Automated Conflict-Free Concurrent Implementation of Timed Component-Based Models

Ahlem Triki, Borzoo Bonakdarpoor, Jacques Combaz

Verimag Research Report n^o TR-2015-2

Febrary 9, 2015

Automated Conflict-Free Concurrent Implementation of Timed Component-Based Models

AUTEURS = Ahlem Triki, Borzoo Bonakdarpoor, Jacques Combaz

Febrary 9, 2015

Abstract

Correct implementation of concurrent real-time systems has always been a tedious task due to their inherent complex structure; concurrency introduces a great deal of non-determinism, which can potentially conflict with meeting timing constraints. In this paper, we focus on model-based concurrent implementation of timed models. Our *abstract* models consist of a set of components interacting with each other using multi-party interactions. Each component is internally subject to a set of timing constraints. We propose a chain of transformations that starts with an abstract model as input and generates correct-by-construction executable code as output. We show that all transformed models are observationally equivalent to the abstract model through bisimulation proofs and, hence, all functional properties of the abstract model are preserved. To facilitate developing the proofs of correctness, each transformation obtains a model by incorporating a subset of *physical* constraints (e.g., type of communication and global clock synchronization). Our method is fully implemented and validated on a real-time multi-sensor image reconstruction system outperforms the corresponding concurrent implementation managed by a centralized scheduler.

Keywords: Concurrent Real-Time Systems, BIP, correct-by-construction

Reviewers: Saddek Bensalem

How to cite this report:

@techreport {TR-2015-2,
title = {Automated Conflict-Free Concurrent Implementation of Timed Component-Based Models},
author = { Ahlem Triki, Borzoo Bonakdarpoor, Jacques Combaz },
institution = {{Verimag} Research Report},
number = {TR-2015-2},
year = {}
}

1 Introduction

Although concurrent computing is widely used nowadays, especially due to the recent advances in the multi-core and GPU technologies, implementation and deployment of correct concurrent applications are still time-consuming, error-prone, and hardly predictable tasks. This problem becomes even more challenging when the concurrent application is required to meet a set of timing constraints as well, for instance, in computation-intensive real-time embedded systems. This is due to the fact that the developer of a real-time concurrent application not only has to consider typical problems in concurrency (e.g., deadlock/livelock freedom, race conditions, etc), but also should ensure that all subtle interleavings of the application meet the timing constraints.

Model-based software development is a promising approach, where a chain of steps starting from a specification leads to an implementation on a given execution platform. It involves the use of transformation methods and tools for progressively deriving the implementation by making adequate design choices. Such transformations ensure functional correctness, software line productivity, and incorporate extra-functional properties such as timing constraints. Although there have recently been plausible efforts in model-based automated implementation of distributed (e.g., [7, 13]) and real-time (e.g., [1, 11]) systems, we currently lack techniques that obtain executable real-time concurrent code from an abstract model of a system. This problem is particularly challenging, as one has to develop transformations for different levels of abstractions, each taking into account certain physical constraints (e.g., time, communication, synchronization, etc), and each transformation should add minimal overhead while maintaining a high level of parallelism. With this motivation, in this paper, we propose an automated method for producing efficient and correct-by-construction multi-threaded real-time implementation from an abstract component-based timed model. Our abstract models are expressed in the timed BIP (Behavior, Interaction, Priority) formalism [5]. BIP is a well-founded
component-based framework, where the *behavior* of each component (similar to timed automata [2]) is a Petri net or transition system subject to local timing constraints expressed by Boolean expressions over logical dense-time clock variables. A BIP model encompasses high-level multi-party *interactions* for synchronizing components (e.g., rendezvous and broadcast) and dynamic *priorities* for scheduling between interactions.

Our method consists of successive transformations that starts with a timed BIP model and terminates with an implementation. Intermediate transformation steps augment the output model with communication constraints and a physical time watching mechanism, such that each step results in a model closer to an actual implementation. These transformations are described as follows:

Decentralization In the abstract model, each component may depend upon global synchronization with other components to execute a local step. Indeed, executing a component transition is possible only when an interaction involving that transition is executed. To decide whether an interaction can be executed, one has to consider all participating components. In a concurrent setting, however, each component can only rely on its local knowledge to decide whether to execute a transition. Thus, our first transformation builds a model where additional components are responsible for scheduling interactions, based on the information received from the input model's components. Our transformation creates *conflict-free* schedulers, where schedulers do not need to interact with each other in order to resolve distributed conflicts. A distributed conflict refers to the situation where two or more interactions are enabled in the distributed implementation, but the abstract model semantics allows execution of only one.

Logical clock removal This transformation step builds a model that is robust to execution delay. This is done through decoupling logical and physical time. At this step, the two of them are assumed to be identical; i.e., communication occurs instantaneously (no delay) and component clocks are perfect (no drifts). This is the main reason that our target concurrent execution platform is *multi-process* applications, where all processes reside in the same machine and share a single clock. Unlike the logical clocks, the single clock introduced in this step is never reset and measures the absolute real time elapsed since the system starts executing. This transformation step is parametrized by a set of (observable) interactions whose constraints have to be met. We show that the model obtained in this step is observationally equivalent to the input abstract model through a bisimulation proof and, hence, all functional properties of the abstract model are preserved.

Implementation This transformation creates a set of independent executables that communicate through asynchronous message passing and may read the value of the single global hardware clock of the platform.

Our method is fully implemented and we show that the automatically generated concurrent code for a real-time multi-sensor image reconstruction system outperforms the corresponding concurrent implementation managed by a centralized scheduler.

The rest of this paper is structured as follows. In Section 2, we present the preliminary concepts on timed BIP models. Section 3 formalizes the point-to-point communication physical constraints. Our stepwise transformations are formally described in Sections 4 and 6. Implemetation and experimental results are described in Section 7. Related work is discussed in Section 8. Finally, we make concluding remarks in Section 9. .

2 Basic Semantic Model of BIP

In this section, we present the operational *global state* semantics of BIP [4]. BIP is a component framework for constructing systems by superposing three layers of modeling: Behavior, Interaction, and Priority. In this paper we do not consider priorities. In Subsection 2.2, we formally define atomic components. The notion of composite components is presented in Subsection 2.3.

2.1 Notations

Given a variable x, the *domain* of x is the set $\mathcal{D}(x)$ of all values possibly taken by x. Given a set of variables X, a *valuation* of X is a function $v : X \to \bigcup_{x\in X} \mathcal{D}(x)$ assigning a value to each variable of X, that is, such that for all x $v(x) \in \mathcal{D}(x)$. We denote by $\mathcal{V}(X)$ the set of all possible valuations of X. The restriction of $v \in \mathcal{V}(X)$ to a subset of variables $X' \subseteq X$ is the valuation $v_{|X'} \in \mathcal{V}(X')$ that coincides with v on X', that is, $v_{|X'}(x) = v(x)$ for all $x \in X'$. When it is not ambiguous, we write v also for $v_{|X'}$.

Given valuations $v \in \mathcal{V}(X)$ and $v' \in \mathcal{V}(X')$ of variables X and X' such that $X' \subseteq X$, we denote by $v[X' \leftarrow v']$ the valuation of X that coincides with v' for all variables of X', and with v for all variables of $X \setminus X'$. It is defined by:

$$v[X' \leftarrow v'](x) = \begin{cases} v'(x) \text{ if } x \in X' \\ v(x) \text{ otherwise.} \end{cases}$$

When all variables in X have the same domain $\mathcal{D}$, and given value $k \in \mathcal{D}$, we also denote by k the constant valuation assigning k to all variables of X.

A *guard* is a predicate on a set of variables X. Given a guard g on X and a valuation $v \in \mathcal{V}(X)$, we denote by $g(v) \in \{\texttt{false}, \texttt{true}\}$ the evaluation of g for v. An *update function* $f : \mathcal{V}(X) \to \mathcal{V}(X)$ for variables X is used to assign new values $f(v)$ to variables in X from their current values v. It extends to any larger set of variables $X' \supseteq X$ considering that extra variables $X' \setminus X$ are unchanged, i.e., f transforms $v \in \mathcal{V}(X')$ into $v[X \leftarrow f(v)]$.

Timing Constraints and Time Progress Conditions. In order to measure time progress, we use *clocks* that are variables advancing with the same rate [2] and ranging over real numbers. We denote by $\mathbb{R}_{\geq 0}$ the set of non-negative reals, and by $\mathbb{Z}_{\geq 0}$ the set of non-negative integers.

Timing constraints are used to specify when actions of a system are enabled. Given a set of clocks C, we consider atomic constraints $c \sim k$ where $c \in \mathsf{C}$, $k \in \mathbb{Z}_{\geq 0}$, and $\sim$ is a comparison operator such that $\sim \in \{\leq, =, \geq\}$. They are used to build *timing constraints* defined by the following grammar: $\mathsf{tc} := \texttt{true} \mid \texttt{false} \mid c \sim k \mid \mathsf{tc} \wedge \mathsf{tc}$. Notice that any timing constraint tc can be put into a conjunction of the form:

$$\mathsf{tc} = \bigwedge_{c\in\mathsf{C}} l_c \leq c \leq u_c, \tag{1}$$

such that for all $c \in \mathsf{C}$, $l_c \in \mathbb{Z}_{\geq 0}$ and $u_c \in \mathbb{Z}_{\geq 0} \cup \{+\infty\}$. The evaluation of a timing constraint tc for a valuation $t \in \mathcal{V}(\mathsf{C})$ of clocks C is the Boolean value $\mathsf{tc}(t)$ obtained by replacing in tc each clock c by its value $t(c)$.

Time progress conditions are used to specify whether time can progress at a given state of the system. They correspond to a special case of timing constraint in which atomic constraints are restricted to the form $c \leq k$. Notice that a time progress condition put in the form of (1) is such that for all $c \in \mathsf{C}$, $l_c = 0$.

2.2 Atomic Components

An *atomic component* is described as a *1-Safe Petri net* extended with local *variables* and clocks, consisting of a set of *places* and a set of *transitions*. Each transition is labeled by a *port*, a *guard* on local variables

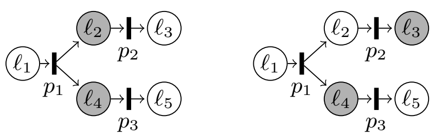


Figure 1: A simple Petri net

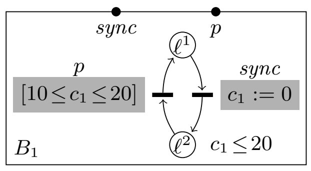


Figure 2: An atomic component.

combined with a timing constraint on clocks, and an update function. Ports are used for communication among different components. Each port *exports* a subset of variables of the component.

Definition 1 *A* Petri net *is defined by a triple $S = (L, P, T)$, where L is a set of* places*, P is a set of ports, and $T \subseteq 2^L \times P \times 2^L$ is a set of transitions. A transition τ is a triple $({}^\bullet\tau, p, \tau^\bullet)$, where ${}^\bullet\tau$ is the set of* input places *of τ and $\tau^\bullet$ is the set of* output places *of τ.*

A Petri net is often modeled as a directed bipartite graph $G = (L \cup T, E)$. Places are represented by circular vertices and transitions are represented by rectangular vertices (see Figure 1). The set of directed edges E is the union of the sets $\{(\ell, \tau) \in L \times T \mid \ell \in {}^\bullet\tau\}$ and $\{(\tau, \ell) \in T \times L \mid \ell \in \tau^\bullet\}$. We depict the *state* of a Petri net by marking its places with tokens [12]. We say that a place is *marked* if it contains a token. A transition τ is *enabled* at a state if all its input places ${}^\bullet\tau$ are marked. Upon the execution of τ, tokens of input places ${}^\bullet\tau$ are removed and tokens in output places in $\tau^\bullet$ are added.

Given an initial state $m_0 \subseteq L$, a Petri net (L, P, T) is *1-Safe* if for any execution from m_0, output places of enabled transitions are never marked. The behavior of a 1-Safe Petri net (L, P, T) is defined as a finite labeled transition system $(2^L, P, \rightarrow)$, where 2^L is the set of states, P is the set of labels, and $\rightarrow \subseteq 2^L \times P \times 2^L$ is the set of transitions defined as follows. We have $(m, p, m') \in \rightarrow$, denoted by $m \xrightarrow{p} m'$, if there exists $\tau = ({}^\bullet\tau, p, \tau^\bullet) \in T$ such that ${}^\bullet\tau \subseteq m$ and $m' = (m \backslash {}^\bullet\tau) \cup \tau^\bullet$. In this case, we say that p is *enabled* at m. We say that the Petri net (L, P, T) is *deterministic*, if for any execution from m_0 any two transitions $\tau_1 \neq \tau_2$ labeled by same port p are not simultaneously enabled at any state.

An *atomic component* is essentially a timed automaton [2] labeled by ports and extended with variables, whose states and transitions are given by the behavior of a deterministic 1-Safe Petri net.

Definition 2 (Atomic Component) *An* atomic component *B is defined by $B = (L, P, T, \mathsf{C}, X, \{X_p\}_{p\in P}, \{g_\tau\}_{\tau\in T}, \{\mathsf{tc}_\tau\}_{\tau\in T}, \{f_\tau\}_{\tau\in T}, \{\mathsf{tpc}_\ell\}_{\ell\in L}$) where:*

- *(L, P, T) is a* deterministic 1-Safe Petri net.
- *C is a set of clocks.*
- *X is a set of discrete variables.*
- *For each port $p \in P$, $X_p \subseteq X$ is the set of variables* exported *by p (i.e., variables visible from outside the component through port p).*
- *For each transition $\tau \in T$, g_τ is a guard on X, tc_τ is a timing constraint over C, and $f_\tau : \mathcal{V}(X) \times \mathcal{V}(\mathsf{C}) \rightarrow \mathcal{V}(X) \times \mathcal{V}(\mathsf{C})$ is a function that updates the set of variables X and may reset a subset of clocks $\mathsf{R}_\tau \subseteq \mathsf{C}$.*
- *For each place $l \in L$, tpc_l is a time progress condition.*

Example 1 *Figure 2 shows an atomic component. The set of clocks is $\{c_1\}$. The set of places is $\{\ell^1, \ell^2\}$ where ℓ^1 has time progress condition $c_1 \leq 20$. The set of ports is $\{p, sync\}$ and there is no discrete variable. There are two transitions: $\tau_1 = (\ell^1, sync, \ell^2)$ and $\tau_2 = (\ell^2, p, \ell^1)$. The transition τ_1 resets clock c_1 and the transition τ_2 is guarded by a timing constraint on clock c_1.*

Definition 3 (Atomic Component Semantics) *The semantics of an atomic component $B = (L, P, T, \mathsf{C}, X, \{X_p\}_{p\in P}, \{g_\tau\}_{\tau\in T}, \{\mathsf{tc}_\tau\}_{\tau\in T}, \{f_\tau\}_{\tau\in T}, \{\mathsf{tpc}_l\}_{l\in L})$ is defined as the labeled transition system $(Q_B, P_B, \underset{B}{\longrightarrow})$, where*

- *$Q_B = 2^L \times \mathcal{V}(X) \times \mathcal{V}(\mathsf{C})$ is the set of states.*
- *$P_B = P \cup \mathbb{R}_{\geq 0}$ is set of labels: ports or time values.*

- $\xrightarrow[B]{} \subseteq Q_B \times P_B \times Q_B$ *is the set of labeled transitions defined as follows. Let* (m, v, t) *and* (m', v', t') *be two states,* $p \in P$, *and* $\delta \in \mathbb{R}_{\geq 0}$ *be a delay.*

  Jump transitions. *We have* $(m, v, t) \xrightarrow[B]{p} (m', v', t')$, *iff transition* $\tau = ({}^\bullet\tau, p, \tau^\bullet)$ *is enabled at* m *in the Petri net* (L, P, T) *and* $g_\tau(v) \wedge \mathsf{tc}_\tau(t)$ *is* `true`. *In this case, we say that* p *is enabled from* (m, v, t). *Notice that* t' *satisfies* $t' = t[\mathsf{R}_\tau \leftarrow 0]$, *where* R_τ *is the set of clocks reset by* τ.

  Delay transitions. *We have* $(m, v, t) \xrightarrow[B]{\delta} (m, v, t + \delta)$ *if we have* $\bigwedge_{\ell \in m} \mathsf{tpc}_\ell(t + \delta)$ *is* `true`, *where* $t + \delta$ *is the usual notation for the valuation defined by* $(t + \delta)(c) = t(c) + \delta$ *for any* $c \in \mathsf{C}$.

An atomic component B can execute a transition $\tau = ({}^\bullet\tau, p, \tau^\bullet)$ from a state (m, v, t) if its guard is met by the valuation v and its timing constraint is met by the valuation t. From state (m, v, t), B can also wait for $\delta > 0$ time units if $\bigwedge_{\ell \in m} \mathsf{tpc}_\ell(t + \delta)$ stays `true`. Waiting for δ time units increases all the clock values by δ. Notice that the execution of a jump transition is instantaneous and time elapses only on states. The semantics presented here is slightly different from the one found in [2], as we consider time progress conditions instead of invariants. Unlike invariants, an atomic component B may reach a state (m, v, t) violating the corresponding time progress condition $\bigwedge_{\ell \in m} \mathsf{tpc}_\ell$. In this case B cannot wait and is forced to execute a transition from (m, v, t). In the following, we consider systems that cannot reach states violating time progress conditions.

2.3 Composite Components

A composite component is built from a set of n atomic components $\{B_i = (L_i, P_i, T_i, \mathsf{C}_i, X_i, \{X_p\}_{p \in P_i}, \{g_\tau\}_{\tau \in T_i}, \{\mathsf{tc}_\tau\}_{\tau \in T_i}, \{f_\tau\}_{\tau \in T_i}, \{\mathsf{tpc}_\ell\}_{\ell \in L_i})\}_{i=1}^n$, such that their respective sets of places, ports, clocks, and discrete variables are pairwise disjoint; i.e., for any two $i \neq j$ from $\{1, \ldots, n\}$, we have $L_i \cap L_j = \emptyset$, $P_i \cap P_j = \emptyset$, $\mathsf{C}_i \cap \mathsf{C}_j = \emptyset$, and $X_i \cap X_j = \emptyset$. We denote $P = \bigcup_{i=1}^n P_i$ the set of all the ports in the composite component, $L = \bigcup_{i=1}^n L_i$ the set of all places, $\mathsf{C} = \bigcup_{i=1}^n \mathsf{C}_i$ the set of all clocks, and $X = \bigcup_{i=1}^n X_i$ the set of all variables.

Definition 4 (Interaction) *An* interaction a *between atomic components* $\{B_i\}_{i=1}^n$ *is a subset of ports* $a \subseteq P$, *such that it contains at most one port of every component, that is,* $|a \cap P_i| \leq 1$ *for all* $i \in \{1, \ldots, n\}$.

The set X_a *of variables available to an interaction* a *is given by* $X_a = \bigcup_{p \in a} X_p$. *We associate to* a *its guard* G_a *and its update function* F_a *over* X_a.

Since an interaction a uses at most one port of every component, we denote $a = \{p_i\}_{i \in I}$, where $I \subseteq \{1, \ldots, n\}$. A component B_i is *involved* in a if $i \in I$.

Definition 5 (Composite Component) *We denote by* $B \overset{def}{=} \gamma(B_1, \ldots, B_n)$ *the* composite *component obtained by applying a set of interactions* γ *to the set of atomic components* $\{B_i\}_{i=1}^n$. *It is defined by the atomic component* $B = (L, \gamma, T, \mathsf{C}, X, \{X_a\}_{a \in \gamma}, \{g_\tau\}_{\tau \in T}, \{\mathsf{tc}_\tau\}_{\tau \in T}, \{f_\tau\}_{\tau \in T}, \{\mathsf{tpc}_\ell\}_{\ell \in L})$ *as follows.*

- *Given an interaction* $a = \{p_i\}_{i \in I}$ *of* γ, *a transition* $\tau = (\ell, a, \ell')$ *is in* T *if its projection* $\tau_i = (\ell_i, p_i, \ell'_i) = (\ell \cap L_i, a \cap P_i, \ell' \cap L_i)$ *on* B_i *is a transition of* B_i *(i.e.* $\tau_i \in T_i$*), for all* $i \in I$.

- *The guard* g_τ *of transition* τ *is* $g_\tau = G_a \wedge \bigwedge_{i \in I} g_{\tau_i}$.

- *The timing constraint* tc_τ *of* τ *is* $\mathsf{tc}_\tau = \bigwedge_{i \in I} \mathsf{tc}_{\tau_i}$.

- *We have* $f_\tau(v, t) = (f_{\tau_1} \circ \cdots \circ f_{\tau_n})(F_a(v), t)$, *where* f_{τ_i} *is the identity function, for* $i \notin I$. *Notice that functions* f_{τ_i} *modify disjoint sets of variables and clocks and, hence, can be composed in any order.*

- *For a control location* $\ell = (\ell_1, \ldots, \ell_n) \in L$, *the time progress condition* tpc_ℓ *is* $\mathsf{tpc}_\ell = \bigwedge_{i \in \{1..n\}} \mathsf{tpc}_{\ell_i}$.

A composite component $B = \gamma(B_1, \ldots, B_n)$ can execute an interaction $a = \{p_i\}_{i \in I} \in \gamma$ from a state (m, v, t) iff (1) for each port p_i, the corresponding atomic component B_i can execute a transition labeled by p_i from the projection $(m_i, v_i, t_i) = (m \cap L_i, v_{|X_i}, t_{|\mathsf{C}_i})$ of (m, v, t) on B_i, and (2) the guard G_a of the interaction evaluates to `true` on the variables exported by the ports participating in interaction a. Execution of interaction a triggers the function F_a which modifies the variables of the components exported by ports p_i. The new values obtained are then processed by the components' transitions. Note that the components also reset clocks according to the update function associated to their transition. The states of components that do not participate in the interaction remain unchanged. We say that an interaction $a \in \gamma$ is *enabled* at state $q \in Q_B$ of B, if there exists state $q' \in Q_B$ such that $q \xrightarrow[B]{a} q'$.

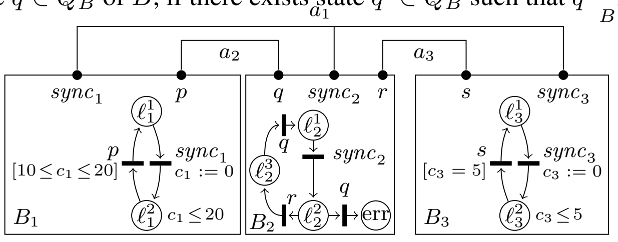


Figure 3: Example of BIP composite component.

Example 2 *Figure 3 illustrates a composite component $\gamma(B_1, B_2, B_3)$. The set γ of interactions is $\{a_1, a_2, a_3\}$ with no guards nor update functions. Initially, the system is in state $(\ell_1^1, \ell_2^1, \ell_3^1)$, where c_1 and c_3 are set to 0. The only enabled interaction is a_1. Since time progress condition at this state is* `true`*, any delay $\delta \in \mathbb{R}_{\ge 0}$ can be taken. If interaction a_1 is executed, the next state is $(\ell_1^2, \ell_2^2, \ell_3^2)$ and clocks c_1 and c_3 are reset. At this state, the time progress condition and the timing constraint in B_3 impose that a_3 has to be executed after a delay of $\delta = 5$ time units. Once a_3 is executed, a_2 can execute after a delay of $\delta \in [5, 15]$ time units according to the time progress condition and the timing constraint in B_1.*

3 Target Architecture

In this section, we describe the overall architecture of the source-to-source transformation of BIP models. Since we target concurrent execution of interactions, if two interactions are simultaneously enabled, they can be executed in parallel only if the semantics of the initial global state model is met. That is, if they involved disjoint sets of components. This leads to the notion of *conflict* between interactions. Two interactions are conflicting if they involve a shared component and they are potentially enabled at the same time.

Definition 6 *Let $\gamma(B_1, \ldots, B_n)$ be a BIP model. We say that two interactions a and b of γ are in* structural conflict *iff there exists an atomic component B_i that has two transitions $\tau_1 = ({}^\bullet\tau_1, p_1, \tau_1^\bullet)$ and $\tau_2 = ({}^\bullet\tau_2, p_2, \tau_2^\bullet)$ such that (1) $p_1 \in a$ and $p_2 \in b$, and (2) there exists a reachable state in the Petri net (L_i, P_i, T_i) of B_i at which both τ_1 and τ_2 are enabled.*

Note that structural conflicts as defined in Definition 6 are an over-approximation of conflicts, since some structural conflicts may not be reachable due to guards and timing constraints. A special case of conflict is when two interactions a and b share a common port, that is, $a \cap b \neq \emptyset$. As already discussed, handling conflicting interactions in a BIP model executed by a centralized Engine is quite straightforward [4, 15]. However, in a concurrent setting, detecting and avoiding conflicts is not trivial [7].

Consider a composite component $B = \gamma(B_1 \cdots B_n)$ in the BIP model and a *partition* of the set of interactions $\{\gamma_j\}_{j=1}^m$ (i.e., m *classes* of interactions γ_j are disjoint and cover all the interactions of γ). In our target concurrent model, atomic components B_i are transformed into atomic components B_i^{SR}. We also add *Interaction Protocol* components to implement interactions, such that each class of interaction γ_j is handled by a single Interaction Protocol component IP_j. The partition $\{\gamma_j\}_{j=1}^m$ allows the designer to enforce load-balancing and to improve the performance of the given model when running in a concurrent fashion. It also determines whether or not a conflict between interactions can be resolved locally. Consider conflicting interactions $a \in \gamma_j$ and $b \in \gamma_k$. We distinguish between two types of conflict for a and

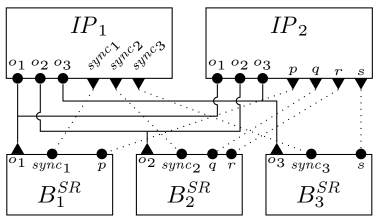
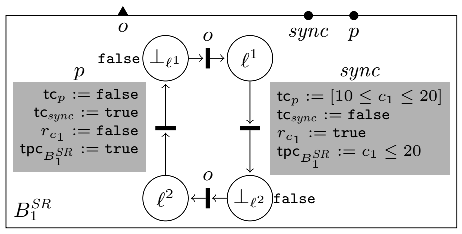


Figure 4: Concurrent model of Figure 3.



o
o
$sync$ p
p
false
$\perp_{\ell^1}$
ℓ^1
$sync$
$\mathtt{tc}_p := \mathtt{false}$
$\mathtt{tc}_{sync} := \mathtt{true}$
$r_{c_1} := \mathtt{false}$
$\mathtt{tpc}_{B_1^{SR}} := \mathtt{true}$
$\mathtt{tc}_p := [10 \leq c_1 \leq 20]$
$\mathtt{tc}_{sync} := \mathtt{false}$
$r_{c_1} := \mathtt{true}$
$\mathtt{tpc}_{B_1^{SR}} := c_1 \leq 20$
o
ℓ^2
$\perp_{\ell^2}$
false
B_1^{SR}

Figure 5: Transformation of the atomic component in Figure 2.

b, according to the partition $\{\gamma_j\}_{j=1}^m$. A conflict is *internal* if a and b belong to the same class of the partition, i.e., $j = k$. In this case, it can be resolved by the Interaction Protocol component IP_i responsible for a and b. A conflict is *external* if a and b belong to the different classes of the partition, i.e., $j \neq k$. External conflicts cannot be resolved by a single Interaction Protocol component IP_j, and requires additional synchronizations and components [7]. This is beyond the scope of this paper.

Consider again the example from Figure 3. Interaction a_1 is conflicting with neither a_2 nor a_3. However, a_2 and a_3 are conflicting because port q involved in a_2 and port r involved in a_3 are both enabled from place ℓ_2^2. Partition $\gamma_1 = \{a_1\}$ and $\gamma_2 = \{a_2, a_3\}$ is such that all conflicts between interactions are internal. The overall architecture of the concurrent model built for this partition is given in Figure 4. Notice that IP_1 and IP_2 share B_2^{SR}, as the later is involved in both $a_1 \in \gamma_1$ and $a_2 \in \gamma_2$. However, this is not a problem since a_1 and a_2 are never enabled at the same time.

From now on, we consider partitions $\{\gamma_j\}_{j=1}^m$ of interactions γ such that conflicts are always *internal*, that is, if two interactions $a, b \in \gamma$ are conflicting, then they belong to the same class γ_j. We also target Send/Receive BIP models. Intuitively, a *Send/Receive* model is a set of independent components communicating through asynchronous message passing defined next.

Definition 7 *We say that $B^{SR} = \gamma^{SR}(B_1^{SR}, \ldots, B_n^{SR})$ is a* Send/Receive *BIP composite component iff we can partition the set of ports of B^{SR} into three sets P_s, P_r, and P_u that are respectively the set of* send-ports, receive-ports, *and* unary interaction ports*, such that:*

- *Each interaction $a \in \gamma^{SR}$, is either (1) a Send/Receive interaction with $a = (s, r_1, r_2, \ldots, r_k)$, $s \in P_s$,*
  $r_1, \ldots, r_k \in P_r$, $G_a = \mathtt{true}$ and F_a copies the variables exported by port s to the variables exported by ports $r_1, r_2, \ldots, r_k$, or, (2) a unary interaction $a = \{p\}$ with $p \in P_u$, $G_a = \mathtt{true}$, F_a is the identity function.

- *If s is a port in P_s, then there exists one and only one Send/Receive interaction $a \in \gamma^{SR}$ with $a = (s, r_1, r_2, \ldots, r_k)$ and all ports $r_1, \ldots, r_k$ are receive-ports. We say that $r_1, r_2, \ldots, r_k$ are the receive-ports associated to s.*

- *If $a = (s, r_1, \ldots, r_k)$ is a Send/Receive interaction in γ^{SR} and s is enabled at some global state of B^{SR}, then all its associated receive-ports $r_1, \ldots, r_k$ are also enabled at that state.*

Definition 7 defines a class of BIP models for concurrent implementation based on asynchronous message passing. In such systems, communication is sender-triggered, where a message is emitted by the sender, regardless of the availability of receivers. The third property of the definition, requires that all receivers are ready to receive whenever the sender may send a message. This ensures that the sender is never blocked and triggers the Send/Receive interaction.

Intuitively, a model that meets properties of Definition 7 can be seen as a set of independent process, communicating through asynchronous message passing. However, execution of this model according to the BIP semantics assumes that clocks of these components advance at the same rate and communication is instantaneous.

4 Step 1: BIP to Send/Receive-BIP

In this section, we describe a method for automated transformation of a timed BIP model $B = \gamma(B_1, \ldots, B_n)$ into a timed *Send/Receive*-BIP model $B^{SR} = \gamma^{SR}(B_1^{SR}, \ldots, B_m^{SR})$ that meets restrictions of Definition 7. Correctness of this transformation is established in Section 5.

4.1 Atomic Components

For the sake of simplicity and clarity, we present the transformation for an atomic component such that its Petri net is an automaton, that is, each of its transitions has a single source and single target place, and its initial state consists in a single place. Notice that the behavior of a 1-Safe Petri net defines a finite automaton, allowing us to apply the following transformation to any arbitrary atomic component.

We transform an atomic component B of a BIP model into a Send/Receive atomic component B^{SR} that is capable of communicating with the Interaction Protocol component(s). To communicate, B^{SR} sends *offers* to the Interaction Protocol that are acknowledged by a *response*. An offer includes necessary information for computing enabled interactions from the current state of B^{SR}, i.e., values of variables exported by the ports, timing constraints of transitions, and resets of clocks. When the interaction protocol selects an interaction involving B^{SR} for execution, B^{SR} is notified by a response sent on the chosen port.

Since each notification from the Interaction Protocol triggers an internal computation in a component, following [4], we split each place ℓ into two places, namely, ℓ itself and a *busy place* $\perp_\ell$. Intuitively, reaching $\perp_\ell$ marks the beginning of an unobservable internal computation. We are now ready to define the transformation from B into B^{SR}.

Definition 8 *Let $B = (L, P, T, \mathsf{C}, X, \{X_p\}_{p\in P}, \{g_\tau\}_{\tau\in T}, \{\mathsf{tc}_\tau\}_{\tau\in T}, \{f_\tau\}_{\tau\in T}, \{\mathsf{tpc}_\ell\}_{\ell\in L})$ be an atomic component. The corresponding Send/Receive atomic component is $B^{SR} = (L^{SR}, P^{SR}, T^{SR}, \emptyset, X^{SR}, \{X_p^{SR}\}_{p\in P}, \{g_\tau\}_{\tau\in T^{SR}}, \emptyset, \{f_\tau\}_{\tau\in T^{SR}}, \emptyset)$, such that:*

- *$L^{SR} = L \cup L^\perp$, where $L^\perp = \{\perp_\ell \mid \ell \in L\}$.*
- *$X^{SR} = X \cup \{\mathsf{tc}_p\}_{p\in P} \cup \{\mathsf{tpc}_{B^{SR}}\} \cup \{r_c\}_{c\in \mathsf{C}}$, where r_c are Boolean variables, tc_p are timing constraint variables and $\mathsf{tpc}_{B^{SR}}$ is time progress condition variable.*
- *$P^{SR} = P \cup \{o\}$, where the* offer *port o exports the variables $X_o^{SR} = \bigcup_{p\in P} X_p \cup \{\mathsf{tc}_p\}_{p\in P} \cup \{r_c\}_{c\in\mathsf{C}}$. For all other ports $p \in P$, we define $X_p^{SR} = X_p$.*
- *For each place $\ell \in L$, we include an intermediate place $\perp_\ell$ and an* offer *transition $\tau_\ell = (\perp_\ell, o, \ell)$ in T^{SR}. The time progress condition tpc_{τ_ℓ} is `false`, both the guard g_{τ_ℓ} and the timing constraint tc_{τ_ℓ} are `true`, and the update function f_{τ_ℓ} is the identity function.*
- *For each transition $\tau = (\ell, p, \ell') \in T$, we include a* response *transition $\tau_p = (\ell, p, \perp_{\ell'})$ in T^{SR} with no guard and timing constraint.*

  The function f_{τ_p} first applies function f_τ of τ, and then sets time progress condition variable to the time progress condition of next location ($i.e.$ $\mathsf{tpc}_B := \mathsf{tpc}_{\ell'}$) and updates the timing constraint and reset variables: $\forall p' \in P \quad \mathsf{tc}_{p'} := \begin{cases} \mathsf{tc}_{\tau'} & \text{if } g_{\tau'} \wedge \tau' = (\ell', p', \ell'') \in T \\ \texttt{false} & \text{otherwise.}\end{cases}$

  $$\forall c \in \mathsf{C} \qquad r_c := \begin{cases} \texttt{true} & \text{if } f_\tau \text{ resets } c \\ \texttt{false} & \text{otherwise.}\end{cases}$$

In the above definition, the execution of a transition $\tau = (\ell, p, \ell')$ of a component B corresponds to the following two execution steps in B^{SR}. Firstly, an offer transition $\tau_\ell = (\perp_\ell, o, \ell)$ transmits for each port $p' \in P$ the current values of its variables $X_{p'}$, the timing constraint $\mathsf{tc}_{p'}$ corresponding to the enabledness of p' at ℓ, as well as the time progress condition tpc_ℓ. These are used by the Interaction Protocol for computing guards and timing constraints of interactions involving B^{SR}. The transition τ_ℓ also transmits for each clock $c \in \mathsf{C}$ the value of its reset variable r_c, such that $r_c = \texttt{true}$, if c has been reset by the previous transition execution. Variables r_c are used to reset clocks in the Interaction Protocol before computing timing constraints of interactions.

Secondly, a response transition $\tau_p = (\ell, p, \perp_{\ell'})$ is executed once the Interaction Protocol decides to execute an interaction involving port p. Similar to τ in B, τ_p updates values of variables X according to f_τ. It also updates variables $\mathsf{tpc}_{B^{SR}}$, $\mathsf{tc}_{p'}$ and r_c to set up-to-date values for the next offer (i.e. starting from ℓ'). Since (L, P, T) is a deterministic 1-Safe Petri net, a port $p' \in P$ enables at most one transition at ℓ'. If no transition labeled by p' is enabled at ℓ', or if the guard $g_{\tau'}$ of the transition τ' enabled by p' at ℓ' evaluates to `false`, $\mathsf{tc}_{p'}$ is set to `false` to disable interactions involving p'. Otherwise, $\mathsf{tc}_{p'}$ is set to the timing constraint $\mathsf{tc}_{\tau'}$ of transition τ' enabled by p' at ℓ'.

Notice that time progress conditions and timing constraints of B^{SR} do not involve clocks C. Thus, according to [9] clocks are no longer *active* and can be removed from B^{SR}. Original time progress conditions and timing constraints of B are stored in variables of B^{SR}, and transmitted to the Interaction Protocol which is responsible for enforcing timeliness in interactions execution. Figure 5 illustrates the transformation of the component B_1 of Figure 2 into its corresponding Send/Receive component B_1^{SR}.

4.2 Interaction Protocol Layer

The Petri net that defines the behavior of an Interaction Protocol component IP_j handling a class γ_j of interactions is constructed as follows. Figure 6 illustrates the construction of the Petri net of component IP_2 handling interaction a_2 and a_3 in example of Figure 4.

Variables and Clocks. For each component B_i, we include a time progress condition variable tpc_{B_i}. For each port p involved in interactions γ_j, we include a timing constraint variable tc_p and a local copy of the variables X_p exported by p. We also include for each clock a Boolean variable r_c that indicates whether clock c has to be reset.

The set of clocks of IP_j contains all the clocks defined initially in components B_i involved in γ_j before being transformed into B_i^{SR}.

Places. The Petri net has two types of places:

- For each component B_i involved in interactions of γ_j, we include *waiting* and *received* places w_i and rcv_i, respectively. Place rcv_i has a time progress condition defined by the variable tpc_{B_i}. Initially the IP_j remains in a waiting place until it receives an offer from the corresponding component. When an offer from component B_i^{SR} is received, IP_j moves from w_i to rcv_i.
- For each port p involved in interactions of γ_j, we include a *sending* place s_p. The time progress condition of s_p is `false`. The response to an offer of a component B_i^{SR} is sent from this place to port p of B_i^{SR}.

Ports.

The set of ports of IP_j is the following:

- For each component B_i, we include a receive-port o_i, to receive offers. Each port o_i is associated to the variables tc_p, and X_p associated to each port p of B_i, the variables r_c for each clock c of B_i as well as the variable tpc_{B_i} of B_i. These variables are updated whenever an offer from B_i is received.
- For each port p involved in interactions γ_j, we include a send-port p, which exports the set of variables X_p.
- We include a unary port for each interaction $a \in \gamma_j$.

Transitions. IP_j receives offers from SR components and responds to them. The following set of transitions of IP_j performs these two tasks:

- In order to receive offers from a component B_i, we include transition (w_i, o_i, rcv_i). We also include a transition (rcv_i, o_i, rcv_i) to receive new offers when B_i takes part in an external interaction. This transition resets all clocks c such that r_c is `true`.
- For each interaction $a = \{p_i\}_{i \in I}$ in γ_j, we include the transition $(\{rcv_i\}_{i \in I},\ a,\ \{s_{p_i}\}_{i \in I})$. This transition is guarded by the predicate G_a, has the timing constraint $\bigwedge_{i \in I} \mathsf{tc}_{p_i}$ and moves the tokens from receiving to sending places. This transition triggers function F_a.

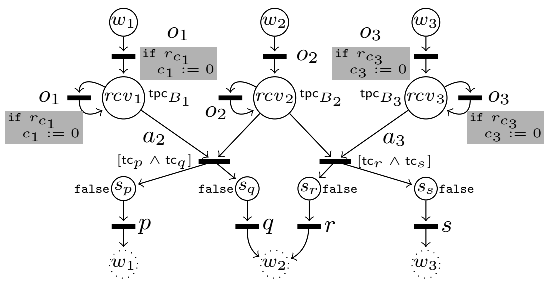


Figure 6: Component IP_2 handling interactions a_2 and a_3 from Figure 3.

- Finally, for each port p involved in interactions γ_j, we include a transition (s_p, p, w_i). This transition notifies the corresponding component to execute the transition labeled p.

Note that in Interaction Protocol components, time progress conditions, timing constraints and resets of clocks depend on variables, which are not permitted by Definition 2. However, there is only a finite number of configurations for the values of these variables, as the number of transitions and states in atomic components B_i is finite. In IP_j, we could include multiple transitions for offers o_i and interactions a to encode all possible combinations of these configurations. In this case, an atomic component B_i^{SR} would send offers indicating in which configuration are its reset, time progress conditions and timing constraints variables, and appropriate guards in IP_j would enable the corresponding transitions.

4.3 Send/Receive Interactions

In this subsection, we define the interactions between the components defined thus far. Following Definition 7, we introduce Send/Receive interactions by specifying only the sender. Given a BIP model $\gamma(B_1 \cdots B_n)$, a partition $\gamma_1 \cdots \gamma_m$ of γ, the transformation gives a Send/Receive BIP model $B^{SR} = \gamma^{SR}(B_1^{SR}, \ldots, B_n^{SR}, IP_1, \ldots, IP_m)$. We define the Send/Receive interactions of γ^{SR} as follows:

- For each component B_i^{SR}, let $IP_{j_1}, \ldots, IP_{j_l}$ be the Interaction Protocol components handling interactions involving B_i^{SR}. We include in γ^{SR} the *offer interaction* $(B_i^{SR}.o, IP_{j_1}.o_i, \ldots, IP_{j_l}.o_i)$.
- For each port p in component B_i^{SR} and for each Interaction Protocol component IP_j handling an interaction involving p, we include in γ^{SR} the *response interaction* $(IP_j.p, B_i^{SR}.p)$.
- For each interaction $a \in \gamma$, we add the unary interaction $(IP_j.a)$ to γ^{SR}, where IP_j is the Interaction Protocol component handling the interaction a.

The concurrent version obtained from the model depicted in Figure 3 is shown in Figure 4. The transformation is parametrized by the partition of the interaction $\gamma_1 = \{a_1\}$ and $\gamma_2 = \{a_2, a_3\}$, yielding two interaction protocol components.

Theorem 1 Given a timed BIP model B, we have $B^{SR} \sim B$, where $\sim$ denotes observational equivalence.

5 Correctness of Step 1

In Subsection 5.1, we show that our transformations indeed result in a well-formed model as defined in Section 3. Then, in Subsection 5.2, we prove that the initial high-level timed BIP model is observationally equivalent to the timed SR-BIP model obtained by the transformation of Section 4.

5.1 Compliance with the Send-Receive Model

We need to show that when a receive-port of B^{SR} is enabled, the corresponding send-port is also enabled. This holds since communications between atomic components and interaction protocol components follow a request/acknowledgement pattern. Whenever an atomic component sends an offer, it enables the receive-port to receive a response and no new offer is sent until the first one is acknowledged.

Lemma 1 *Given a BIP model B, the model B^{SR} obtained by transformation of Section 4 meets the properties of Definition 7.*

Proof: The first two constraints of Definition 7 are trivially met by construction. This is because (1) each interaction has only one send-port and multiple receive-ports, and (2) each send-port is associated with one and only SR interaction.

We now prove that the third constraint also holds; i.e, whenever a send-port is enabled, all its associated receive-ports are enabled as well. Between components and Interaction Protocol layer, for all interactions involving a component B_i, we distinguish between 3 classes of global states:

- The first class contains all states where all the places w_i and rcv_i in all interaction protocol components contain a token, and B_i^{SR} is in a busy place $\perp_\ell$. This class contains the initial state. From that class, the only enabled send-port involved in an interaction with B_i^{SR} is the port o_i. By definition of the class, all associated receive-ports are also enabled, and the Send/Receive interaction can take place to reach a state of the second class.

- In the second class, the component B_i^{SR} is in a place ℓ that is not a busy place, and in each interaction protocol component the rcv_i place contains a token. Two distinct interaction protocol components handle interactions involving distinct set of ports from B_i, otherwise their interaction would be externally conflicting. Furthermore, only one interaction protocol component handles ports from B_i with timing constraints that are not `false` in the last offer from B_i^{SR}, for the same reason. From that configuration, there is no enabled send-port involved in an interaction with B_i^{SR}. The next class of states is reached when the interaction protocol that handles B_i ports with not false timing constraints executes a transition corresponding to an interaction involving B_i.

- In the last class of state, the component B_i^{SR} is not in a busy place, in all interaction protocol components except one, the place rcv_i is active. In the remaining one, there is a token in a place s_p with p a port of B_i. The port p of this interaction protocol component is enabled. By construction, the component B_i^{SR} sends an offer from $\perp_\ell$ with a not false timing constraint for port p only if the receive port p is enabled from place ℓ in B_i^{SR}. Thus the Send/Receive interaction can take place to reach back the first class of states considered. ■

The proof of Lemma 1 ensures that any component ready to perform a transition labeled by a send-port will not be blocked by waiting for the corresponding receive-ports. In other terms, it proves that any Send/Receive interaction is initiated by the sender.

5.2 Observational Equivalence between Original and Transformed BIP Models

We recall the definition of *observational equivalence* of two transition systems $A = (Q_A, P \cup \{\beta\}, \rightarrow_A)$ and $B = (Q_B, P \cup \{\beta\}, \rightarrow_B)$. It is based on the usual definition of weak bisimilarity [**?**], where β-transitions are considered unobservable. The same definition is trivially extended for atomic and composite BIP components.

Definition 9 (Weak Simulation) *A* weak simulation *from A to B, denoted $A \subset B$, is a relation $R \subseteq Q_A \times Q_B$, such that $\forall (q,r) \in R,\ a \in P :\ q \xrightarrow{a}_A q' \implies \exists r' :\ (q', r') \in R \wedge r \overset{\beta^* a \beta^*}{\rightarrow}_B r'$ and $\forall (q,r) \in R :\ q \xrightarrow{\beta}_A q' \implies \exists r' :\ (q', r') \in R \wedge r \xrightarrow{\beta^*}_B r'$*

A weak bisimulation over A and B is a relation R such that R and R^{-1} are both weak simulations. We say that A and B are *observationally equivalent* and we write $A \sim B$ if for each state of A there is a weakly bisimilar state of B and conversely. In this subsection, our goal is to show that B and B^{SR} are observationally equivalent. We consider the correspondence between actions of B and B^{SR} as follows. To each jump transition $a \in \gamma$ of B, we associate the unary interaction a of B^{SR}. To each delay transition $\delta \in \mathbb{R}_{\geq 0}$ of B, we associate the same delay transition δ of B^{SR}. All other interactions of B^{SR} (i.e., offer and response) are unobservable and denoted by β.

We proceed as follows to complete the proof of observational equivalence. We denote by q^{SR} a state of B^{SR} and q a state of B. A state of B^{SR} from where no β action is possible is called a *stable state*, in the sense that any β action from this state does not change the state of the atomic components layer.

Lemma 2 *From any state q^{SR}, there exists a unique stable state $[q^{SR}]$ such that $q^{SR} \xrightarrow{\beta^*} [q^{SR}]$.*

Proof: The state $[q^{SR}]$ exists since each Send/Receive component B_i^{SR} can do at most two β transitions: receive a response and send an offer. Since two β transitions involving two different components are independent (i.e. do not change the same variable or the same place), the same final state is reached independently of the order of execution of β actions. Thus $[q^{SR}]$ is unique. ■

The above lemma proves the existence of a well-defined stable state for any of the transient states reachable by the SR model B^{SR}. This stable state will be used later to define our observational equivalence. Furthermore, combining this lemma with Lemma 1, we obtain the following property:

Lemma 3 *At a stable state $[q^{SR}]$, the Send/Receive model verifies the following properties:*

- *All atomic components are in a non busy place ℓ.*
- *All tokens in interaction protocol components are in receive places rcv_i.*
- *Clocks and variables in the atomic components have the same value than their copies in the interaction protocol components.*

Proof: The two first points comes from the Lemma 1 that guarantees possible execution of a Send/Receive interaction if its send-port is enabled. Therefore no place s_p in interaction protocol components (respectively $\perp_\ell$ in atomic components) can be active at $[q^{SR}]$, otherwise the answer p (respectively the offer from $\perp_\ell$)) could occur. Furthermore, since all offers have been sent, no token can be in a w_i state.

To prove the last point, first we claim that no delay transition are allowed from a non stable state. Indeed, at such a state a β transition is enabled. By construction, All time progress conditions of places enabling β transitions (offer and response) are `false`, which forbids delaying these transitions. Now, whenever a clock is reset or a variable is updated in an atomic component, the immediately following offer propagates the values of the variables to the interaction protocol components. If some clocks were reset, the offer transition in the atomic components resets it as well (because of the r_x boolean), and no delay transition occurred between these two resets. ■

We are now ready to state and prove our central result.

Theorem 1 $B^{SR} \sim B$.

Proof: We define a relation between the set of states Q^{SR} of B^{SR} and the set of states Q of B as follows. For each state $q^{SR} \in Q^{SR}$, we build an equivalent state $equ(q^{SR})$ by

1. considering the unique stable state $[q^{SR}]$ reachable by doing β transitions.
2. taking the control state ℓ of B_i^{SR} as control state for B_i in $equ(q^{SR})$. Lemma 3 ensures that it is a valid control state for B_i.
3. taking the valuation $v_i = v_i^{SR}|_{X_i}$ as data and clocks state for B_i in $equ(q^{SR})$.

We then define the equivalence R by taking:

$$R = \{(q^{SR}, q) \in Q^{SR} \times Q \mid q = equ(q^{SR})\}$$

The three next assertions prove that R is a weak bisimulation:

(i) If $(q^{SR}, q) \in R$ and $q^{SR} \xrightarrow{\beta} r^{SR}$ then $(r^{SR}, q) \in R$.

(ii) If $(q^{SR}, q) \in R$ and $q^{SR} \xrightarrow{a} r^{SR}$ then $\exists r \in Q : q \xrightarrow{a} r$ and $(r^{SR}, r) \in R$.

(iii) If $(q^{SR}, q) \in R$ and $q \xrightarrow{a} r$ then $\exists r^{SR} \in Q^{SR} : q^{SR} \xrightarrow{\beta^* a} r^{SR}$ and $(r^{SR}, r) \in R$.

(i) If $q^{SR} \xrightarrow{\beta} r^{SR}$, then $[q^{SR}] = [r^{SR}]$, and we have by definition $equ(q^{SR}) = equ(r^{SR})$.
(ii) The action a in B^{SR} is either a jump transition a or a delay transition δ.

If a is a jump transition, it corresponds to executing a transition labeled by a unary port a in one of the interaction protocol components. By construction of the interaction protocol component, this transition has the conjunction of timing constraints tc_p sent by the atomic components for each port $p \in a$. Similarly, the guard of this transition is the guard of a evaluated on the same values. By Lemma 3, these values are the same in atomic components, and by extension in $q = equ(q^{SR})$. Thus the guard of a evaluates to true at q. By construction of the atomic components, the timing constraints sent from $\perp_\ell$ in B_i^{SR} are the timing constraints at state ℓ in B_i. Therefore, the timing constraints of a are met at state q. Thus we have $q \xrightarrow{a} r$. Finally, executing a in B^{SR} triggers the execution of the data transfer function F_a, followed by the computation in atomic component upon reception of the response. Thus at $[r^{SR}]$ the values in atomic components are the same as in r, which yields $(r^{SR}, r) \in R$.

If a is a delay transition δ, it can occur only from a stable state. By Lemma 3, all rcv_i places have a token. Therefore any transition a can execute if the time progress conditions tpc_{B_i} are met. In particular, there is no time progress condition tpc_{B_i} that is false, otherwise the δ delay transition would not be allowed. The same reasoning as in the previous case applies to state that in the equivalent state $equ(q^{SR})$ the time progress conditions are the same. Therefore $q \xrightarrow{\delta} r$. Executing δ has the same effect on the clocks in both models, therefore $(r^{SR}, r) \in R$.

(iii) If a can be executed in B at state q, then from an equivalent state q^{SR}, one can reach the state $[q^{SR}]$ where the state, data and clocks of atomic components coincide with those of q. By Lemma 3, clocks, port, timing constraints and data values are the same in components of B and interaction protocols components in B^{SR}. Furthermore, all rcv_i places are active. As previously, we distinguish the cases where a is a jump transition a or a delay transition δ.

If a is a jump transition, the corresponding transition in the interaction protocol can be executed as the guard and timing constraints of a are also true. As previously, the execution of a in both model leads to equivalent states. Thus we have $q^{SR} \xrightarrow{\beta^*} [q^{SR}] \xrightarrow{a} r^{SR}$ with $(r^{SR}, r) \in R$.

If a is a delay transition δ, then no interaction a is urgent at q. Similarly, no unary transition a is urgent in interaction protocol components at $[q^{SR}]$, and no β transition is possible. After waiting δ in both model, clocks are updated identically. Thus we have $q^{SR} \xrightarrow{\beta^*} [q^{SR}] \xrightarrow{\delta} r^{SR}$ with $(r^{SR}, r) \in R$. ■

6 Step 2: Use of a Single Clock

In this section, we explain how we refine Send/Receive-BIP models presented in Section 4 into *Single-Clock* Send/Receive-BIP. In a *Single-Clock* Send/Receive-BIP, all the time progress conditions and timing constraints of the model are expressed based on a single global clock g that is never reset. This clock measures the absolute time elapsed since the system starts executing.

The transformation from a Send/Receive model to a *Single-Clock* Send/Receive-BIP model involves the following steps:

1. We add the global clock g to each component.

2. For each clock c of a component B, we introduce a real variable ρ_c in order to store the absolute time of the last reset of the clock c with respect to the clock g. Whenever the clock c is reset by a transition of B, we assign to ρ_c the current value of g, denoted by $\rho_c := t(g)$, where $t(g)$ represents the valuation of the clock g at the current state of the system. Notice that the value of c can be computed from the current value of g and ρ_c by using the equality $c = g - \rho_c$.

3. We express any timing constraints tc using the clock g instead of clocks C. Using (1) we rewrite tc as follows: $\mathsf{tc} = \bigwedge_{c \in \mathsf{C}_i} l_c + \rho_c \leq g \leq u_c + \rho_c$. That is, tc is an interval constraint on g of the form: $\mathsf{tc} = \max\{l_c + \rho_c\}_{c \in \mathsf{C}_i} \leq g \leq \min\{u_c + \rho_c\}_{c \in \mathsf{C}_i}$.

4. Due to the previous transformation, local clocks C are no longer used by timing constraints, that is, they are not active [9]. Thus, we keep only the global clock g and the variables ρ_c.

Notice that steps 2, 3, and 4 apply only to Interaction Protocol components, since distributed atomic components have no clock.

Algorithm 1: Code generated for SR-BIP components.

```
Initialize()                     // initialize the Petri net and connections
G:= 0                            // initialize logical time
g := 0                           // reset real-time clock
while true do
    while $P_s \cap$ enabled $\neq \emptyset$ do          // send messages
        choose $p \in P_s \cap$ enabled
        send($p$)
        s:=NextState()
    $D$ :=nextDeadline($s$)
    $G := g$
    if $G > D$ then exit(DEADLINE_MISS)
    $N$ :=nextEnabled($P_u$, $G$)
    if $N > G$ then
        wait newMessage() $\vee$ $g \geq$ min$(N, D)$
        if newMessage() then                       // received message
            recv(message())
            s:=NextState()
            continue
    if $g \geq N \wedge D \geq g$ then
    G:= N
    choose $p \in P_u$ such that nextEnabled($p, G$) $= G$
    DoInternalComputation($p$)          // internal transition
    s:=NextState()

    if $g > D$ then exit(DEADLINE_MISS)
```

Single-Clock Send/Receive-BIP models are easier to map on a platform than Send/Receive-BIP, as they require a single real-time clock to be implemented. However, they are based on the fact that atomic components respond instantaneously to notification of Interaction Protocol components by sending offers. This assumption cannot be met in practice since execution of transitions as well as transmission of messages may take significant time.

7 Implementation and Experimental Results

7.1 Enforcing Robustness

Before generating code from components of a Single-Clock Send/Receive-BIP model, we transform the model to make it robust to execution time of transitions and communication primitives. To this end, we use the notion of *logical time*, which is the theoretical time instant at which transitions must be executed in the model. Logical time and actual time must coincide only for timing constraints of the initial model, that is, for transitions corresponding to interactions in the Interaction Protocol.

We transform Single-Clock Send/Receive-BIP as follows. In any Interaction Protocol component IP_j, we use a real variable G that stores the logical time of the last interaction execution. Initially, we set G to 0. Unary transitions in IP_j components update the current value of the logical time using $G := t(g)$. Moreover, when IP_j notifies a component B_i for executing a port p, it also sends the current value of the logical time G. Boolean reset variables r_c of atomic components B_i^{SR} are replaced by real valued variables ρ_c. B_i^{SR} keeps track of the last reset time of a clock c by executing $\rho_c := G$ instead of $r_c := \texttt{true}$, whenever c is reset. When Interaction Protocol component IP_j receives offers from a component B_i^{SR}, they update their local copy of variables ρ_c based on the values received in the offers.

In such a model, components can let time progress before sending messages, which may violate time progress conditions of states reached by components. Before executing an interaction a in IP_j at the current time $t(g)$ we check before updating the logical time G that the value $t(g)$ is allowed by the semantics, that is, no time progress condition is evaluated to $\texttt{false}$ at $t(g)$. If this condition is violated, we say that a deadline has been missed and the platform was not fast enough to execute the model.

7.2 Code Generation

The C++ code generated from a component B_i is shown in Algorithm 1. It uses the clock g giving access to the current value of the global time, and maintains the logical time in variable G. The function NextState() updates the state of the component upon execution of a transition. We denote by enabled the set of ports enabled at the current state. Given a port p , $\mathsf{nextEnabled}(p, G)$ denotes the next global time value from G at which p is enabled. This value is computed from the timing constraint tc_p of p using $\mathsf{nextEnabled}(p, G)$ $=$ **min** $\{t \geq G \mid \mathsf{tc}_p(t)\} \cup \{+\infty\}$. This notion is extended to sets of ports by taking the minimal value. Given a state s, $\mathsf{nextDeadline}(s, G)$ denotes the maximal global time value until which the component is allowed to stay at s. This value is computed from time progress conditions tpc_ℓ of ℓ, where ℓ is a place belonging to state s, as following: $\mathsf{nextDeadline}(s, G) =$ **max** $\{t \geq G \mid \bigwedge_{\ell \in s} \mathsf{tpc}_\ell(t)\} \cup \{G\}$.

First the component is initialized (Lines 1 to 3). Then, the main loop then starts by sending as soon as possible messages corresponding to all enabled send-ports (Lines 5 to 8). Unary ports P_u may have timing constraints, and have to be executed in real-time. We update the logical time along with checking for violations of deadlines at Lines 9 to 11. We stop if a deadline is missed, i.e. if there is a time progress condition that is evaluated to `false` at the new value of logical time provided by g. If no unary transition can be executed at the current value of the logical time (i.e. $N = \mathsf{nextEnabled}(P_u, G) > G$), we wait for a unary transition to be enabled (i.e. g reaches N), for the deadline to expire(i.e. g reaches D) or for a message to be received (Line 14). In the later case, we execute the corresponding port to take the message into account (Lines 15 to 18). If no message has been received until N (i.e $g \geq N$) and the deadline did not expire ($g \leq D$), we execute a unary transition enabled at logical time $G = N$ (Lines 20 to 23). Otherwise, we report a deadline miss if $g > D$ (Line 25). Notice that unary transitions can only be found in Interaction Protocol components IP_j and correspond to interactions. An Interaction Protocol component computes the deadlines according to the pending offers it received and schedules an enabled interaction ASAP. In atomic components, Lines 15 to 18 cannot be reached as N and D are always $+\infty$.

7.3 Demosaicing Algorithm

Our case study to evaluate our framework is a real-time image reconstruction system. A demosaicing algorithm transforms the raw data from a camera sensor into an actual image. A camera sensor is an array of light sensors, each of them outputting a single value. A color filter placed over the sensor array ensures that each sensor receives either red, green, or blue light. The filtering is done according to the pattern presented on the left of Figure 7. The obtained raw image contains a single color component for each pixel. Demosaicing yields an RGB image by interpolating for each pixel the missing color components from the values of the neighbor pixels. In Figure 7, the neighborhood contains only adjacent pixels. Depending upon the interpolation algorithm used, this neighborhood may change. In our case, we use a neighborhood of size 5×5, centered on the interpolated pixel. We do not detail here the interpolation function used.

Figure 8 shows the BIP model of demosaicing for 4 parallel blocks. Initially, the image is loaded by the component I, which trigger a $\mathsf{global\_start}$ interaction (not shown on the Figure) between components D_{ij}, J_i, I and O. These components reset their respective clocks upon execution of $\mathsf{global\_start}$.

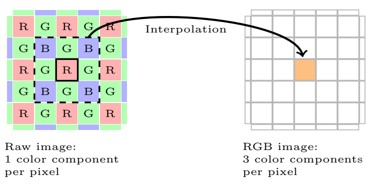


Figure 7: Demosaicing raw data to RGB image.

This algorithm is parallelized by cutting the raw image into blocks, each being demosaiced concurrently. The image is first split into lines by the interaction named split, then each line i is split into blocks by the interaction named $\mathsf{split\_line\_}i$. The detailed behavior of components D_{ij} and C_{ij} is depicted in Figure 9.

Component D_{ij} performs actual demosaicing of the image block located at i, j, whenever it receives an image block through its port $\mathsf{get\_block}$. Component C_{ij} controls timing of the component D_{ij}. We use timing constraints and time progress conditions to enforce delivery of a RGB image within a given amount of time after the raw image has been provided. The parameter k allows to control the time to demosaice a block. When D_{ij} finishes the demosacing of a block, if less time than k elapsed since the last global start, C_{ij} allows interaction $\mathsf{trans\_}ij$ that transmits the demosaiced block to J_i. Otherwise, the block is not transmitted, C_{ij} declares a $\mathsf{timeout}$ through unary interaction $\mathsf{timeout}$ and returns with D_{ij} to the initial state through interaction cancel. The component J_i joins blocks to form the line i and transmits it through $\mathsf{trans\_}i$. Its transition from the receiving state to the transmitting state is triggered if all blocks are received or if too much time elapsed since $\mathsf{global\_start}$.

Finally, component O merges received lines and outputs the image. The image is outputted either if

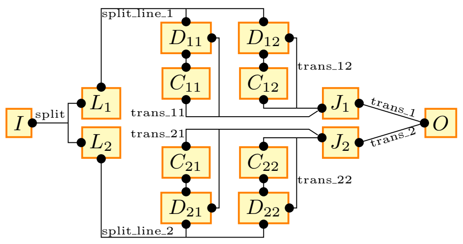


Figure 8: BIP model for demosaicing.

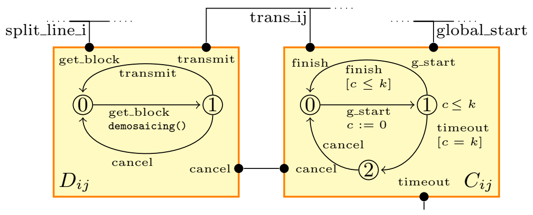


Figure 9: Detail of the C_{ij} and D_{ij} components.

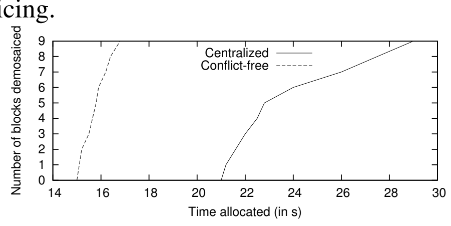


Figure 10: Time needed to process a 25MP image.

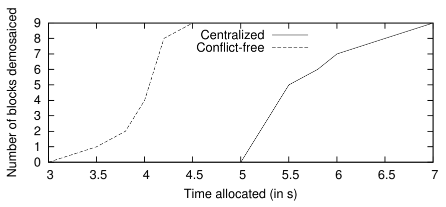


Figure 11: Time needed to process a 6MP image.

all lines are received or if too much time elapsed since global_start. If some blocks or lines were not transmitted, the outputted image is incorrect on the corresponding blocks.

We consider two different partitions for generating the distributed model. The first one is *centralized* partition and puts all interactions in the same Interaction Protocol component. The second one is *conflict-free* partition, and is such that:

- Interaction split, start and all interactions split_line_i define the first class of the partition.
- For each i, the interactions trans_ij are grouped in the same class, together with reset interactions.
- The interactions trans_i form the last class.

7.4 Experimental Results

We demosaice raw images of size 25×10^6 pixels and 6×10^6 pixels. The BIP model splits each image into 9 blocks, that are demosaiced concurrently. We generate distributed code for both the centralized and the conflict-free versions. We run the code on a UltraSparc T1 that allows parallel execution of 24 processes. The parameter k in the component C_{ij} controls the amount of time after which the image must be outputted.

Figures 10 and 11 show the number of blocks processed depending on the amount of time allowed k, respectively for 25M pixels and 6M pixels raw images, for both centralized and conflict-free partition. The conflict-free partition exhibits a speedup ranging between 1.5 to 2 comparatively to the centralized partition. The conflict-free partition allows more parallelism between interactions, for instance, interactions split_line_i can be executed in parallel, where the centralized does not. Since demosaicing components run concurrently in both implementations, the speedup stems solely from the possible parallelism between interactions.

8 Related Work

LOTOS [10] is a specification language based on process algebra, that encompasses multiparty interactions. In [16], the authors describe a method of executing a LOTOS specification in a distributed fashion. This implementation is obtained by constructing a tree at runtime. The root is the main connector of the LOTOS specification and its children are the subprocesses that are connected. A synchronization between two processes is handled by their common ancestor. Another framework that offers automatic distributed code generation is described in [13]. The input model consists of composition of I/O automata, from which a Java implementation using MPI for communication is generated. The model, as well as the implementation, can interact with the environment. However, connections between I/O automata (binary synchronization) are less expressive than BIP interactions, as proved in [6]. Finally, the framework in [13] requires the designer to specify low-level elements of a distributed system such as channels and schedulers.

In the context of the framework, automated implementation of distributed applications from BIP models has been addressed in [7, 8]. The authors propose a 3-layer architecture or, where the first layer is concerned with behavior of components, the second layer handles execution of interactions, and the third layer resolves distributed conflicts. However, this line of work is not concerned with notion of time and timing constraints. On the timed models side, in [1], the authors study the problem of model-based implementation of sequential timed BIP models. The closest work to this paper is the approach in [15]. This technique transforms a timed BIP model into a parallel time-aware code. The main difference is unlike our approach, the method in [15] augments the code with only one centralized engine. Such an engine can potentially become a bottleneck and consequently make the generated code inefficient.

Finally, TIMES is a tool for modelling and schedulability analysis of embedded real-time systems [3]. The tool is featured with a code generator for sequential C-code synthesis on LegoOS platform from the input model. Unlike our approach in this paper, TIMES is not able to generate concurrent code.

9 Conclusion

Concurrent real-time systems have numerous applications in today's embedded computing systems. However, correct development of such systems is known to be a notoriously difficult problem. In this paper, we focused on model-based automated and correct-by-construction development of multi-process applications that are subject to timing constraints. We proposed a chain of transformations that starts from an abstract model of the application expressed in terms of a set of interacting components. Each component is constrained by a set of local logical timing requirements. In each step, a transformation obtains a model that encompasses platform constraints, such as point-point communication and physical real time. Each transformation ensures that all functional properties of the input model are preserved. Our transformations are fully implemented and validated on a framework for real-time image reconstruction system.

For future work, there are several research directions. An important extension of this work is to design transformations, in which schedulers are not necessarily conflict-free. Such schedulers potentially result in better levels of parallelism. A more challenging (but highly needed) research direction is model-based development of distributed real-time applications, where a global perfect clock cannot be assumed.

References

[1] T. Abdellatif, J. Combaz, and J. Sifakis. Model-based implementation of real-time applications. In *ACM International Conference on Embedded Software (EMSOFT)*, pages 229–238, 2010. 1, 8

[2] R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994. 1, 2.1, 2.2, 2.2

[3] T. Amnell, E. Fersman, L. Mokrushin, P. Pettersson, and W. Yi. TIMES - a tool for modelling and implementation of embedded systems. In *Proceedings 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 460–464, 2002. 8

[4] A. Basu, P. Bidinger, M. Bozga, and J. Sifakis. Distributed semantics and implementation for systems with interaction and priority. In *FORTE*, pages 116–133, 2008. 2, 3, 4.1

[5] A. Basu, M. Bozga, and J. Sifakis. Modeling heterogeneous real-time components in BIP. In *Software Engineering and Formal Methods (SEFM)*, pages 3–12, 2006. 1

[6] S. Bliudze and J. Sifakis. A notion of glue expressiveness for component-based systems. In *Concurrency Theory (CONCUR)*, pages 508–522, 2008. 8

[7] B. Bonakdarpour, M. Bozga, M. Jaber, J. Quilbeuf, and J. Sifakis. A framework for automated distributed implementation of component-based models. *Springer Journal on Distributed Computing (DC)*, 25(1):383–409, October 2012. 1, 3, 3, 8

[8] B. Bonakdarpour, M. Bozga, and J. Quilbeuf. Automated distributed implementation of component-based models with priorities. In *ACM International Conference on Embedded Software (EMSOFT)*, pages 59–68, 2011. 8

[9] C. Daws and S. Yovine. Reducing the number of clock variables of timed automata. In *RTSS*, pages 73–81. IEEE Computer Society, 1996. 4.1, 4

[10] ISO/IEC. *Information Processing Systems – Open Systems Interconnection: LOTOS, A Formal Description Technique Based on the Temporal Ordering of Observational Behavior*, 1989. 8

[11] E. Jee, S. Wang, J.-K. Kim, J. Lee, O. Sokolsky, and I. Lee. A safety-assured development approach for real-time software. In *Proceedings of the 16th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 133–142, 2010. 1

[12] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541 –580, apr 1989. 2.2

[13] J. A. Tauber, N. A. Lynch, and M. J. Tsai. Compiling IOA without global synchronization. In *Symposium on Network Computing and Applications (NCA)*, pages 121–130, 2004. 1, 8

[14] A. Triki, B. Bonakdarpoor, J. Combaz, and S. Bensalem. Automated conflict-free concurrent implementation of timed component-based models. Technical report, Verimag Research Report.

[15] A. Triki, J. Combaz, S. Bensalem, and J. Sifakis. Model-based implementation of parallel real-time systems. In *FASE*, pages 235–249, 2013. 3, 8

[16] G. von Bochmann, Q. Gao, and C. Wu. On the distributed implementation of lotos. In *FORTE*, pages 133–146, 1989. 8