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 and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*Logic of programs*; I.2.2 [**Artificial Intelligence**]: Automatic Programming—*Program transformation*

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

*The research leading to these results has received funding from the European Community's Seventh Framework Programme [FP7/2007-2013] under grant agreement no 248776 (PRO3D) and no 257414 (ASCENS), from ARTEMIS JU grant agreement ARTEMIS-2009-1-100230 (SMECY) and from Canada ORF RE03-045, NSERC DG 357121-2008, and IS09-06-037 grants.

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 [2] 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 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* [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.

[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.

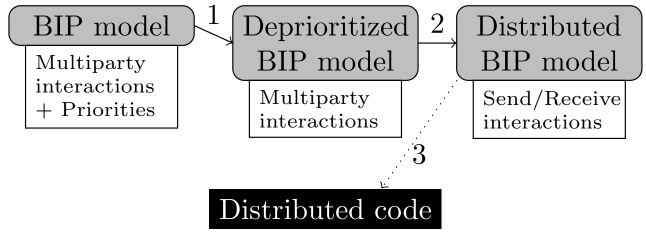


Figure 1: Steps for generating a distributed implementation from a high-level BIP model.

- **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* τ *and* f *is the* update function *of* τ*.* g *is a predicate defined over the variables in* X *and* f *is a function that computes new values for* X *according to the previous ones.*

We denote $\mathbf{X}$ 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 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 $i \in I$, $p_i \in P_i$, where $I \subseteq \{1..n\}$. We denote by F_a^i the projection of F_a on X_{p_i}.

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

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

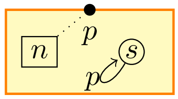
Note that if the system does not contain any priority, we may omit π.

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

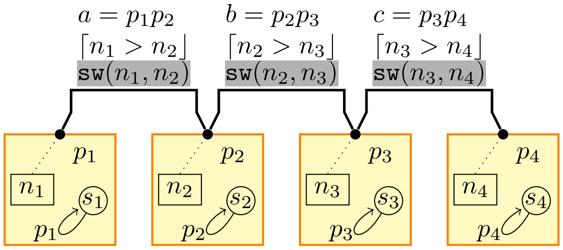
$$\frac{\begin{array}{c} a = (\{p_i\}_{i \in I}, G_a, F_a) \in \gamma \\ G_a(v_1, \ldots, v_n) \qquad \forall i \notin I.\ (q_i, v_i) = (q'_i, v'_i) \\ \forall i \in I.\ (q_i, v_i) \xrightarrow{p_i(v''_{p_i})}_i (q'_i, v'_i), v''_{p_i} = F_a^i(v_1, \ldots, v_n) \end{array}}{((q_1, v_1), \ldots, (q_n,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

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

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



(a) An atomic component



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

Figure 2: Atomic and composite components in BIP

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 inside each component. The states of components that do not participate in the interaction stay unchanged. The second inference rule simply filters out transitions which are not maximal with respect to priorities. A transition is executed only if no other one with higher priority is enabled.

Figure 2(b) illustrates a composite component $\gamma(B_1, \cdots, B_4)$, where each B_i is identical to component B in Figure 2(a). The set γ 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.

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

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

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

This kind of conflict is called *weak* because it is weaker than the definition of conflict in [7], that we call here *strong conflict*. Two interactions are strongly conflicting iff 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 $B^\perp = (Q^\perp, X^\perp, P^\perp, T^\perp)$, such that:*

- $Q^\perp = Q \cup \{\perp_s | s \in Q\}$.
- $X^\perp = X \cup \{x_p\}_{p \in P}$, *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., $X_o = X^\perp$).*

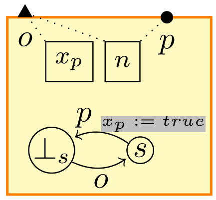


Figure 3: Transformed version of one atomic component from Figure 2(b)

- *For each transition $\tau = (q, p, g, f, q') \in T$, we include the following two transitions in $T^{\perp}$:*
  1. offer $\tau_o^q = (\perp_q, o, g_o, f_o, q)$ *where g_o 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_τ on X and for each port $r \in P$, it sets x_r to true if $\tau' = (q', r, g', f', q'') \in T$ for some q'' and g' is true. Otherwise, x_r is set to false.*

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

Figure 3 shows the transformed version of the atomic component shown in Figure 2(a). Initially, the component is in busy state $\perp_s$ and the value of x_p is true; i.e., the component is willing to interact on port p. Then, it sends an offer through port o containing the current values of x_p and n and reaches stable state s. The reception of a notification corresponds to the p-labelled transition that brings back the component to the initial busy state.

3.2 Interaction Managers

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

Table 1: Ports of a manager component

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

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

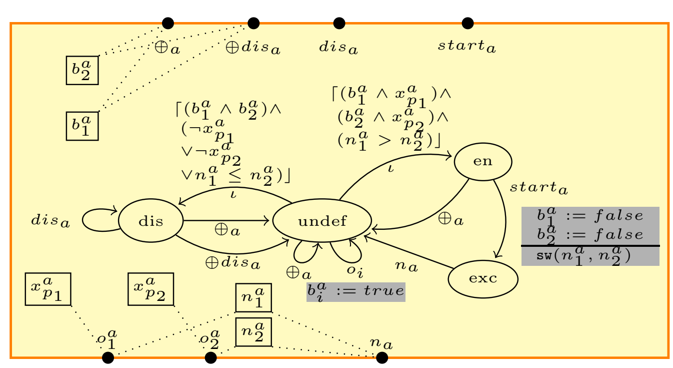


Figure 4: The manager component for interaction a between components B_1 and B_2 in Figure 2(b).

Table 2: Transitions of a manager component

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

Figure 4 represents the manager for interaction a in Figure 2(b). It contains the variables b_1^a and b_2^a since interaction a involves components B_1 and B_2. The manager contains two offer ports o_1^a and o_2^a. Port o_i^a, $i \in \{1, 2\}$, is associated with variables (1) $x_{p_i}^a$, which indicates the status of port p_i in B_i, and (2) n_i^a, that are local copies of variables n_i associated to ports p_i in Figure 2(b). All these variables are refreshed upon receiving an offer through ports o_i^a. The transition from *undef* to *en* guarded by $(b_1^a \wedge x_{p_1}^a) \wedge (b_2^a \wedge x_{p_2}^a) \wedge (n_1^a > n_2^a)$ switches from *undefined* to *enabled* state. The two first conjuncts ensures that (1) B_1 and B_2 are in *stable* state, and (2) p_1 and p_2 are enabled. The latter conjunct corresponds to the guard of interaction a in Figure 2(b). Likewise, the transition from *undef* to *dis* allows reaching the state where the interaction a is disabled. The update function associated to τ_{start} sets b_1^a and b_2^a to false and then swaps the variables n_1^a and n_2^a. Both n_1^a and n_2^a are associated to the notification port n_a, so their new values are sent back to the component.

3.3 Connecting Managers

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

We now formally define the deprioritized model, by specifying how we connect the components defined so far. Let $\gamma^{(i)}$ denote the set of all interactions in γ that involve 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 $\tilde{B} = \tilde{\gamma}(B_1^{\perp}, \cdots, B_n^{\perp}, M_{a_1}, \cdots, M_{a_m})$, where $B_i^{\perp}$ is obtained from B_i as explained in definition 5, M_{a_j} is obtained from a_j as explained in definition 6, and $\tilde{\gamma}$ contains the following interactions:*

- **Offer interactions.** *For each $i \in \{1 \cdots n\}$, $\tilde{\gamma}$ contains the interaction off_i, where $P_{\mathit{off}_i} = \{o_i\} \cup \bigcup_{a \in \gamma^{(i)}} \{o_i^a\}$. For each interaction $a \in \gamma^{(i)}$, the update function F_{off_i} sets the values of variables $\{x_{p_i}^a\} \cup X_{p_i}^a$ to the values of $\{x_p\} \cup X_p$ associated to o_i, where p is the of port B_i involved in a. 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 $a = \{p_i\}_{i \in I}$, $\tilde{\gamma}$ contains the interaction not_a, such that $P_{not_a} = n_a \cup \{p_i\}_{i \in I}$. This interaction notifies each component which port has been selected. The update function F_{not_a} copy back data to each component B_i involved in a. That is, the values of X_{p_i} (in B_i) are set to the values of $X_{p_i}^a$ (from M_a).*
- **Schedule interactions.** *For each interaction $a \in \gamma$, $\tilde{\gamma}$ contains the interaction $\tilde{a}$:*

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

This interaction has no guard. For each interaction c weakly conflicting with a, the update function $F_{\tilde{a}}$ sets variable b_i^c of manager M_c to false through the port $\oplus_c$ if $\{a, c\} \subseteq \gamma^{(i)}$. In other terms, the $start_a$ 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^c variable in each manager M_c and the actual state of component B_i.

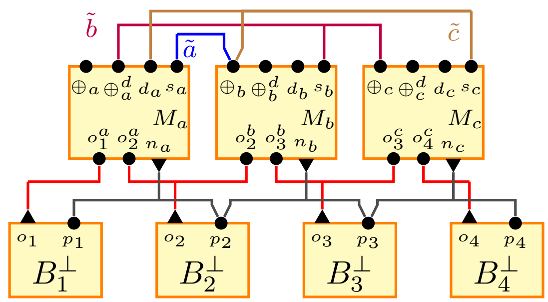


Figure 5: Deprioritized version of model from Figure 2(b).

Figure 5 presents the deprioritized model from Figure 2(b). Please note that the port names have been shortened for space reasons (e.g. s_a and d_a stand for $start_a$ and dis_a respectively. For offer and notification interactions, we interpret a triangle port as a *send port* (i.e., for sending offers) and bullet port as a *receive port* (i.e., for receiving offers). Note that offers and notifications only copy variables between components and managers.

If we assume priorities $b\pi a$ and $c\pi a$ for the model in Figure 2(b), we obtain the following schedule interactions: a has no higher priority interaction and is weakly conflicting with b, thus $P_{\tilde{a}} = \{start_a, \oplus_b\}$. Executing $\tilde{a}$ will set the variable b_2^b to false in M_b, since B_2 will become busy. 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 $P_{\tilde{c}} = \{start_c, dis_a, \oplus_b\}$.

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 $B = \pi\gamma(B_1, \cdots, B_n)$ 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 γ. The observable actions of $\tilde{B}$ are only the schedule interactions, that is $\{\tilde{a} | a \in \gamma\}$. The remaining interactions in $\tilde{B}$, namely offers off_i and notifications not_a, are unobservable and are denoted β. We denote $\tilde{q} \xrightarrow{\beta} \tilde{q}'$ if a β action brings the system from state $\tilde{q}$ to state $\tilde{q}'$.

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

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

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

PROOF. In any reachable state, if a manager reaches the state *exc* then the corresponding notification 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. $\square$

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

We recall the definition of *observational equivalence* of two transition systems $A = (Q_A, P \cup \{\beta\}, \rightarrow_A)$ and $B = (Q_B, P \cup \{\beta\}, \rightarrow_B)$. It is based on the usual definition of weak bisimilarity [11], where β-transitions are considered unobservable. The same definition is trivially extended for atomic and composite BIP components.

DEFINITION 8 (WEAK SIMULATION). *A weak simulation over A and B, denoted $A \subset B$, is a relation $R \subseteq Q_A \times Q_B$, such that we have $\forall (q, r) \in R,\ a \in P:\ q \xrightarrow{a}_A q' \implies \exists r' : (q', r') \in R \wedge r \xrightarrow{\beta^* a \beta^*}_B r'$ and $\forall (q, r) \in R:\ q \xrightarrow{\beta}_A q' \implies \exists r' :\ (q', r') \in R \wedge r \xrightarrow{\beta^*}_B r'$*

A weak bisimulation over A and B is a relation R such that R and R^{-1} are both weak simulations. We say that A and B are *observationally equivalent* and we write $A \sim B$ if for each state of A there is a weakly bisimilar state of B and conversely. We consider the correspondence between observable actions of B and $\tilde{B}$ as follows. To each interaction $a \in \gamma$, where γ is the set of interactions of B, we associate the schedule interaction $\tilde{a}$ of $\tilde{B}$.

THEOREM 1. *$B \sim \tilde{B}$.*

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

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

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

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

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

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

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

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

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

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

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 $\gamma_{sched}(M_a, M_b, M_c)$ 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 6. 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), and

- $B_1^\perp \cdots B_n^\perp$ are copied from the deprioritized model, since they have already been transformed by the deprioritization.

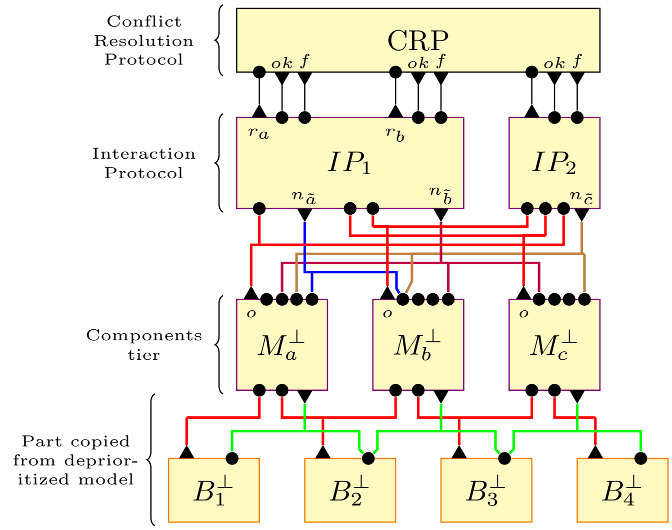


Figure 6: Distributed version of the deprioritized model from Figure 5.

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 $n_{\tilde{a}}, n_{\tilde{b}}, n_{\tilde{c}}$).

Conflict Resolution Protocol. This tier accommodates an algorithm that solves the *committee coordination problem* [9] to resolve 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 *reserve* an interaction (labelled r) and responding by either success (labelled *ok*) or failure (labelled f).

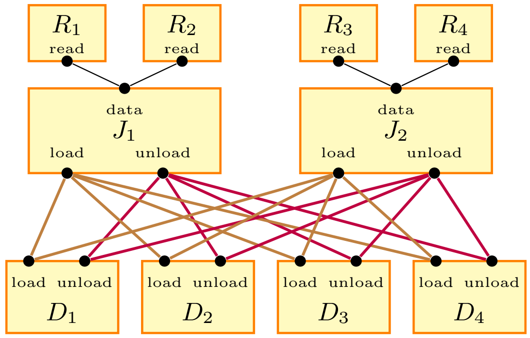


Figure 7: BIP Model for the jukebox example.

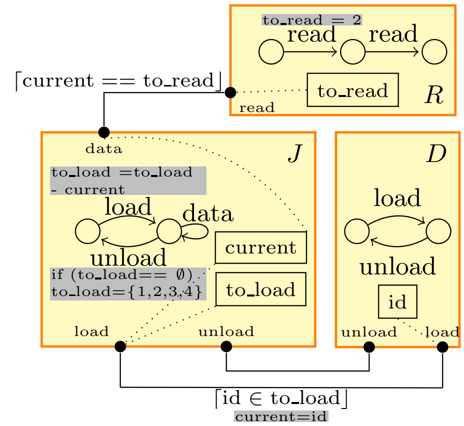


Figure 8: Behaviour of jukebox components and interactions.

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), $i \in \{1 \cdots 4\}$ and $k \in \{1, 2\}$, has two interactions: (1) a $load_{i,k}$ interaction for loading the disc in the jukebox and an $unload_{i,k}$ interaction for unloading it. Each reader R_j is connected to a jukebox through a $read_j$ interaction. During the test, we simulate execution of interactions by waiting a given amount of time. Namely, we wait 100ms for *load*/*unload* and 500ms for *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 *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., $1 \ldots 4$) 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 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 $B_\emptyset$, does not contain priorities. The second model, denoted B_π, is the $B_\emptyset$ restricted by 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 $\{unload_{i,1} \; \pi \; read_j \mid i \in \{1, \cdots, 4\}, j \in \{1, 2\}\}$ and $\{unload_{i,2} \; \pi \; read_j \mid i \in \{1, \cdots, 4\}, j \in \{3, 4\}\}$, for each jukebox. This ensures that any enabled *read* interaction will be executed before the disc is unloaded. Since each disc is eventually loaded, each *read* interaction will take place and the execution terminates. Otherwise, sequences of *load*/*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 *load* interactions. Here, we give higher priority to Disc 1 in Jukebox 1 by adding the following set of priorities: $\{load_{i,1} \; \pi \; load_{1,1} \mid i \in \{2, 3, 4\}\}$.

For both versions $B_\emptyset$ and B_π, we generate the corresponding deprioritized models $\tilde{B}_\emptyset$ and $\tilde{B}_\pi$. 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 $B_\emptyset$, $\tilde{B}_\emptyset$, and $\tilde{B}_\pi$ 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 $B_\emptyset$ although we can directly obtain a distributed model. By comparing the execution times of $B_\emptyset$ and $\tilde{B}_\emptyset$ 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_π 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.**

	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

More importantly, the distributed execution of $\tilde{B}_\emptyset$ is almost twice faster than $B_\emptyset$. 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.

Furthermore, the model $\tilde{B}_\pi$ runs faster than $B_\emptyset$ 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

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

[2] Rajive Bagrodia. A distributed algorithm to implement n-party rendevouz. In *Foundations of Software Technology and Theoretical Computer Science, Seventh Conference (FSTTCS)*, pages 138–152, 1987.

[3] 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)*, pages 209–218, 2010.

[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] Marcin 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 *International Conference on Concurrency Theory (CONCUR)*, pages 518–533, 1992.

[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.