# Automated Distributed Implementation of Component-based Models with Priorities

*Borzoo Bonakdarpour, Marius Bozga, Jean Quilbeuf*

**Verimag Research Report n$^o$ TR-2011-3**

February 2011

# Automated Distributed Implementation of Component-based Models with Priorities

*Borzoo Bonakdarpour, Marius Bozga, Jean Quilbeuf*

February 2011

## Abstract

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 system due to their inherent complexity. Our three-phase method's input is a model specified in terms of a set of behavioural components that interact through a set of high-level synchronization primitives (e.g., rendezvous and broadcasts) and priority rules for scheduling purposes. Our technique, first, transforms the input model into a model that has no priorities. Then, it transforms the deprioritized model into another model that resolves distributed conflicts by incorporating a solution to the committee coordination problem. Finally, it generates distributed code using asynchronous point-to-point send/receive primitives. All transformations preserve the properties of their input model by ensuring observational equivalence. The transformations are implemented and our experiments validate their effectiveness.

**Keywords:** Correct-by-construction; Distributed implementation; Transformation; Priorities; BIP

**Reviewers:**

**How to cite this report:**

```
@techreport {TR-2011-3,
   title = {Automated Distributed Implementation of Component-based
Models with Priorities},
   author = {Borzoo Bonakdarpour, Marius Bozga, Jean Quilbeuf},
   institution = {{Verimag} Research Report},
   number = {TR-2011-3},
   year = { }
}
```

## 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 due to a number of factors such as non-determinism, non-atomic execution of processes, race conditions, and occurrence of faults. Model-based development of such applications aims to increase the integrity of these applications through the usage of explicit models employed in clearly defined transformation steps leading to correct-by-construction artifacts. This approach is 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 an actual executable code is obtained.

In this paper, we focus on the BIP framework [5] as our formal modelling language. BIP (Behaviour, Interaction, Priority) is based on a semantic model encompassing composition of heterogeneous components. The *behaviour* of components 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[1]. 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.
3. **Enforcing priorities.** When two interactions can execute simultaneously, the one with higher priority must be executed.

These issues introduce challenging problems in a distributed setting. The conflict resolution issue can be addressed by incorporating solutions to the *committee coordination problem* [9] for implementing multiparty interactions. For example, Bagrodia [1] proposes different solutions with different degrees of parallelism. The most distributed solution is based on the drinking philosophers problem [8], and has inspired the approaches of Pérez et al. [14] and Parrow et al. [13]. In the context of BIP, a transformation addressing all the three challenges through employing *centralized scheduler* is proposed in [4]. Moreover, in [6, 7], we propose transformations that address the concurrency issue by breaking the atomicity of interactions and conflict resolution by embedding a solution to the committee coordination problem in a distributed fashion. On the contrary, designing transformations that enforce priorities between interactions in a distributed setting remains unaddressed in spite of the vital role specifying priorities plays in designing systems.

In Subsection 1.1, we discuss the importance of incorporating priorities as a scheduling tool to solve a wide range of 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, such bad things are often due to existence of a set of processes competing over a resource. Priorities can be used to resolve such 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.

[1]Although our focus is on BIP, 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.

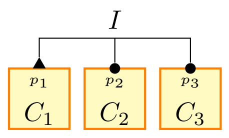


Figure 1: A component-based model with broadcast interaction.

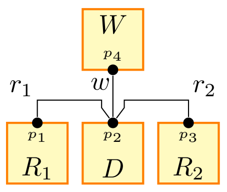


Figure 2: A simple BIP model for multiple readers/single writer problem.

- **Improving performance.** In distributed systems, it is often the case that certain resources have higher demands. For example, in *group mutual exclusion* [10], as Mittal and Mohan argue [12], 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 resources 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 behaviours. In many scenarios, it is desirable to guide the system to behave in a certain way. 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 if their corresponding port is 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 ensuring priorities in a distributed setting is their correct implementation. This is due to the fact that components need to obtain a common knowledge about enabledness of interactions, so the interaction with highest priority is executed. In [3], 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., [12]).

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 [6, 7], one can automatically generate a distributed implementation that is observationally equivalent to the high-level model. However, the solutions in [6, 7] 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.

This example clearly shows that it is highly desirable for designers and developers of distributed systems to have access to methods that automatically construct a correct distributed implementation from a high-level model along with a set of priority rules, such as the one in Figure 2. This way, all implementation issues are dealt with by transformation algorithms and designers need to make minimal effort to develop models.

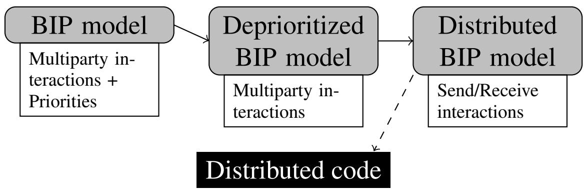


Figure 3: Steps for generating a distributed implementation from a high-level BIP model.

## 1.2 Contributions

With this motivation, our contributions in this paper are as follows:

- We propose a transformation that, given a high-level BIP model with priorities, generates a BIP model without priorities, that behaves equivalently. 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 transformation introduced in [7] 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 [6, 7], 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 *jukebox* application in BIP and conducting experiments on the generated distributed code. The jukebox application incorporates priorities to manage demands on reading discs and our experiments show that the overhead of our transformations has minimal effect on the benefit of using priorities.

**Organization.** The rest of the paper is organized as follows. In Section 2, we present the basic semantics model of BIP. Section 3 is dedicated to formalize our transformation problem. Then, in Section 4, 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 5. We discuss our case study and experimental results in Section 6. Finally, we conclude in Section 7. All proofs are presented in the appendix.

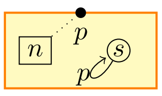
# 2 Basic Semantic Models of BIP

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

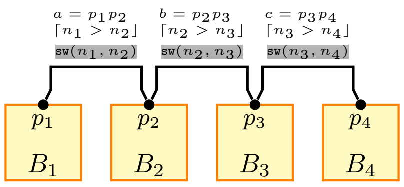
**Atomic Components.** We define *atomic components* as transition systems extended with a set of ports and a set of variables. Each transition is guarded by a predicate on the variables, triggers an update function, and is labelled by a port. The ports are used for communication among different components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



(a) An atomic component



(b) A composite component (BIP model)

Figure 4: Atomic and composite components in BIP

- *$T$ 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 $\tau$ and $f$ is the* update function *of $\tau$. $g$ 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 $\mathbf{X}$ the set of valuations on $X$, and $Q \times \mathbf{X}$ the set of local states. 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 $(q, v) \rightarrow (q', v')$. Similarly, $(q, v) \xrightarrow{p}$ means that there exists a transition $\tau = (q, p, g, f, q')$ such that $g(v)$ is true; i.e., $p$ is *enabled* in state $(q, v)$.

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 $\{B_i = (Q_i, P_i, X_i, T_i)\}_{i=1}^n$, 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 $\gamma$ 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 $\gamma$ and a priority partial order $\pi \subseteq \gamma \times \gamma$. We denote $B \overset{def}{=} \pi\gamma(B_1, \ldots, B_n)$ the component obtained by composing components $B_1, \cdots, B_n$ using the interactions $\gamma$ and the priorities $\pi$.*

Note that if the system does not contain any priority, we may omit $\pi$.

**Definition 3** (Composite Component Semantics)**.** *The behaviour of a composite component without priority $\gamma(B_1, \cdots, B_n)$ is a transition system $(Q, \gamma, X, \rightarrow_\gamma)$, where $Q = \bigotimes_{i=1}^n Q_i$, $X = \bigcup_{i=1}^n X_i$ and $\rightarrow_\gamma$ is the least set of transitions satisfying the rule:*

$$\frac{a = (\{p_i\}_{i \in I}, G_a, F_a) \in \gamma \\ G_a(v_1, \ldots, v_n) \qquad \forall i \notin I.\ (q_i, v_i) = (q'_i, v'_i) \qquad \forall i \in I.\ (q_i, v_i) \xrightarrow{p_i(v''_{p_i})}_i (q'_i, v'_i), v''_{p_i} = F_a^i(v_1, \ldots, v_n)}{((q_1, v_1), \ldots, (q_n, v_n)) \xrightarrow{a}_\gamma ((q'_1, v'_1), \ldots, (q'_n, v'_n))}$$

*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:*

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

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. Execution of the interaction modifies components' variables by first applying update function $F_a$ to associated variables and then function $f_i$ in each component. The states of components that do not participate in the interaction stay unchanged.

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 $\gamma$ of interactions is $\{a, b, c\}$, where $a = (\{p_1, p_2\}, n_1 > n_2, \mathtt{sw}(n_1, n_2))$ and function 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.

In the component presented in Figure 4(b), 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 $\pi\gamma(B_1, \ldots, B_4)$.

## 3 The Problem of Enforcing Priorities in Distributed BIP Models

The key characteristic of a distributed system is that its components run concurrently. Since semantics of BIP models require atomic execution of interactions and transitions, one has to break this atomicity in order to obtain a concurrent model. In this section, we describe the challenge of enforcing priorities in BIP models, where transitions of atomic and composite components are not executed atomically. In Subsection 3.1, we recall our method from [6, 7] for transforming a component into one whose transitions are non-atomic. Then, in Subsection 3.2, we describe why breaking the atomicity of transitions makes it difficult to enforce priorities.

### 3.1 Components with Non-atomic Transitions

In order to break the atomicity of transitions, the method in [7] splits each transition into two consecutive steps: (1) an offer that publishes the current state of the component, and (2) a notification that triggers the update function. The intuition behind this transformation is that the offer transition correspond to sending information about component's intention to interact to some scheduler and the notification transition correspond to receiving the answer from the scheduler, once some interaction has been completed. Update functions can be then executed concurrently and independently by components upon notification reception.

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

**Definition 4** (Transformed atomic component)**.** *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 the 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* $T^\perp$*:*

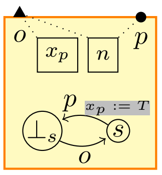


Figure 5: Transformed version of the atomic component from Figure 4(a)

1. offer $\tau_o^q = (\perp_q, o, g_o, f_o, q)$ *where* $g_o$ *is true,* $f_o$ *is the identity function, and*
2. notification $\tau_p^q = (q, p, g_p, f_p, \perp_{q'})$ *where* $g_p$ *is true and* $f_p$ *applies* $f_\tau$ *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.*

Figure 5 shows the transformed version of the atomic component presented in Figure 4(a). Initially, the component is in control 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 state $s$. In that state, it waits for a notification on port $p$, which updates the value of $n$. The notification also triggers the update function which consists here in only setting $x_p$ to true, since no guard and no update function were present in the high-level model.

In Definition 4, states $\{\perp_s \,|s \in Q\}$ 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*.

## 3.2 Priorities in Non-atomic Models

In [4], the authors show that in general, if a BIP model has priorities, its transformation to the distributed setting, as prescribed in Subsection 3.1, does not behave in the same way as the original model. This is due to the fact that breaking the atomicity of transitions introduces unstable states to components. In such states components have not yet sent their offers and, hence, some enabledness and disabledness of interactions become uncertain. An interaction can be wrongly detected maximal and executed, because higher priority interactions are (still) waiting for some (slow) components. Thus, applying priority rules at partially busy states results in producing behaviours that are not allowed in the initial BIP model. In [4], the authors propose a solution for the case where the scheduler is centralized.

In order to handle priorities in a decentralized fashion, we introduce the notion of *conflicting interactions*. Intuitively, two interactions $a_1$ and $a_2$ are *weakly conflicting* iff they share a common component.

**Definition 5** (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* $q \in P_{a_2}$*.*

If two interactions are weakly conflicting, then executing one of them can change the status of the other one, for instance, from enabled to disabled. Weak conflicts play an important role in maintaining a coherent view on the status of interactions and consequently in handling priorities efficiently. For example, knowing that an interaction is disabled allows to execute a lower priority interaction.

Nevertheless, for distributed implementation of BIP models without priorities as discussed in [7] one has to consider also *strong conflicts*. Two interactions are strongly conflicting if they synchronize on the same transition or two internally conflicting transitions (starting at the same state) of some component. A correct implementation, that is, one that conforms to the operational semantics of the BIP, requires that amongst two strongly conflicting interactions enabled simultaneously at most one can execute.

**Definition 6** (Strong Conflict)**.** *Two interactions* $a_1$ *and* $a_2$ *are said to be* strongly conflicting *iff one of the two following conditions holds:*

- *There exists a port* $p$ *in some component* $B$*, such that* $p \in P_{a_1} \cap P_{a_2}$*.*
- *There exist two ports* $p$ *and* $q$ *in some component* $B$*, and a state* $s$ *of* $B$*, such that* $p \in P_{a_1}$*,* $q \in P_{a_2}$*,* $s \xrightarrow{p}$ *and* $s \xrightarrow{q}$*.*

Clearly, strong conflict implies weak conflict. Weak conflicts are used for propagating changes of status between interactions. Strong conflicts are used to characterize the cases where a mechanism to ensure consistent execution (i.e., conflict resolution) is needed.

## 4 Deprioritizing a BIP Model

In this section, we describe our approach to transform a BIP model $B$ into an equivalent model without priorities, denoted $\tilde{B}$. In our construction, we use multiparty interactions to coordinate and synchronize our components. In Section 5, we explain how we replace multiparty interactions to obtain a model that uses only asynchronous message passing as interactions.

Our transformation is as follows. First, we transform atomic components as prescribed in Subsection 3.1. Then, we build a *manager* component for each interaction, which detects enabledness of the interaction, communicates with other managers to check priority rules, and, if allowed, executes the interaction. We show that our transformation preserves the semantics of the initial BIP model. By preserving the original semantics, we mean *observational equivalence* between the original model and the transformed model. This is addressed as correctness of our transformation in Appendix 4.2.

### 4.1 Building and Connecting Interaction Managers

Given a BIP model $B = \gamma(B_1 \cdots B_n)$, for each interaction $a \in \gamma$, we build a manager component $M_a$. This manager component:

- detects enabledness of $a$ by listening to the offers sent by atomic components involved in $a$,
- executes the update function of $a$,
- notifies atomic components involved in $a$, if the interaction has been selected.

The set of managers must schedule interactions according to the global semantics of the original BIP model described in Section 2. To this end, each manager component maintains the following control states: *undefined*, *enabled*, *disabled*, and *executing*, for the interaction it represents. Interactions between managers enforce priority rules and update the state of managers.

In this context, note that if two interactions are weakly conflicting, executing one can change the state of the other. For instance, let $a$ and $b$ be two interactions, such that $a \oplus b$, because they share component $B$. Obviously, executing $a$ implies a move in component $B$. This move results in changing the state of interaction $b$ to *undefined*, because until component $B$ completes its local execution and sends an offer, it is unknown which ports and, hence, interactions will be enabled.

With this intuition, we now formalize our deprioritization method. In the sequel, let $B = \pi\gamma(B_1, \cdots, B_n)$ be a BIP model and $a \in \gamma$ be an interaction, where $P_a = \{p_i\}_{i \in I}$. We define the manager $M_a = (Q, X, P, T)$ as follows. Our construction is generic in that the only given parameter is an interaction $a = (P_a, F_a, G_a)$, which indicates the set of ports, associated variables, guard, and update function (in Figure 6, we construct the manager component that handles interaction $a$ in Figure 4(b) as a running example):

#### Control States and Variables

- We let $Q = \{undef, en, dis, exc\}$. Intuitively, in state $undef$, 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 $en$ and $dis$, we know for sure that $a$ is enabled or disabled, respectively. In state $exc$, the interaction $a$ is executing.
- For each component $B_i$ involved in $a$, (i.e., $i \in I$), $X$ contains a Boolean variable $b_i^a$. This variable is true when component $B_i$ is in a stable state, that is, waiting for a notification. For instance, in Figure 6, we have the variables $b_1^a$ and $b_2^a$ since interaction $a$ involves components $B_1$ and $B_2$.

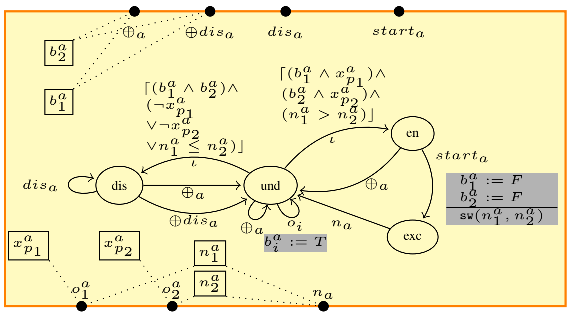


Figure 6: A manager for an interaction between $B_1$ and $B_2$.

**Ports and Transitions**

We describe our construction with respect to ports and transitions based on the functional behaviour of the manager component:

- **Receiving offers.** For each port $p_i$ that participates in interaction $a$, we include a port $o_i^a$ in $P$. Moreover, $X$ contains a Boolean variable $x_{p_i}^a$ and the set of variables $X_{p_i}^a$ (i.e., a local copy of the variables $X_{p_i}$ associated to $p_i$ in component $B_i$). The set $T$ of transitions includes a transition $\tau_i = (undef, o_i^a, true, f_{\tau_i}, undef)$. This transition takes place when an offer from component $B_i$ is received. In this case, variables from $\{x_{p_i}^a\} \cup X_{p_i}^a$ are updated through the port $o_i^a$ by the values received in the offer. The update function $f_{\tau_i}$ sets $b_i^a$ to true, since the manager component has just received an offer from $B_i$. In Figure 6, the manager contains two ports $o_1^a$ and $o_2^a$. Port $o_i, i \in \{1, 2\}$, 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 are refreshed upon receiving an offer through ports $o_i^a$.

- **Detecting enabled interaction.** We include a port $\iota$ in $P$ and a transition $\tau_{en} = (undef, \iota, g_{\tau_{en}}, id, en)$ in $T$. The port $\iota$ is not synchronized elsewhere. Thus, it is omitted in Figure 6. The guard $g_{\tau_{en}} = \forall i \in I.\ (b_i^a \wedge x_{p_i}^a) \ \wedge\ G_a$ ensures that each component $B_i$ (1) is in a stable state, where port $p_i$ is enabled, and (2) the guard of $a$ holds before switching to the $en$ state. Finally, $id$ is the identity function. In Figure 6, we have a transition from $undef$ to $en$ guarded by $(b_1^a \wedge x_{p_1}^a) \wedge (b_2^a \wedge x_{p_2}^a) \wedge (n_1^a > n_2^a)$. Notice that the latter conjunct corresponds to the guard of interaction $a$ in Figure 4(b).

- **Detecting disabled interaction.** Likewise, $T$ contains the transition $\tau_{dis} = (undef, \iota, g_{\tau_{dis}}, id, dis)$, with the guard $g_{\tau_{dis}} = (\forall i \in I. b_i^a) \ \wedge\ (\exists i \in I. \neg x_{p_i}^a \vee \neg G_a)$. Thus, the manager switches to state $dis$ only when all components $B_i$ (1) are in a stable state and (2) either there exists a port in $P_a$ that is not enabled or the guard of $a$ does not hold. In our example in Figure 6, the guard of the transition from $undef$ to $dis$ is $(b_1^a \wedge b_2^a) \ \wedge\ (\neg x_{p_1}^a \vee \neg x_{p_2}^a \vee n_1^a \leq n_2^a)$.

- **Executing interaction and notifying components.** We include ports $start_a$ and $n_a$ in $P$. We also include transitions $\tau_{start} = (en, start_a, true, f_{\tau_{start}}, exc)$ and $\tau_n = (exc, n_a, true, id, undef)$ in $T$. The transition $\tau_{start}$ is executed when interaction $a$ has been selected for execution (selection procedure explained below). Since components $\{B_i\}_{i \in I}$ are about to leave stable state, first, the update function $f_{\tau_{start}}$ sets all $b_i^a$ variables to false. Then, it applies the update function $F_a$ on variables $\bigcup_{i \in I} X_{p_i}^a$. Next, the transition $\tau_n$ notifies components $\{B_i\}_{i \in I}$ that $a$ has been executed and updated values of variables in $\bigcup_{i \in I} X_{p_i}^a$ are sent through port $n_a$. In our example, the update function associated to $\tau_{start}$ 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 $n_a$, such that their new values are sent back to the component.

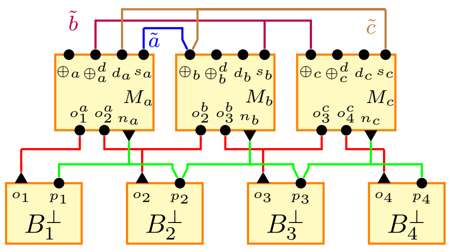


Figure 7: Deprioritized version of model from Figure 4(b)

- **Exporting disabled state.** We introduce a port $dis_a$ in $P$. This port is meant to be enabled only when the interaction $a$ is in disabled state. Thus, we label transition $(dis, dis_a, true, id, dis)$ with this port.
- **Handling weak conflicts.** This is the key element of our construction for enforcing priorities by the manager component.

  1. We include port $\oplus_a$ in $P$. All variables $b_i^a$ are associated to this port. This port takes part in a *schedule interaction* $\tilde{c}$ each time an interaction $c$ in weak conflict with interaction $a$ is selected. Intuitively, a schedule interaction $\tilde{c}$ (described later in detail) checks whether all interactions with higher priority than $c$ are disabled and informs other managers that $c$ will execute. Thus, variables $b_i^a$ that correspond to components involved in $c$ are set to false through the port $\oplus_a$ as well, and the state of manager $M_a$ reaches $undef$. Consequently, The port $\oplus_a$ labels the transitions $(en, \oplus_a, true, id, undef)$, $(dis, \oplus_a, true, id, undef)$ and $(undef, \oplus_a, true, id, undef)$.
  2. Moreover, the port $start_a$ is enabled when interaction $a$ is enabled by the manager $M_a$. This port participates in the schedule interaction $\tilde{a}$.
  3. Finally, we include the port $\oplus dis_a$ in $P$. This port labels the transition $(dis, \oplus dis_a, true, id, undef)$. This transition handles the case where there exists an interaction $c$ which has higher priority than $a$ and is in weak conflict with $a$.

**Interactions Involving a Manager**

Intuitively, our construction has three types of interactions: (1) *offer interactions* where components send their enabled ports to corresponding managers, (2) *notification interactions* where managers notify components after execution of an interaction, and (3) *schedule interactions* where priority rules are handled. We now formally construct the deprioritized model. Given a model $B = \pi\gamma(B_1, \cdots, B_n)$, first, we construct components $B_1^\perp \cdots B_n^\perp$, as prescribed in Subsection 3.1, and components $M_{a_1}, \cdots, M_{a_m}$, where $\{a_1 \cdots a_m\} = \gamma$, as prescribed in this Subsection (see Figure 7 for an example). Let $\gamma(B_i)$ denote the set of all interactions in $\gamma$ that involve the component $B_i$. Then, we define interactions $\tilde{\gamma}$ of the deprioritized model as follows (Figure 7 also shows the deprioritized version of the model in Figure 4(b)):

- **Offer interactions.** For each $i \in \{1 \cdots n\}$, $\tilde{\gamma}$ contains the interaction $\mathit{off}_i$, where $P_{\mathit{off}_i} = \{o_i\} \cup \bigcup_{a\in\gamma(B_i)}\{o_i^a\}$. For each interaction $a \in \gamma(B_i)$, the update function $F_{\mathit{off}_i}$ 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 a port of $B_i$ involved in $a$. We note that in Figure 7, 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). Offer interactions have no guard and they only copy variables from the sender port to variables of the receivers ports.
- **Notifications 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 to each component which port has been selected. The update function $F_{not_a}$ sets for each component $B_i$ involved in $a$ the

value of $X_{p_i}$ to the value of $X^a_{p_i}$. Interpretations of bullet and triangle ports are the same as for offer interactions.

- **Schedule interactions.** For each interaction $a \in \gamma$, $\tilde{\gamma}$ contains the interaction $\tilde{a}$:

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

  This interaction has no guard. For each interaction $c$ weakly conflicting with $a$, the update function $F_{\tilde{a}}$ sets the variable $b^c_i$ to false through the port $\oplus_c$ if $\{a, c\} \subseteq \gamma(B_i)$. For example, for the model in Figure 4(b) with priorities $b\pi a$ and $c\pi a$, we obtain the following schedule interactions:

  – $a$ has no higher priority interaction and is weakly conflicting with $b$. Thus, $P_{\tilde{a}} = \{start_a, \oplus_b\}$.
  – $b$ has less priority than $a$ and is weakly conflicting with both $a$ and $c$. Thus, $P_{\tilde{b}} = \{start_b, \oplus dis_a, \oplus_c\}$.
  – $c$ has less priority than $a$ and is weakly conflicting with $b$. Thus, $P_{\tilde{c}} = \{start_c, dis_a, \oplus_b\}$.

## 4.2 Corectness

We now show the correctness of our approach, where we prove that our construction results in a model that is observationally equivalent to the original BIP model.

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 unprioritized 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 $\gamma$. 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 $\mathit{off}_i$ and notifications $not_a$, are unobservable and are denoted $\beta$. We denote $\tilde{q} \xrightarrow{\beta} \tilde{q}'$ if a $\beta$ action brings the system from state $\tilde{q}$ to state $\tilde{q}'$.

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

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

**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 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 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. ■

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 $[\tilde{q}] \overset{\beta}{\not\to}$.

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 [11], where $\beta$-transitions are considered unobservable. The same definition is trivially extended for atomic and composite BIP components.

**Definition 7** (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 $\forall (q,r) \in R,\ a \in P:\ q \xrightarrow{a}_A q' \implies \exists r':\ (q',r') \in R \wedge r \overset{\beta^* a \beta^*}{\longrightarrow}_B r'$ 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 $\gamma$ 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 $\tau_{en}$ transition is true
- 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 $\beta$ 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 $(\tilde{r}, r) \in R$. ∎

## 4.3 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 use the method presented is section 5, or by replacing each notification and each offer by a set of binary interactions of the type {sender, receiver} (i.e., we need to duplicate offer ports in atomic components and notification ports in managers).

Notice that desynchronization of the notifications has no effect on the behaviour each component since atomic components perform independent computation after receiving the notification. On the contrary, synchronization of offers' receivers ensures that the value 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

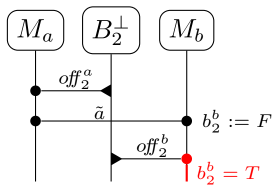


Figure 8: A scenario leading to inconsistency between managers

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 $\tilde{B}$.*

## 5 Building a Distributed Model: The 3 Tier Architecture

Once we construct a model with no priorities as prescribed in Section 4, one can apply the technique presented in [6] to generate distributed code. We now briefly recap this technique. The code generation is accomplished in two steps. First, from a given BIP component, we generate another BIP model that only incorporates asynchronous message passing as interactions (denoted SR-BIP). Then, we transform the SR-BIP model into a set of C++ executables – one per atomic component – that communicate using asynchronous message passing primitives such as MPI or TCP sockets primitives. We only review the first step.

As explained in Subsection 3.2, distributed execution of interactions may introduce strong 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 $M_{a_1}^\perp \cdots M_{a_m}^\perp$ (i.e., manager components obtained by the transformation explained in Subsection 3.1 to break atomicity). The components $B_1^\perp \cdots B_n^\perp$ are copied from the deprioritized model, since they have already been transformed by the deprioritization. Recall that the send-port offer ($o$) shares the list of enabled ports in the component with the upper tier. Each port $p$ of the original

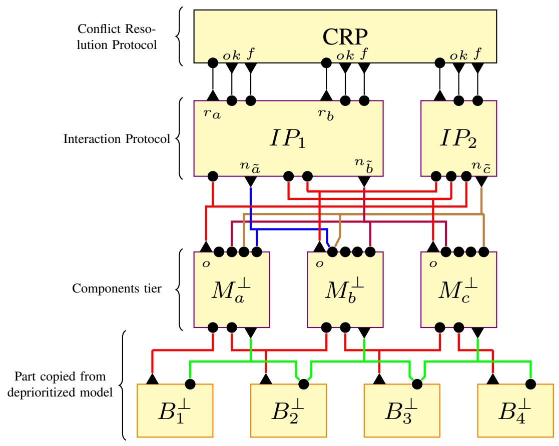


Figure 9: Distributed version of the deprioritized model from Figure 7

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. For example, the bottom tier in Figure 9 includes managers components illustrated in Figure 7.

**Interaction Protocol.** This tier consists of a set of components each hosting a set of interactions from the deprioritized BIP model. Strong 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 $IP_1$. To the contrary, the conflict between $\tilde{b}$ and $\tilde{c}$ has to be resolved using the third tier of our model. 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 $n_{\tilde{a}}, n_{\tilde{b}}, n_{\tilde{c}}$).

**Conflict Resolution Protocol.** This tier accommodates an algorithm that solves the *committee coordination problem* [9] to resolve strong conflicts between interactions hosted in separate interaction protocol components. For instance, the external conflict between interactions $\tilde{b}$ and $\tilde{c}$ 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 [1] or dining philosophers [9, 2] result in different structures in this tier and are discussed in detail in [6]. The interface between this tier and the Interaction Protocol involves ports for receiving request to reserve an interaction (labelled $r$) and responding by either success (labelled $ok$) or failure (labelled $f$).

## 6 Case Study

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 any disc. Access to the disc 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. All interactions take some time to execute. In particular, 100ms for *load/unload* and 500ms for *read*, respectively.

Figure 11 presents the behaviour of atomic components and the data transfer on interactions. To ensure

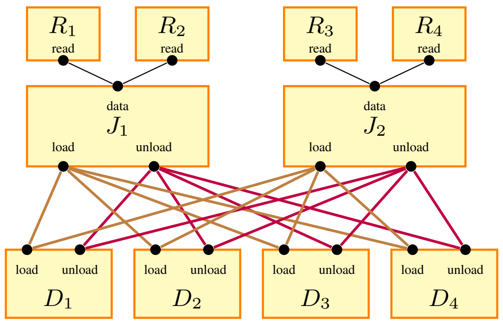


Figure 10: BIP Model for the jukebox example

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 on 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 $\{read, data\}$ interaction holds, only if the disc in the jukebox (*current*) is the one to be read (*to_read*). Each reader has a sequence of 2 discs to read. The variable *to_read* contains the id of the next disc to be read. It is initialized with the first value (not shown on the figure), and is updated after the first read. This model has finite runs: the execution terminates when all readers have read the two discs they needed.

This model comes in two versions. The first one, denoted $B_\emptyset$ does not contain priorities. The second one, denoted $B_\pi$, is the restriction of $B_\emptyset$ using two types of priorities:

- *Priorities to enforce termination*: we give priority to the *read* interactions over the *unload* interactions. Formally, it corresponds to the sets of priorities $\{unload_{i,1} \ \pi \ read_j | i \in \{1, \cdots, 4\}, j \in \{1, 2\}\}$ and $\{unload_{i,2} \ \pi \ read_j | i \in \{1, \cdots, 4\}, j \in \{3, 4\}\}$, one set corresponding to each jukebox. This ensures that any enabled *read* interaction will be executed before the disc is unloaded. Since each disc is eventually loaded, each *read* interaction will take place and the execution terminates. Otherwise, sequences of *load*/*unload* interaction could happen forever. Note that here, assuming fairness ensures that the model eventually terminates.
- *Priorities to speed up execution*: by inspecting the discs requested by the readers, we know that some discs are more often needed than others, therefore we give more priority to the corresponding *load* interactions. Here, we give more priority to the disc 1 in jukebox 1 by adding the following set of priorities: $\{load_{i,1} \ \pi \ load_{1,1} | i \in \{2, 3, 4\}\}$.

For both versions $B_\emptyset$ and $B_\pi$, we generate the corresponding deprioritized models $\tilde{B}_\emptyset$ and $\tilde{B}_\pi$. In Table 1, we present the size – the number of atomic components and the number of interactions – of these different models, in the columns labelled "Orig.". We then apply the transformation provided in [7] to the models $B_\emptyset$, $\tilde{B}_\emptyset$ and $\tilde{B}_\pi$ to obtain for each model a distributed version including a centralized scheduler[2]. The number of Send/Receive components and interactions contained in the distributed version of these models is given in the columns labelled "S/R" in Table 1. We simulate the execution of these models on two different platforms. The first one is *centralized*, where only one processor is available to execute all components. The second one is *fully decentralized*, where each atomic component has its own processor. We assume that executing a load, unload or read interaction completely blocks the processor. For each couple (model, execution platform), we measure the average time of terminating executions. Results are presented in Table 1.

As mentioned earlier, we applied our deprioritization transformation to model $B_\emptyset$ although we can directly obtain a distributed model. By comparing the execution times of $B_\emptyset$ and $\tilde{B}_\emptyset$ on the centralized

[2]We cannot transform directly $B_\pi$ into such a distributed model since the transformation presented in [7] does not take priorities into account.

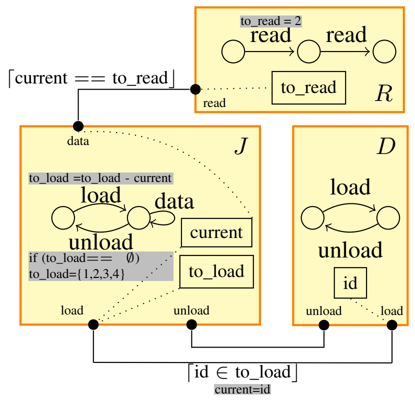


Figure 11: Behaviour of jukebox components and interactions

| | Model Size | | | | Execution time | |
|---|---|---|---|---|---|---|
| | # Atoms | | # Interactions | | | |
| | Orig. | S/R | Orig. | S/R | Cent. | Decent. |
| $B_\emptyset$ | 10 | 11 | 20 | 28 | 15.2 | 11.0 |
| $\tilde{B}_\emptyset$ | 30 | 31 | 70 | 148 | 12.0 | 5.9 |
| $\tilde{B}_\pi$ | 30 | 31 | 70 | 154 | 5.4 | 2.8 |

Table 1: Model size and execution time (in s) for different implementations of Figure 10

platform, we can notice that our deprioritization transformation does not introduce a significant overhead, even if it increases the number of components and interactions.

The distributed execution is almost twice faster for $\tilde{B}_\emptyset$ than for $B_\emptyset$. This is due to the fact all time consuming computations in $B_\emptyset$ are on interactions, which are all executed on the same processor (the one hosting the scheduler). When switching to $\tilde{B}_\emptyset$, these interactions are executed by the manager components and, hence, run concurrently on different processors.

The model $\tilde{B}_\pi$ runs faster than $B_\emptyset$ on the centralized platform. Here, priorities enforce a better scheduling – we first load the discs that are often used, we do not perform an unload if a reader has something left to read – and thus reduce the total execution time. Again, switching to decentralized execution gives almost twice better results, as (time consuming) interactions are now running concurrently.

## 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, (2) a transformation from [6, 7] 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 a non-trivial example and presented encouraging experimental results.

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 [7]. 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., [3]).

## References

[1] 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.

[2] R. Bagrodia. Process synchronization: Design and performance evaluation of distributed algorithms. *IEEE Transactions on Software Engineering (TSE)*, 15(9):1053–1065, 1989.

[3] 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.

[4] 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.

[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.

[6] 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.

[7] 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)*, 2010. To appear.

[8] K. M. Chandy and J. Misra. The drinking philosophers problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 6(4):632–646, 1984.

[9] K. M. Chandy and J. Misra. *Parallel program design: a foundation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1988.

[10] M. Jurdzinski. Small progress measures for solving parity games. In *Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 290–301, 2000.

[11] R. Milner. *Communication and concurrency*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, 1995.

[12] 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.

[13] J. Parrow and P. Sjödin. Multiway synchronizaton verified with coupled simulation. In *Proceedings of the Third International Conference on Concurrency Theory*, CONCUR '92, pages 518–533, London, UK, 1992. Springer-Verlag.

[14] 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.