Component-based Design of Heterogeneous Reactive Systems in Prometheus

Gregor Goessler

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Component-based Design of Heterogeneous Reactive Systems in PROMETHEUS

Gregor Gössler

N° ????

Décembre 2006

Thème COM

ISSN 0249-6399

INRIA
RHÔNE-ALPES

Component-based Design of Heterogeneous Reactive Systems in PROMETHEUS

Gregor Gössler

Thème COM — Systèmes communicants
Projet Pop Art

Rapport de recherche n° ???? — Décembre 2006 — 18 pages

Abstract: Designing embedded systems increasingly demands coping with *heterogeneous* systems, involving different models of computation, communication, and execution, on different levels of abstraction and different time scales. The component model BIP (Behavior, Interaction model, Priority) has been designed to support the construction of heterogeneous reactive systems. It enables heterogeneous modeling by separating the notions of behavior, interaction model, and execution model.

We present here the design tool PROMETHEUS, which implements the BIP component model, along with a set of algorithms for compositional verification. The use of the component framework is illustrated with two case studies involving different models of computation and communication.

Key-words: component framework, heterogeneous modeling, incrementality, correctness by construction.

Unité de recherche INRIA Rhône-Alpes
655, avenue de l'Europe, 38334 Montbonnot Saint Ismier (France)
Téléphone : +33 4 76 61 52 00 — Télécopie +33 4 76 61 52 52

Construction de systèmes réactifs hétérogènes à partir de composants en PROMETHEUS

Résumé : La conception de systèmes embarqués doit aujourd'hui faire face à des systèmes *hétérogènes*, composés de modules utilisant différents modèles de calcul, de communication et d'exécution, à différents niveaux d'abstraction et échelles de temps. Le modèle à composant BIP a été conçu pour supporter la construction de systèmes réactifs hétérogènes. La modélisation hétérogène est rendue possible par la séparation des notions de comportement, de modèle d'interaction et de modèle d'exécution.

Nous présentons ici l'outil de conception PROMETHEUS qui met en œuvre le modèle à composants BIP et un ensemble d'algorithmes pour la vérification compositionnelle. L'utilisation du cadre à composants est illustrée par deux études de cas qui utilisent différents modèles de calcul et de communication.

Mots-clés : composants, modélisation hétérogène, incrémentalité, correction par construction.

1 Introduction

Designing embedded systems increasingly demands coping with *heterogeneous* reactive systems that interact continuously with their environment. The heterogeneous nature comes from the need to integrate components using different models of computation (for instance, automata, continuous dynamic systems), communication (blocking or non-blocking), and execution (asynchronous, run-to-completion, constraints), on different levels of abstraction and different time scales.

Component-based techniques are crucial to overcome the complexity of embedded systems design. By equipping modules with an *interface*, they introduce a notion of encapsulation and abstraction, aiming at making the design of a component independent of its deployment. Existing component frameworks for designing heterogeneous systems include *reactive modules* [2], *Ptolemy* [16] — oriented towards the definition and composition of heterogeneous components —, and *Metropolis* [3], designed to support the design flow of embedded heterogeneous systems. Verification and correctness by construction of heterogeneous models has so far benefited from less attention. An interesting recent formalism for the composition of heterogeneous models is that of tag machines [5], adopting a lower level of description than the frameworks previously mentioned. Other formal component models such as interface automata [8], E-Lotos [9], IF-2 [6], SaveCCM [1], and Fractal [7], provide no or only limited support for heterogeneous modeling.

Two important requirements for a component framework to be useful in practice, are its support for an *incremental* design flow, and its *structural expressiveness*. Both are closely linked.

Incrementality of the design flow is crucial to cope with the complexity of design and verification. Incrementality means that at any stage of the design process, the subsystem constructed so far can be verified, simulated, or guaranteed correct by construction, and new components be added without back-tracking in the design process in order to modify the existing subsystem.

The notion of structural expressiveness is orthogonal to the classical notion of expressiveness (given a property P, is there a component k such that $k \vDash P$?), as it considers what can be expressed using only existing components: given a property P and a set K of components, is it possible to define a composition operation (or "glue") gl to assemble $K' \subseteq K$ such that $gl(K') \vDash P$? In other words, a structurally expressive component framework will help a designer to construct a correct system without modifying existing components or designing new ones, and thus facilitate component reuse, and enable an incremental design flow. To this end, structural expressiveness requires a powerful composition operation.

We argue that *separation of concerns* is the key to heterogeneity, structural expressiveness, and incremental construction. Clearly, separation of concerns enforces a separate and explicit description and composition of different aspects of the model, and thus enables heterogeneity and structural expressiveness.

For composition of components to be incremental, the composition operation $\|$ should be *associative* in the sense that for any components K_1, K_2, K_3 and parameterizations of the composition operation gl_{12}, $gl_{12,3}$, there exist parameterizations gl_{23}, $gl_{1,23}$ such that

$$(K_1 \|_{gl_{12}} K_2) \|_{gl_{12,3}} K_3 = K_1 \|_{gl_{1,23}} (K_2 \|_{gl_{23}} K_3)$$

Separation of concerns enables this requirement of associativity, since different aspects can composed separately by one single composition operation, in contrast to two non-commuting operations for composition and restriction used by some frameworks.

The component model BIP (Behavior, Interaction model, Priority) presented in [13, 14] has been designed to support incremental construction of heterogeneous reactive systems. It enables heterogeneous modeling by separating the notions of behavior, interaction model, and execution model. We present here the design tool PROMETHEUS, which implements the BIP component model, along with a set of algorithms for compositional verification of different properties like deadlock-freedom, liveness, and reachability.

This paper is structured as follows. Section 2 introduces the component framework. Section 3 explains the part of the work done by PROMETHEUS which is common to all implemented verification and synthesis algorithms. Section 4 discusses two case studies involving different models of computation and communication, and Section 5 concludes.

2 Component Model and Language

In the following, we present an enhanced version of the component model BIP presented in [13, 14]. For a set of variables X, let $V(X)$ denote the set of valuations of X, let $\mathcal{P}(X) = 2^{V(X)}$ be the set of predicates on $V(X)$.

2.1 Components

A *component* is a tuple $K = (B, I, P)$ consisting of a *behavior* B, an *interface* I, and a set of *constraints* P.

2.1.1 Behavior

The behavior of a component is defined in terms of a *transition system*. In contrast to the general component framework which allows for arbitrary domains, PROMETHEUS currently supports only Boolean and bounded non-negative integer variables.

Definition 1 (Transition system) *A* transition system S *is a tuple* (X, A, G, F) *where*

- X *is a set of variables;*
- $A = A^c \cup A^u$ *is a set of actions, also called* ports*, partitioned into* controllable *and* uncontrollable *actions;*
- $G : A \to \mathcal{P}(X)$ *associates with every action its* guard *specifying when the action can occur;*
- *For each action* $a \in A$*, let* $F^a : X \to \big(V(X) \to \mathbb{B} \cup \mathbb{N}\big)$ *be a partial function defining an* assignment *to variables, depending on the current state. The assignment function is total (defined on all valuations* $V(X)$*).*

Definition 2 (Postcondition) *Given a transition system* (X, A, G, F) *and state* $\mathbf{x} \in V(X)$*, let* $post_a(\mathbf{x})$ *be the* postcondition *of* $\mathbf{x}$ *under* $a \in A$*, that is,* $post_a(\mathbf{x}) = \mathbf{x}'$ *such that* $\mathbf{x}'_i = F^a(x_i)(\mathbf{x})$ *if* $F^a(x_i)$ *is defined, and* $\mathbf{x}'_i = \mathbf{x}_i$ *otherwise.*

Definition 3 (Semantics of a transition system) *A transition system* $S = (X, A, G, F)$ *defines a* transition relation $\to$: $V(X) \times A \times V(X)$ *such that* $\forall \mathbf{x}, \mathbf{x}' \in V(X)\ \forall a \in A\ .\ \mathbf{x} \xrightarrow{a} \mathbf{x}' \iff G(a)(\mathbf{x}) \wedge \mathbf{x}' = post_a(\mathbf{x})$.

The syntax of a basic component, whose behavior is defined by a transition system, can be found in appendix B.1.2. Declarations of integer variables must specify, in parentheses, an upper bound. Any assignment between integer variables in PROMETHEUS requires that both variables have the same upper bound. Actions are typed as uncontrollable by the envt modifier.

Example 1 *We define a* producer *component which produces an infinite sequence of integers* $..., 41, 42, 0, 1, ...$

```
component producer
    bool idle;
    int item(50);

    action init1 if idle and item=42 do idle:=false, item:=0;
    action init2 if idle and item<42 do idle:=false, item+=1;
    action send if not idle do idle:=true;
end;
```

2.1.2 Interface

The *interface* defines what part of a component is visible from outside, and how the component can interact with its environment. A fundamental idea of component-based design is that components are designed independent of the context in which they are going to be deployed [20]. Components should therefore allow for *inputs*, variables that can be used in the component, and connected with visible variables of other components once the component is integrated.

Formally, the interface of a component K with behavior $B = (X, A, G, F)$ is a tuple $I = (X_{in}, X_v, IM, IC_v)$ consisting of

- the set of *input variables* X_{in} such that $X_{in} \cap X = \emptyset$;
- the set of *visible variables* $X_v \subseteq X$;
- the *interaction model* $IM = (C, IC^+, f_C)$;
- the set of *visible interactions* $IC_v \subseteq I(C) = \{\alpha \mid \exists c \in C \,.\, \emptyset \neq \alpha \subseteq c\}$.

The *interaction model IM* specifies how the sub-components of K interact with each other. Intuitively, $C \subseteq 2^A$ is a set of *connectors*. Each of them is a maximal set of component actions that can be executed jointly. The set of connectors C uniquely defines the set of interactions $I(C)$. $IC^+ \subseteq I(C)$ is the set of *complete interactions*. Complete or maximal interactions (that is, connectors) do not need to synchronize with other actions to take place, whereas non-maximal incomplete interactions cannot occur alone. Communication between component only takes place through interactions, so as to separate communication from computation. Each connector $c \in C$ may define a communication function $f_C(c) : X \to (V(X) \to \mathbb{B} \cup \mathbb{N})$ making assignments to variables of components interacting in c.

The interaction model is fundamental for incremental construction: while specifying which interactions can occur at any stage of construction, it also provides information about yet incomplete interactions that require synchronization with further components. For instance, a non-maximal and incomplete interaction $\alpha \in I(C) \setminus (C \cup IC^+)$, although it may not occur alone, may be visible in order to allow for an interaction $\alpha \cup \alpha'$ once the component is integrated in an environment proposing α'.

A "pure" BIP component as defined in [13, 14] is a component (B, I, P) with $B = (X, A, G, F)$, $I = \big(\emptyset, X, IM, I(C)\big)$, and $IM = (C, IC^+, \emptyset)$, that is, a component without input variables, hiding, and communication functions.

2.1.3 Constraints

Constraints are used to model the fact that part of the behavior of a component may be forbidden depending on the state of other components. For instance, constraints may be used to express restrictions on the access to resources, or scheduling policies.

Definition 4 (Constraint) *A* constraint *on a component with behavior $B = (X, A, G, F)$ and interface $I = (X_{in}, X_v, (C, IC^+, f_C), IC_v)$ is a tuple $\big(U, (U^\alpha)_{\alpha \in C \cup IC^+}\big)$ where*

- *U is a predicate on $X \cup X_{in}$ called* state constraint*;*
- *for any $\alpha \in C \cup IC^+$, U^α is a predicate on $X \cup X_{in}$ called* interaction constraint*.*

Usually, the interaction model expresses how components *cooperate*, while constraints specify how they *constrain* each other, for instance on shared resources.

2.2 Semantics

Each component has a unique semantics defined as follows.

Definition 5 (Precondition) *Given a transition system (X, IC, G, F) (where transitions are labeled with interactions) and a predicate $P \in \mathcal{P}(X)$, let $pre_\alpha(P)$ be the* precondition *of P under $\alpha \in IC$, that is, $\forall \mathbf{x} \in V(X) \,.\, pre_\alpha(P)(\mathbf{x}) \iff P\big(post_\alpha(\mathbf{x})\big)$.*

Definition 6 (Existential abstraction) *Let X and X' be sets of variables such that $X' \subseteq X$, and let P be a predicate on X. We write $P|^\exists_{X'}$ for the* existential abstraction of P on X'*, that is, the smallest predicate on X' implied by P.*

Definition 7 (Semantics of a component) *The semantics of a component (B, I, P) with $B = (X, A, G, F)$, $I = (X_{in}, X_v, IM, IC_v)$, $IM = (C, IC^+, f_C)$, and $P = (U, (U^\alpha)_{\alpha \in A})$ is given by the behavior $B' = (X, C \cup IC^+, G', F')$ (where transitions are labeled with interactions) with $F'(\alpha) = \bigcup_{a \in \alpha} F^a \cup f_C(\alpha)$ for any $\alpha \in C \cup IC^+$, and $G'(\alpha) = \bigwedge_{a \in \alpha} G(a) \wedge \big(U^\alpha \wedge pre_\alpha(U)\big)|^\exists_X$.*

Notice that constraints on unconnected input variables are abstracted away, so as to obtain, for any component with open inputs, the most restrictive behavior over-approximating its behavior once the component is further integrated and the inputs are connected.

2.3 Operations on Components

The atoms of the component model are *basic components* — as in example 1 — consisting of a behavior, a trivial interface, and no constraints.

Definition 8 (Basic component) *A* basic component *is a component* $K = (B, I, P)$ *where* $B = (X, A^c \cup A^u, G, F)$, $I = (\emptyset, X_v, IM, IC_v)$, $IM = (C, IC^+, \emptyset)$ *with* $\{\{a\} \mid a \in A^u\} \subseteq IC^+ \subseteq C = \{\{a\} \mid a \in A^c \cup A^u\}$, *and* $P = (true, (true)_{\alpha \in C \cup IC^+})$.

2.3.1 Composition

Composition of components is defined by a composition operation which is parameterized by a *glue interaction model* $IM = (C, IC^+, f_C)$ on the components to be composed.

Definition 9 (Glue interaction model) *A* glue interaction model *over a set of components* $K_i = (B_i, I_i, P_i)$, $i \in \{1, \ldots, n\}$, *with* $B_i = (X_i, A_i, G_i, F_i)$ *and* $I_i = (X_{in,i}, X_{v,i}, IM_i, IC_{v,i})$ *is an interaction model* $IM_{glue} = (C, IC^+, f_C)$ *such that*

- *All connectors and complete interactions encompass only visible interactions of the components:* $\forall \alpha \in C \cup IC^+$, $\alpha \subseteq \bigcup_i A_i$, *and* $\alpha \cap A_i \in IC_{v,i} \cup \{\emptyset\}$.
- *For any glue interaction* $\alpha \in C \cup IC^+$ *there exist two components* $i \neq j$ *such that* $\alpha \cap A_i \neq \emptyset \neq \alpha \cap A_j$.
- *Each connector* $c \in C$ *may define a communication function* $f_C(c) : X \to (V(X) \to \mathbb{B} \cup \mathbb{N})$ *making assignments to variables of components interacting in* c.
- *Each complete glue interaction* $\alpha \in IC^+ \setminus C$ *that is part of two connectors* $c_1, c_2 \in C$, $\alpha \subseteq c_1 \cap c_2$, *must have a unique communication function: for any component* i *involved in* α, $x \in X_i$, *and* $(c_1, x, f_C(c_1, x)) \in f_C$ *we have* $(c_2, x, f_C(c_2, x)) \in f_C$ *with* $f_C(c_1, x) = f_C(c_2, x)$. *Thus,* $f_C(c_1)$ *uniquely defines* $f_C(\alpha)$ *by setting* $f_C(\alpha, x) = f_C(c_1, x)$.

The set of connectors C, complete interactions IC^+ and communication functions f_C are defined by a sequence of declarations in the following syntax:

connector *c* [complete *interactions*] do *assignments*;

Example 2 *Let us connect the producer of example 1 with a consumer by synchronizing* producer.send *with* consumer.receive, *such that the former interaction is complete, whereas the latter must interact. An interaction* $\{a_1, a_2\}$ *is written as* $a_1|a_2$.

```
component producer ... end;

component consumer
    bool consuming;
    int item(50);

    action receive if not consuming do consuming:=true;
    action done if consuming do consuming:=false;
end;

connector producer.send|consumer.receive complete producer.send
    do consumer.item::=producer.item;
```

Graphically, components are represented as boxes with the visible variables and actions. Connectors are represented by a line between its member actions. Complete and incomplete interactions are represented with a filled triangle and circle, respectively.

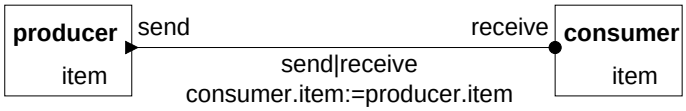

Figure 1: Connector producer.send|consumer.receive with complete interaction {producer.send}.

Inputs. Compound components can have *inputs*, making it possible to define constraints on the component behavior depending on input variables, without a priori knowledge of where the input comes from. Inputs are declared as a list of declarations of the form *type ident* in the component definition. When the component is assembled, the input can be connected using the statement

connect *input* = *variable*;

Example 3 *Coming back to example 2, suppose that the producer-consumer system is required to distinguish items of a particular value, which depends on the "outside" of the system and may vary. This is done by declaring an input* special*, and then connecting it with a visible variable of another component:*

```
system prodcons_controlled
    system prodcons (int special(50))
        component producer ... end;
        component consumer ... end;
        connector producer.send|consumer.receive ...;
    end;
    component controller
        int value(50);
        ...
    end;
    connect prodcons.special = controller.value;
end
```

Definition 10 (Consistency) *Given a set of components $K_i = (B_i, I_i, P_i)$, $i \in \{1, \ldots, n\}$ with $B_i = (X_i, A_i, G_i, F_i)$, $I_i = (X_{in,i}, X_{v,i}, IM_i, IC_{v,i})$, and $IM_i = (C_i, IC_i^+, f_{C,i})$, a glue interaction model $IM = (C, IC^+, f_C)$ on $\bigcup_{i=1,\ldots,n} IC_{v,i}$ is* consistent *with the set of components $(K_i)_{i=1,\ldots,n}$ if $dom(f_C) = \emptyset$ (the connectors do not define any communication functions), or the following three conditions hold:*

- *Communication functions update only variables of participating components: for any $\big(c, x, f_C(c,x)\big) \in f_C$ and interaction $\alpha \in C \cup IC^+$, $\alpha \subseteq c$, there is some component K_i such that $x \in X_{v,i}$ and $A_i \cap \alpha \neq \emptyset$.*
- *For each interaction $\alpha \in C \cup IC^+$ with $\alpha' = \alpha \cap A_i \neq \emptyset$ for some $i = 1, ..., n$ such that α' is part of two local connectors $c_1, c_2 \in C_i$ (i.e. $\alpha' \subseteq c_1 \cap c_2$), α' must have a unique communication function: for any $x \in X_i$ and $\big(c_1, x, f_{C,i}(c_1, x)\big) \in f_{C,i}$ we have $\big(c_2, x, f_{C,i}(c_2, x)\big) \in f_{C,i}$ with $f_{C,i}(c_1, x) = f_{C,i}(c_2, x)$. Thus, $f_{C,i}(c_1)$ uniquely defines $f_{C,i}(\alpha')$ by setting $f_{C,i}(\alpha', x) = f_{C,i}(c_1, x)$.*
- *Communication functions of IM are composable with transition and communication functions of the component behaviors: for any $\alpha \in C \cup IC^+$ and $a \in \alpha \cap A_i$ for some i, the sets $dom\big(f_C(\alpha)\big)$, $dom(F_i^a)$, and $dom\big(f_{C,i}(\alpha \cap A_i)\big)$ are pairwise disjoint.*

The communication functions of non-maximal complete glue interactions α are derived from the communication functions of the connectors containing α. The second and third condition above make sure that the communication function of glue interactions is well defined.

Consistency is automatically checked by PROMETHEUS. In particular, PROMETHEUS issues an error message if transition functions and communication functions define contradictory assignments for some legal (complete or maximal) interaction.

Example 4 *Consider the interaction model $IM = \big(\{a_1, a_2, a_3\}, \big\{\{a_1, a_2, a_3\}, \{a_1, a_2\}, \{a_1, a_3\}, \{a_1\}\big\}, \{x_2 := x_1, x_3 := x_1\}\big)$ defined by*

$$\text{connector } a_1|a_2|a_3 \text{ complete } a_1|a_2|a_3, a_1|a_2, a_1|a_3, a_1 \text{ do } x_2 := x_1,\ x_3 := x_1\ ;$$

between three components K_1, K_2, K_3, such that indexes of variables and actions refer to their owners. IM is consistent with the three components if none of the three actions updates any of the variables x_1, x_2, x_3. The complete interaction $\{a_1, a_2\}$ then encompasses the assignment $x_2 := x_1$.

Definition 11 (Composition of assignments) *Given two partial functions $f, g : X \to (V(X) \to \mathbb{B} \cup \mathbb{N})$ such that $dom(g) \cap dom(g) = \emptyset$, let $f \cup g$ be their composition such that $dom(f \cup g) = dom(f) \cup dom(g)$, $(f \cup g)(x) = f(x)$ if $x \in dom(f)$, and $(f \cup g)(\mathbf{x}) = g(x)$ if $x \in dom(g)$.*

Definition 12 (Composition $\|$) *Let $K_i = (B_i, I_i, P_i)$, $i \in \{1, \ldots, n\}$, be components with pairwise disjoint variables and actions, $B_i = (X_i, A_i, G_i, F_i)$, $I_i = (X_{in,i}, X_{v,i}, IM_i, IC_{v,i})$, and $P_i = (U_i, (U^\alpha)_{\alpha \in C_i \cup IC_i^+}$. Let $IM_{glue} = (C_{glue}, IC_{glue}^+, f_{glue})$ be a consistent glue interaction model over $\{K_i \mid i = 1, ..., n\}$. Furthermore, let $\sigma : \bigcup_i X_{in,i} \to \bigcup_i X_{v,i}$ be a partial substitution of input variables with visible variables.*

The composition of the components K_i under IM_{glue} and σ is the component

$$K = \|_{IM_{glue},\sigma}\{K_i \mid i = 1, ..., n\} = \big(\bigcup_i B_i,\ (X_{in}, \bigcup_i X_{v,i}, IM, IC_v),\ P\big)$$

where

- *the union of the behaviors is defined by $B_i \cup B_j = (X_i \cup X_j, A_i \cup A_j, G_i \cup G_j, F_i \cup F_j)$;*
- *$X_{in} = \bigcup_i X_{in,i} \setminus dom(\sigma)$;*
- *$IM = (C, IC^+, f_C)$ such that $IC = \bigcup_i I(C_i) \cup I(C_{glue})$, $C = \max IC$, $IC^+ = \bigcup_i IC_i^+ \cup IC_{glue}^+$, and $f_C(\alpha) = f_{C,i}(\alpha)$ if $\alpha \in C_i$ for some i, $f_C(\alpha) = \bigcup_i f_{C,i}(\alpha \cap A_i) \cup f_{glue}(\alpha)$ if $\alpha \in C_{glue}$.*
- *$IC_v = \bigcup_i IC_{v,i} \cup IC_{glue}$;*
- *$P = \big((\bigwedge_i U_i)[\sigma], (U^\alpha[\sigma])_{\alpha \in C \cup IC^+}\big)$ where $U^\alpha = true$ for $\alpha \in IC_{glue}$.*

Copy constructor. The copy constructor is syntactic sugar useful for compact modeling of systems with many identical components:

$$\text{component } \mathit{ident} = \mathit{ident}_c;$$

It creates, in the current scope, a new component with name ident as a copy of the component ident_c. Its use is illustrated in example 7 and in the example of section 4.2.

2.3.2 Restriction

Definition 13 (Restriction) *Let $K = (B, I, P)$ be a component with $B = (X, A, G, F)$, $I = (X_{in}, X_v, (C, IC^+, f_C), IC_v)$, and $P = \big(U, (U^\alpha)_{\alpha \in C \cup IC^+}\big)$. The* restriction *of K with a constraint $P_1 = \big(U_1, (U_1^\alpha)_{\alpha \in IC_v}\big)$ with $U \in \mathcal{P}(X_v)$ is the component $K/P_1 = \big(B, I, \big(U \wedge U_1, (U^\alpha \wedge U_1^\alpha)_{\alpha \in C \cup IC^+}\big)\big)$, where we set $U_1^\alpha = true$ for $\alpha \notin IC_v$.*

The concrete syntax of state and interaction constraints, respectively, is

assert *predicate*;
disable *interaction* (if | unless) *predicate*;

Example 5 *Coming back to example 3, suppose that items whose value is equal to* special, *are used to re-synchronize producer and consumer, and may not be discarded even if the consumer is not ready. This is done by restricting* producer.send *with an interaction constraint depending on* special*:*

```
system prodcons (int special(50)=0)
    component producer ... end;
    component consumer ... end;
    connector producer.send|consumer.receive ...;
    disable producer.send if producer.item=sync;
end;
```

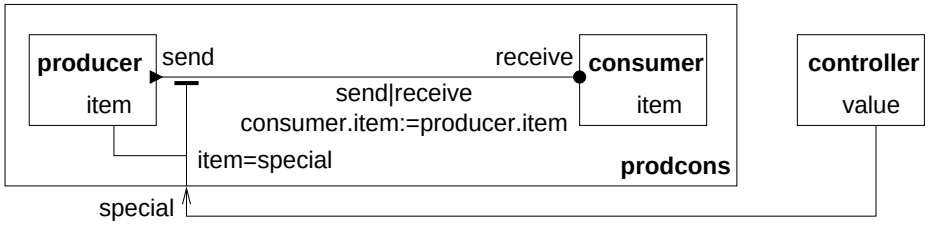

Figure 2: Interaction constraint on producer.send.

Graphically, the system composed of prodcons *and* controller *is represented as shown in Figure 2.*

Priorities are a special case of restriction, where interactions are disabled depending on which actions are enabled. Priorities are a convenient way to model coordination and have some interesting properties, discussed in [14].

Definition 14 (Priority) *Given an interaction model* $IM = (C, IC^+, f_C)$ *over a set of actions* A*, a* priority *is a relation* $\prec \subseteq (C \cup IC^+) \times 2^A$.

Definition 15 (Closure) *Let* $\prec$ *be a priority. The* closure *of* $\prec$ *is the least priority* $\prec^+$ *containing* $\prec$ *such that if* $\alpha_1 \prec^+ A_1$ *and* $\alpha_2 \prec^+ A_2$*, then* $\alpha_1 \prec^+ (A_1 \cup A_2) \smallsetminus \alpha_2$.

Example 6 *For the sake of readability, we write drop the set accolades for the interactions on the left-hand side and the sets of actions on the right-hand side. For* $\prec = \{a \prec bc, b \prec ad\}$, $\prec^+ = \{a \prec^+ bc, b \prec^+ ad, a \prec^+ acd, a \prec^+ cd, b \prec^+ bcd, b \prec^+ cd\}$.

A priority $\prec$ is automatically translated into the constraint $(true, (U^\alpha)_{\alpha \in C \cup IC^+})$ where

$$U^\alpha = \neg \bigvee_{\alpha \prec^+ A'} \bigwedge_{a_i \in A'} G(a_i)$$

Example 7 *We illustrate the composition and use of priorities with three processes sharing two resources.*

```
component c1
    bool waiting, running;
    action arrive if not (waiting or running) do waiting:=true;
    action p if waiting do waiting:=false, running:=true;
    action v if running do running:=false;
end;
component c2 = c1;
component c3 = c2;

c1.p < c2.v * c3.v;  // disable c1.p if c2 and c3 are not idle
c2.p < c1.v * c3.v;
c3.p < c1.v * c2.v;
c3.p < c2.p; c2.p < c1.p;  // fixed priority scheduling
```

2.3.3 Hiding

Hiding elements of a component is a feature most component frameworks offer. From a software engineering point of view, hiding enables encapsulation, and thus a conceptual abstraction from implementation details of a component: hidden variables and interactions are only visible and usable within the scope of the component. From a verification point of view, hiding helps to enable compositionality, by separating part of the behavior of a component from interaction of the component with the system. The syntax is

hide *variables [: interactions]*

Definition 16 (Hiding) *For a component $K = (B, I, P)$ with $B = (X, A, G, F)$, $I = (X_{in}, X_v, IM, IC_v)$, a set of variables $X' \subseteq X_v$, and a set of interaction $IC' \subseteq IC_v$, the component obtained by hiding X' and IC' in K is $(B, I, P)/(X', IC') = (B, I', P)$ with $I' = (X_{in}, X_v \setminus X', IM, IC_v \setminus IC')$.*

3 How Prometheus Works

The goal of Prometheus is to ensure, as far as possible, correctness by construction, while being minimally restrictive to allow for incremental construction. Therefore, Prometheus first restricts the system to ensure *necessary conditions* for the required properties to hold. In a second step, Prometheus checks for efficiently verifiable *sufficient conditions* entailing properties such as liveness and reachability.

3.1 Model Construction

Abstraction. Small integer variables are internally represented by vectors of Booleans. For the sake of efficiency, large integer variables — variables whose upper bound is greater than or equal to a constant THRESHOLD_ABSTRACT —, are represented only by an abstraction. Assignments and comparisons with such variables are interpreted on the abstraction. Since the compositional verification algorithms implemented in Prometheus use sufficient conditions on both under- and over-approximations of the behavior, abstraction is required to provide, for each concrete predicate, a *pair* of abstract predicates bracketing the concrete predicate.

Abstract predicates are derived automatically from the use of integer variables. Consider a concrete predicate of the form $x\#c$ or an assignment of the form $x := c$ (explicit or by value passing between interacting components), where x is an integer variable with an upper bound $b \geqslant$ THRESHOLD_ABSTRACT, c is a constant, and $\# \in \{<, \leqslant, =, \geqslant, >\}$. Prometheus automatically defines a tuple of abstract variables $\left((x\#c)^u, (x\#c)^o\right)$ such that $(x\#c)^u \implies x\#c \implies (x\#c)^o$. Similarly, consider a concrete predicate of the form $x\#y$ or an assignment of the form $x := y$, where x and y are integer variables with the same upper bound $b \geqslant$ THRESHOLD_ABSTRACT. If both variables belong to the same basic component, then Prometheus defines a tuple of abstract variables $\left((x\#y)^u, (x\#y)^o\right)$ as above. If both variables belong to different components, then Prometheus defines a tuple of abstract variables $\left((x\#y)^u_1, (x\#y)^u_2, (x\#y)^o_1, (x\#y)^o_2\right)$ such that $(x\#y)^u_1 \wedge (x\#y)^u_2 \implies x\#y \implies (x\#y)^o_1 \vee (x\#y)^o_2$.

Initial states. In a basic component, a local predicate P specifying a set of initial states can be defined with the statement initially P. The local state space of the component is then explored from P. The effect of the statement is equivalent to restricting the system with the state constraint characterizing all states of the component reachable from P. Notice that the statement does *not* define the initial states of the component as P in the global system, as most of the algorithms implemented in Prometheus do not distinguish initial states.

Example 8 *Adding an* initially *statement to the producer of example 1 restricts the state space of* producer *to item $\leqslant 42$, making sure that* item *has been defined when action* send *is enabled.*

```
component producer
    bool idle;
    int item(50);
    initially idle and item=0;

    action init1 if idle and item=42 do idle:=false, item:=0;
    action init2 if idle and item<42 do idle:=false, item+=1;
    action send if not idle do idle:=true;
end;
```

Invariants. As Prometheus focuses on the construction and verification of reactive systems, we suppose all components to be required deadlock-free. Therefore, Prometheus locally computes, as a first synthesis step, for each basic component k the most general deadlock-free control invariant (DLFCI) of k. Properties are then verified within these invariants.

Definition 17 (Deadlock-free control invariant) *A predicate $U^X \neq false$ is a* deadlock-free control invariant *of a basic component (B, I, P) with $B = (X, A, G, F)$ and $A = A^c \cup A^u$ in a system with semantics $B' = (X', IC, G', F')$ if (1) $U^X \implies \left(\bigvee_{\alpha \in IC \atop \alpha \cap A \neq \emptyset} G'(\alpha) \wedge pre_\alpha(U^X)\right)|_X^\exists$, and (2) for $\forall \alpha \in IC$ such that $\alpha \cap A^u \neq \emptyset$. $U^X \wedge G'(\alpha) \implies pre_\alpha(U^X)$.*

Intuitively, the first condition ensures that for any state of the basic component satisfying U^X, there is some global state enabling some interaction α involving the component and whose post-condition again satisfies U^X. The second condition ensures that U^X is invariant under interactions encompassing some uncontrollable action.

For any connector $c \in C$ and interaction $\alpha \subseteq c$, the communication function $f_C(\alpha)$ may make assignments to visible component variables, potentially violating invariance of the local invariant of c. Invariance of the local DLFCI under (global) interactions is called *receptiveness.*

Definition 18 (Receptiveness) *A component $K = (B, I, P)$ with $B = (X, A, G, F)$ is* receptive *in a system with semantics $B' = (X', IC, G', F')$ with respect to a local invariant dlf if $\forall \alpha \in IC$ such that $\alpha \cap A \neq \emptyset$, $G'(\alpha) \implies pre_\alpha(dlf)$.*

That is, interactions preserve the invariant of the component, independent of the values assigned by communication functions. PROMETHEUS guarantees receptiveness of basic components by construction.

3.2 Algorithms

PROMETHEUS implements verification algorithms for several properties, among them for determinism, liveness, and reachability. The discussion of these algorithms is not in the scope of this paper. Simplified versions of the reachability algorithm and the liveness criterion can be found in [11] and [12], respectively.

3.3 Interactive Mode

After successful construction of the model and verification of the properties specified in the command line parameters, PROMETHEUS enters an interactive/batch mode, where it is possible to interactively navigate through the state space of the model. The command interpreter supports the following requests:

- state *pred*; set the current state to *pred*
- let *ident=pred*; define the predicate named *ident* as *pred*
- path *pred*; try to find a path from the current state to *pred*
- *interaction;* execute *interaction* from the current state;
- undo cancel the last command that updated the current state;
- unexec *interaction*; set the current state to $pre_{interaction}$ of the current state
- ? print the defined predicates and interactions enabled in the current state;
- help print this list of available commands.

The grammar of *pred* is defined in section B.2.

Example 9 *Suppose we want to find, for the system of example 5, a path from state* prodcons.producer.item=0 and prodcons.producer.idle and not prodcons.consumer.consuming and ctrl.value=1 *to some state where* prodcons.producer.send *is enabled. This can be done as in the following dialog (commands are in italic):*
@0 = true > *state prodcons.producer.item=0 and prodcons.producer.idle and not prodcons.consumer.consuming and ctrl.value=1;*
@1 = !prodcons.consumer.consuming and prodcons.producer.idle and (ctrl.value=1) and (prodcons.producer.item=0) > *path precond prodcons.producer.send;*

PROMETHEUS returns the path prodcons.producer.init2; prodcons.producer.send|prodcons.consumer.receive; prodcons.producer.init2, which is the shortest path leading to the requested state.

4 Examples

4.1 SystemC

SystemC is a class library for C++ dedicated to the design and verification of hardware and software, providing constructs like modules, processes, and communication primitives [19, 15]. We informally show how SystemC models can be represented in our component framework. A prototype tool for automatic translation of a subset of SystemC to the component language of PROMETHEUS has been developed.

SystemC distinguishes between *logical time* and *physical time*. Each instant of physical time can consist of a sequence of logical time instants called *delta-cycles*.

An example. We illustrate the translation with a model of two master processes that can call slave processes via a shared bus. The example, adapted from [18], is written in SystemC, using a transaction-level modeling library provided by ST Microelectronics. It consists of six components: two processes status_master and signal_master, a bus arbiter bus, a signal, and two slave processes status_slave and signal_slave. In order to trigger their respective slave process, the master processes send, at each physical time unit, a request to the bus arbiter, along with the address of the slave process to be called, and some optional data to be transmitted. The bus arbiter calls the slave process mapped to the address, or sets a status flag if the address does not correspond to any of the slave processes, and then returns control to the invoking process. status_slave checks whether the received data has the expected value. Before accessing to the bus, signal_master sets the signal to *false*. Component signal_slave checks whether the signal reads *false*. The component model of the system is shown in figure 3.

Processes, modules, and channels. The basic building blocks of SystemC for behavior and communication are processes and primitive channels, respectively. Both are translated into basic components, by translating the control structure. SystemC modules allow to structure a design, and are translated into components with the same hierarchical structure.

SystemC provides two means of communication between processes, namely channels and events. In order to communicate through a channel, a process calls a method of the channel. On termination, the method may return some value. We translate this into two rendez-vous between the process and the channel: a call interaction where the former which can pass parameters such as an address to the latter, and a return interaction where the channel can pass parameters such as a status, back to the process. In order to avoid handling call stacks, the current translation scheme does not allow for recursion in the SystemC model.

Signals are a particular case of channels. They provide two methods, write and read. The former assigns a new Boolean value to an internal state variable new. The latter returns the current value current. The current value is updated to new at the end of the delta-cycle. We represent each signal by a simple component with this behavior.

Events. Events allow for asymmetric communication: when a process emits some event, all processes blocked on a wait statement waiting for that event, are notified and activated. We translate this by instantiating an *event* component for each SystemC event. Occurrence of the event is modeled by an interaction between the emitting component and the event component. Notification can be *immediate*, in the next delta-cycle, or after some physical time delay. Notification is modeled by interaction constraints disabling the actions waking up the waiting components unless the event has been notified. Our running example does not use events.

Run to completion. According to the (informally specified) simulation semantics, the execution model of SystemC is *run-to-completion*, that is, a running process runs exclusively and is not preempted until it encounters a wait statement suspending it until some specified event occurs or time progresses, or — in the case of methods — until termination. In order to model run-to-completion, we partition, for each basic component K_i representing a SystemC process or channel, all states $V(X_i)$ in the sets of *running* states $R(i)$ and *waiting* states $W(i)$, where waiting states are those where some notification is awaited. Based on this classification of component states, we partition the interactions of the model into *starting* interactions $ST(i)$ and *executing* interactions $EX(i)$, defined as the interactions of component i entering $R(i)$ and enabled in $R(i)$, respectively. Let $ST = \bigcup_i ST(i)$ and $EX = \bigcup_i EX(i)$. Run-to-completion is then achieved by restricting the system with the priority $\{\alpha_1 \prec \alpha_2 \mid \alpha_1 \in ST \wedge \alpha_2 \in EX\}$, which we note here $ST \prec EX$ for short.

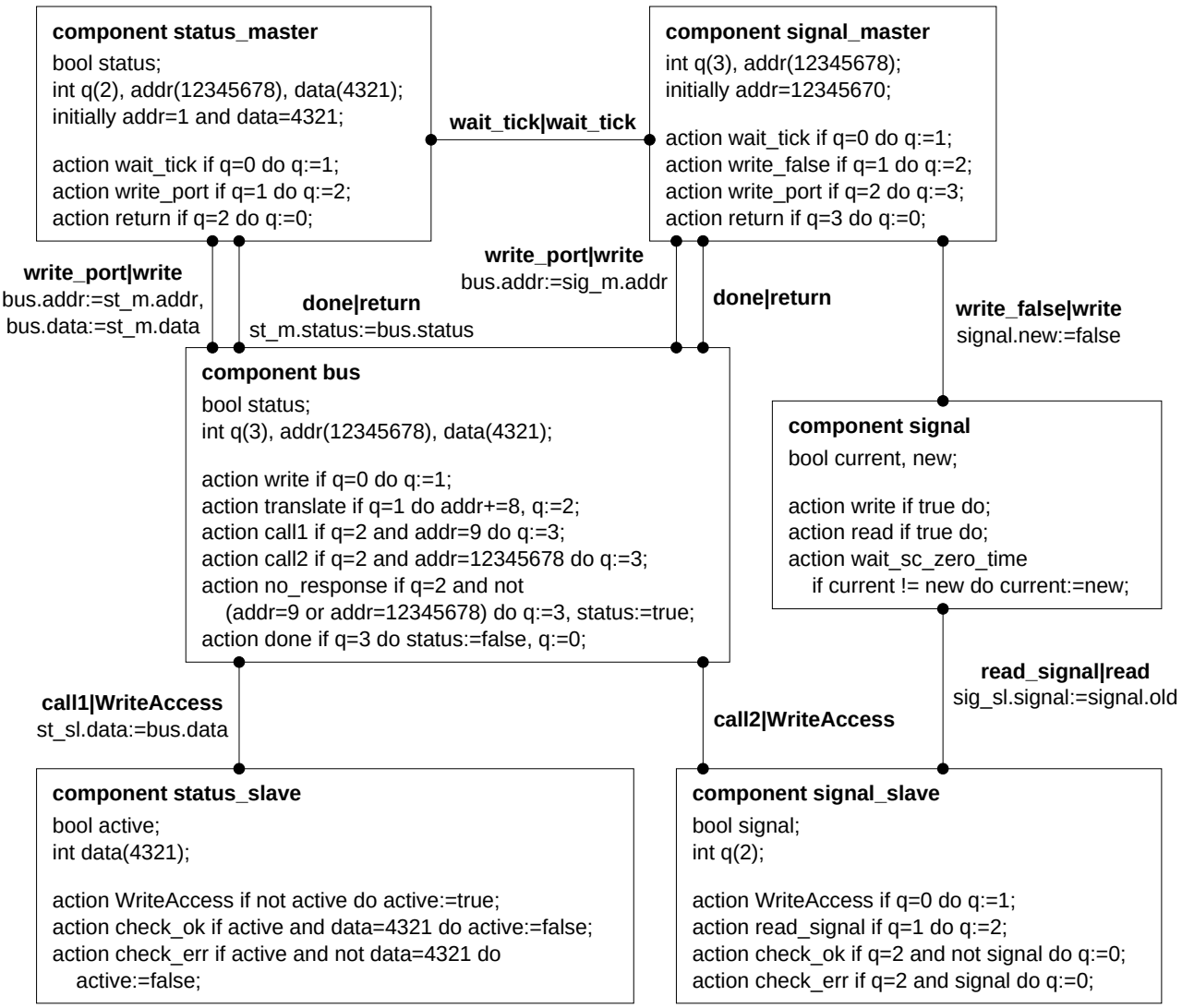

Figure 3: Four processes communicating through a shared bus and a signal.

SystemC scheduler. Slightly simplifying, the SystemC simulation semantics are as follows:

1. As long as there are eligible processes, run one of them until completion.
2. If no more process is eligible, the current delta-cycle ends. Update signals and notify events.
3. If there are processes waiting for logical time progress, make one step of logical time, and return to 1.
4. Otherwise, if some process is waiting for physical time progress, advance physical time and return to 1.

We model logical time progress by an interaction between the sc_wait_zero_time actions of all concerned components (in our example, the signal component). Physical time progress is modeled by a wait_tick interaction between all concerned components.

As the algorithm above describes a priority between the different stages of simulation, we have chosen to model part of the scheduler as priorities: wait_tick $\prec$ sc_wait_zero_time $\prec ST \prec EX$. However, as notifications of different events must be atomic (step 2.), it would be incorrect to run a process which has been waiting for a notified event, before *all* events have been notified. This is taken into account by a basic component implementing a simple automaton which represents the alternation of step 2 and the other steps of

the scheduler. In the case of our example, there are no events, such that we do not need a scheduler component here.

Properties. Properties of SystemC models one may be interested in, include:

- Deadlock-freedom, that is, absence of states where the processes mutually await events from each other;
- Time progress: does physical time always diverge? This may not be the case if a set of processes keep activating each other through events.
- Liveness: are all of the processes guaranteed to progress?
- Reachability of "bad" states implying an error in the specification or model.

All of these properties can be checked with PROMETHEUS. In the case of our example, the system is deadlock-free, time diverges, and all components are live.

4.2 Genetic Regulatory Networks

Proteins fulfill a huge number of functions in any living organism. Any protein is encoded by a gene. In order to produce the protein, the corresponding gene has to be *transcribed* into messenger RNA, which is then *translated* to obtain the protein. This production mechanism is regulated by the concentration of other proteins, which can *activate* or *inhibit* the production. The dynamics of the protein concentrations is thus defined by a regulatory network which usually encompasses a multitude of highly complex feedback loops. Being able to analyze its structure and behavior is crucial for understanding the functions of the proteins and their interactions.

We illustrate modeling of genetic networks in PROMETHEUS with a well-studied example: cell differentiation by delta-notch lateral inhibition [17, 10]. Cell differentiation is an important step in embryonic development, as it causes initially uniform cells to assume different functions. For each cell we consider the concentrations of two trans-membrane proteins, *delta* and *notch*. High concentrations of *notch* inhibit production of *delta* in the same cell. High *delta* levels activate *notch* production in the neighboring cells.

Figure 4 shows a component model with four identical cells. Each cell is modeled by a compound component consisting of two basic components, each of them being a counter for the concentration of one protein, where concentration levels are discretized to the set $\{0, 1, 2\}$. The model makes extensive use of interaction constraints to represent the influence of protein concentrations on each other.

The component model shown in Figure 4 is obtained by the following PROMETHEUS program:

```
system delta_notch
    system cell1 (int in1(2), int in2(2))
        component delta
            int x(2);
            action inc if x<=1 do x+=1;
            action dec if x>=1 do x-=1;
        end;

        component notch
            int x(2);
            action inc if x<=1 do x+=1;
            action dec if x>=1 do x-=1;
        end;

        let u = in1=2 or in2=2;

        disable delta.inc if notch.x>=1;
        disable delta.dec if notch.x<=1;
        disable notch.inc if not $u;
        disable notch.dec if $u;
    end;

    component cell2 = cell1;
    component cell3 = cell1;
    component cell4 = cell1;
```

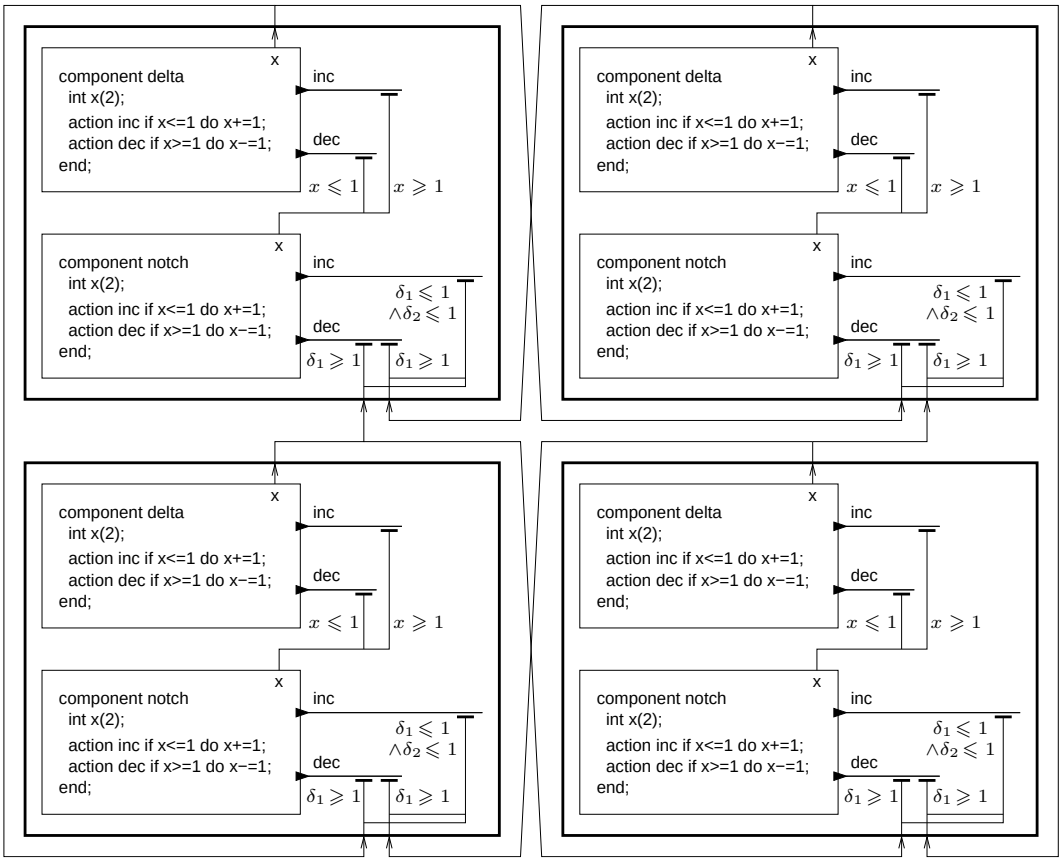

Figure 4: Delta-notch network.

```
    CONNECT cell1.in1 = cell2.delta.x;
    CONNECT cell1.in2 = cell3.delta.x;
    CONNECT cell2.in1 = cell1.delta.x;
    CONNECT cell2.in2 = cell4.delta.x;
    CONNECT cell3.in1 = cell1.delta.x;
    CONNECT cell3.in2 = cell4.delta.x;
    CONNECT cell4.in1 = cell2.delta.x;
    CONNECT cell4.in2 = cell3.delta.x;
end
```

Modeling genetic networks in a mathematically well-founded way based on piecewise linear differential inclusions [11] is supported by the high-level primitive equilibrium. For instance, equilibrium $cell1.delta.x = 0$ if $cell1.notch.x = 2$ defines a set of interaction constraints modeling the fact that provided $cell1.notch.x = 2$, $cell1.delta.x$ converges towards zero. The definition and further explanation can be found in [11].

5 Discussion

We believe that the framework for heterogeneous components BIP, and its implementations, will both help to better understand heterogeneous reactive systems, and prove is use in practice. The two examples shown in this paper only give a first glance at the versatility of the framework.

Translators from SystemC, and towards CADP, connect PROMETHEUS with other tool platforms, enabling larger case studies. Both translators are under development at INRIA and currently exist as prototypes.

Another tool implementing the BIP component framework, focusing on code generation, is itself called BIP [4]. This paper also shows how timed and synchronous components can be modeled in our component framework.

Both the component framework and PROMETHEUS are ongoing work and likely to evolve, especially by supporting new language primitives further improving separation of concerns, and by providing new algorithms for verification and correctness by construction.

References

[1] M. Akerholm, J. Carlson, J. Fredriksson, H. Hansson, J. Hakansson, A. Möller, P. Pettersson, and M. Tivoli. The SAVE approach to component-based development of vehicular systems. *Journal of Systems and Software*, 2006.

[2] R. Alur and T.A. Henzinger. Reactive modules. In *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science (LICS'96)*, pages 207–218, 1996.

[3] F. Balarin, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, M. Sgroi, and Y. Watanabe. *Modeling and Designing Heterogeneous Systems*, volume 2549 of *LNCS*, pages 228–273. Springer-Verlag, 2002.

[4] A. Basu, M. Bozga, and J. Sifakis. Modeling heterogeneous real-time components in BIP. In *proc. SEFM'06 (invited paper)*, 2006.

[5] A. Benveniste, B. Caillaud, L. Carloni, and A. Sangiovanni-Vincentelli. Tag machines. In W. Wolf, editor, *proc. EMSOFT'05*, pages 255–263, 2005.

[6] M. Bozga, S. Graf, and L. Mounier. If-2.0: A validation environment for component-based real-time systems. In E. Brinksma and K.G. Larsen, editors, *proc. CAV'02*, volume 2404 of *LNCS*, pages 343–348. Springer-Verlag, 2002.

[7] E. Bruneton, T. Coupaye, and J.B. Stefani. Recursive and dynamic software composition with sharing. In *proc. WCOP'02*, 2002.

[8] L. de Alfaro and T.A. Henzinger. Interface automata. In *Proc. 9th Annual Symposium on Foundations of Software Engineering (FSE)*, pages 109–120. ACM Press, 2001.

[9] H. Garavel and M. Sighireanu. Towards a second generation of formal description techniques - rationale for the design of E-LOTOS. In *proc. FMICS'98*, pages 187–230, 1998.

[10] R. Ghosh, A. Tiwari, and C. Tomlin. Automated symbolic reachability analysis; with application to delta-notch signaling automata. In O. Maler and A. Pnueli, editors, *proc. HSCC'03*, volume 2623 of *LNCS*, pages 233–248. Springer-Verlag, 2003.

[11] G. Gössler. Compositional reachability analysis of genetic networks. In C. Priami, editor, *CMSB'06*, volume 4210 of *LNBI*, pages 212–226. Springer, 2006.

[12] G. Gössler, S. Graf, M. Majster-Cederbaum, M. Martens, and J. Sifakis. An approach to modelling and verification of component based systems. In *proc. SOFSEM'07*, LNCS. Springer-Verlag, 2007.

[13] G. Gössler and J. Sifakis. Component-based construction of deadlock-free systems (extended abstract). In *proc. FSTTCS'03*, volume 2914 of *LNCS*. Springer-Verlag, 2003.

[14] G. Gössler and J. Sifakis. Priority systems. In F. de Boer, M. Bonsangue, S. Graf, and W.-P. de Roever, editors, *proc. FMCO'03*, volume 3188 of *LNCS*, pages 314–329. Springer-Verlag, 2004.

[15] T. Grötker, S. Liao, G. Martin, and S. Swan. *System Design with SystemC*. Kluwer, 2002.

[16] E.A. Lee. Overview of the Ptolemy project. Technical Report UCB/ERL M03/25, University of California at Berkeley, 2003.

[17] G. Marnellos, G.A. Deblandre, E. Mjolsness, and G. Kintner. Delta-Notch lateral inhibitory patterning in the emergence of ciliated cells in *Xenopus*: Experimental observations and a gene network model. In *proc. PSB'00*, volume 5, pages 326–337. World Scientific Publishing, 2000.

[18] M. Moy, F. Maraninchi, and L. Maillet-Contoz. LusSy: A toolbox for the analysis of system-on-a-chip at the transactional level. In *proc. ACSD'05*, 2005.

[19] Open SystemC Initiative. SystemC. `http://www.systemc.org`.

[20] C. Szyperski, D. Gruntz, and S. Murer. *Component Software.* Addison Wesley, 2nd edition, 2002.

A How to Use Prometheus

A.1 Command Line Parameters

prometheus [<options>] <model file>, where <options> may include:

-h to get this help

-p for verbose parsing

-v verbose mode

-d to check for determinism

-l to check liveness

-u to under-approximate behavior (if the equilibrium statement is used, see [11])

-cadp to translate the model into a CADP model (under development).

B Syntax

In the following sections, terminals are written **bold**. **ident** denotes an identifier, and **nat** a non-negative integer. $ident_b$, $ident_i$, $ident_p$, $ident_a$, $ident_v$, and $ident_c$ stand for identifiers representing a boolean variable, an integer variable, a predicate, an action, a (boolean or integer) variable, and a component, respectively. The axioms are underlined. Deprecated non-terminals are written in *italic*.

B.1 System Definition

Expressions in brackets "[x]" are to be read as $(\epsilon |\ \mathrm{x})$.

B.1.1 Boolean Expressions

bool_cst ::= **TRUE** | **FALSE**

pred ::= bool_cst | **ident**$_b$ | pred bool_op pred | **NOT** pred | **(** pred **)** | arith | **$** **ident**$_p$

bool_op ::= **AND** | **OR** | == | **!=**

arith ::= **ident**$_i$ arith_cmp **nat** | **ident**$_i$ arith_cmp **ident**$_i$

arith_cmp ::= $<$ | $\leqslant$ | $=$ | $\geqslant$ | $>$

B.1.2 Basic Components

basic_component ::= **COMPONENT ident** behavior **END**

behavior ::= [bool_vars **;**] [int_vars **;**] [initial_cond] transition$^+$

bool_vars ::= **BOOL ident**$_b$ $($**,** **ident**$_b)^*$

int_vars ::= **INT** int_var $($**,** int_var$)^*$

int_var ::= **ident**$_i$ **(** **nat** **)**

initial_cond ::= **INITIALLY** pred **;**

transition ::= $(\epsilon$ | **ENVT**$)$ **ACTION ident**$_a$ [param] **IF** pred **DO** [assignments] **;**

assignments ::= assignment $($**,** assignment $)^*$

assignment ::= **ident**$_b$ **:=** pred | int_assignment

int_assignment ::= **ident**$_i$ **::=** **ident**$_i$ | **ident**$_i$ **:=** **nat** | **ident**$_i$ **+=** **nat** | **ident**$_i$ **−=** **nat**

B.1.3 Compound Components

system ::= **SYSTEM ident** system_decl
system_decl ::= [inputs] component$^+$ composition **END**
inputs ::= **(** (input_var ,)* input_var **)**
input_var ::= **BOOL ident** | **INT ident (nat)**
composition ::= connections connector* [interaction_model] pred_decl* constraints
connections ::= (**CONNECT ident**$_v$ **= ident**$_v$ **;**)*
connector ::= **CONNECTOR** interaction [complete] **DO** [assignments] **;**
complete ::= **COMPLETE** interactions
interactions ::= interaction (**,** interaction)*
interaction_model ::= **INTERACTIONS** interactions [incomplete] **;**
constraints ::= [state_constraint] action_constraint* prio* equilibrium*
component ::= (basic_component | copy | system) [hiding] **;**
copy ::= **COMPONENT ident = ident**$_c$
param ::= **!** bool_cst | **! ident** | **? ident**
interaction ::= **ident**$_a$ ('|' **ident**$_a$)*
incomplete ::= **INCOMPLETE** interactions **;**
pred_decl ::= **LET ident**$_p$ **=** pred **;**
state_constraint ::= **ASSERT** pred **;**
prio ::= interaction < interaction (***** interaction)* **;**
action_constraint ::= **DISABLE** interaction (**IF** | **UNLESS**) pred **;**
equilibrium ::= **EQUILIBRIUM ident**$_i$ **= nat IF** pred **;**
hiding ::= **HIDE ident**$_v$ (**,** **ident**$_v$)$^+$ [**:** interactions]

B.2 Script Commands

Non terminals not defined here are as in the system declaration grammar.

commands ::= (state_decl | pred_decl_c | path | exec | **UNDO** | unexec | **?** | **HELP**)*
state_decl ::= **STATE** pred_c **;**
pred_c ::= **@ nat** | **PRECOND** interaction | **POSTCOND** interaction
| **EXABSTRACT** pred_c **ON ident**$_i{}^+$
| **UNIVABSTRACT** pred_c **ON ident**$_i{}^+$
| bool_cst | **ident**$_b$ | pred_c bool_op pred_c | **NOT** pred_c | **(** pred_c **)**
| arith | **$ ident**$_p$
pred_decl ::= **LET ident**$_p$ **=** pred_c **;**
path ::= **PATH** pred_c (**UNTIL** pred_c)* **;**
exec ::= interaction **;**
unexec ::= **UNEXEC** interaction **;**

Unité de recherche INRIA Rhône-Alpes
655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399