Noname manuscript No.
(will be inserted by the editor)

Model-based Implementation of Distributed Systems with Priorities

Borzoo Bonakdarpour · Marius Bozga · Jean Quilbeuf

Abstract Model-based application development aims at increasing the application's integrity by using models employed in clearly defined transformation steps leading to correct-by-construction artifacts. In this paper, we introduce a novel model-based approach for constructing correct distributed implementation of component-based models constrained by priorities. We argue that model-based methods are especially of interest in the context of distributed embedded systems due to their inherent complexity (e.g., caused by non-deterministic nature of distributed systems). Our method is designed based on three phases of transformation. The input is a model specified in terms of a set of behavioral components that interact through a set of high-level synchronization primitives (e.g., rendezvous and broadcasts) and priority rules for scheduling purposes. The first phase transforms the input model into a model that has no priorities. Then, the second phase transforms the deprioritized model into another model that resolves distributed conflicts by incorporating a solution to the committee coordination problem. Finally, the third phase generates distributed code using asynchronous point-to-point message passing primitives (e.g., TCP sockets). All transformations preserve the properties of their input model by ensuring observational equivalence. All the transformations are implemented and our experiments validate their effectiveness.

Keywords Component-based modeling, Automated transformation, Distributed systems, BIP, Correctness-by-construction, Committee coordination, Conflict resolution.

The research leading to these results has received funding from Canada under NSERC Discovery Grant 418396-2012, from the European Community's Seventh Framework Programme [FP7/2007-2013] under grant agreement no 248776 (PRO3D) and no 257414 (ASCENS), and from ARTEMIS JU grant agreement ARTEMIS-2009-1-100230 (SMECY).

B. Bonakdarpour
School of Computer Science
University of Waterloo
200 University Avenue West, Waterloo, Ontario, Canada, N2L 3G1
E-mail: borzoo@cs.uwaterloo.ca

M. Bozga and J. Quilbeuf
UJF-Grenoble 1 / CNRS, VERIMAG UMR 5104
Grenoble, F-38041 France
E-mail: {marius.bozga, jean.quilbeuf}@imag.fr

1 Introduction

Correct design and implementation of computing systems has been an ongoing research topic in the past three decades. This problem is significantly more challenging in the context of distributed systems. This challenge is due to the inherent complex nature of distributed systems caused by a number of factors, such as non-determinism, non-atomic execution of processes, race conditions, and occurrence of faults. Correctness of distributed implementations is of significant importance in the context of embedded applications, as such applications are often employed in safety/mission-critical systems.

A promising approach to address the challenges in correct development of a distributed application is to automatically generate code from abstract models. Model-based development of distributed applications aims at increasing their integrity by using explicit models employed in clearly defined transformation steps leading to correct-by-construction artifacts. This approach is highly beneficial, as one can ensure functional correctness of the system by dealing with a high-level formally specified model that abstracts implementation details and then derive a correct implementation through a series of transformations that terminates when actual executable code is obtained.

In this paper, we focus on the BIP framework [6, 16] as our formal modelling language. BIP (Behaviour, Interaction, Priority) is based on a semantic model encompassing composition of heterogeneous components. The *behaviour* of a component in BIP is described as an automaton or Petri net extended by data and functions given in C++. BIP uses a diverse set of composition operators for obtaining composite components from a set of components. The operators are parametrized by a set of *interactions* between the composed components. Finally, *priorities* are used to specify different scheduling mechanisms. Transforming a BIP model into a distributed implementation involves addressing three fundamental issues:

1. *(Concurrency)* Components and interactions should be able to run concurrently while respecting the sequential semantics of the high-level model.
2. *(Conflict resolution)* Interactions that share a common component can potentially conflict with each other. Such interactions should execute in mutual exclusion.
3. *(Enforcing priorities)* When two interactions are simultaneously ready to execute, only the one with higher priority can execute.

These issues introduce challenging problems in a distributed setting. The conflict resolution issue can be addressed by incorporating solutions to the *committee coordination problem* [14] for implementing multiparty interactions. For example, Bagrodia [2] proposes different solutions with different degrees of parallelism. The most distributed solution is based on the drinking philosophers problem [13], and has inspired the approaches by Pérez et al. [25] and Parrow et al. [24]. In the context of BIP, a transformation addressing all the three challenges through employing a *centralized scheduler* is proposed in [5]. Moreover, in [8–10], the authors propose transformations that address the concurrency issue by breaking the atomicity of interactions, and, the conflict resolution problem by designing an architecture to automatically augment an implementation with a solution to the committee coordination problem in a distributed fashion. However, designing transformations that enforce priorities between interactions in a distributed setting remains unaddressed in spite of the vital role priorities plays in designing systems.

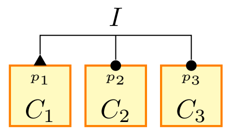


Fig. 1 A component-based model with broadcast interaction.

In Subsection 1.1, we discuss the importance of incorporating priorities as a scheduling tool to solve a wide range of computing problems and the main difficulty in their implementation. In Subsection 1.2, we state our contributions in this paper.

1.1 Motivation

Priorities are widely used in system design as a way of scheduling events. Below, we present examples of how applying priorities can guide a system to satisfy certain properties:

- **Ensuring safety.** Safety properties are normally of the form "nothing bad happens during the system execution". In the context of concurrent and distributed computing, an example of such a bad thing is often due to the existence of a set of processes competing over a resource. Priorities can be used to resolve race conditions. For instance, one way to prevent two processes to enter a critical section simultaneously is to give explicit priority to one process. Dynamic priorities can then be used to ensure non-starvation.
- **Improving performance.** In distributed systems, it is often the case that certain resources have higher demands. For example, in *group mutual exclusion* [19], as Mittal and Mohan argue [23], in many commonly considered systems, group access requests are non-uniform. Hence, in order to improve the performance, it is reasonable to devise algorithms that give priority to groups that require a resource with higher demand. A concrete example of group mutual exclusion is the well-known readers/writers problem. In most cases, we give priority to readers to improve the performance.
- **Reducing non-determinism.** Non-determinism in distributed and concurrent computing is one of the sources of obtaining a diverse set of behaviors. In many scenarios and in particular, in embedded applications, it is desirable to constrain the system, so that it behaves in a predictable fashion. For example, consider the model in Figure 1 with the following semantics. Port p_1 is an active port (e.g., a trigger), whereas ports p_2 and p_3 are passive (e.g., synchrons). Connector I is enabled if port p_1 is enabled and other components can optionally participate in the interaction (i.e., if their corresponding ports are enabled). Thus, connector I allows interactions of the following set: $\{p_1, p_1p_2, p_1p_3, p_1p_2p_3\}$. Now, if we are to build a *broadcast* interaction out of I, all passive ports that are listening (enabled) have to be activated whenever this interaction takes place. This can be achieved when interaction $p_1p_2p_3$ is given higher priority than p_1p_2 and p_1p_3 that are given higher priority that p_1 alone.

The main challenge in utilizing priorities in a distributed setting is their correct implementation while ensuring efficiency. This is due to the fact that components need

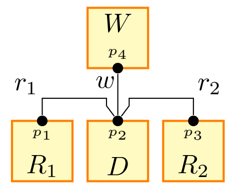


Fig. 2 A simple BIP model for multiple readers/single writer problem.

to obtain a reliable knowledge about enabledness of interactions, so that only the interaction with the highest priority is executed. In [4], the authors propose a model checking approach that determines whether actions of a given Petri net can be executed without violating priority rules. However, the downside of this approach is (1) it has scaling issues, as it uses model checking, and (2) in most cases the local knowledge of processes is shown to be insufficient to decide whether or not an action can be executed. Other approaches include applying customized algorithms to implement priority rules for specific problems in distributed computing (e.g., [23]).

To better describe our idea in this paper, consider the multiple readers/single writer problem. A high-level component-based model to solve the problem is shown in Figure 2. Component D contains shared data, component W is a writer, and components R_1 and R_2 are two readers. Components W, R_1, and R_2 access the shared data through binary rendezvous interactions w, r_1, and r_2, respectively. The semantics of this model requires that these interactions are executed atomically, ensuring sequential consistency of the shared data. Using the approach introduced in [8, 9], one can automatically generate a distributed implementation that is observationally equivalent to the high-level model. However, the solutions in [8, 9] come short in implementing a priority rule such as $(w < r_1) \wedge (w < r_2)$, where the writer has to wait as long as readers are reading the shared data.

These examples clearly demonstrate the demand for developing methods that automatically construct a correct distributed implementation by starting from a high-level model along with a set of priority rules. This way, all implementation issues are dealt with by transformation algorithms. Thus, a designer only needs to make minimal effort to develop an abstract model of the distributed application.

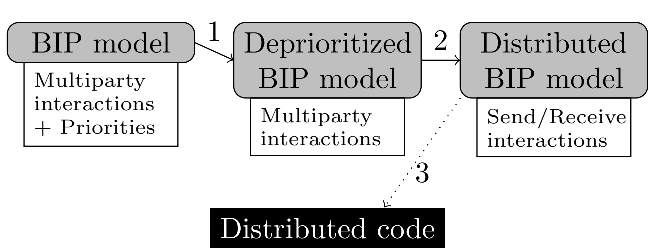


Fig. 3 Steps for generating a distributed implementation from a high-level BIP model.

1.2 Contributions

Our contributions in this paper are as follows:

- We propose a transformation that takes a high-level BIP model with priorities as input and generates a BIP model without priorities as output. This corresponds to the first step in Figure 3.
- We show the correctness of this transformation by proving that the initial and transformed models are observationally equivalent.
- We apply the transformations introduced in [8–10] to derive a distributed model, where multiparty interactions are implemented in terms of asynchronous point-to-point send/receive primitives. This corresponds to the second step in Figure 3. From this distributed model, we generate distributed code, as explained in [8–10], which completes the design flow from the initial BIP model with priorities to a correct distributed implementation.
- Finally, we validate the effectiveness of our approach by modelling a distributed *jukebox* application and *dining philosophers* in BIP and conducting experiments on the generated distributed code. The jukebox application incorporates priorities to manage demands on reading discs. Our experiments show that the overhead of our transformations has minimal effect on the benefit of using priorities.

We emphasize that although our focus is on the BIP framework, all results in this paper can be applied to any model that is specified in terms of a set of components synchronized by broadcast and rendezvous interactions. We note that our focus in this paper is on the class of BIP models that are not augmented with timing constraints. The problem of generating real-time distributed code from BIP models is outside the scope of this paper.

Organization. The rest of the paper is organized as follows. In Section 2, we present the basic semantics model of BIP. Then, in Section 3, we describe our transformation for deriving a model that has no priorities. Our approach for deriving a distributed model and code is presented in Section 4. Section 5 is dedicated to our case studies and experimental results, while related work is discussed in Section 6. Finally, we conclude and discuss future work in Section 7.

2 Basic Semantic Models of BIP

In this section, we present operational *global state* semantics of BIP [6]. BIP is a component framework for constructing systems by superposing three layers of modelling: *Behaviour*, *Interaction*, and *Priority*.

Atomic Components. We define an *atomic component* as a transition system extended with a set of ports and a set of variables. Each transition is guarded by a predicate on the variables, triggers an update function, and is labelled by a port. The ports are used for communication among different components and each port is associated with a subset of variables of the component.

Definition 1 (Atomic Component) An *atomic component* B is a labelled transition system represented by a tuple (Q, X, P, T) where:

- Q is a set of *control states*.
- X is a set of *variables*.
- P is a set of *communication ports*. Each port is a pair (p, X_p) where p is a label and $X_p \subseteq X$ is the set of variables bound to p. By abuse of notation, we denote a port (p, X_p) by p.
- T is a set of *transitions* of the form $\tau = (q, p, g, f, q')$, where $q, q' \in Q$ are control states, $p \in P$ is a port, g is the *guard* of τ and f is the *update function* of τ. g is a predicate defined over the variables in X and f is a function that computes new values for X according to the previous ones.

We denote the set of valuations of X by $\mathbf{X}$, and the set of local states of a component by $Q \times \mathbf{X}$. Let (q, v) and (q', v') be two states in $Q \times \mathbf{X}$, p be a port in P, and v''_p be a valuation in $\mathbf{X}_p$ of X_p. We write $(q, v) \xrightarrow{p(v''_p)} (q', v')$ iff $\tau = (q, p, g, f, q') \in T$, $g(v)$ is true, and $v' = f(v[X_p \leftarrow v''_p])$, (i.e., v' is obtained by applying f after updating variables X_p associated to p by the values v''_p). When the communication port is irrelevant, we simply write $(q, v) \to (q', v')$. Similarly, $(q, v) \xrightarrow{p}$ means that there exists a transition $\tau = (q, p, g, f, q')$ such that $g(v) = true$; i.e., p is *enabled* in state (q, v).

Example 1 Figure 4(a) shows an atomic component B, where $Q = \{s\}$, $X = \{n\}$, $P = \{(p, \{n\})\}$, and $T = \{(s, p, g, f, s)\}$. Here g is always true and f is the identity function.

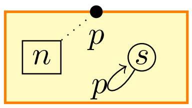
Interactions. For a model built from a set of n atomic components $\{B_i = (Q_i, X_i, P_i, T_i)\}_{i=1}^n$, we assume that their respective sets of ports and variables are pairwise disjoint; i.e., for any two $i \neq j$ in $\{1..n\}$, we require that $P_i \cap P_j = \emptyset$ and $X_i \cap X_j = \emptyset$. Thus, we define the set $P = \bigcup_{i=1}^n P_i$ of all ports in the model as well as the set $X = \bigcup_{i=1}^n X_i$ of all variables. An *interaction* a is a triple (P_a, G_a, F_a), where $P_a \subseteq P$ is a set of ports, G_a is a guard, and F_a is an update function, both defined on the variables associated by the ports in P_a (i.e., $\bigcup_{p \in P_a} X_p$). By $P_a = \{p_i\}_{i \in I}$, we mean that for all $i \in I$, $p_i \in P_i$, where $I \subseteq \{1..n\}$. We denote by F_a^i the projection of F_a on X_{p_i}.

Priorities. Given a set γ of interactions, a priority between two interactions specifies which one is preferred over the other. We define such priorities through a partial order $\pi \subseteq \gamma \times \gamma$. We write $a \pi b$ if $(a, b) \in \pi$, which means that a has less priority than b.

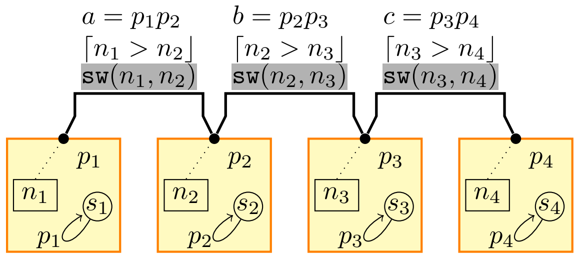
Definition 2 (Composite Component) A *composite component* (or simply *component*) is defined by a set of components, composed by a set of interactions γ and a priority partial order $\pi \subseteq \gamma \times \gamma$. We denote $B \stackrel{def}{=} \pi\gamma(B_1, \ldots, B_n)$ the component obtained by composing components $B_1, \ldots, B_n$ using the interactions γ and the priorities π.

Note that if the system does not contain any priority, we may omit π.

Definition 3 (Composite Component Semantics) The behaviour of a composite component without priority $\gamma(B_1, \ldots, B_n)$, where $B_i = (Q_i, X_i, P_i, T_i)$ and $\to_i$ is the



(a) An atomic component



(b) A BIP composite component that sorts integers n_i, obtained by gluing 4 atomic components using 3 interactions.

Fig. 4 Atomic and composite components in BIP

transition relation between states of B_i, is a transition system $(Q, \gamma, X, \rightarrow_\gamma)$, where $Q = \times_{i=1}^{n} Q_i$, $X = \bigcup_{i=1}^{n} X_i$ and $\rightarrow_\gamma$ is the least set of transitions satisfying the rule:

$$\frac{\begin{array}{c} a = (\{p_i\}_{i \in I}, G_a, F_a) \in \gamma \qquad G_a(v_1, \ldots, v_n) \\ \forall i \notin I.\ (q_i, v_i) = (q_i', v_i') \qquad \forall i \in I.\ (q_i, v_i) \xrightarrow{p_i(v''_{p_i})}_i (q_i', v_i'), v''_{p_i} = F_a^i(v_1, \ldots, v_n) \end{array}}{((q_1, v_1), \ldots, (q_n, v_n)) \xrightarrow{a}_\gamma ((q_1', v_1'), \ldots, (q_n', v_n'))}$$

A state (q, v) of $\gamma(B_1, \ldots, B_n)$ corresponds to states $(q_1, v_1), \cdots, (q_n, v_n)$ of components $B_1, \cdots, B_n$. We define the behaviour of the composite component $B = \pi\gamma(B_1, \ldots, B_n)$ as the transition system $(Q, \gamma, X, \rightarrow_\pi)$, where $\rightarrow_\pi$ is the least set of transitions satisfying the rule:

$$\frac{(q, v) \xrightarrow{a}_\gamma (q', v') \qquad \forall a' \in \gamma.\ a\pi a' \implies (q, v) \overset{a'}{\not\rightarrow}_\gamma}{(q, v) \xrightarrow{a}_\pi (q', v')}$$

Intuitively, the first inference rule specifies that a composite component $B = \gamma(B_1, \ldots, B_n)$ can execute an interaction $a \in \gamma$ iff (1) for each port $p_i \in P_a$, the corresponding atomic component B_i can execute a transition labelled by p_i, and (2) the guard G_a of the interaction evaluates to true in the current state. Any interaction verifying these two conditions is *enabled*. Execution of the interaction modifies components' variables by first applying update function F_a to associated variables and then function f_i inside each component. The states of components that do not participate in the interaction stay unchanged. The second inference rule simply filters out interactions which are not maximal with respect to priorities. An interaction can execute only if no other one with higher priority is enabled.

Example 2 Figure 4(b) illustrates a composite component $\gamma(B_1, \cdots, B_4)$, where each B_i is identical to component B in Figure 4(a). The set γ of interactions is $\{a, b, c\}$, where $a = (\{p_1, p_2\}, n_1 > n_2, \texttt{sw}(n_1, n_2))$ and function $\texttt{sw}$ swaps the values of its arguments. Interactions b and c are defined in a similar fashion. Interaction a is enabled when ports p_1 and p_2 are enabled and the value of n_1 (in B_1) is greater than the value of n_2 (in B_2). Thus, the composite component B sorts variables $n_1 \cdots n_4$, such that n_1 contains the smallest and n_4 contains largest values.

It may be desirable to always execute interaction a when possible. This can be done by adding the two priority rules $b\pi a$ and $c\pi a$. We denote the obtained component by $\pi\gamma(B_1, \ldots, B_4)$. We will use this example to illustrate the transformations presented in this paper.

We now introduce the notion of *conflicting interactions*. Intuitively, two interactions a_1 and a_2 are *weakly conflicting* iff they share a common component.

Definition 4 (Weak Conflict) Two interactions a_1 and a_2 are *weakly conflicting* (denoted $a_1 \oplus a_2$) iff there exist two ports p and q in some component B such that $p \in P_{a_1}$ and $q \in P_{a_2}$.

This kind of conflict is called *weak* because its constraint is weaker than the definition of conflicting interactions in [9] that we call here a *strong conflict*. Two interactions are strongly conflicting iff they share a component B and either (1) they share a common port in B, or (2) there exist two ports, each in a distinct interaction, such that these two ports label two conflicting transitions (i.e., transitions that originate from the same state). Clearly, a strong conflict implies a weak conflict, but the converse is not true.

3 Deprioritizing a BIP Model

In this section, we describe our approach to transform a BIP model B into an equivalent model without priorities, denoted $\tilde{B}$. Intuitively, our transformation proceeds as follows:

1. First, it replaces each atomic component in B by a functionally equivalent send/receive atomic component, where atomicity of transitions and interactions is broken. This first transformation, already used in [5,8,9] separates the synchronization from internal computation on component transitions and enables concurrent execution of atomic components.
2. Then, it inserts *manager* components for handling interactions. These managers detect enabledness of interactions and schedule them for execution according to priority rules. Managers interact with each other through multi-party interactions in order to maintain a consistent view on the state of the system.

This section is organized as follows. Subsection 3.1 describes the first step of the transformation (breaking atomicity). Subsections 3.2 and 3.3 present the second step of the transformation (inserting and connecting manager components). Subsection 3.4 demonstrates the correctness of our approach, while Subsection 3.5 discusses binary versus n-ary interactions.

3.1 Breaking Atomicity

The transformation of atomic components splits each transition into two consecutive steps: (i) an *offer* that publishes the current state of the component, and (ii) a *notification* that triggers the update function. The intuition behind this transformation is that the offer transition corresponds to sending information about component's intention to interact with the other components. The notification transition receives the response

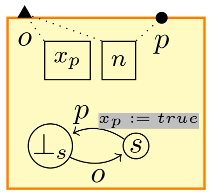


Fig. 5 Transformed version of one atomic component from Figure 4(b)

from the scheduler components (described later), once some interaction has been completed. Local update functions can then be executed concurrently and independently by components upon receiving a notification.

The offer transition publishes its enabled ports through a special port named o. Enabled ports are encoded through a list of Boolean variables. After the computation of the local function, this list is updated to the ports that are enabled at the next control state. Notification transitions are triggered by corresponding ports from the original atomic component.

Definition 5 (Transformed atomic components) Let $B = (Q, X, P, T)$ be an atomic component. The corresponding transformed atomic component is $B^{\perp} = (Q^{\perp}, X^{\perp}, P^{\perp}, T^{\perp})$, such that:

- $Q^{\perp} = Q \cup \{\perp_s \mid s \in Q\}$.
- $X^{\perp} = X \cup \{x_p\}_{p \in P}$, where each x_p is a Boolean variable indicating whether port p is enabled.
- $P^{\perp} = P \cup \{o\}$, where o is an offer port. All variables in $X^{\perp}$ are associated to o (i.e., $X_o = X^{\perp}$).
- For each transition $\tau = (q, p, g, f, q') \in T$, we include the following two transitions in $T^{\perp}$:
  1. *offer* $\tau_o^q = (\perp_q, o, g_o, f_o, q)$, where $g_o = true$ and f_o is the identity function, and
  2. *notification* $\tau_p^q = (q, p, g_p, f_p, \perp_{q'})$, where $g_p = true$ and f_p applies f_τ on X and for each port $r \in P$, it sets x_r to true if $\tau' = (q', r, g', f', q'') \in T$ for some q'' and g' is true. Otherwise, x_r is set to false.

In Definition 5, states $\{\perp_s \mid s \in Q\}$ from where the component sends offers, are called *busy* or *unstable states*. States Q, from where the component is waiting to receive a notification, are called *stable states*.

Example 3 Figure 5 shows the transformed version of the atomic component shown in Figure 4(a). Initially, the component is in busy state $\perp_s$ and the value of x_p is true; i.e., the component is willing to interact on port p. Then, it sends an offer through port o containing the current values of x_p and n and reaches stable state s. The reception of a notification corresponds to the p-labelled transition that brings back the component to the initial busy state.

3.2 Interaction Managers

The set of managers are introduced to execute interactions according to the global semantics of the original BIP model described in Section 2. To this end, a manager

port	*variables*	*description*
o_i^a	$\{x_{p_i}^a\} \cup X_{p_i}^a$	receives offers from atomic component B_i
ι	$\emptyset$	change status to *enabled* or *disabled* (internal port)
$start_a$	$\emptyset$	triggers interaction execution
n_a	$\{X_{p_i}^a\}$	notifies atomic components upon execution
dis_a	$\emptyset$	signals *disabled* status to other managers
$\oplus_a$	$\{b_i^a\}$	gets notified about execution of a weakly conflicting interaction by other managers
$\oplus dis_a$	$\{b_i^a\}$	similar to port $\oplus_a$, but for interactions with higher priority

Table 1 Ports of a manager component

component for an interaction a has to (i) detect enabledness of a by listening to offers sent by atomic components, (ii) trigger the execution of a, (iii) notifies atomic components as well as the other conflicting managers, whenever the interaction is executed.

Observe that if two interactions are weakly conflicting, then executing one can change the status of the other. For instance, let a and b be two interactions, such that $a \oplus b$; i.e., they share some component B. Obviously, executing a triggers a transition in component B. This transition can result in changing the status of interaction b. That is, until component B completes its local execution and sends a new offer, the status of enabled ports and values of variables in B can change.

Definition 6 (Interaction Manager) Let $a \in \gamma$ be an interaction, where $P_a = \{p_i\}_{i \in I}$. The interaction manager M_a is an atomic component $M_a = (Q, X, P, T)$ defined as follows:

- The set of control states is $Q = \{undef, en, dis, exc\}$. Intuitively, in state *undef* (*undefined*), the manager does not have enough information to decide whether or not interaction a is enabled. This is normally because some offers have not been received yet. In states *en* (*enabled*) and *dis* (*disabled*), the manager knows that a is enabled or disabled, respectively. In state *exc* (*executing*), the interaction a is executing.
- The set of variables is $X = \{b_i^a\}_{i \in I} \cup \left\{\{x_{p_i}^a\} \cup X_{p_i}^a\right\}_{p_i \in a}$. For every component B_i, the manager holds a Boolean variable b_i^a which is true iff component B_i is in a stable state, that is, waiting for a notification. For every port $p_i \in a$, the manager holds respectively, a Boolean variable $x_{p_i}^a$ which indicates the status of the port (i.e., enabled or disabled) and variables $X_{p_i}^a$ that is, data associated to the port p_i.
- The set of ports P and their associated variables is presented in Table 1.
- The set of transitions T and their intuitive meaning is presented in Table 2.

Example 4 Figure 6 shows the manager component for interaction a in Figure 4(b). It contains variables b_1^a and b_2^a since interaction a involves components B_1 and B_2. The manager contains two offer ports o_1^a and o_2^a. Port o_i^a, $i \in \{1, 2\}$, is associated with variables (1) $x_{p_i}^a$, which indicates the status of port p_i in B_i, and (2) n_i^a, that are local copies of variables n_i associated to ports p_i in Figure 4(b). All these variables

Transition	*Guard / Function*	*Description*
$undef \xrightarrow{o_i^a} undef$	- / $b_i^a := true$	receive offer from B_i
$undef \xrightarrow{\iota} en$	$G_a \wedge$ $\forall i \in I.\ (b_i^a \wedge x_{p_i}^a)$ / -	change state to *enabled*
$undef \xrightarrow{\iota} dis$	$(\forall i \in I. b_i^a) \wedge (\neg G_a \vee \exists i \in I. \neg x_{p_i}^a)$ / -	change state to *disabled*
$en \overset{start_a}{\rightarrow} exc$	- / $\{b_i^a\} := false$; $\{X_{p_i}^a\} := F_a(\{X_{p_i}^a\})$	execute interaction, apply update function.
$exc \overset{n_a}{\rightarrow} undef$	- / -	notifies atomic components on execution
$dis \overset{dis_a}{\rightarrow} dis$	- / -	signals *disabled* state
$dis \overset{\oplus_a}{\rightarrow} undef$ $undef \overset{\oplus_a}{\rightarrow} undef$ $en \overset{\oplus_a}{\rightarrow} undef$	- / -	gets notified about execution of a weakly conflicting interaction
$dis \overset{\oplus dis_a}{\rightarrow} undef$	- / -	gets notified about execution of a higher priority weakly conflicting interaction

Table 2 Transitions of a manager component

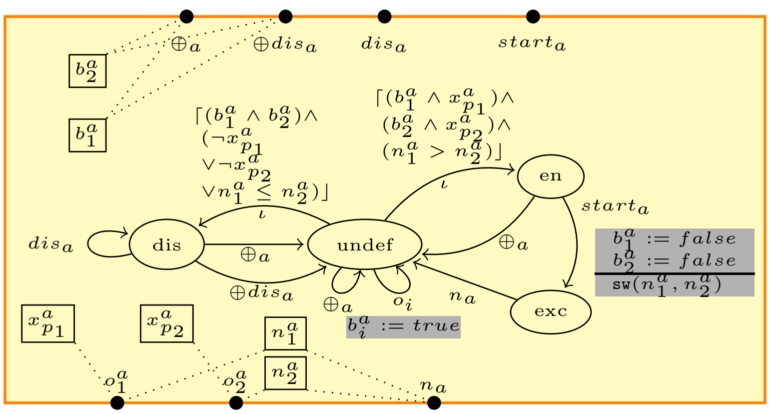


Fig. 6 The manager component for interaction a between components B_1 and B_2 in Figure 4(b).

are refreshed upon receiving an offer through ports o_i^a. The transition from *undef* to *en* guarded by $(b_1^a \wedge x_{p_1}^a) \wedge (b_2^a \wedge x_{p_2}^a) \wedge (n_1^a > n_2^a)$ switches from *undefined* to *enabled* state. The two first conjuncts ensures that (1) B_1 and B_2 are in *stable* state, and (2) p_1 and p_2 are enabled. The latter conjunct corresponds to the guard of interaction a in Figure 4(b). Likewise, the transition from *undef* to *dis* is possible only when B_1 and B_2 are in *stable* state and the interaction a is disabled. The update function associated to τ_{start} sets b_1^a and b_2^a to false and then swaps the variables n_1^a and n_2^a. Both n_1^a and n_2^a are associated to the notification port n_a, so their new values are sent back to the component.

3.3 Connecting Managers

The transformed atomic components and interaction managers are interconnected using three types of interactions: (i) *offer interactions*, where components send their enabled ports to corresponding managers, (ii) *notification interactions*, where managers notify components after execution of an interaction, and (iii) *schedule interactions*, where priority rules are handled.

We now formally define the deprioritized model, by specifying how we connect the components defined so far. Let $\gamma^{(i)}$ denote the set of all interactions in γ that involve component B_i.

Definition 7 (Deprioritized model) Given a model $B = \pi\gamma(B_1, \cdots, B_n)$, with $\gamma = \{a_1 \cdots a_m\}$, we define its deprioritized version as $\tilde{B} = \tilde{\gamma}(B_1^\perp, \cdots, B_n^\perp, M_{a_1}, \cdots, M_{a_m})$, where $B_i^\perp$ is obtained from B_i as explained in definition 5, M_{a_j} is obtained from a_j as explained in definition 6, and $\tilde{\gamma}$ contains the following interactions:

- **Offer interactions.** For each $i \in \{1 \cdots n\}$, $\tilde{\gamma}$ contains interaction off_i, where $P_{\mathit{off}_i} = \{o_i\} \cup \bigcup_{a \in \gamma^{(i)}} \{o_i^a\}$. For each interaction $a \in \gamma^{(i)}$, the update function F_{off_i} sets the values of variables $\{x_{p_i}^a\} \cup X_{p_i}^a$ to the values of $\{x_p\} \cup X_p$ associated to o_i, where p is the of port B_i involved in a. An offer interaction has no guard and only copies data from the sender component to the manager.
- **Notification interactions.** For each interaction $a \in \gamma$, where $a = \{p_i\}_{i \in I}$, $\tilde{\gamma}$ contains the interaction not_a, such that $P_{not_a} = n_a \cup \{p_i\}_{i \in I}$. This interaction notifies each component which port has been selected. The update function F_{not_a} copy back data to each component B_i involved in a. That is, the values of X_{p_i} (in B_i) are set to the values of $X_{p_i}^a$ (from M_a).
- **Schedule interactions.** For each interaction $a \in \gamma$, $\tilde{\gamma}$ contains the interaction $\tilde{a}$:

$$
\begin{aligned}
P_{\tilde{a}} &= \{start_a\} \\
&\cup \ \{\oplus_c | c \oplus a, \ c \not\succ a\} \\
&\cup \ \{dis_c | c \not\oplus a, \ a\pi c\} \\
&\cup \ \{\oplus dis_c | c \oplus a, \ a\pi c\}
\end{aligned}
$$

  This interaction has no guard. For each interaction c weakly conflicting with a, the update function $F_{\tilde{a}}$ sets variable b_i^c of the manager M_c to false through the port $\oplus_c$ (or $\oplus dis_c$) if $\{a, c\} \subseteq \gamma^{(i)}$. In other words, interaction $start_a$ informs manager M_c that the components causing the weak conflict with a have moved and are not in their stable state anymore. This information maintains coherence between the b_i^c variable in each manager M_c and the actual state of component B_i.

Example 5 Figure 7 presents the deprioritized model from Figure 4(b). Note that the port names have been shortened for space reasons (e.g., s_a and d_a stand for $start_a$ and dis_a respectively). For offer and notification interactions, we interpret a triangle port as a *send port* (i.e., for sending offers) and bullet port as a *receive port* (i.e., for receiving offers). Note that offers and notifications only copy variables between components and managers.

If we assume priorities $b\pi a$ and $c\pi a$ for the model in Figure 4(b), we obtain the following schedule interactions: a has no higher priority interaction and is weakly conflicting with b, thus $P_{\tilde{a}} = \{start_a, \oplus_b\}$. Executing $\tilde{a}$ will set variable b_2^b to false in M_b,

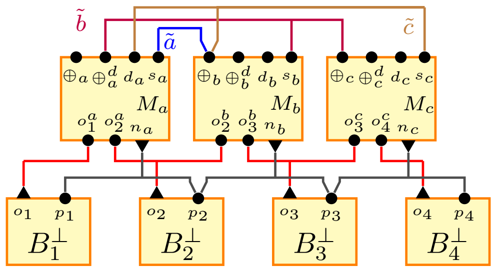


Fig. 7 Deprioritized version of model from Figure 4(b).

since B_2 will become busy. Interaction b has less priority than a and is weakly conflicting with both a and c, thus $P_{\tilde{b}} = \{start_b, \oplus dis_a, \oplus_c\}$. Interaction c has less priority than a and is weakly conflicting with b, thus $P_{\tilde{c}} = \{start_c, dis_a, \oplus_b\}$.

Reducing the number of added components. Since our transformation builds a manager for each interaction, the number of manager components in the deprioritized model is equal to the number of interactions in the input model. This number can be reduced by *merging* components with the method presented in [12]. Given a user-defined partition of components, merging consists in generating for each class of the partition a single component which is strongly bisimilar to their product. Merging components may improve performance of the obtained system by suppressing communication between the merged components. It may also deteriorate performance. For example, suppose that two components can execute in parallel. Merging these components, may result in degrading their performance to sequential execution in the product component. A guideline for merging components in the context of this article is provided at the end of Section 5.

3.4 Correctness

We now show that the 2-step transformation presented in Subsections 3.1-3.3 preserves the semantics of the original BIP model. By preserving the original semantics, we mean ensuring *observational equivalence* between the original model and the transformed model. This is proved in Theorem 1.

Let $B = \pi\gamma(B_1, \cdots, B_n)$ be a BIP model and $\tilde{B} = \tilde{\gamma}(B_1^\perp, \cdots, B_n^\perp, M_{a_1}, \cdots, M_{a_m})$ be its deprioritized version. We denote $q = (q_1, \cdots, q_n)$ a state of B and $\tilde{q} = (\tilde{q}_1, \cdots, \tilde{q}_n, s_1, \cdots, s_m)$ a state of $\tilde{B}$. We show that $\tilde{B}$ is observationally equivalent to B.

The observable actions of B are the interactions γ. The observable actions of $\tilde{B}$ are only the schedule interactions, that is $\{\tilde{a} | a \in \gamma\}$. The remaining interactions in $\tilde{B}$, namely offers off_i and notifications not_a, are unobservable and are denoted β. We denote $\tilde{q} \xrightarrow{\beta} \tilde{q}'$ if a β action brings the system from state $\tilde{q}$ to state $\tilde{q}'$.

Proposition 1 $\xrightarrow{\beta}$ *is terminating.*

Proof Each β action involves at least a component. Each component can take part in at most 2 β actions, 1 notification and 1 offer, then no other β action is possible until an $\tilde{a}$ action is executed. Thus at most $2n$ consecutive β-steps are possible. $\square$

Proposition 2 *From any reachable state $\tilde{q}$ of $\tilde{B}$, $\xrightarrow{\beta}$ is confluent.*

Proof In any reachable state, if a manager reaches the state *exc* then the corresponding notification interaction is enabled, since schedule interactions and boolean variables b_i ensure that each component may receive only one notification after each offer. Similarly, if any component reaches an unstable state, then the corresponding offer interaction is enabled.

Offer interactions are independent since they do not share any port nor change a common variable. Thus, the order of their execution does not change the final state.

Notification interactions (that correspond to interactions of the original model, augmented by a notification port) enabled from a reachable state are not conflicting since schedule interactions handle weak conflicts. Thus, notification interactions are independent and their order of execution does not change the final state. We can conclude that $\xrightarrow{\beta}$ is confluent. $\square$

From proposition 1 and 2, for each reachable state $\tilde{q}$ of $\tilde{B}$, there is a unique state denoted $[\tilde{q}]$ such that $\tilde{q} \xrightarrow{\beta^*} [\tilde{q}]$ and $[\tilde{q}] \stackrel{\beta}{\not\rightarrow}$.

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 [22], where β-transitions are considered unobservable. The same definition is trivially extended for atomic and composite BIP components.

Definition 8 (Weak Simulation) A *weak simulation* over A and B, denoted $A \subset B$, is a relation $R \subseteq Q_A \times Q_B$, such that we have $\forall (q, r) \in R,\ a \in P :\ q \xrightarrow{a}_A q' \implies \exists r' : (q', r') \in R \wedge r \xrightarrow{\beta^* a \beta^*}_B r'$ 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. We consider the correspondence between observable actions of B and $\tilde{B}$ as follows. To each interaction $a \in \gamma$, where γ is the set of interactions of B, we associate the schedule interaction $\tilde{a}$ of $\tilde{B}$.

Theorem 1 $B \sim \tilde{B}$.

Proof We define the relation R between the states of B and the states of $\tilde{B}$ as follows: the couple $(\tilde{q}, q)$ is in the relation R if the states of atomic components $B_1^\perp, \cdots, B_n^\perp$ in $[\tilde{q}]$ are the same as in q. Formally, we have $(\tilde{q}, q) \in R$ if $[\tilde{q}] = (q_1, \cdots, q_n, s_1, \cdots, s_m)$ and $q = (q_1, \cdots, q_n)$. We show that R is an observational equivalence by proving the next three assertions:

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

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

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

The point (i) comes from the definition of R.

(ii) If the interaction $\tilde{a}$ is enabled, then manager M_a is in state *en*, which implies that at equivalent state q:

- All ports of a are enabled and the guard G_a is true, since the guard of the τ_{en} transition is true

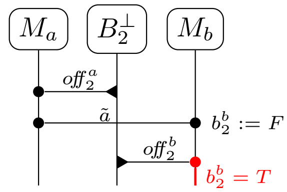


Fig. 8 A scenario leading to inconsistency between managers

– No higher priority interaction is enabled since $\tilde{a}$ is enabled only when managers corresponding to such interactions are in state *dis*.

Thus we have $q \xrightarrow{a} r$, and the reader can easily check that $(\tilde{r}, r) \in R$.

(iii) From $\tilde{q}$ we can reach $[\tilde{q}]$ by using only β transitions. In state $[\tilde{q}]$, since every atomic component has sent an offer, the state of each manager will be either *en* or *dis*, according to the status of the corresponding interaction at state q in B. Then since a is enabled at state q, M_a is in state *en* at state $[\tilde{q}]$. If there is any interaction b with higher priority than a, then it is disabled in state q, thus the manager M_b is in state *dis* at state $[\tilde{q}]$. Thus $\tilde{a}$ is enabled at state $[\tilde{q}]$ and we have $\tilde{q} \xrightarrow{\beta^* \tilde{a}} \tilde{r}$. Executing the notification interaction n_a and the offer interactions from components involved in a lead $\tilde{B}$ in a state where atomic components have the same state as in r. Thus $(\tilde{r}, r) \in R$. $\square$

3.5 Binary Versus n-ary Offers and Notifications

In a realistic distributed implementation, the offer and notification interactions may be implemented using a primitive ensuring synchronization of the receivers (e.g., atomic multicast). However, if such a primitive is not available, we have to refine our $\tilde{B}$ model, so that it resolves the issue in asynchronous networks as well. To this end, we can either use the method presented is section 4, or replace each notification and each offer by a set of binary interactions of the type {sender, receiver}. The latter option, which is studied in this subsection, requires to duplicate offer ports in atomic components and notification ports in managers, as well as the corresponding transitions, so that a previously n-ary offer (or notification) is transformed into a sequence of binary offers (or notifications). This transformation adds new transient states where offers (or notifications) have been sent to a part of the recipients, and remain to be sent for the others.

Notice that desynchronization of the notifications has no effect on the behaviour of each component since atomic components perform independent computation after receiving the notification. On the contrary, synchronization of offers' receivers ensures that the values of b_i variables are consistent among managers. This is, in fact, a crucial requirement to ensure correctness of our construction. As an example, consider the scenario presented in Figure 8. This scenario presents interactions between M_a, $B_2^{\perp}$, and M_b from the model in Figure 7, with desynchronized offers. Once $B_2^{\perp}$ has sent its offer to M_a, this latter one switches to enabled state, interaction $\tilde{a}$ becomes enabled, and is executed. Then, M_b receives the offer from $B_2^{\perp}$ and sets b_2^b to true, which is an inconsistency, as the offer from $B_2^{\perp}$ has already been consumed by a.

To prevent this inconsistency, we enforce synchronization of the offers by adding a guard to each schedule interaction. Let a be an interaction, we defined $\tilde{a} = (P_{\tilde{a}}, G_{\tilde{a}}, F_{\tilde{a}})$ and we define $\tilde{a}_{SR} = (P_{\tilde{a}}, G_{\tilde{a}}^{SR}, F_{\tilde{a}})$ as:

$$G_{\tilde{a}}^{SR} = \bigwedge_{c \oplus a} \bigwedge_{\{a,c\} \subset \gamma(B_i)} b_i^c$$

This guard checks that, for each interaction c weakly conflicting with a, and each component B_i that is involved in both a and c, the value of b_i^c is true. It means that the manager M_c has received the offer from B_i.

We denote $\tilde{B}_{SR}$ by the model built from $\tilde{B}$, where we replace each offer and each notification by a set of binary interactions, and, we replace each schedule interaction $\tilde{a}$ by the interaction $\tilde{a}_{SR}$.

Theorem 2 *Let B be a BIP model. $\tilde{B}_{SR}$ is observationally equivalent to $\tilde{B}$.*

Proof We define an equivalence between states of $\tilde{B}_{SR}$ and $\tilde{B}$ by giving the following application. From a state of $\tilde{B}_{SR}$, we build a state of $\tilde{B}$ by considering the transient states that are active in $\tilde{B}_{SR}$. If an atomic component of $\tilde{B}_{SR}$ is in a transient state introduced by the transformation of a n-ary offer into binary offers, we consider the offer as unsent. More precisely, in the built state of $\tilde{B}$, components involved in this offer are set to the state where the offer interaction is enabled. If a manager of $\tilde{B}_{SR}$ is in a transient state introduced by the transformation of a n-ary notification into binary notifications, we consider the notification as sent. More precisely, in the built state of $\tilde{B}$, components involved in this notification are set to the state reached after executing the notification interaction.

In order to have observational equivalence for this state equivalence, we map each n-ary offer to the last binary offer of the corresponding sequence and each n-ary notification to the first binary notification of the corresponding sequence. Other binary offers and notifications are unobservable. It is clear that any unobservable action in $\tilde{B}_{SR}$ does not change the equivalent state in $\tilde{B}$.

If a schedule interaction $\tilde{a}_{SR}$ is enabled in $\tilde{B}_{SR}$, the guard $G_{\tilde{a}}^{SR}$ ensures that all components involved in this interaction have executed all possible binary offers. That is, $\tilde{a}$ is possible in the equivalent state of $\tilde{B}$. Execution of the schedule interaction brings the two systems in equivalent states. By definition of equivalent states and visible interactions, if a visible offer or notification is enabled in $\tilde{B}_{SR}$, then the corresponding offer or notification is also enabled in $\tilde{B}$. Again execution of these interactions brings the two systems in equivalent states.

Finally, if a schedule interaction, or notification interaction is possible in $\tilde{B}$, then the definition of equivalent states ensures that these interactions are also possible in any equivalent state of $\tilde{B}_{SR}$. If an offer interaction is possible in $\tilde{B}$, it may not be directly possible in $\tilde{B}_{SR}$, but can be reached by executing (if needed) the unobservable notification to the component sending the offer and then the unobservable offers from that component. $\square$

4 Building a Distributed Model: The 3-Tier Architecture

Once we construct a model with no priorities as prescribed in Section 3, one can apply the technique presented in [8–10] to generate distributed code. We now briefly recap

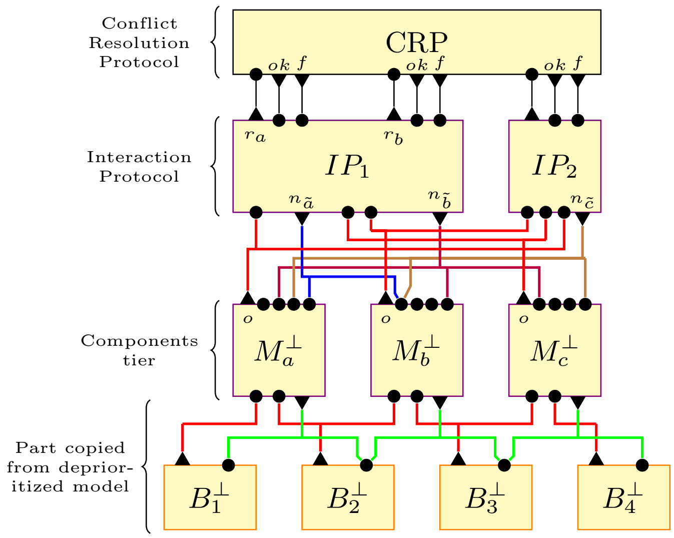


Fig. 9 Distributed version of the deprioritized model from Figure 7.

this technique. Distributed code generation is accomplished in two steps. First, from a given BIP model (e.g., a deprioritized BIP model), we generate another BIP model that only incorporates asynchronous message passing as interactions (denoted SR-BIP, where SR stands for send-receive). Then, we transform the SR-BIP model into a set of executables – one per atomic component – that communicate using asynchronous message passing primitives such as MPI or TCP sockets.

4.1 BIP to SR-BIP Transformation

Recall that distributed execution of interactions may introduce conflicts even if we do not consider priorities. Thus, our target SR-BIP model in a transformation should have the following three properties: (1) preserving the behaviour of each atomic component, (2) preserving the behaviour of interactions, and (3) resolving conflicts in a distributed manner. Moreover, we require that interactions in the target model are asynchronous message passing.

We design our target BIP model based on the three tasks identified above, where we incorporate one tier for each task. Since several distributed algorithms exist in the literature for conflict resolution, we design the tier corresponding to conflict resolution so that it provides appropriate interfaces with minimal restrictions. As a running example, we use the part of the model presented in Figure 7 formed by $\gamma_{sched}(M_a, M_b, M_c)$ where $\gamma_{sched} = \{\tilde{a}, \tilde{b}, \tilde{c}\}$ to describe the concepts of our transformation. The distributed version of $\gamma_{sched}(M_a, M_b, M_c)$ is presented in Figure 9. Our 3-tier architecture consists of the following.

Components Tier. Let $\tilde{B} = \tilde{\gamma}(B_1^\perp \cdots B_n^\perp, M_{a_1} \cdots M_{a_m})$ be a deprioritized BIP model. The component tier includes components:

- $M^{\perp}_{a_1} \cdots M^{\perp}_{a_m}$ (i.e., manager components obtained by the transformation explained in Subsection 3.1 to break atomicity), and
- $B^{\perp}_1 \cdots B^{\perp}_n$ are copied from the deprioritized model, since they have already been transformed by the deprioritization.

Recall that the send-port offer (o) publishes the list of enabled ports in the component for the upper tier (i.e., the interaction protocol). These ports are active ports, meaning that the holder component can send messages through them. Send ports are shown by triangles in Figure 9. Each port p of the original component becomes a receive-port p through which the component is notified to execute the transition labelled by p once the upper tiers resolve conflicts and decide on which components can execute on what port. Receive ports are passive, meaning that the holder component gets blocked on them until a message arrives. Receive ports are shown by bullets in Figure 9.

Interaction Protocol. This tier consists of a set of components each hosting a set of interactions from the deprioritized BIP model. Conflicts between interactions included in the same component are resolved by that component locally. For instance, interactions $\tilde{a}$ and $\tilde{b}$ in Figure 7 are grouped into component IP_1 in Figure 9. Thus, the conflict between $\tilde{a}$ and $\tilde{b}$ is handled locally in IP_1. On the contrary, the conflict between $\tilde{b}$ and $\tilde{c}$ has to be resolved using the third tier of our model (i.e., conflict resolution protocol). The interaction protocol also evaluates the guard of each interaction and executes the code associated with an interaction that is selected locally or by the upper tier. The interface between this tier and the component tier provides ports for receiving enabled ports from each component and notifying the components on permitted port for execution (ports $n_{\tilde{a}}$, $n_{\tilde{b}}$, and $n_{\tilde{c}}$).

Conflict Resolution Protocol. This tier accommodates an algorithm that solves the *committee coordination problem* [14] to resolve conflicts between interactions hosted in separate interaction protocol components. In this problem, a committee consists of a set of professors (i.e., components) and committee meetings (i.e., rendezvous synchronization) have to convene, so that each professor participates in at most one committee meeting at a time. Committee coordination is a classic problem of distributed mutual exclusion.

Consider the external conflict between interactions $\tilde{b}$ and $\tilde{c}$. This conflict is referred to the conflict resolution protocol and is resolved by the central component *CRP* in Figure 9. We emphasize that the structure of components in this tier solely depends upon the augmented committee coordination algorithm. Incorporating a centralized algorithm results in one component *CRP* as illustrated in Figure 9. Other algorithms, such as ones that use a circulating token [2] or dining philosophers [3, 14] result in different structures in this tier and are presented in [10]. The interface between this tier and the Interaction Protocol involves ports for receiving requests to *reserve* an interaction (labelled r) and responding by either success (labelled *ok*) or failure (labelled f). For the token ring and dining philosophers the conflict resolution protocol contains one component for each interaction. In the token ring version, a token circulates through all components. Only the component having the token may respond an *ok* message. In the dining philosophers version, if two interactions are conflicting, the corresponding components are neighbors and share a fork. Only the component having all the forks from his neighbors may respond an *ok* message.

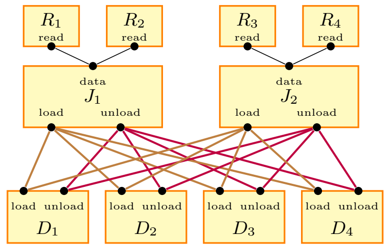


Fig. 10 BIP Model for the jukebox example.

4.2 Distributed Code Generation

Given an SR-BIP model, generating distributed code is a straightforward task. First, each component needs to initialize communication connections with respect to the platform communication primitives library. For instance, in case of TCP sockets, this step establishes connection-oriented stream sockets between components that need to send and receive messages to each other. Then the transition system representing the behavior of the component is used to generate execution loop code as follows. The code first tries to execute transitions labeled by send-ports. In this case, the generated code calls a communication primitive that sends a message, for instance, the TCP primitive send(). The recipient of this message is given by the Send/Receive interaction that is bound to the send-port. If no transition labeled by a send port or an internal port (such as ι in manager components) is enabled, the generated code calls a primitive that blocks and waits for messages from other components, for instance, the TCP primitive select(). Then, the code for transitions labeled by receive-ports is executed. This code calls a communication primitive for fetching the received data, such as the TCPprimitive receive().

It is possible to generate multi-threaded implementation as well. To this end, BIP uses shared memory, mutexes, and semaphores of POSIX library for implementing send, receive, and select primitives. More precisely, for each atomic component, we create a shared-memory FIFO buffer, a semaphore, and a mutex. The communication primitives used in the code are implemented as follows:

– The send primitive: The source component writes the message in the FIFO buffer of the destination component and increments the value of its semaphore. These actions are protected by a mutex of the destination component.
– The select primitive: The component waits on its semaphore.
– The receive primitive: The component reads the message from its buffer protected by its corresponding mutex.

5 Case Studies

5.1 Distributed Jukebox

In this section, we use a jukebox example to illustrate our deprioritization transformation and conduct experiments to study the effectiveness of our method (see the models in Figures 10 and 11). This model represents a system, where a set of readers $(R_1, \ldots, R_4)$ need to access the data located on discs $(D_1, \ldots, D_4)$. A reader may need

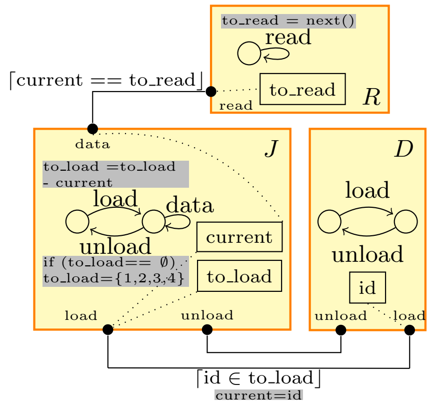


Fig. 11 Behaviour of jukebox components and interactions.

any disc. Access to the discs is managed by jukebox components (J_1, J_2) that can load any disc to make it available for reading. Each pair (D_i, J_k), $i \in \{1 \cdots 4\}$ and $k \in \{1, 2\}$, has two interactions: (1) a $load_{i,k}$ interaction for loading the disc in the jukebox and an $unload_{i,k}$ interaction for unloading it. Each reader R_j is connected to a jukebox through a $read_j$ interaction. During the test, we simulate execution of interactions by adding an artificial 10ms wait for *load*/*unload* and a 50ms wait for *read*.

Figure 11 presents the behaviour of atomic components and the data transfer on interactions. To ensure that all discs are eventually loaded, each jukebox keeps a list of discs to load, namely *to_load*. Each time a disc is loaded, it is removed from the list by the *load* transition in the jukebox component. The guard of a *load* interaction prevents the disc to be loaded if it is not on the list. When the *to_load* list becomes empty, it is reinitialized to the set of all discs by the *unload* interaction. The variable *current* contains the identity (i.e., $1 \dots 4$) of the disc currently loaded in the jukebox, and is updated by the *load* interaction. In order to ensure that the reader gets the correct data, a guard on the $\{read, data\}$ prevents the interaction if the disc in the jukebox (*current*) is not the one to be read (*to_read*). Each reader has a sequence of discs to read. The variable *to_read* contains the *id* of the next disc to be read. This value is updated after each read.

We consider two versions of the model. The first model, denoted $B_\emptyset$, does not contain priorities. The second model, denoted B_π, is the model $B_\emptyset$ augmented with two types of priorities:

– **Priorities to enforce progress.** For this particular model, we assume that progress is done whenever a *read* interaction takes place. In the $B_\emptyset$ model, an infinite sequence of *load*/*unload* interactions is a valid behavior. To avoid this, we give priority to the *read* interactions over the *unload* interactions. Formally, it corresponds to the sets of priorities $\{unload_{i,1} \; \pi \; read_j \mid i \in \{1, \cdots, 4\}, j \in \{1, 2\}\}$ and $\{unload_{i,2} \; \pi \; read_j \mid i \in \{1, \cdots, 4\}, j \in \{3, 4\}\}$, for each jukebox. This ensures that any enabled *read* interaction will be executed before the disc is unloaded.
– **Priorities to speed up execution.** By inspecting the discs requested by the readers, we know that some discs are more often needed than others. Thus, we give higher priority to the corresponding *load* interactions. Here, we give higher priority

to Disc 1 in Jukebox 1 by adding the following set of priorities: $\{load_{i,1}\ \pi\ load_{1,1} \mid i \in \{2,3,4\}\}$.

For both versions $B_\emptyset$ and B_π, we generate the corresponding deprioritized models $\tilde{B}_\emptyset$ and $\tilde{B}_\pi$. Note that we apply the deprioritization to $B_\emptyset$, even if it does not contain priority, in order to measure the overhead induced by this transformation. For each of the models $B_\emptyset$, $\tilde{B}_\emptyset$ and $\tilde{B}_\pi$, we generate a distributed implementation by using the transformation presented in Section 4 [10]. For the $B_\emptyset$ model, this transformation is applied to the entire model. For the $\tilde{B}_\emptyset$ and $\tilde{B}_\pi$ models, the transformation is applied only to the managers and schedule interactions. Note that an interaction from $B_\emptyset$ corresponds to a schedule interaction from $\tilde{B}_\emptyset$.

The distributed model obtained is parametrized by (1) a partition of the interaction and (2) a conflict resolution protocol. For the $B_\emptyset$ model, we consider a partition of the original interactions. For the deproritized models, we consider the partition obtained by taking the corresponding schedule interactions. We consider the following three different partitions to conduct our experiments:

– $P1$ is the partition obtained by grouping all interactions. It results in a distributed model with only one interaction protocol component.
– $P2$ is the partition obtained by grouping all interactions that involve a given scheduler. Thus, we have two interaction protocol components: IP_1 and IP_2. IP_1 is responsible for the interactions (or the schedule interactions corresponding to) $\{read_i\}_{i=1,2}$ and $\{load_{i,1}, unload_{i,1}\}_{i=1..4}$. IP_2 is responsible for the remaining interactions. Since there are conflicts between interactions that are not in the same interaction protocol components (e.g., between $load_{1,1}$ and $load_{1,2}$ that both need disc 1), this partition needs external conflict resolution.
– $P3$ is the partition obtained by building a separate interaction protocol component for each interaction. Again, this partition needs external conflict resolution.

For partitions $P2$ and $P3$, we use the three conflict resolution protocols presented in [10], namely centralized (CT), token ring (TR), and dining philosopher (DP). For each setting, we generate a distributed version of $B_\emptyset$, $\tilde{B}_\emptyset$, and $\tilde{B}_\pi$. For each version, we count the number of *read* interactions that occur during 60s. The results are presented in Figure 12.

Overhead. We evaluate the overhead of deprioritization by comparing the number of *read* interactions for distributed executions of models $B_\emptyset$ and $\tilde{B}_\emptyset$. In the distributed version of $\tilde{B}_\emptyset$, interaction code is executed by managers, thus executing one interaction does not block the others. In the distributed version of $B_\emptyset$, interaction code is executed within interaction protocol components. This restricts parallelism since executing one interaction blocks all other interactions handled by the same interaction protocol component. Since the computation load (i.e., 10ms for *load* and 50ms for *read*) is placed in the interaction code, it is fair to compare performance of $B_\emptyset$ and $\tilde{B}_\emptyset$ only for partition $P3$, where in both cases each component handles at most one interaction. When using the centralized conflict resolution protocol (CT), the deprioritized model does more *read* interactions. However, with the token ring (TR) and dining philosophers (DP), we can see that our transformation incurs a great overhead. This can be explained by the number of concurrent processes: the execution of $\tilde{B}_\emptyset$ needs 20 more processes (i.e., the managers) than execution of $B_\emptyset$. For the setting $P3$/TR and $P3$/DP, the number

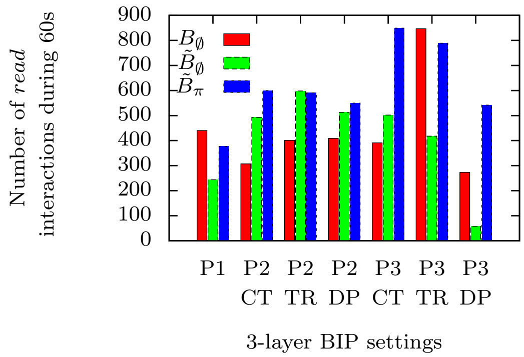


Fig. 12 Performance of different implementations of the Jukebox model.

of processes is 70 for $\tilde{B}_\emptyset$ against 50 for $B_\emptyset$, thus we may have some performance limitation due to the platform. Also, the additional tier made of the managers adds some overhead.

Performance improvement. The distributed execution of $\tilde{B}_\pi$ outperforms execution of $B_\emptyset$ except for partition $P1$ and partition $P3$ with the token ring conflict resolution protocol (TR). Note that there are more conflicts between schedule interactions than between interactions of the original model because (1) priorities add conflicts between schedule interactions, and (2) two interactions that are weakly conflicting but not strongly conflicting (as defined in [10]) from the original model will result in two strongly conflicting schedule interactions. For the $P3$/TR setting, where distributed execution of $B_\emptyset$ is the more efficient, this additional conflicts may explain the worse performance of $\tilde{B}_\pi$. The other factor is the number of processes, as explained above. In other cases, the priorities that we added to enforce progress and speed up execution are useful. In particular, for the settings $P3$/CT and $P3$/DP, twice more *read* interactions are performed.

5.2 Distributed Dining Philosophers

Our second example is based on the dining philosophers problem. A fragment of the model is presented in Figure 13. The model contains N philosopher components and N fork components. Each philosopher P_i first grabs the left fork through interaction $\mathit{grableft}_i$, then it grabs the right fork through interaction $\mathit{grabright}_i$. Whenever it has the two forks, it can eat through interaction eat_i.

Note that this model contains one deadlock state, that is reached from the initial state when each philosopher grabs the fork to its left. One way to resolve this deadlock is by adding the following priorities: $\mathit{grableft}_i \,\pi\, \mathit{grabright}_{i-1}$, for all i in $\{1, \ldots, N\}$. This set of priorities ensures that each philosopher can take the fork to its left only if the philosopher on its left has not taken its left fork yet. This type of priorities can be obtained automatically [15].

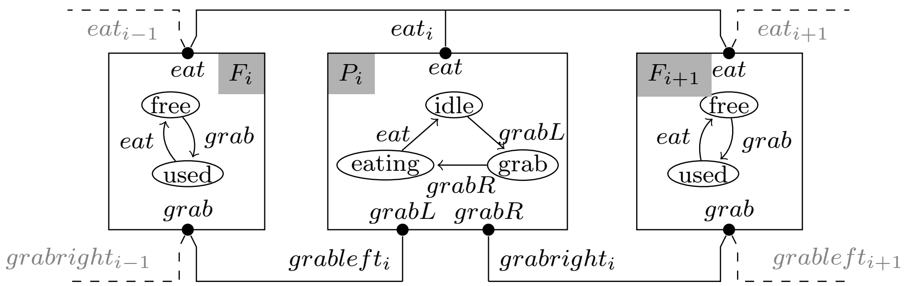


Fig. 13 A fragment of the dining philosopher example.

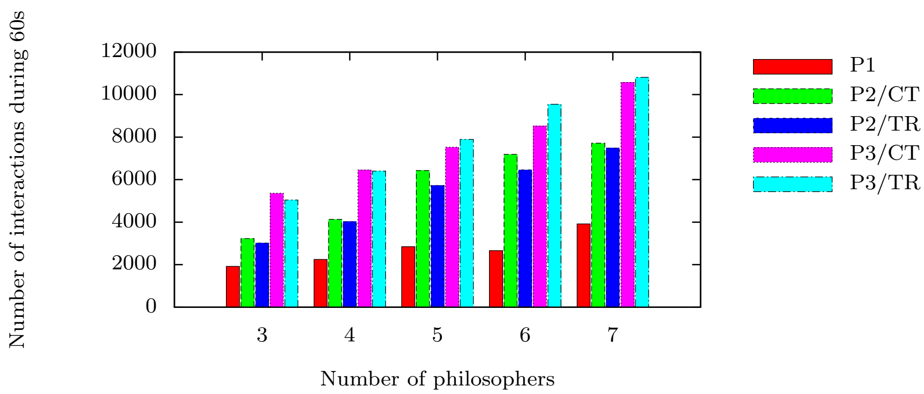


Fig. 14 Total number of interactions executed for different distributed implementations of the dining philosopher example.

Distributed execution of this model without priorities is not possible since it leads to a deadlock. Hence, we used the deprioritized model to generate a distributed implementations for the model, for different settings of the 3-tier transformation. We used the three following partitions for the schedule interactions:

- $P1$ contains all schedule interactions from the model.
- $P2$ contains one class for each philosopher. For philosopher P_i this class contains the schedule interactions $\{\tilde{eat}_i, gra\tilde{bl}eft_i, grab\tilde{r}ight_i\}$.
- $P3$ contains one class for each schedule interaction.

We propose the following experiments: for each model containing from 3 to 7 philosophers, we generate the corresponding distributed model. Then, for each of these models, we compare the performance obtained with each of the partitions presented above. For partitions $P2$ and $P3$, we used both centralized (CT) and token ring (TR) conflict resolution protocols. The total number of interactions executed during 60s of execution is presented in Figure 14.

First, increasing the number of philosophers increases the parallelism, and thus increases the global number of interactions. Moreover, notice that more decentralized partitions give better performance. Indeed, for each size, the worst performance is obtained for partition $P1$ and the best performance is obtained for partition $P3$. For partition $P2$, the centralized conflict resolution protocol gives better results than the token ring protocol, because there are few requests and no need to receive them in parallel. For partition $P3$, it is not clear which conflict resolution protocol gives better results.

More decentralized implementations give rise to many components. In the most decentralized case, for each interaction we have 3 components: 1 priority manager, 1 interaction protocol component and 1 conflict resolution protocol. However, we can reduce the number of components by *merging* this three components for each interaction into a single one, as explained at the end of Section 3.3 and in [12]. As a more general guideline, for each class of the partition, we might merge all managers of interactions in this class with the corresponding interaction protocol component.

6 Related Work

In this section, we report on the work related to automated code generation for distributed systems from high-level models. To the best of our knowledge, this paper presents the first work on automated implementation of distributed application, where synchronization primitives are regulated by dynamic priority rules. We first discuss solutions to the *committee coordination problem* in Subsection 6.1. Then, we present frameworks for automatic generation of distributed code in Subsection 6.2.

6.1 Algorithms for Solving the Committee Coordination Problem

As mentioned in the introduction, resolving distributed conflicts in the context of our framework leads us to solving the *committee coordination problem* [14], where a set of professors organize themselves in different committees. Two committees that have a professor in common cannot meet simultaneously. The original distributed solution to the committee coordination problem assigns one *manager* to each interaction [14]. Conflicts between interactions are resolved by reducing the problem to the dining or drinking philosophers problems [13], where each manager is mapped onto a philosopher. Bagrodia [2] proposes an algorithm where message counts are used to solve synchronization and exclusion is ensured by a circulating token. In a follow-up paper [3], Bagrodia modifies the solution in [2] by using message counts to ensure synchronization and reducing the conflict resolution problem to dining or drinking philosophers. Also, Perez et al [25] and Parrow et al [24] propose another approach that essentially implements the same idea using a lock-based synchronization mechanism.

In [20], Kumar proposes an algorithm that replaces the managers by tokens, one for each interaction, traversing the processes. An interaction executes whenever the corresponding token manages to traverse all involved processes. Another solution without managers has been provided in [18] based on a randomized algorithm. The idea is that each process randomly chooses an enabled interaction and sends its choice to all other processes. If some process detects that all participants in an interaction have chosen it, then the interaction is executed. Otherwise, this procedure gets restarted.

In [11], the authors propose *snap-stabilizing* committee coordination algorithms. Snap-stabilization is a versatile technique allowing to design algorithms that efficiently tolerate transient faults. The authors show that it is impossible to implement an algorithm that respects both fairness and maximal concurrency amongst meetings or professors. Consequently, they propose two snap-stabilizing algorithms that respect either fairness or maximal concurrency.

6.2 Automated Generation of Distributed Code

Our previous work in [8–10] focuses on model-based implementation of distributed application from BIP models without priorities. This paper complements the techniques presented in [8–10], that is, our method is an independent transformation for deprioritizing BIP models. The output model can then be used to generate distributed code that embodies the priorities while preserving the functional properties of the original model. Priorities for component-based models can be synthesized from a high-level specification using the methods in [15].

LOTOS [17] is a specification language based on process algebra, that encompasses multiparty interactions. In [28], 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. This approach is not suitable for BIP where there is no 'main' interaction. Also, the idea of a parent to be responsible for ensuring synchronization makes solutions more centralized than distributed with greater communication overhead.

Another framework that offers automatic distributed code generation is described in [27]. The input model consists of composition of I/O automata [21], 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 are less expressive than BIP interactions, as described in [7]. Indeed, to express an n-ary rendezvous between n I/O automata, one would need to add an automaton in charge of synchronization, which is not needed in BIP. Another difference with our work is that the designer has to provide some function that resolves nondeterminism. Finally, the framework in [27] requires the designer to specify low-level elements of a distributed system such as channels and schedulers.

Finally, [26] provides a distributed implementation method for the Reo framework [1]. In this framework, components are black boxes with input and output ports synchronized by data-flow connectors. A distributed implementation is obtained by deploying connectors, according to a user-defined partition, on a set of generic engines, implemented in Scala. At execution, a consensus algorithm between all the engines chooses a subset of connectors to be executed, based on the set of currently enabled ports. Unlike BIP interactions, data-flow connectors in Reo do not provide support for guard conditions and arbitrary data transfer functions. Moreover, the consensus algorithm used enforces a global agreement between all engines, whereas in the 3-layer BIP decisions are taken independently by each engine, or by communication with the Reservation Protocol.

7 Conclusion

In this paper, we proposed an automated method to derive correct distributed implementation from high-level component-based models encompassing prioritized multiparty interactions. Our method consists of three steps: (1) one transformation to *deprioritize* the initial model, which is the main contribution of this paper, (2) a transformation from [8–10] that generates a distributed model from the deprioritized model by *resolving interaction conflicts*, and (3) a final transformation from the distributed model into C++ code. All steps preserve observational equivalence between the input and output models. We illustrated our approach using non-trivial case studies and presented encouraging experimental results.

The transformation from [8–10] is parameterized by a partition of the interactions and a conflict resolution protocol. These parameters have a strong influence on the performance of the obtained implementation since partitioning preconditions parallelism between interactions and volume of communication. The target platform (number and speed of processors, network characteristics, etc) is a crucial parameter for building the partition. In this paper, we conducted our experiments on a multi-core distributed platform, and experiments gave mixed results. Automatically computing a partition while ensuring optimal performance for a given platform remains an open problem.

There exist several research directions for future work. First, more rigorous and deeper case studies and experiments are needed to completely understand the overheads introduced by our transformations. Since deprioritization is an independent step of our method and is isolated from conflict resolution (i.e., step two), one can study the overhead of each step separately. Another direction is to devise a committee coordination algorithm for conflict resolution that takes priority issues into account. This allows us to incorporate such an algorithm directly in our 3-tier model [10]. This approach can potentially have less overhead than the one presented in this paper. Finally, one can speed-up distributed execution of models with priorities by detecting disabled interactions as early as possible. Such detection can benefit from knowledge-based methods (e.g., [4]). Another interesting line of work is model-based development of distributed applications that are subject to timing constraints.

References

1. F. Arbab. Reo: a channel-based coordination model for component composition. *Mathematical Structures in Computer Science*, 14(3):329–366, 2004.
2. R. Bagrodia. A distributed algorithm to implement n-party rendevouz. In *Foundations of Software Technology and Theoretical Computer Science, Seventh Conference (FSTTCS)*, pages 138–152, 1987.
3. R. Bagrodia. Process synchronization: Design and performance evaluation of distributed algorithms. *IEEE Transactions on Software Engineering (TSE)*, 15(9):1053–1065, 1989.
4. A. Basu, S. Bensalem, D. Peled, and J. Sifakis. Priority scheduling of distributed systems based on model checking. In *Computer Aided Verification (CAV)*, pages 79–93, 2009.
5. A. Basu, P. Bidinger, M. Bozga, and J. Sifakis. Distributed semantics and implementation for systems with interaction and priority. In *Formal Techniques for Networked and Distributed Systems (FORTE)*, pages 116–133, 2008.
6. 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.
7. S. Bliudze and J. Sifakis. A notion of glue expressiveness for component-based systems. In *Concurrency Theory (CONCUR)*, pages 508–522, 2008.
8. B. Bonakdarpour, M. Bozga, M. Jaber, J. Quilbeuf, and J. Sifakis. Automated conflict-free distributed implementation of component-based models. In *IEEE Symposium on Industrial Embedded Systems (SIES)*, pages 108 – 117, 2010.
9. B. Bonakdarpour, M. Bozga, M. Jaber, J. Quilbeuf, and J. Sifakis. From high-level component-based models to distributed implementations. In *ACM International Conference on Embedded Software (EMSOFT)*, pages 209–218, 2010.

10. B. Bonakdarpour, M. Bozga, M. Jaber, J. Quilbeuf, and J. Sifakis. A framework for automated distributed implementation of component-based models. *Distributed Computing*, 2012. To appear.
11. B. Bonakdarpour, S. Devismes, and F. Petit. Snap-stabilizing committee coordination. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 231–242, 2011.
12. M. Bozga, M. Jaber, and J. Sifakis. Source-to-source architecture transformation for performance optimization in bip. *Industrial Informatics, IEEE Transactions on*, 6(4):708 –718, nov. 2010.
13. K. M. Chandy and J. Misra. The drinking philosophers problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 6(4):632–646, 1984.
14. K. M. Chandy and J. Misra. *Parallel program design: a foundation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1988.
15. C.-H. Cheng, S. Bensalem, Y.-F. Chen, R. Yan, B. Jobstmann, H. Ruess, C. Buckl, and A. Knoll. Algorithms for synthesizing priorities in component-based systems. In *Automated Technology for Verification and Analysis (ATVA)*, pages 150–167, 2011.
16. G. Gössler and J. Sifakis. Composition for component-based modeling. *Science of Computer Programming*, 55(1-3):161–183, 2005.
17. ISO/IEC. *Information Processing Systems – Open Systems Interconnection: LOTOS, A Formal Description Technique Based on the Temporal Ordering of Observational Behavior*, 1989.
18. Y.-J. Joung and S. A. Smolka. Strong interaction fairness via randomization. *IEEE Transactions on Parallel and Distributed Systems*, 9(2):137–149, 1998.
19. M. Jurdzinski. Small progress measures for solving parity games. In *Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 290–301, 2000.
20. D. Kumar. An implementation of n-party synchronization using tokens. In *IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 320–327, 1990.
21. N. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, San Mateo, CA, 1996.
22. R. Milner. *Communication and concurrency*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, 1995.
23. N. Mittal and P. K. Mohan. A priority-based distributed group mutual exclusion algorithm when group access is non-uniform. *Journal of Parallel Distributed Computing*, 67(7):797–815, 2007.
24. J. Parrow and P. Sjödin. Multiway synchronizaton verified with coupled simulation. In *International Conference on Concurrency Theory (CONCUR)*, pages 518–533, 1992.
25. J. A. Pérez, R. Corchuelo, and M. Toro. An order-based algorithm for implementing multiparty synchronization. *Concurrency and Computation: Practice and Experience*, 16(12):1173–1206, 2004.
26. J. Proença. *Synchronous Coordination of Distributed Components*. PhD thesis, Faculteit der Wiskunde en Natuurwetenschappen, May 2011.
27. 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.
28. G. von Bochmann, Q. Gao, and C. Wu. On the distributed implementation of LOTOS. In *Formal Techniques for Networked and Distributed Systems (FORTE)*, pages 133–146, 1989.