Modeling Synchronous Systems in BIP

Marius Bozga Vassiliki Sfyrla Joseph Sifakis
VERIMAG – Centre Equation, 2 Avenue de Vignate, 38610 Gières, France
FirstName.LastName@imag.fr

ABSTRACT

We present a general approach for modeling synchronous component-based systems. These are systems of synchronous components strongly synchronized by a common action that initiates steps of each component. We propose a general model for synchronous systems. Steps are described by acyclic Petri nets equipped with data and priorities. Petri nets are used to model concurrent flow of computation. Priorities are instrumental for enforcing run-to-completion in the execution of a step.

We study a class of *well-triggered* synchronous systems which are by construction deadlock-free and their computation within a step is confluent. For this class, the behavior of components is modeled by *modal flow graphs*. These are acyclic graphs representing three different types of dependency between two events p and q: strong dependency (p must follow q), weak dependency (p may follow q), conditional dependency (if both p and q occur then p must follow q).

We propose a translation of Lustre into well-triggered synchronous systems. This translation is modular and exhibits not only data-flow connections between nodes but also their synchronization by using clocks.

Keywords

synchronous systems, priority Petri nets, modal flow graphs, Lustre, BIP (Behavior-Interaction-Priority)

1. INTRODUCTION

Synchronous systems are composed of strongly synchronized parallel components. Their global behavior is characterized by runs consisting of successive computation steps. In each step, all components perform some quantum of computation. This ensures a built-in fairness between components in sharing resources, usually enforced by using static scheduling policies. Synchronous computation models are particularly adequate for hardware, real-time systems and streaming systems. Their main advantage over asynchronous computation models is efficiency and predictability (determinacy), in particular thanks to lightweight analysis techniques for deciding deadlock-freedom and timeliness. Nonetheless, for general applications an adequate mix of synchronous and asynchronous computation is necessary for optimal use of resources e.g. GALS models [2].

A non trivial open problem is the design of systems consistently integrating synchronous and asynchronous subsystems e.g. one in Simulink and another in ADA. This requires in principle, the use of a common semantic model encompassing both the synchronous and the asynchronous formalism.

The BIP (Behavior, Interaction, Priority) component framework is a formalism for the description of component-based systems consisting of heterogeneous components [1]. It allows the description of systems as the composition of generic atomic components characterized by their behavior and their interfaces. It supports a system construction methodology based on the use of two families of composition operators: interactions and priorities. Interactions are used to specify multiparty synchronization between components as the combination of two protocols: rendezvous (strong symmetric synchronization) and broadcast (weak asymmetric synchronizations). Priorities between interactions are used to restrict non determinism inherent to parallel systems. They are particularly useful to model scheduling polices.

In contrast to existing formal frameworks, BIP is expressive enough to directly model any coordination mechanism between components [3]. It has been successfully used to model complex systems including mixed hardware/software systems and complex software applications.

In this paper, we show how the basic execution mechanisms underlying synchronous data-flow systems can be modeled in BIP. We define a notion of *synchronous BIP component* which differs from general components in that its behavior is described by a step. Steps of components are delimited by a specific transition labeled by a port *sync* and executed synchronously by all components. The behavior of a component in a *step* is described by a safe extended priority Petri net. This is a safe Petri-net whose transitions are labeled with elements of a set of ports P and a *priority order*, a strict partial order $\prec \subset P \times P$. Furthermore, transitions may be labeled with guards and functions representing data transformations. The Petri net has a set of initial and a set of final places. When only final places are marked, a step can terminate by executing the specific transition labeled by *sync*. Termination of a step consists in removing the tokens from final places and putting a token in each initial place. Implicitely, the priority order requires that *sync* has lower priority than any other port to ensure maximal computation in a step.

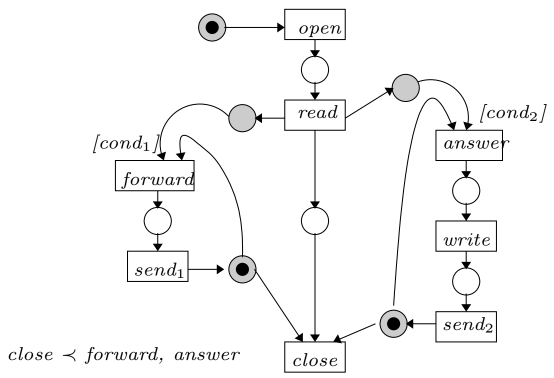


Figure 1: Email treatment

Figure 1 shows a priority Petri net describing the treatment of an email in one computation step. The *sync* transition is not explicitly represented. Initial places are marked with a token; final places are grayed. Transitions are enabled when their input transitions are marked and the associated condition is true. The priority order restricts choices amongst enabled transitions (ports). That is if both *forward* and *close* are enabled then *forward* is executed. Therefore, depending on the conditions $[cond_1]$ and $[cond_2]$, the possible execution sequences are : *open read close*, *open read forward* $send_1$ *close*, *open read answer write* $send_2$ *close*, *open read (forward* $send_1$ $||$ *answer write* $send_2$*) close* where $||$ is the interleaving of sequences.

We define composition of synchronous components as a partial internal operation parameterized by a set of *interactions*. Given a set of synchronous components, we get a product component by composing their Petri nets and their priority orders.

An essential property of synchronous systems is termination of steps, in particular steps must be deadlock-free. Another requirement is confluence of computation within a step which means that the overall behavior is deterministic when system states are observed only at the end of each step. For some synchronous languages e.g. Lustre, these properties can be ensured by checking very simple sufficient conditions [12].

We provide results guaranteeing deadlock-freedom and confluence for a class of synchronous systems encompassing most of the existing executable synchronous formalisms. We define the class of *modal flow components*. They are a subclass of synchronous components where priority Petri nets are replaced by *modal flow graphs*. These correspond to a subclass of priority Petri nets for which deadlock-freedom and confluence can be decided at low cost. Modal flow graphs are structures expressing dependency relations between events. Similar structures such as [13, 16, 17] have been proposed and used in different contexts. An important difference between modal flow graphs and related formalisms is the use of three different *modalities* characterizing dependency between events. For a given set of ports P, a modal flow graph is a directed acyclic graph with nodes P and edges representing the union of three binary relations. Each relation expresses a different kind of causal dependency (modality) between pairs of ports p and q:

- q *strongly depends* on p if the execution of p *must* be followed by the execution of q. That is, p and q cannot be executed independently, only the sequence pq is possible.
- q *weakly depends* on p if the execution of p *may* be followed by q. That is either p can be executed alone or the sequence pq.
- q *conditionally depends* on p if when both p and q are executed, then q must follow p. Conditional dependency requires that if p and q occur then only the sequence pq is possible; otherwise p or q may be independently executed.

In Figure 2, we show the modal flow graph corresponding to the Petri net of Figure 1. Bold, simple and dashed arrows represent respectively strong, weak and conditional dependency relations.

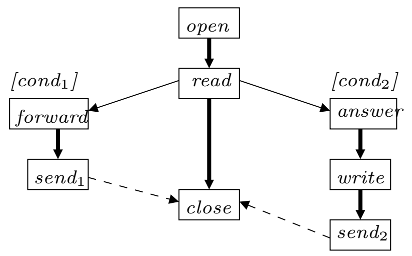


Figure 2: Modal flow graph for email treatment

The semantics of a modal flow graph is a priority Petri net. It associates with each port a transition of the Petri net. The associated priority order is the inverse of the causality order, that is causes have higher priority than consequences.

We show that modal flow graphs are deadlock-free if they are *well-triggered*. This property expresses consistency between the three types of dependency. It also guarantees confluence under some conditions of non interference of concurrent computations.

To illustrate the results, we translate the Lustre language into synchronous BIP. The translation method follows the BIP system construction methodology and involves:

- *Definition of the set of atomic components.* We need atomic components for modeling *flows* (variables) and *clocks* as well as for modeling *combinatorial*, *delay*, *sampling* and *interpolation* Lustre operators. As in [15], atomic components have two kinds of ports: 1) *data ports* associated with data variables and used to input and output data between components and 2) *control ports*, used to enforce (partially) the control flow of computation within a step.
- *Definition of the interactions between atomic components.* We use two classes of interactions: 1) interactions which implement the data flow relation between

data ports and 2) interactions realizing strong synchronization between control ports of components.

The translation is modular and makes explicit all the interactions needed to perform a synchronous computation in a inherently parallel (component-based) system.

The paper is structured as follows. Synchronous components and their composition are presented in section 2. Section 3 is the main section of the paper. It presents the sub-class of synchronous components defined by using modal flow graphs as well as sufficient conditions for deadlock-freedom and confluence. Section 4 presents the concrete application of modal flow graphs for modeling of synchronous systems. Related work is presented in Section 5 and in Section 6, we conclude and present future work directions. Detailed proofs of propositions and theorems can be found in the technical report [5].

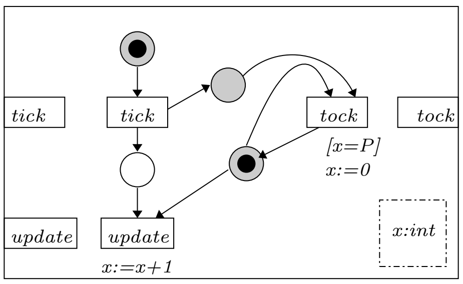
2. SYNCHRONOUS COMPONENTS

We present synchronous components and their semantics. The behavior of a synchronous component within a synchronous computation step is a safe extended priority Petri net with given sets of initial and final places. When only final places are marked, termination may terminate by removing tokens from final places and putting tokens to initial places.

Definition 1 (synchronous component: syntax). *A synchronous component B is a tuple $(X, P, N, \prec)$ where:*

- *X is a set of data variables,*
- *P is a set of ports p, each one labelled with a subset of variables $X_p \subseteq X$, the ones exported on interactions through p,*
- *$N = (L, T, F, L_0, L_f)$ is an extended 1-safe Petri net:*
  - *L is a finite set of places,*
  - *T is a finite set of transitions τ labelled by (p_τ, g_τ, f_τ) where $p_\tau \in P$ is the port triggered by the transition τ, g_τ is the guard of τ, that is a predicate on X and f_τ is the update function associated with τ, that is a state transformer defined on X,*
  - *$F \subseteq L \times T \cup T \times L$ is the token flow relation,*
  - *$L_0 \subseteq L$ is the set of initial places,*
  - *$L_f \subseteq L$ is the set of final places,*
- *$\prec \subseteq P \times P$ is a priority order on ports, that is a strict partial order on the set of ports.*

Example 1. *Figure 3 shows a synchronous BIP component that produces a tock every P ticks. At every step, it executes the* tick *transition and then, during the same step, it increases the local variable x by executing the* update *transition. Whenever x reaches the value P, the component can also execute the* tock *transition and reset x to 0. In this situation, the* tock *and* update *transitions are conflicting, however, the associated priorities enforce the execution of* tock *before* update *if both transitions are possible.*



update $\prec$ *tock* $\prec$ *tick*

Figure 3: Tick-tock synchronous component

In order to define the operational semantics for synchronous components, let us first introduce some notations. Given a Petri net $N = (L, T, F, L_0, L_f)$ we define the set of 1-safe markings $\mathcal{M}$ as the set of functions $m : L \to \{0, 1\}$. Given two markings m_1, m_2, we define inclusion $m_1 \leq m_2$ iff for all $l \in L$, $m_1(l) \leq m_2(l)$. Also, we define addition $m_1 + m_2$ as the marking m_{12} such that, for all $l \in L$, $m_{12}(l) = m_1(l) + m_2(l)$. Given a set of places $K \subseteq L$, we define its characteristic marking m_K by $m_K(l) = 1$ for all $l \in K$ and $m_K(l) = 0$ for all $l \in L \setminus K$. Moreover, when no confusion is possible from the context, we will simply use K to denote its characteristic marking m_K. Finally, for a given transition τ, we define its pre-set ${}^\bullet\tau$ (resp. post-set $\tau^\bullet$) as the set of places flowing to (resp. from) that transition ${}^\bullet\tau = \{l \mid (l, \tau) \in F\}$ (resp. $\tau^\bullet = \{l \mid (\tau, l) \in F\}$).

Definition 2 (synchronous component: semantics). *The semantics of a synchronous component $B = (X, P, N, \prec)$ with $N = (L, T, F, L_0, L_f)$ is defined as the labelled transition system $S = (Q, \Sigma, \to)$ where*

- *$Q = \mathcal{M} \times \mathcal{V}$ is the set of states defined by:*
  - *$\mathcal{M} = \{m : L \to \{0, 1\}\}$ the set of 1-safe markings,*
  - *$\mathcal{V} = \{v : X \to \mathcal{D}\}$ the set of valuations of variables,*
- *$\Sigma = P \cup \{sync\}$ is the set of labels,*
- *$\to \subseteq Q \times \Sigma \times Q$ is the set of transitions defined by the rules in Figure 4*

$$1)\ \frac{\tau \in T \quad {}^\bullet\tau \leq m \quad m' = m - {}^\bullet\tau + \tau^\bullet \quad g_\tau(v) = true \quad v' = f_\tau(v)}{(m, v) \xrightarrow{p_\tau}_0 (m', v')} \qquad 2)\ \frac{m \leq m_{L_f}}{(m, v) \xrightarrow{sync}_0 (m_{L_0}, v)}$$

$$3)\ \frac{(m, v) \xrightarrow{p}_0 (m', v') \quad \neg(\exists p'. p \prec p' \ \wedge\ (m, v) \xrightarrow{p'}_0)}{(m, v) \xrightarrow{p} (m', v')} \qquad 4)\ \frac{(m, v) \xrightarrow{sync}_0 (m', v') \quad \neg(\exists p. (m, v) \xrightarrow{p}_0)}{(m, v) \xrightarrow{sync} (m', v')}$$

Figure 4: Operational Semantics Rules

Rules (1) and (2) define moves $\to_0$ of the behavior without priorities. Rule (1) is the usual firing rule of transitions in Petri nets extended with global data. Rule (2) defines *sync* transitions which denote the end of a step and the beginning of the next one. *Sync* transitions can be executed whenever the current marking does not contain tokens in non-final places, and their effect is to restore the initial marking, while keeping the data unchanged. Rules (3) and (4) define the moves $\rightarrow$ of a synchronous component, by restricting $\rightarrow_0$ with respect to priorities. Rule (3) is simply the application of the priority rule specified by the priority order $\prec$. Rule (4) enforces highest priority of all the transitions ports of the component with respect to *sync* transitions, denoting step termination.

Definition 3 (interaction). *An interaction a is a triple (P_a, G_a, F_a) where*

- *P_a, is a set of ports, the support set of the interaction,*
- *G_a is the interaction guard, that is a boolean predicate defined on variables $(X_p)_{p\in P_a}$ exported through ports belonging to the interaction,*
- *F_a is an update (or transfer) function, that is a state transformer on variables $(X_p)_{p\in P_a}$*

We define composition parameterized by interactions as an internal operation of synchronous components. This operation is partial: the result of the composition is defined as a synchronous component only if the priority order associated to it is acyclic.

Definition 4 (synchronous component: composition). *Let $\{B_i = (X_i, P_i, N_i, \prec_i)\}_{i=1,n}$ be a set of synchronous components defined on disjoint sets of variables and ports. Let γ be a set of interactions on ports $\cup_{i=1}^n P_i$ such that each interaction uses at most one port of every component, that is for all $a \in \gamma$, for all $i \in \overline{1,n}$, $|P_a \cap P_i| \leq 1$. The composition $\gamma(B_1, ..., B_n)$ is a partial operation defining the synchronous component $B = (X, P, N, \prec)$ where*

- *the set of variables $X = \cup_{i=1}^n X_i$,*
- *the set of ports P is the set of interactions γ. Moreover, for each interaction $a \in \gamma$, we define its set of exported variables $X_a = \cup_{p\in P_a} X_p$,*
- *the Petri net $N = (L, T, F, L_0, L_f)$ is obtained from the set of the Petri nets $\{N_i = (L_i, T_i, F_i, L_{0i}, L_{f_i})\}_{i=1,n}$ as follows:*
  - *the set of places $L = \cup_{i=1}^n L_i$,*
  - *the set of transitions T corresponds to sets of interacting transitions*

    $$T = \left\{ \langle a, \{\tau_i\}_{i\in I}\rangle \mid \begin{array}{l} a \in \gamma,\ I \subseteq \overline{1,n} \text{ such that} \\ \forall i \in I. \tau_i \in T_i \text{ and } P_a = \{p_{\tau_i}\}_{i\in I} \end{array} \right\}$$

    Moreover, for any transition $\tau = \langle a, \{\tau_i\}_{i\in I}\rangle$ the associated port p_τ is the interaction a, the guard $g_\tau = \wedge_{i=1}^n g_{\tau_i} \wedge G_a$, and the update function $f_\tau = (\sqcup_{i=1}^n f_{\tau_i}) \circ F_a$, where $\sqcup$ is the function consisting in computing each function f_{τ_i} - the order of computation is irrelevant as the data of the components are disjoint,
  - *the token flow relation F of the net is defined as*

    $$F = \begin{array}{l} \{(l, \langle a, \{\tau_i\}_{i\in I}\rangle) \mid \exists j \in I. l \in {}^{\bullet}\tau_j\} \cup \\ \{(\langle a, \{\tau_i\}_{i\in I}\rangle, l) \mid \exists j \in I. l \in \tau_j^{\bullet}\} \end{array}$$
  - *the set of initial places L_0 is $\cup_{i=1}^n L_{0i}$,*
  - *the set of final places L_f is $\cup_{i=1}^n L_{f_i}$,*
- *the relation $\prec$ is the strict transitive closure of the relation $\prec_0$ defined as the extension of individual priority orders $\prec_i$ to interactions: $a_1 \prec_0 a_2$ iff $\exists i = \overline{1,n}$. $\exists p_{i_1} \in P_{a_1} \cap P_i$. $\exists p_{i_2} \in P_{a_2} \cap P_i$ such that $p_{i_1} \prec_i p_{i_2}$. The composition is defined only if this relation is a strict partial order.*

Example 2. *Composition of synchronous components is illustrated in Figure 5. Two tick-tock components are composed by synchronizing the* tock *of the first component and the* tick *of the second one. The resulting component produces a* tock$_2$ *every P^2 ticks.*

3. MODAL FLOW COMPONENTS

Modal flow components are synchronous components defined by modal flow graphs, which correspond to a particular class of priority Petri nets. We define the semantics of atomic flow components as well as their composition.

Definition 5 (modal flow component: syntax). *A modal flow component B^f is defined as a tuple (X, P, D):*

- *X is a set of data variables,*
- *P is a set of ports p, each one being associated with a triple (X_p, g_p, f_p) where*
  - *$X_p \subseteq X$, the set of variables exported through p,*
  - *g_p, the triggering condition of p, that is a predicate defined on X,*
  - *f_p, an update function, that is a state transformer function on X*
- *$D = (D_s, D_w, D_c)$ is a triple of causal dependency relations between ports. The relations $D_s, D_w, D_c \subseteq P \times P$ denote respectively strong, weak and conditional dependency and are such that their union $D_s \cup D_w \cup D_c$ is acyclic.*

Example 3. *In Figure 6, the tick-tock synchronous component of Figure 3, is represented as a modal flow component. The* tock *transition is weakly dependent on* tick *transition. Also, the* update *transition is strongly dependent on* tick *and conditionally dependent on* tock. *The only possible executions within a step are therefore* tick update *or* tick tock update.

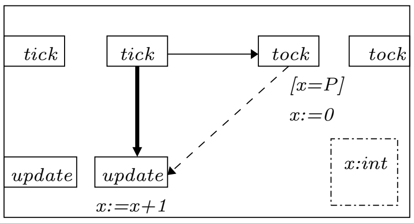


Figure 6: Tick-tock modal flow component

We use the following notations. For fixed $x = s, w, c$, we write $p \overset{x}{\rightsquigarrow} q$ to denote $(p, q) \in D_x$. We write $\overset{x}{\rightsquigarrow}{}^*$ to denote the reflexive and transitive closure of $\overset{x}{\rightsquigarrow}$. We write $p \rightsquigarrow q$ to denote $(p, q) \in D_s \cup D_w \cup D_c$ and $\rightsquigarrow^*$ for its reflexive and

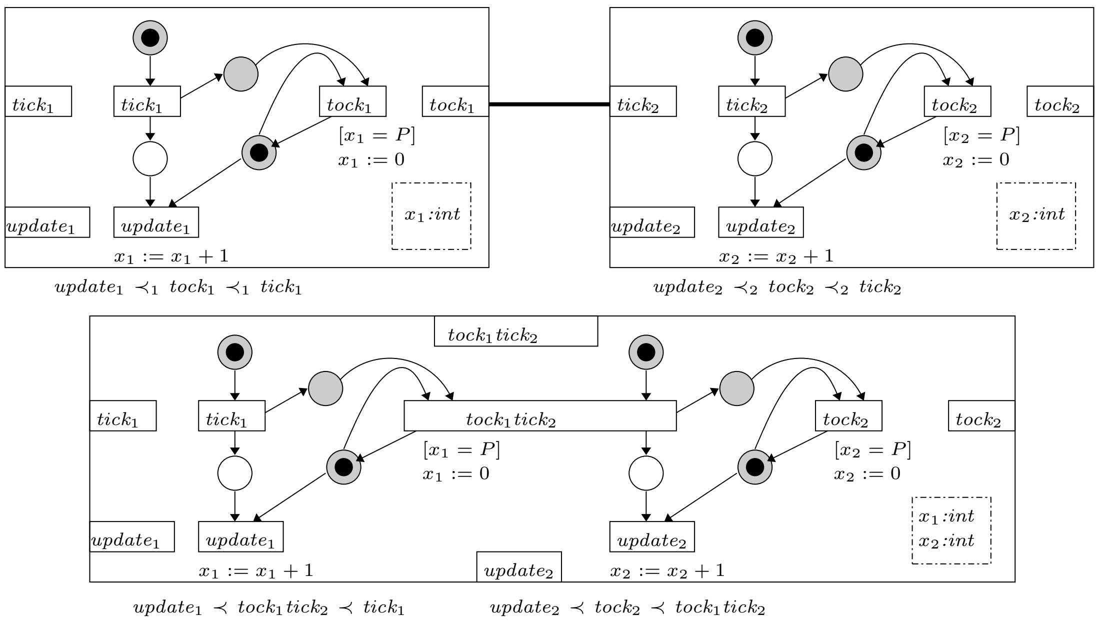


Figure 5: Example of composition

transitive closure. Two ports p and q are called independent (noted $p \sharp q$) iff neither $p \rightsquigarrow^* q$ nor $q \rightsquigarrow^* p$.

For fixed $x = s, w, c$, we denote by $\min_x P$ the set of minimal ports with respect to D_x, that is $\min_x P = \{q \mid \neg \exists p . p \overset{x}{\rightsquigarrow} q\}$. We write $\min P$ to denote the set of minimal ports with respect to $D_s \cup D_w \cup D_c$, that is $\min P = \{q \mid \neg \exists p . p \rightsquigarrow q\}$.

We call a modal flow component *well triggered* iff:

1. each port p has a unique minimal strong cause $|\{q \in \min_s P \mid q \overset{s}{\rightsquigarrow}{}^* p\}| = 1$
2. each port p has exclusively either strong or weak causes.

For a port p, we denote its minimal strong cause by $root(p)$.

Notice that, well-triggered modal flow graphs can be decomposed as shown in Figure 7. The strong dependency relation defines a set of connected subgraphs involving all the ports of the component. Each one of these subgraphs has a single root which is the common cause for its ports. Weak dependencies express triggering of the root of a subgraph by some port of another subgraph. Finally, conditional dependencies may relate ports of different subgraphs provided the acyclicity property is not violated.

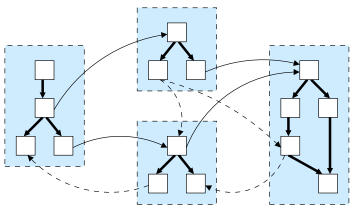


Figure 7: Well-triggered components.

We define the semantics of modal flow components which are well-triggered in terms of synchronous components.

Definition 6 (modal flow component: semantics). *The semantics of a well-triggered modal flow component $B^f = (X, P, D)$ is the synchronous component $B = (X, P, N, \prec)$:*

- *the set of variables is X,*
- *the set of ports is P; moreover, for each port p the associated set of exported variables is X_p,*
- *the Petri net $N = (L, T, F, L_0, L_f)$ is defined by:*
  - *the set of places L is isomorphic to the set $D_s \cup D_w \cup D_c$ augmented with the set of minimal ports. That is $L = \{l^x_{p,q} \mid p \overset{x}{\rightsquigarrow} q\} \cup \{l_p \mid p \in \min P\}$,*
  - *the set of transitions T is isomorphic to the set of ports P, that is $T = \{t_p \mid p \in P\}$. Moreover, for any transition t_p we associate its port p, the guard g_p, and the update function f_p,*
  - *the token flow relation $F \subseteq L \times T \cup T \times L$, is constructed as follows:*
    - *for each $p \in \min P$ add (l_p, t_p) to F,*
    - *for each dependency $p \overset{x}{\rightsquigarrow} q$ add $(t_p, l^x_{p,q}), (l^x_{p,q}, t_q)$ to F,*
    - *for each conditional dependency $p \overset{c}{\rightsquigarrow} q$ add $(l^c_{p,q}, t_{root(p)})$ to F,*
  - *the set of initial places L_0 corresponds to minimal ports and conditional dependencies that is $L_0 = \{l_p \mid p \in \min P\} \cup \{l^c_{p,q} \mid p \overset{c}{\rightsquigarrow} q\}$,*
  - *the set of final places L_f consists of all places corresponding to all but strong dependencies $L_f = L \setminus \{l^s_{p,q} \mid p \overset{s}{\rightsquigarrow} q\}$.*
- *the priority order $\prec = (\rightsquigarrow^*)^{-1} \setminus Id$, that is $p \prec q$ iff $q \rightsquigarrow^* p$ and $q \neq p$, for all $p, q \in P$.*

Example 4. *The tick-tock modal flow component shown in Figure 6 is well-triggered. Its semantics is defined by the tick-tock synchronous component in Figure 3.*

The Petri nets representing modal flow components satisfy the following trivial properties: 1) every place has at most one incoming transition, 2) every place $l^c_{p,q}$ corresponding to a conditional dependency belongs to a cycle, 3) initially, there is precisely one token in every cycle of the net.

Notice that the above contruction rules of the Petri net enforce the three kinds of dependencies between ports. Strong and weak dependencies are obviously enforced by the net. An initial empty place $l_{p,q}$ between t_p and t_q will prevent the execution of t_q before t_p. Moreover, if the place is not final, the execution of t_p will require the execution of t_q before the end of the step. Concerning conditional dependencies $p \overset{c}{\rightsquigarrow} q$, the Petri net ensures that the execution of t_q disables any further execution of $t_{root(p)}$ and consequently of t_p.

The following proposition gives additional properties.

Proposition 1. *Priority Petri nets representing modal flow graphs meet the following properties*

1. *Every reachable marking has at most one token in every cycle of the net.*
2. *Each transition is executed at most once in every step.*
3. *Are 1-safe.*

Definition 7 (modal flow components: composition). *Let $\{B^f_i = (X_i, P_i, D_i)\}_{i=1,n}$ be a set of modal flow components defined on disjoint sets of variables and ports. Let γ be a set of interactions on ports $\cup_{i=1}^n P_i$ such that*

- *each interaction uses at most one port of every component, that is for all $a \in \gamma$, for all $i \in \overline{1,n}$ $|P_a \cap P_i| \leq 1$,*
- *each port belongs to at most one interaction, that is for all $p \in \cup_{i=1}^n P_i$ $|\{a \mid p \in P_a\}| \leq 1$,*

We define the composition $\gamma(B^f_1, ..., B^f_n)$ as the modal flow component $B^f = (X, P, D)$ where

- *the set of variables X is $\cup_{i=1}^n X_i$,*
- *the set of ports P is the set of interactions γ. Moreover, for every interaction a of γ, we define the guard $g_a = (\wedge_{p \in P_a} g_p) \wedge G_a$, the exported variables $X_a = \cup_{p \in P_a} X_p$ and the transfer function $f_a = (\sqcup_{p \in P_a} f_p) \circ F_a$*
- *the set of dependencies $D = (D_s, D_w, D_c)$ are inherited from atomic components, that is for every $x = s, w, c$ we have $D_x = \{(a_1, a_2) \mid \exists i \in \overline{1,n}. \exists p_1 \in P_{a_1} \cap P_i, p_2 \in P_{a_2} \cap P_i$ such that $(p_1, p_2) \in D_{xi}\}$*

Notice that composition amounts to merging nodes belonging to the same interaction without changing the dependency relations. Composition is a partial operation because, its result is a valid modal flow component only if the set of derived dependencies is acyclic.

Example 5. *The composition of modal flow components is illustrated in Figure 8. Three tick-tock modal flow components are composed* sequentially *by synchronizing the* tock *of each component with the* tick *of its left neighbour and by keeping all the other transitions unchanged.*

Moreover, let us observe that composition of modal flow graphs is not the same operation as composition of Petri nets. These differ because conditional dependencies do not have a local interpretation e.g., $p \overset{c}{\rightsquigarrow} q$ implies that execution of t_q disables further execution of $t_{root(p)}$. But, the minimal strong cause *root*(p) of p can denote different actions within the modal flow graph of p and within the composed graph.

The following theorems give sufficient conditions for deadlock-freedom and confluence of computation (i.e, determinism) for synchronous steps of modal flow components.

Theorem 1 (deadlock-freedom). *A well-triggered modal flow component $B^f = (X, P, D)$ is* deadlock-free *if every port p with strong causes has its guard true: $g_p = true$*

Theorem 2 (confluence). *A well-triggered modal flow component $B^f = (X, P, D)$ is confluent if for every independent ports $p_1 \sharp p_2$, their associated guarded actions are independent, that is:*

- $X_{p_1} \cap X_{p_2} = \emptyset$
- $use(g_{p_1}) \cap (X_{p_2} \cup def(f_{p_2})) = \emptyset$
- $use(g_{p_2}) \cap (X_{p_1} \cup def(f_{p_1})) = \emptyset$

4. APPLICATIONS

To illustrate the use of modal flow graphs for modeling synchronous systems, we provide a modular translation of Lustre [12] into modal flow graphs. Similar translations can be made for other synchronous languages or graphical formalisms [10, 6].

Lustre [12] is a dataflow synchronous language for programming reactive systems. Lustre programs operate on flows of values, that are infinite sequences $(x_0, x_1, \cdots, x_n, \cdots)$ of values at logical time instants $0, 1, \cdots, n$. An abstract syntax for Lustre programs is shown below. *In* (resp. *Out*) denotes the set of input (resp. output) flows of a program node. Symbols N, E, x, v, b denote respectively node names, expressions, flows, boolean flows and constant values.

$$\begin{array}{rcl}
\textit{program} & ::= & \textit{node}^+ \\
\textit{node} & ::= & \textbf{node}\ N\ (\textit{In})\ (\textit{Out})\ \textit{equation}^+ \\
\textit{equation} & ::= & x = E \mid \\
 & & x, \cdots, x = N(E, \cdots, E) \\
E & ::= & x \mid v \mid op(E, \cdots, E) \mid \textbf{pre}(E,\ v) \mid \\
 & & E\ \textbf{when}\ b \mid \textbf{current}\ E
\end{array}$$

A Lustre program is structured as a set of *nodes*. Each node computes output flows from input flows. Output flows are defined either directly by means of equations of the form $x = E$, meaning $x_n = E_n$ for any time instant $n \geq 0$ or, as the output of other (already defined) nodes instantiated with particular inputs $x, ... = N(E, ...)$.

The basic operators used in expressions E, are combinatorial operator (op), unit delay (**pre**), sampling (**when**) and interpolation (**current**). Combinatorial (memory-less) operators include usual boolean, arithmetic and relational operators. The unit delay **pre** operator gives access to the value of its argument at the previous time instant. For example, the

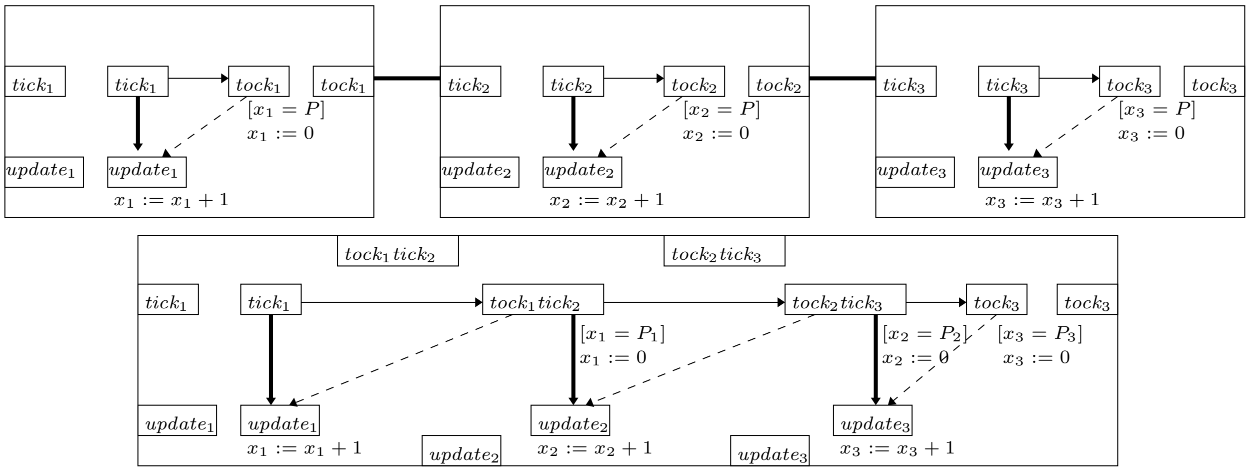


Figure 8: Example of composition

expression $E' = \mathbf{pre}(E, v)$ means $E'_0 = v$ and $E'_i = E_{i-1}$, for all $i \geq 1$.

In Lustre each flow (and expression) is associated with a logical clock. Implicitely, there always exists a unique, fastest, *basic clock* which defines the step (or basic clock cycle) of a synchronous program. Depending on this clock, other slower clocks can be defined as the sequences of time instants where boolean flow variables take the value *true*. In order to define and manipulate flows operating on slower clocks, Lustre provides two additional operators. The *sampling* operator **when**, samples a flow depending on a boolean flow. The expression $E' = E$ **when** b, is the sequence of values E when the boolean flow b is *true*. The expression E and the boolean flow b have the same clock, while the expression E', operates on a slower clock defined by the instants at which b is true. The *interpolation* operator **current**, interpolates an expression on the clock which is immediately faster than its own clock. The expression $E' =$ **current** E, takes the value of E at the last instant when b was true, where b is the boolean flow defining the slower clock of E.

We consider statically correct programs which satisfy the static semantics rules of Lustre [11]. These rules exclude programs containing cyclic, dependent equations, recursive calls of nodes as well as combinatorial operators applied to expressions having different clocks.

We define modular operational semantics for Lustre, first for single-clock programs and then for multi-clock programs.

4.1 Single-clock synchronous programs

The single-clock subset of Lustre is generated by using only combinatorial and unit delay operators. All flows are sampled (indexed) by the basic clock.

The translation from Lustre to modal flow graphs is modular. Each node is represented by a well-triggered modal flow component with two kinds of ports: *act* control ports and *input* (in) or *output* (out) data ports. An *act* port is triggered by the basic clock and initiates the step of the node. The *in* (resp. *out*) data ports carry data input (resp. output) read (resp. produced) by the node. Additionally, modal flow graphs may contain internal ports and variables, depending on the specific computation carried by the node.

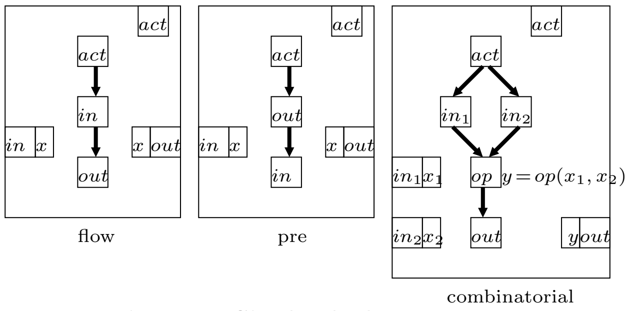


Figure 9: Single-clock operators

The modal flow components shown in Figure 9 correspond to basic Lustre elements: flow, pre operator and combinatorial operator. The flow component whenever activated through the *act* port, reads a value through the *in* port and outputs this value through the *out* port in the same step. The **pre** component has a local variable x. Whenever it is activated through *act*, it outputs the current value x, then it inputs and assigns a new value to x to be used in the next step. The combinatorial component starts a step when it is triggered through the *act* port. Then it reads input values in some arbitrary order, performs its specific computation, and finally, produces an output value.

The modal flow component representing a single-clock Lustre node is obtained by composing a set of components by using a set of interactions defined as follow.

• **components**: For each input and output flow declared in the node we add a *flow* component. For each **pre** (resp. combinatorial) expression occuring within the equations, we add a *pre* (resp. *combinatorial*) component. Moreover, for each subnode called within equations we add its corresponding modal flow component.

• **interactions**: Interactions are of two types: *control flow* and *data flow*. A single control flow interaction realizes strong synchronization between all the *act* ports of all components. Data flow interactions synchronize one *out* port to one or more *in* ports. They are used to propagate data from input flow components to expression components and from expression components to output flow components or other expression components according to the syntactic structure of expressions and equations.

Example 6. *Figure 10 shows a discrete integrator written in Lustre and its corresponding synchronous network of operators.*

node `Integrator(i:  int)`
  returns o: int;
let o = i + **pre**(o,0); **tel**;

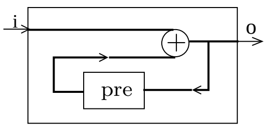
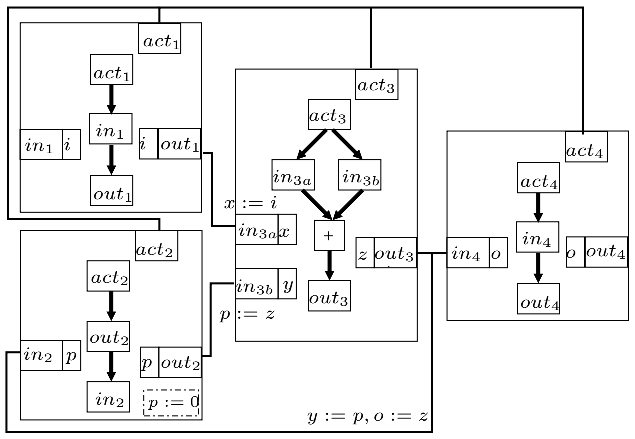


Figure 10: Integrator

The representation of this node as a composition of modal flow components is shown in Figure 11 (top). The atomic modal flow components correspond to the **pre** *operator, the combinatorial + operator, the input flow and the output flow. In addition to the* act *interaction, there are three interactions for data transfer from outputs to inputs: 1) from the input flow component to the + component, 2) from the* **pre** *component to the + component and 3) from the + operator to the output flow component and back to the* **pre** *component. The result of the composition is shown in Figure 11 (bottom).*



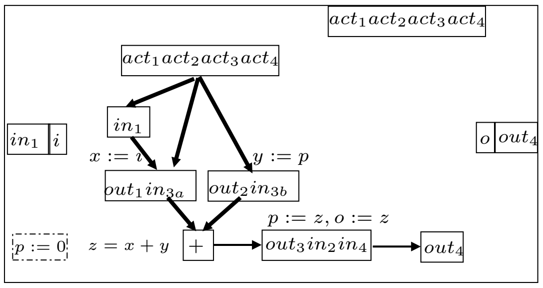


Figure 11: The Integrator modal flow component

The following theorem is a consequence of modularity of translation and of the following facts: 1) the modal flow graphs corresponding to the basic constructs of Lustre are well-triggered; 2) for statically correct Lustre programs [12], composition of the basic modal flow graphs preserves well-triggeredness.

Theorem 3. *Every statically correct single-clock Lustre node N is represented by a well-triggered modal flow component B^f_N such that:*

1. *it has a unique root which is an* act *port;*
2. *all its dependencies are strong;*
3. *it is deadlock-free and confluent;*
4. *simulates the micro-step Lustre semantics [11] of N.*

4.2 Multi-clock synchronous programs

In Figure 12, we provide two components modeling respectively the sampling and interpolation operators of Lustre. Both components have two control ports, act_i and act_o triggering respectively the input *in* and the output *out* data ports. For a sampling component, act_o depends weakly on act_i, and moreover, the output *out* dependends conditionally on the input *in*. Thus an input is always read and whenever required, an output is produced with the most recent value of the input – which is precisely the interpretation of sampling. For the interpolation component, we have the opposite: act_i depends weakly on act_o but *out* conditionally depends on *in*. Thus the output is always produced with the most recent value of the input. The last modal flow graph in Figure 12 describes an additional component, the *derived clock* component corresponding to a boolean flow b. This component is used to initiate all the computations carried on the clock b. Intuitively, it triggers the slower *clock* port only after its base clock *act* has been triggered and if the value obtained through the data input *in* port is true.

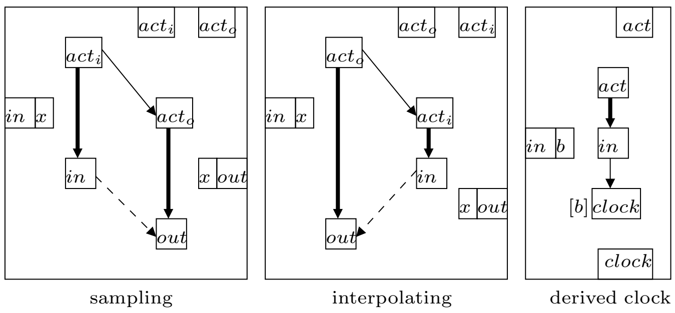


Figure 12: Multi-clock operators

We apply a similar modular construction method for building modal flow components for multi-clock nodes.

• **components:** First, we add a *derived clock* component for each clock (i.e, **when** b). Second, we add a *sampling* (resp. *interpolation*) component for each sampling (resp. interpolation) expression occuring within the equations of the node.

• **interactions:** The data flow interactions are the same as for the single-clock case, with the addition that data is also propagated to the input port of derived clocks. Regarding control flow interactions, we add one interaction which synchronizes all the *act* ports of flows and expressions sampled on the basic clock. In addition, for each derived clock component, we add an interaction which synchronizes its *clock* port with all *act* ports of flows and expressions sampled by that clock.

Example 7. *Consider the following Lustre program:*

```
node input_handler(a: bool, x: int when a)
returns y: int;
let   y = if a then current x else pre(y, 0); tel ;

node output_handler(c: bool, y: int) returns z: int when c;
var   yc: int when c;
let   yc = y when c; z = yc * yc ; tel ;

node  input_output(a,c: bool, x: int when a)
returns z: int when c;
var   y: int;
let   y = input_handler(a, x); z = output_handler(c, y); tel;
```

Depending on an input value x triggered by an input clock a, the `input_output` *node produces a corresponding output value z triggered by an output clock c, by using the most recent available value of the input.*

The main node is the `input_output` *node which interconnects the two nodes,* `input_handler` *and* `output_handler`*. The* `input_handler` *node receives at every moment the boolean value a. An integer value x is received only when a is* true*. The output value y is an integer produced at every moment by interpolating the value of x. The* `output_handler` *node receives at every moment a boolean c and an integer variable y. It produces an output z by sampling y when c is* true*. Finally, the* `input_output` *top node connects the output of the* `input_handler` *to the input of the* `output_handler`.

Figure 13 shows the modal flow component representing the system. Its modal flow graph is obtained after composition and static simplification of the modal flow graphs of the **input_output** *node. It can be decomposed into three subgraphs with activation ports* act, act_a *and* act_c *corresponding respectively to the basic,* **when a***, and* **when c** *clocks.*

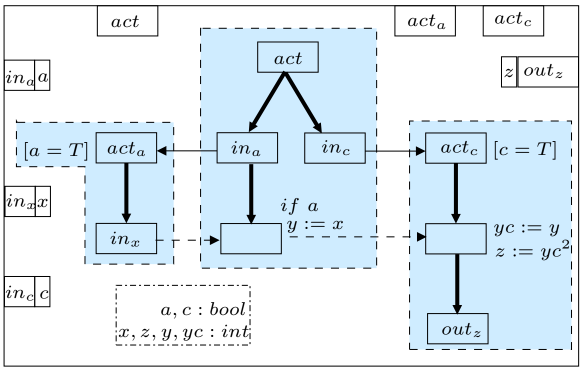


Figure 13: The input/output handler

The following theorem establishes the correctness of our translation.

THEOREM 4. *Every statically correct multi-clock Lustre node N is represented by a well-triggered modal flow component B_N^f which:*

1. *has multiple (control) root* act *ports, one for each clock in the Lustre program, and multiple data* in/out *ports;*
2. *the subgraphs defined by strong dependencies are connected through weak dependencies into a tree;*
3. *is deadlock-free and confluent;*
4. *simulates the micro-step Lustre semantics [11] of N*

4.3 Experimental work

We have studied and implemented a translator from Lustre to BIP synchronous components which directly generates Petri nets without using modal flow graphs. The translator is currently fully operational. It takes as input Lustre programs and produces full-fledged BIP systems, that can be simulated and analyzed using the BIP toolset [1].

This first approach has several important drawbacks. For the generated BIP programs it is not easy to verify properties guaranteed by construction for some synchronous programs e.g. deadlock-freedom and confluence. Moreover, the BIP compilation chain cannot easily recover the information that the system is indeed synchronous and consequently, it cannot produce optimized code. For example, experiments with concrete Lustre programs show an $600:1$ overhead of execution time between the C code produced by the BIP compiler and executed by the BIP engine, and the flat C code produced by the Lustre compiler. Although, this overhead can be diminished to $20:1$ by applying static composition of components in BIP [4], it still remains high.

Modal flow graphs allow coping with these drawbacks. We are now investigating the possibility to integrate directly modal flow components in BIP. Our results about confluence and deadlock-freedom of modal flow components provide syntactic conditions, easily implementable in an automatic tool. Moreover, modal flow components keep all the data-flow explicit and can be used to generate efficient code, monolithic or not, as synchronous language compilers do.

5. RELATED WORK

Our work as a tentative to bridge the gap between synchronous and asynchronous computation, is related to approaches with similar objectives.

In [15], a model for synchronous components is proposed where steps are described by using automata with final states. Another similarity is the distinction between data ports and control ports. Nonetheless, the latter are activated by controllers which are specific components. The synchronous/reactive domain of the Ptolemy system-level design framework [9] allows component-based description of synchronous systems where synchronous execution is orchestrated by a *director*. Finally, our work has the same general objectives as [2] which studies a compositional framework heterogeneous reactive systems. In contrast to BIP, the framework is denotational and is based on the concept of tags marking the events of the signals of a system.

There are several differences between our work and existing results. Our work is based on operational semantics. It considers synchronous component-based systems as a particular case of the BIP framework which also encompasses general asynchronous computation. Furthermore, we believe that our framework is expressive enough to allow modular translation of synchronous languages into BIP by preserving the structure of the source, as shown for Lustre.

Modal flow graphs without data and only strong dependencies are acyclic partial orders on events. They correspond to acyclic marked graphs which are Petri nets without forward and backward conflicts. Theorem 2 generalizes well-known results for marked graphs [8].

Modal flow graphs with strong dependencies and their composition operation are also similar to *synchronous structures* used in a study of the synchronous model of computation [16]. This model has also some similarities with models such as *modal automata* [14] which distinguish between *must* and *may* transitions or *live sequence charts* [7] which distinguish between *hot* and *cold* events. Nonetheless, modal flow graphs encompass three independent modalities which are all necessary for modular description of synchronous systems, as shown in the paper. Furthermore, for a reasonably general class of modal flow graphs we proposed sufficient conditions for deadlock-freedom and confluence.

6. CONCLUSIONS

We present a general approach for modeling synchronous component-based systems. These are systems of synchronous components strongly synchronized by a common action *sync* that initiates execution steps of each component. Steps can be described by priority Petri nets. Priorities are instrumental for enforcing run-to-completion in the execution of a step. Modal flow graphs are used to define a particular class of Petri nets for which deadlock-freedom and confluence are met by construction provided some easy-to-check conditions hold. This result is the generalization of existing results for classes of Petri nets without conflicts. It allows more general behavior for components given that the semantics of conditional dependencies lead to Petri nets with backward conflicts and priorities.

The definition of synchronous components as a subset of the BIP framework allows their combination with other asynchronous languages that can be translated into BIP. The proposed semantics has maximal parallelism, that is it shows only the absolutely necessary dependencies between events. Execution in a step is non deterministic. However, if the behavior is confluent and the order of execution is irrelevant. The translation of Lustre shows the interplay between data flow and control flow and allows understanding how strict synchrony can be weakened to get more less synchronous computation models.

This work opens the way for exploring problems regarding relations between synchronous and asynchronous systems. It allows integration of synchronous systems theory in an all encompassing component framework [3] without losing advantages such as correctness-by-construction and efficient code generation. This makes possible modeling mixed synchronous/asynchronous systems without artefacts. In Figure 14, we show the principle for modeling GALS. A synchronous system sending data to another synchronous system through a FIFO queue. The input of the queue is triggered by the clock of the sender while its output is triggered by the clock of the receiver. The meaningful integration of synchronous and asynchronous models in this framework is the object of future work.

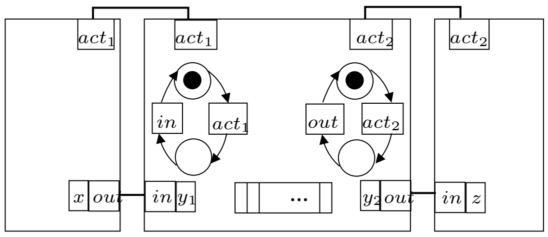


Figure 14: A GALS system in BIP

7. REFERENCES

[1] A. Basu, M. Bozga, and J. Sifakis. Modeling heterogeneous real-time systems in BIP. In *Proceedings of SEFM'06*, pages 3–12. invited talk.

[2] A. Benveniste, B. Caillaud, L. P. Carloni, P. Caspi, and A. L. Sangiovanni-Vincentelli. Composing heterogeneous reactive systems. *ACM-TECS*, 7(4), 2008.

[3] S. Bliudze and J. Sifakis. A notion of glue expressiveness for component-based systems. In *Proceedings of CONCUR'08, LNCS 5201*, pages 508–522, 2008.

[4] M. Bozga, M. Jaber, and J. Sifakis. Source-to-source architecture transformation for performance optimization in BIP. Technical Report TR-2009-3, Verimag.

[5] M. Bozga, V. Sfyrla, and J. Sifakis. Modeling synchronous systems in BIP. Technical Report TR-2009-8, Verimag.

[6] S. L. Campbell, J.-P. Chancelier, and R. Nikoukhah. *Modeling and Simulation in Scilab/Scicos*. Springer.

[7] P. Combes, D. Harel, and H. Kugler. Modeling and verification of a telecommunication application using live sequence charts and the play-engine tool. In *Proceedings of ATVA'05, LNCS 3707*, pages 414–428.

[8] F. Commoner, A. W. Holt, S. Even, and A. Pnueli. Marked directed graphs. *Computer System Sciences*, 5(5):511–523, 1971.

[9] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong. Taming heterogeneity - the Ptolemy approach. *Proceedings of IEEE*, 91(1):127–144, 2003.

[10] P. L. Guernic, T. Gautier, M. L. Borgne, and C. L. Maire. Programming real time applications with Signal. *Proceedings of IEEE*, 79(9):1321–1336, 1991.

[11] N. Halbwachs. About synchronous programming and abstract interpretation. *SCP*, 31(1):75–89, 1998.

[12] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language lustre. *Proceedings of IEEE*, 79(9):1305–1320, 1991.

[13] D. Harel and S. Maoz. Assert and negate revisited: Modal semantics for UML sequence diagrams. *Software and System Modeling*, 7(2):237–252, 2008.

[14] K. G. Larsen, U. Nyman, and A. Wasowski. Modal i/o automata for interface and product line theories. In *Proceedings of ESOP'07, LNCS 4421*, pages 64–79.

[15] F. Maraninchi and T. Bouhadiba. 42: Programmable models of computation for a component-based approach to heterogeneous embedded systems. In *Proceedings of ACM-GPCE'07*.

[16] D. Nowak. Synchronous structures. *Information and Computation*, 204(8):1295–1324, 2006.

[17] Y. Zhou and E. A. Lee. Causality interfaces for actor networks. *ACM-TECS*, 7(3), 2008.