Automated Distributed Implementation of Component-Based Models with Priorities
Borzoo Bonakdarpour, Marius Bozga, Jean Quilbeuf

To cite this version:

HAL Id: hal-00722405
https://hal.archives-ouvertes.fr/hal-00722405
Submitted on 1 Aug 2012

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Automated Distributed Implementation of Component-based Models with Priorities

Borzoo Bonakdarpour
School of Computer Science
University of Waterloo
200 University Avenue West
Waterloo, Canada, N2L3G1
borzoo@cs.uwaterloo.ca

Marius Bozga
VERIMAG, Centre Équation
2 avenue de Vignate
38610, GIÈRES, France
marius.bozga@imag.fr

Jean Quilbeuf
VERIMAG, Centre Équation
2 avenue de Vignate
38610, GIÈRES, France
jean.quilbeuf@imag.fr

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 embedded 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.

Categories and Subject Descriptors

C.2.4 [Computer-Communication Networks]: Distributed Systems [Distributed applications]; D.1.3 [Programming Techniques]: Concurrent Programming—Distributed programming, Parallel programming; D.2.13 [Software Engineering]: Reusable Software—Reuse models; D.4.7 [Operating Systems]: Organization and Design—Real-time and embedded systems; F.3.1 Logics

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

EMSOFT’11, October 9–14, 2011, Taipei, Taiwan.
Copyright 2011 ACM 978-1-4503-0714-7/11/10 ...$10.00.

General Terms

Theory, Design, Languages, Reliability, Performance

Keywords

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

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. Correctness of distributed implementations is of significant importance in the context of embedded applications, as such applications are often employed in safety-critical systems. Model-based development of embedded 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 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 derives 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
Concurrent 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.

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 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, and has inspired the approaches by 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.

### 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.

- **Improving performance.** In distributed systems, it is often the case that certain resources have higher demands. For example, in group mutual exclusion, as Mittal and Mohan argue, 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 and in particular, in embedded applications, it is desirable to guide the system to behave in a certain predictable fashion.

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 reliable knowledge about enabledness of interactions, so that only 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]).

These examples 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 and designers only need to make minimal effort to develop models.

### 1.2 Contributions

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 1.

- 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 1. 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. 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. We discuss our case study and experimental results in Section 5. Finally, we conclude in Section 6.

2. BASIC SEMANTIC MODELS OF BIP

In this section, we present operational global state semantics of BIP [5]. 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 \( p \).
- \( T \) 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 \( X \) the set of valuations on \( X \), and \( Q \times X \) the set of local states. Let \( (q, v) \) and \( (q', v') \) be two states in \( Q \times X \), \( p \) be a port in \( P \), and \( v_p'' \) be a valuation in \( X_p \) of \( X_p \). We write \( (q, v) \xrightarrow{p(v_p'')} (q', v') \), if \( \tau = (q, p, g, f, q') \in T \), \( g(v) \) 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, \( (q, v) \xrightarrow{\gamma} \) 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 2(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 \( i \in I \), \( p_i \in P_a \), where \( I \subseteq \{1...n\} \). We denote by \( F_a^n \) 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 (Component Interaction). 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 \( B \overset{\text{def}}{=} \pi \gamma(B_1, \ldots, B_n) \) the component obtained by composing components \( B_1, \ldots, B_n \) using the interactions \( \gamma \) and the priorities \( \pi \).

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

Definition 3 (Component Interaction 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 \( \rightarrow_i \) is the transition relation between states of \( B_i \), is a transition system \((Q, \gamma, X, \rightarrow)\), where \( Q = \bigcup_{i=1}^n Q_i \), \( X = \bigcup_{i=1}^n X_i \), and \( \rightarrow \) is the least set of transitions satisfying the rule:

\[
\forall i \in I, (q_i, v_i) \xrightarrow{p_i(v_i')} (q_i', v_i') \quad \forall v_i \in X_i, (q_i, v_i) \xrightarrow{(p_i, v_i')} (q_i, v_i') \\
\xrightarrow{\gamma} (q_i, v_i) \xrightarrow{\pi a} (q_i', v_i') \xrightarrow{\gamma} (q_i, v_i) \xrightarrow{\pi b} (q_i', v_i') \\
((q_i, v_i), \ldots, (q_n, v_n)) \xrightarrow{\gamma} (q_i', v_i') \xrightarrow{\pi a} (q_i', v_i') \xrightarrow{\gamma} (q_i, v_i) \xrightarrow{\pi b} (q_i', v_i')
\]

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

\[
(q, v) \xrightarrow{\pi a} (q', v') \quad \forall a' \in \gamma, \alpha \pi a' \implies (q, v) \xrightarrow{\alpha a'} (q', v')
\]
Intuitively, two interactions 
\( a \) and \( b \) are weakly conflicting (denoted \( a \sqcup b \)) if there exist two ports \( p \) and \( q \) in some component \( B \) such that \( p \in P_a \), and \( q \in P_b \).

This kind of conflict is called weak because it is weaker than the definition of conflict in [7], that we call here strong conflict. Two interactions are strongly conflicting if they share a common port or there is a couple of ports with one member in each interaction such that these two ports label two conflicting transitions of the same component. Clearly, strong conflict implies 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 atomic components in \( B \) by functionally equivalent send/receive atomic components, where atomicity of transitions and interactions is broken. This first transformation, already used in [4, 6, 7] separates the synchronization from computation on component transitions and enables the concurrent execution of atomic components.

2. Secondly, 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.

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 from the scheduler, once some interaction has been completed. Local update functions can then be executed concurrently and independently by components upon notification reception.

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 \( B^+ = (Q^+, X^+, P^+, T^+) \), such that:

- \( Q^+ = Q \cup \{ \bot \mid s \in Q \} \).
- \( X^+ = X \cup \{ x_p \mid p \in P \} \), where each \( x_p \) is a Boolean variable indicating whether port \( p \) is enabled.
- \( P^+ = P \cup \{ o \} \), where \( o \) is the offer port. All variables in \( X^+ \) are associated to \( o \) (i.e., \( X_o = X^+ \)).
For each transition $\tau = (q, p, g, f, q') \in T$, we include the following two transitions in $T^+$:

1. offer $\tau_o^g = (\bot_q, o, g_o, f_o, q)$ where $g_o$ is true, $f_o$ is the identity function, and
2. notification $\tau_n^f = (q, p, g_p, f_p, \bot_q')$ where $g_p$ is true and $f_p$ applies $f_r$ 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 $q'$ is true. Otherwise, $x_r$ is set to false.

In Definition 5, states $\{\bot_q \mid s \in Q\}$ 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.

Figure 3 shows the transformed version of the atomic component shown in Figure 2(a). Initially, the component is in busy state $\bot_q$ 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 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.

Let us 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 $Q = \{\text{undef}, \text{en}, \text{dis}, \text{exc}\}$. 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 being executed.

- The set of variables is $X = \{b^a_i\}_{i \in I} \cup \{x_{p_i}^a\}_{i \in I, p_i \in a}$. For every component $B_a$ in $a$, the manager holds a Boolean variable $b^a_i$ 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 $x^a_{p_i}$ which indicates the status of the port (i.e., enabled or disabled) and variables $X^a_{p_i}$ 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.
Figure 4 represents the manager for interaction a in Figure 2(b). It contains the variables \( b_1 \) and \( b_2 \) 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 \( 1 \) \( x_{p_i} \), 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 2(b). All these variables are refreshed upon receiving an offer through ports \( o_i^a \). The transition from \( \text{undef} \) to \( \text{en} \) guarded by \( \{ b_1^a \land x_{p_1}^a \land (b_2^a \land x_{p_2}^a) \land (n_1^a > n_2^a) \} \) switches from \( \text{undefined} \) to \( \text{enabled} \) state. The two first conjuncts ensures that \( 1 \) \( B_1 \) and \( B_2 \) are in \( \text{stable} \) state, and \( 2 \) \( p_1 \) and \( p_2 \) are enabled. The latter conjuncts corresponds to the guard of interaction a in Figure 2(b). Likewise, the transition from \( \text{undefined} \) to \( \text{dis} \) allows reaching the state where the interaction a is disabled. The update function associated to \( \tau_{\text{start}} \) 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 \( \gamma^{(1)} \) denote the set of all interactions in \( \gamma \) that involve the 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 \( \tilde{B} = \tilde{\gamma}(B_1, \cdots, B_n, M_1, \cdots, M_n) \), where \( B_i \) is obtained from \( B_i \) as explained in definition 5, \( M_a \) is obtained from \( a_j \) as explained in definition 6, and \( \tilde{\gamma} \) contains the following interactions:

- **Offer interactions.** For each \( i \in \{1, \cdots, n\} \), \( \gamma \) contains the interaction \( o_i \), where \( P_{\text{off}} = \{a\} \cup \bigcup_{a \in \gamma^{(1)}}\{o_a\} \). For each interaction \( a \in \gamma^{(1)} \), the update function \( F_{\text{off}} \) sets the values of variables \( \{x_{p_i}^a \} \cap X_{p_i}^a \) to the values of \( \{x_j\} \cup X_j \) associated to \( a \), where \( p \) is the of port \( B_i \) involved in \( a \). Offer interactions have no guard and they only copy data from the sender component to the manager.

- **Notifications interactions.** For each interaction \( a \in \gamma \), where \( a = \{p_i\}_{i \in I} \), \( \gamma \) contains the interaction \( n_{a} \), such that \( P_{\text{not}} = \{a\} \cup \bigcup_{a \in \gamma^{(1)}}\{n_a\} \). This interaction notifies each component which port has been selected. The update function \( F_{\text{not}} \) 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 \( a \in \gamma \), \( \gamma \) contains the interaction \( a \),

\[
P_a = \{\text{start}_a\}
\cup \{\text{not}_a^c | c \neq a\}
\cup \{\text{dis}_c | c \neq a, a \neq c\}
\cup \{\text{dis}_c, c \neq a, a \neq c\}
\]

This interaction has no guard. For each interaction \( c \) conflicting with \( a \), the update function \( F_{\text{not}} \) sets variable \( b_i^a \) of manager \( M_c \) to false through the port \( \oplus_c \) if \( \{a, c\} \subset \gamma^{(1)} \). In other terms, the \( \text{start}_a \) interaction informs the 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 \( b_i^a \) variable in each manager \( M_a \) and the actual state of component \( B_i \).
3.4 Correctness

We now show that the above transformation 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 $B = \pi q(B_1, \ldots, B_m)$ be a BIP model and $\tilde{B} = \tilde{\gamma}(B^+_1, \ldots, B^+_n, M_1, \ldots, M_m)$ be its unprioritized version. We denote $q = (q_1, \ldots, q_n)$ a state of $B$ and $\tilde{q} = (\tilde{q}_1, \ldots, \tilde{q}_n, s_1, \ldots, s_m)$ 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 $\{\tilde{\alpha}[a] \in \gamma\}$. The remaining interactions in $\tilde{B}$, namely offers $\tilde{\alpha}i$ and notifications $\tilde{\alpha}a$, 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.** $\beta \xrightarrow{\gamma} \text{is terminating}.$

**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 $\tilde{\alpha}$ action is executed. Thus at most $2n$ consecutive $\beta$-steps are possible. \qed

**Proposition 2.** From any reachable state $\tilde{q}$ of $\tilde{B}$, $\beta \xrightarrow{\gamma} \text{is confluent}.$
• All ports of a are enabled and the guard \( G_a \) is true, since the guard of the \( \tau_e \) transition is true.

• No higher priority interaction is enabled since \( \tilde{a} \) is enabled only when managers corresponding to such interactions are in state \( \text{dis} \).

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 \( \text{en} \) or \( \text{dis} \), 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 \( \text{en} \) at state \([\tilde{q}]\). If there is any interaction \( \tilde{b} \) with higher priority than \( a \), then it is disabled in state \( q \), thus the manager \( M_b \) is in state \( \text{dis} \) 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 \((\tilde{r}, r) \in R\). \( \square \)

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 [6] to generate distributed code. We now briefly recap this technique. The code generation is accomplished in two steps. First, from a given BIP model, 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 executables – one per atomic component – that communicate using asynchronous message passing primitives such as MPI or TCP sockets. We only review the first step.

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 5 formed by \( \gamma_{\text{sched}}(M_a, M_b, M_c) \) where \( \gamma_{\text{sched}} = \{\tilde{a}, \tilde{b}, \tilde{c}\} \) to describe the concepts of our transformation. The distributed version of \( \gamma_{\text{sched}}(M_a, M_b, M_c) \) is presented in Figure 6. Our 3-tier architecture consists of the following.

Components Tier. Let \( \tilde{B} = (\tilde{B}_1, \tilde{B}_2, \tilde{B}_3, \tilde{B}_4) \) be a deprioritized BIP model. The component tier includes components:

- \( M_{a_1}, \ldots, M_{a_m} \) (i.e., manager components obtained by the transformation explained in Subsection 3.1 to break atomicity), and

- \( B_1, \ldots, B_n \) are copied from the deprioritized model, since they have already been transformed by the deprioritization.

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 5 are grouped into component \( IP_1 \) in Figure 6. 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 \( n_a, n_b, n_c \)).

Conflict Resolution Protocol. This tier accommodates an algorithm that solves the \textit{committee coordination problem} [9] to resolve 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 6. 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 6. Other algorithms, such as ones that use a circulating token [2] or dining philosophers [1, 9] result in different structures in this tier and are discussed in detail in [7]. The interface between this tier and the Interaction Protocol involves ports for receiving requests to reservation (labelled \( r \)) and responding by either success (labelled \( ok \)) or failure (labelled \( f \)).
5. 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 7 and 8). 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 \( (D_i, J_k), i \in \{1 \ldots 4\} \) and \( k \in \{1, 2\} \), has two interactions: (1) a \( \text{load}_{i,k} \) interaction for loading the disc in the jukebox and an \( \text{unload}_{i,k} \) interaction for unloading it. Each reader \( R_i \) is connected to a jukebox through a \( \text{read}_i \) interaction. During the test, we simulate execution of interactions by waiting a given amount of time. Namely, we wait 100ms for \( \text{load}/\text{unload} \) and 500ms for \( \text{read} \).

Figure 8 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 \( \text{to_load} \). Each time a disc is loaded, it is removed from the list by the \( \text{load} \) transition in the jukebox component. The guard of a \( \text{load} \) interaction prevents the disc to be loaded if it is not on the list. When the \( \text{to_load} \) list becomes empty, it is reinitialized to the set of all discs on the \( \text{unload} \) interaction. The variable \( \text{current} \) contains the identity (i.e., \( 1 \ldots 4 \)) of the disc currently loaded in the jukebox, and is updated by the \( \text{load} \) interaction. In order to ensure that the reader gets the correct data, a guard on the \( \{\text{read}, \text{data}\} \) interaction holds, only if the disc in the jukebox (\( \text{current} \)) is the one to be read (\( \text{to_read} \)). Each reader has a sequence of 2 discs to read. The variable \( \text{to_read} \) contains the id of the next disc to be read. It is initialized with the first value (not shown in 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.

We consider two versions of the model. The first model, denoted \( B_0 \), does not contain priorities. The second model, denoted \( B_\pi \), is the \( B_0 \) restricted by two types of priorities:

- **Priorities to enforce termination.** We give priority to the \( \text{read} \) interactions over the \( \text{unload} \) interactions. Formally, it corresponds to the sets of priorities \( \{\text{unload}_{i,1} \pi \text{read}_{i} | i \in \{1, \ldots, 4\}, j \in \{1, 2\}\} \) and \( \{\text{unload}_{i,2} \pi \text{read}_{i} | i \in \{1, \ldots, 4\}, j \in \{3, 4\}\} \), for each jukebox. This ensures that any enabled \( \text{read} \) interaction will be executed before the disc is unloaded.

Since each disc is eventually loaded, each \( \text{read} \) interaction will take place and the execution terminates. Otherwise, sequences of \( \text{load}/\text{unload} \) interaction could occur 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. Thus, we give higher priority to the corresponding \( \text{load} \) interactions. Here, we give higher priority to Disc 1 in Jukebox 1 by adding the following set of priorities: \( \{\text{load}_{1,1} \pi \text{load}_{1,1} | i \in \{2, 3, 4\}\} \).

For both versions \( B_0 \) and \( B_\pi \), we generate the corresponding deprioritized models \( \tilde{B}_0 \) and \( \tilde{B}_\pi \). In Table 3, 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 \( B_0, \tilde{B}_0 \), and \( B_\pi \) to obtain a distributed version of each model 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 3. 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. The results are presented in Table 3.

As mentioned earlier, we applied our deprioritization transformation to model \( B_0 \) although we can directly obtain a distributed model. By comparing the execution times of \( B_0 \) and \( B_\pi \) on the centralized platform, we observe that our deprioritization transformation does not introduce a significant overhead, even if it increases the number of components and interactions.

\(^2\)We cannot transform directly \( B_\pi \) into such a distributed model since the transformation presented in [7] does not take priorities into account.
Table 3: Model size and execution time (s) for different implementations of Figure 7.

<table>
<thead>
<tr>
<th></th>
<th>Model Size</th>
<th>Execution time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td># Atoms</td>
<td># Interactions</td>
</tr>
<tr>
<td>Orig. S/R</td>
<td>Orig.</td>
<td></td>
</tr>
<tr>
<td>$B_0$</td>
<td>10 11</td>
<td>20 28</td>
</tr>
<tr>
<td>$B_0^*$</td>
<td>30 31</td>
<td>70 148</td>
</tr>
<tr>
<td>$B_*$</td>
<td>30 31</td>
<td>70 154</td>
</tr>
</tbody>
</table>

More importantly, the distributed execution of $B_0^*$ is almost twice faster than $B_0$. This is due to the fact all time consuming computations in $B_0^*$ are on interactions, which are all executed on the same processor (the one hosting the scheduler). When switching to $B_0^*$ these interactions are executed by the manager components and, hence, run concurrently on different processors.

Furthermore, the model $B_*$ runs faster than $B_0^*$ on a centralized platform. In this scenario, priorities enforce a better scheduling – we first load the discs that are often used and 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.

6. 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]).

7. REFERENCES