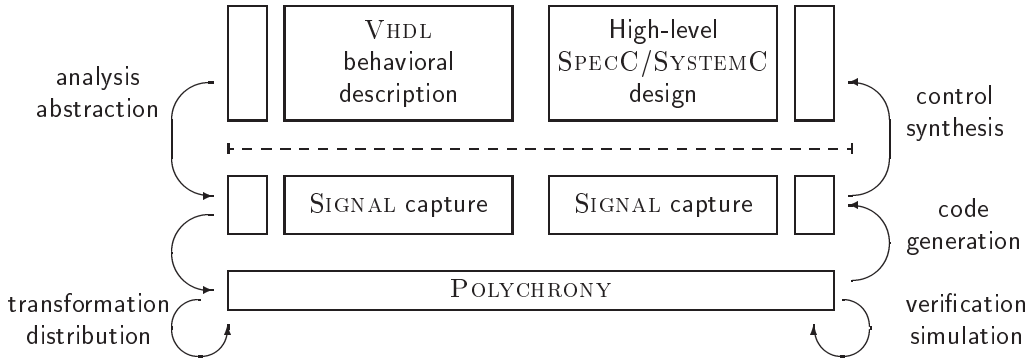

Figure 3. Polychrony for high-level system design

has a behavior b, written $b \in [\![P]\!]$, then it admits any stretching $c \geq b$ (relaxation $c \sqsupseteq b$) 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].

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

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 $c \in p \parallel q$, $(b|_I)_\approx = (c|_I)_\approx$ implies $b \approx c$ 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 $(p|_I)\|(q|_I)$ 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 $\forall b, c \in p$, $b \approx c \Rightarrow b \lesssim c$ 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(istart);
                idata = data; icount = 0;
                while (idata != 0) { icount += data & 1;
                                     idata >>= 1;
                }
                ocount = icount;
                notify(idone); }}};
```

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

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).

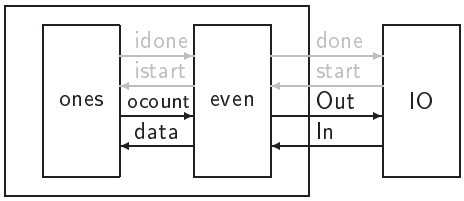

Figure 5. Functional architecture of the EPC

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)
   | idata  := (data default rshift (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{idone}(when c when idata=0)
   |) where    integer idata, icount; event c ;
```

```
process even =  (? integer In, ocount; event tick ! boolean Out,
                  data; boolean  start, istart, done, idone)
  (| c1     ::= wait{start}(tick)
   | data   := In 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 $x ::= f(\boldsymbol{y})$ when c partially defines x by $f(\boldsymbol{y})$ at the clock c. Composed to $x ::= f(\boldsymbol{z})$ when d, it is equivalent to the equation $x := f(\boldsymbol{y})$ when c default $f(\boldsymbol{z})$ when d iff the exclusion of the clocks c and d, denoted by the constraint $c\hat{}\#d$, can be checked satisfiable by the clock resolution engine of the compiler (meaning that the assignment to x is deterministic). We note $\boldsymbol{x} := f\{\boldsymbol{c}\}(\boldsymbol{y})$ for a call to a SIGNAL process of module f that takes the static parameters $\boldsymbol{c}$.

```
process notify={boolean lock}( ? event request ! )
                (| lock ::= true when request |);
process wait  ={boolean lock}( ? event request ! event ack)
                (| ack  ::= 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 $n\ y + 1$ when c (when y references a value computed during the previous transition in this block) or $x = y+1$ when c (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;
process ones = (? integer data ! integer ocount )          % boolean istart, idone          %
   (| idata  := data default rshift (pre InitData idata)    % istart ^= data                 %
    | icount := (pre 0 icount) + xand (idata, 1)            % idone ^= ocount                %
    | ocount := icount when idata=0
    |) where     integer idata, icount end;
process even = (? integer In, ocount ! boolean Out, data ) % boolean start, istart, done, idone %
   (| data   := In                                          % istart ^= In ^= start          %
    | Out    := (xand (ocount, 1) = 1)                       % ocount ^= done ^= idone        %
    |);
function rshift= ( ? i1 ! i2 )     spec (| i1 ^= i2 |)       pragmas C_CODE "&i2 = &i1 >> 1"end pragmas;
function xand = ( ? i1, i2 ! i3 ) spec (| i1 ^= i2 ^= i3 |)pragmas C_CODE "&i3=&i1 & &i2"  end pragmas;
```

Figure 7. Polychronous model of the EPC-core

finished (i.e. when idata is 0), icount 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 □ stands for a register).

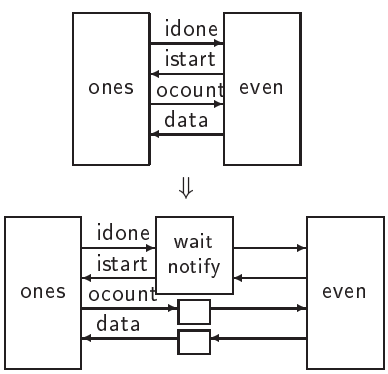

Figure 8. Refinement of the polychronous model by the specification model

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_{ones} and S_{ones}, flow equivalence of the input signal In, i.e. $b|_{\mathsf{In}} \approx c|_{\mathsf{In}}$ implies flow equivalence of the signal Out, i.e. $b|_{\mathsf{Out}} \approx c|_{\mathsf{Out}}$.

$$(1) : \forall b \in [\![P_{\mathsf{ones}}]\!],\ \forall c \in [\![S_{\mathsf{ones}}]\!],\ b|_{\mathsf{In}} \approx c|_{\mathsf{In}} \Rightarrow b|_{\mathsf{Out}} \approx c|_{\mathsf{Out}}$$

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.

$$P \equiv (\mathsf{data\hat{}= start} \parallel \mathsf{idata := data} \parallel \mathsf{start\hat{}= In} \parallel \mathsf{data := In})$$

$$S \equiv \begin{pmatrix} \mathsf{c} \quad \mathsf{::= wait\{start\}(clock)} \\ \parallel \mathsf{idata := data\ when\ c} \end{pmatrix} \parallel \begin{pmatrix} \mathsf{notify\{start\}(clock)} \\ \parallel \mathsf{data := In\ default\ pre\ InitData\ data\ when\ clock} \end{pmatrix}$$

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).

$$(2) : \forall b \in [\![P]\!],\ \forall c \in [\![S]\!],\ b|_{\mathsf{In}} \approx c|_{\mathsf{In}} \Rightarrow b|_{\mathsf{idata}} \approx c|_{\mathsf{idata}}$$

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 In and idata using, e.g. booleans (providing the corresponding implementations of the parameters xand, rshift 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 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; };
```

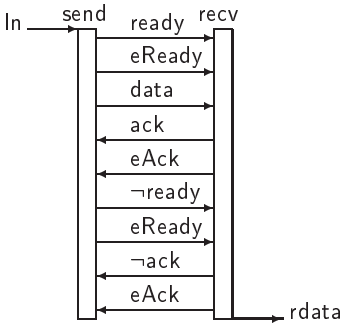

Figure 9. Implementation of an architecture-level channel in SpecC

```
process observer = (? boolean i ! boolean invariant)
 (| 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, o !)
              (| i ^= when flop
               | o ^= when not flop
               | flop := not (pre true flop)
               |) where boolean flop;
end;
```

clock	·	·	·
i	1	0	1
start	·	·	·
o	1	0	1

⇒

clock	·	·	·	·	·	·
i	1		0		1	
start	*tt*	*ff*	*tt*	*ff*	*tt*	*ff*
o		1		0		1

Figure 10. Refinement-checking observer

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 design-flow 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 ! )
  (| c1      := when (event In) when clock
   | data    ::= In when c1
   | ready   :=          true when c1
                default false when c2 when not(pre false ack)
                default pre false ready when clock
   |            notify{eReady}(c1)
   | 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 c1 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_{ones} to the architecture level A_{ones} (figure 12) is correct amounts to checking flow-invariance between the two designs.

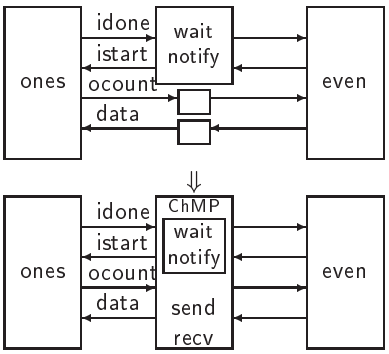

Figure 12. Refinement of the specification by an architecture layer

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

$$(3) : \forall b \in [\![S_{\mathsf{ones}}]\!],\ \forall c \in [\![A_{\mathsf{ones}}]\!],\ b|_{\mathsf{In}} \approx c|_{\mathsf{In}} \Rightarrow b|_{\mathsf{Out}} \approx c|_{\mathsf{Out}}$$

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.

$S \equiv$(data := (In when c default pre InitData data) when clock)
$A \equiv$(data := recv(clock) | send(In, 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 In and output data using booleans). The tool SIGALI allows to prove that the corresponding invariant never becomes false. Notice again that (4) implies (3).

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

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.

```
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 13. Communication-level bus in SPECC

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.

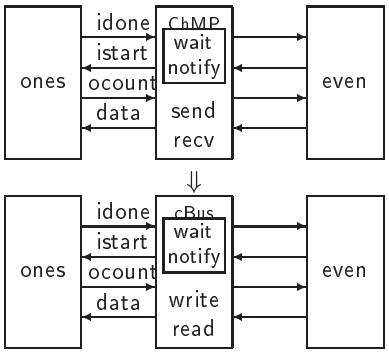

Figure 14. Refinement of an architecture-level channel by a communication-level bus

RTL-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(in event 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_istart = 0b;
                 if (start == 1b) state=S1 else state=S0;
                 break;
        case S1: ack_istart = 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_idone == 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 assembly 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 POLYCHRONY in that design chain. The novelty of integrating POLYCHRONY 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 fora compositional and formally-checked transformational refinements, automating the most difficult design steps toward implementation using efficient clock resolution and synthesis

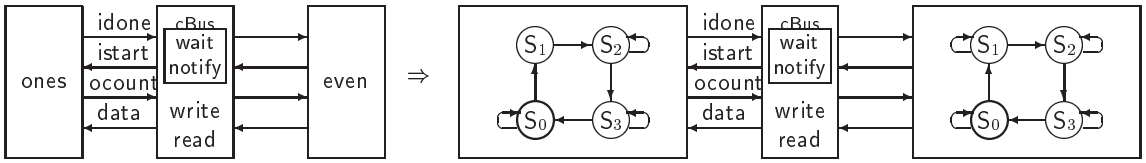

Figure 16. Refinement of the communication-level design by an RTL-level design

techniques, implemented in the SIGNAL compiler.

References

[1] 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.

[2] AUBRY, P. "Mises en oeuvre distribues de programmes synchrones" *Thèse de l'Université de Rennes 1*. October 1997.

[3] BENVENISTE, A., CAILLAUD B., AND LE GUERNIC, P. "Compositionality in dataflow synchronous languages: specification and distributed code generation". In *Information and Computation*, v. 163, pp. 125-171. Academic Press, 2000.

[4] 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.

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

[6] CARLONI, L. P., MCMILLAN, K. L., SANGIOVANNI-VINCENTELLI, A. L. "Latency-Insensitive Protocols". In *Proceedings of the 11th. International Conference on Computer-Aided Verification*. Lecture notes in computer science v. 1633. Springer Verlag, July 1999.

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

[8] F. DOUCET, M. OTSUKA, R. GUPTA AND S. SHUKLA "Efficient System Level Co-Design Environment using Split Level Programming". *Technical Report 01-34*. UC Irvine, June 2001.

[9] GAMATIÉ, A., GAUTIER, T. Modeling of modular avionics architectures using the synchronous language. In *proceedings of the 14th. Euromicro Conference on Real-Time Systems, work-in-progress session*. IEEE Press, 2002.

[10] P. GARG, S. SHUKLA, R. GUPTA "Efficient Usage of Concurrency Models in an Object Oriented Co-Design Framework". In *Design Automation and Test in Europe*. IEEE Press, 2001.

[11] D. GAJSKI, F. VAHID, S. NARAYAN, AND J. GONG. "Specification and Design of Embedded Systems". Prentice Hall, 1994.

[12] LEE, E. A., 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.

[13] 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.

[14] 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.

[15] 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.

[16] TALPIN, J.-P., LE DEZ, B., GAMATI, A., LE GUERNIC, P., BERNER, D. Component-based engineering of real-time JAVA applications on a polychronous design platform. In *Submitted for publication*. Available as INRIA research report n. 4744, February 2003.

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

[18] E. A. LEE. Overview of the Ptolemy project. *Technical Memorandum M01/11*. UC Berkeley, 2001.

[19] SPECC. `http://www.specc.org`, 2003.

[20] SYSTEMC. `http://www.systemc.org`, 2003.