A framework for automated distributed implementation of component-based models

Borzoo Bonakdarpour, Marius Bozga, Mohamad Jaber, Jean Quilbeuf, Joseph Sifakis

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

A Framework for Automated Distributed Implementation of Component-based Models

Borzoo Bonakdarpour · Marius Bozga · Mohamad Jaber · Jean Quilbeuf · Joseph Sifakis

Received: date / Accepted: date

Abstract Although distributed systems are widely used nowadays, their implementation and deployment are still time-consuming, error-prone, and hardly predictable tasks. In this paper, we propose a method for producing automatically efficient and correct-by-construction distributed implementations from a model of the application software in BIP. BIP (Behavior, Interaction, Priority) is a well-founded component-based framework encompassing high-level multi-party interactions for synchronizing components (e.g., rendezvous and broadcast) and dynamic priorities for scheduling between interactions.

Our method transforms an arbitrary BIP model into a Send/Receive BIP model that is directly implementable on distributed execution platforms. The transformation consists in (1) breaking the atomicity of actions in components by replacing synchronous multiparty interactions with asynchronous Send/Receive interactions; (2) inserting distributed controllers that coordinate the execution of interactions according to a user-defined partition of interactions, and (3) adding a distributed algorithm for handling conflicts between controllers. The obtained Send/Receive BIP model is proven observationally equivalent to its corresponding initial model. Hence, all functional properties of the initial BIP model are preserved by construction in the implementation. Moreover, the obtained Send/Receive BIP model can be used to automatically derive distributed executable code. The proposed method is fully implemented. Currently, it is possible to generate C++ implementations for (1) TCP sockets for conventional distributed communication, (2) MPI for multiprocessor platforms, and (3) POSIX threads for deployment on multi-core platforms. We present four case studies and report experimental results for different design choices including partition of interactions and choice of algorithm for distributed conflict resolution.

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

1 Introduction

Design and validation of computing systems often start with developing a *high-level model* of the system which abstracts the implementation details. Using an abstract model is beneficial, as they can be validated with respect to a set of requirements through different techniques such as formal verification, simulation, and testing. However, deriving a *correct* implementation from a model is always challenging, since adding implementation details involves many subtleties that can potentially introduce errors in the resulting system. In the context of distributed systems, these diffi-

This article extends the results in two papers that appeared in ACM International Conference on Embedded Software (EMSOFT'10) and IEEE International Symposium on Industrial Embedded Systems (SIES'10). This work is partially supported by EU FP7 projects no. 215543 (COMBEST), no. 248776 (PRO3D), and by Canada NSERC DG 357121-2008, ORF RE03-045, ORE RE-04-036, and ISOP IS09-06-037.

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

Marius Bozga, Mohamad Jaber, Jean Quilbeuf, Joseph Sifakis
UJF-Grenoble 1 / CNRS UMR 5104
VERIMAG, Grenoble, F-38041, France
E-mail: name.surname@imag.fr

culties are significantly amplified because of inherently concurrent and non-deterministic behavior, as well as the occurrence of unanticipated physical and computational events such as faults. Thus, it is highly advantageous for designers to automatically derive a correct-by-construction implementation from a high-level model. It is, nonetheless, unclear how to transform a high-level abstract model based on global state semantics and multiparty interactions into a real distributed implementation.

In this paper, we present a novel method for automatically transforming a high-level model in BIP [6,19] into a distributed implementation. The BIP (Behavior, Interaction, Priority) language is based on a semantic model encompassing composition of heterogeneous components. The *behavior* of components is described as a Petri net extended with data and functions given in C/C++. Transitions of the Petri net are labeled by port names and functions computing data transformations when they are executed. If a transition can be executed, we say that the associated port is *enabled*. BIP uses a composition operator to obtain composite components from a set of atomic components. The operator is parametrized by a set of *interactions* between the composed components. According to the operational semantics of BIP, the execution of a composite component is done by a sequential *Engine* as follows:

1. The Engine receives the sets of enabled ports from all components.
2. It computes the set of possible interactions, that is the set of interactions whose ports label only enabled transitions.
3. The Engine then chooses one amongst the possible interactions non-deterministically and executes the corresponding update function. This execution of an interaction may involve data sharing between the interacting components.
4. Finally the Engine sequentially executes the transitions in components involved in the chosen interaction.

We emphasize that the BIP semantics does not enforce any fairness condition between interactions.

The principle of the method for transforming a BIP model into a distributed implementation is illustrated through the following example. Consider the BIP model in Fig. 1 with components $B_1 \cdots B_5$ synchronized by using four interactions $a_1 \cdots a_4$. According to the semantics of BIP, interactions are executed sequentially by the Engine. Getting a distributed implementation for this model requires addressing the following issues:

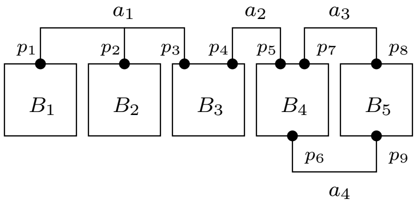


Fig. 1 A simple BIP model with conflicts.

- (*Partial observability*) Suppose that interaction a_1 involving components B_1, B_2, and B_3 is executing in a distributed implementation of the model. If component B_3 completes its computation before B_1 and B_2, and, ports p_4 and p_5 are enabled, then interaction a_2 is enabled as well. In this case, a distributed Engine must be designed, so that concurrent execution of interactions does not introduce a behavior that is not allowed in the initial model.
- (*Resolving conflicts*) BIP semantics assumes that each component has a sequential behavior and participates in a single interaction at each execution step. Since interactions a_1 and a_2 share component B_3, they cannot be executed concurrently. We call such interactions *conflicting*. Obviously, a correct distributed implementation must ensure that execution of conflicting interactions is mutually exclusive.
- (*Performance*) On top of correctness issues, a real challenge is to ensure that the transformation leads to efficient implementations. After all, one crucial goal for developing distributed and parallel systems is to exploit concurrency in order to achieve execution speed up.

Partial observability is achieved by breaking the atomicity of transitions; i.e., each transition is decomposed into two steps. The first step consists in participating in some interaction. The second step is an internal unobservable computation [5].

Resolving conflicts boils down to solving the *committee coordination problem* [15], where a set of professors organize themselves in different committees and two committees that have a professor in common cannot meet simultaneously. The original distributed solution to the committee coordination problem assigns one *manager* to each interaction [15]. Conflicts between interactions are resolved by reducing the problem to the dining or drinking philosophers problems [14], where each manager is mapped onto a philosopher. Bagrodia [4] points out a family of solutions to the committee coordination problem, ranging from fully centralized to fully decentralized ones, depending upon the mapping of sets of committees to the managers. Applying different solutions results in different distributed implementations of the initial BIP model. Each solution exhibits advantages and disadvantages and, hence, fits a specific

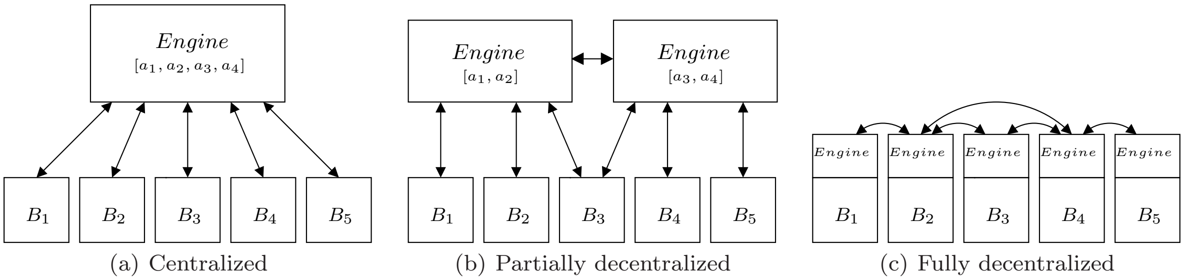


Fig. 2 Spectrum of transformations applied on the example presented in Fig. 1.

type of applications on a target architecture and platform. As a rule, three types of solutions are possible while designing an Engine for a distributed setting:

- *(Centralized)* This solution leads to transformations where distributed components are coordinated by a single centralized Engine (see Fig. 2(a)) no matter how the components are distributed.
- *(Partially decentralized)* This solution leads to transformations where a set of Engines collaborate in order to resolve conflicts in a distributed fashion (see Fig. 2(b)).
- *(Fully decentralized)* This solution leads to transformations where each component acts as a local Engine and reaches an agreement with other components on which non-conflicting interaction(s) can be executed (see Fig. 2(c)).

Existing methods [3,4,9,11,15,16,22,23,33] provide only principles of solutions, algorithms and preliminary simulation results. They partially focus on preservation of functional properties, level of concurrency, fairness, fault-tolerance, efficiency, or performance issues. Nevertheless, the state-of-the-art lacks a deep understanding of the impact of all these issues and their correlation on the generation of concrete distributed implementations. For example, existing implementation methods either do not achieve decentralization [9], are inefficient [16], or require the designer to explicitly specify communication mechanisms of the distributed implementation [33].

Contributions. With this motivation, we propose a framework for transforming BIP models into distributed implementations that allow parallelism between components as well as parallel execution of non-conflicting interactions by using solutions to the committee coordination problem. To the best of our knowledge, this is the first fully automated general implementation method. The works mentioned above focus only on impossibility results, abstract algorithms, and in one instance [4] simulation of an algorithm. Our method involves the following sequence of transformations preserving *observational equivalence* [27]:

1. First, we transform a BIP model into another BIP model that (1) operates in partial-state semantics, and (2) expresses multi-party interactions in terms of asynchronous message passing (Send/Receive primitives). Moreover, the target BIP model is structured in three layers, whose last two layers constitute a distributed Engine:
   (a) The *Components Layer* includes the components of the initial model where each port involved in a strong synchronization is replaced by a pair of Send/Receive ports.
   (b) The *Interaction Protocol* detects enabledness of interactions of the original model and executes them after resolving conflicts either locally or by the help of the third layer. The Interaction Protocol layer consists of a set of components, each handling a user-defined subset of interactions from the original BIP model[1].
   (c) The *Reservation Protocol* treats conflict resolution requests from the Interaction Protocol. It implements a committee coordination algorithm and our design allows employing any such algorithm. We, in particular, consider three committee coordination algorithms: (1) a fully centralized algorithm, (2) a token-based distributed algorithm, and (3) an algorithm based on a reduction to distributed dining philosophers.
2. Then, we generate C++ code from the 3-layer BIP model. Currently, it is possible to generate C++ implementations using (1) TCP sockets for conventional distributed communication, (2) MPI primitives for multi-processor platforms, and (3) POSIX threads for deployment on multi-core platforms[2].

[1] We note that although the BIP framework includes the notion of priorities, transformations that respect priority rules of the given BIP model are outside the scope of this paper. The issue of priorities has been addressed in [10].

[2] Notice that all code generation schemes are based on a reliable communication mechanism. However, one can use

We also conduct a set of simulations and experiments to analyze the behavior and performance of the generated code for different implementation choices (i.e., different partition of interactions and choice of committee coordination algorithm). Our simulations and experiments show that each implementation choice may be suitable for a different topology, size of the distributed system, communication load, and of course, structure of the initial BIP model.

Organization. In Section 2, we present the global state operational semantics of BIP. We describe our 3-layer model in Section 3. Section 4 is dedicated to detailed description of the transformation from BIP to Send/Receive BIP. In Section 5, we prove correctness of the transformation. Section 6 describes the toolchain implementing the transformation. Sections 7 and 8 present the results of our experiments. Related work is discussed in Section 9. Finally, in Section 10, we make concluding remarks and discuss future work.

2 Basic Semantic Model of BIP

In this section, we present the operational *global state* semantics of BIP [5]. BIP is a component framework for constructing systems by superposing three layers of modeling: Behavior, Interaction, and Priority. In this paper we do not consider priorities. In Subsection 2.1, we formally define atomic components. The notion of composite components is presented in Subsection 2.2.

2.1 Atomic Components

An *atomic component* is described as a *1-Safe Petri net* extended with data. It consists of a set of *places*, a set of *transitions*, and a set of local *variables*. Each transition is labelled by a *port*, a *guard* which is a predicate on local variables, and an *update function*. Ports are used for communication among different components. Each port *exports* a subset of variables of the component.

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

A Petri net is often modeled as a directed bipartite graph $G = (L \cup T, E)$. Places are represented by circular vertices and transitions are represented by rectangular vertices (see Fig. 3). The set of directed edges E is the union of the sets $\{(l, \tau) \in L \times T \mid l \in {}^\bullet\tau\}$ and $\{(\tau, l) \in T \times L \mid l \in \tau^\bullet\}$.

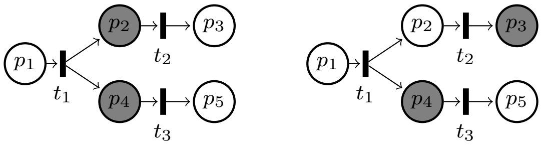


Fig. 3 A simple Petri net

We depict the state of a Petri net by *marking* its places with *tokens*. We say that a place is *marked* if it contains a token. A transition τ can be executed if all its input places in ${}^\bullet\tau$ contain a token and all its output places do not contain a token. Upon the execution of τ, tokens in input places ${}^\bullet\tau$ are removed and tokens in output places in $\tau^\bullet$ are added. Formally, let $\underset{S}{\rightarrow}$ be the set of triples (m, p, m'), such that $\exists \tau = ({}^\bullet\tau, p, \tau^\bullet) \in T$, where ${}^\bullet\tau \subseteq m$ and $m' = (m \backslash {}^\bullet\tau) \cup \tau^\bullet$. The behavior of a Petri net S can be defined as a labeled transition system $(2^L, P, \underset{S}{\rightarrow})$, where 2^L is the set of states, P is the set of labels, and $\underset{S}{\rightarrow}$ is the set of transitions. The reader may find more information about Petri Nets in [29].

Example 1 Fig. 3 shows an example of a Petri net in two successive markings. It has five places $\{p_1, \ldots, p_5\}$ and three transitions $\{t_1, t_2, t_3\}$. The places containing a token are depicted with gray background. The Petri net on the right shows the resulting state of the Petri net on the left after executing transition t_2.

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

- (L, P, T) is a *1-Safe Petri net.*
- X is a set of variables.
- For each port $p \in P$, $X_p \subseteq X$ is the set of variables exported by p (i.e., variables visible from outside the component through port p).
- For each transition $\tau \in T$, g_τ is a predicate defined over X and f_τ is a function that updates the set of variables X.

Example 2 Fig. 4(a) shows an atomic component, where the set of places is $\{s\}$, the set of ports is $\{p\}$, the set of variables is $\{n\}$, the set of transitions is $\{(s, p, s)\}$ with no update function and guard equal to logical true. Variable n is exported by port p.

Given a set X of variables, we denote by $\mathbf{X}$ the set of valuations defined on X. Formally, $\mathbf{X} = \{\sigma : X \rightarrow \mathit{Domain}\}$, where *Domain* is the set of all values possibly taken by variables in X.

committee coordination algorithms that are resilient to faults (e.g., [11, 26]) as well.

Definition 3 The semantics of an atomic component $B = (L,\ P,\ T,\ X,\ \{X_p\}_{p\in P},\ \{g_\tau\}_{\tau\in T},\ \{f_\tau\}_{\tau\in T})$ is defined as the labeled transition system $S_B = (Q_B, P_B, \underset{B}{\longrightarrow})$ where

- $Q_B = 2^L \times \mathbf{X}$, where $\mathbf{X}$ denotes the set of valuations on X.
- $P_B = P \times \mathbf{X}$ denotes the set of labels, that is, ports augmented with valuations of variables.
- $\underset{B}{\longrightarrow}$ is the set of transitions defined as follows. Let (m, v) and (m', v') be two states in $2^L \times \mathbf{X}$, p be a port in P, and v''_p be a valuation in $\mathbf{X}_p$ of X_p. We write $(m, v) \underset{B}{\xrightarrow{p(v''_p)}} (m', v')$, iff $\tau = (m, p, m')$ is a transition of the behavior of the Petri net (L, P, T), $g_\tau(v)$ is true, and $v' = f_\tau(v[X_p \leftarrow v''_p])$, (i.e., v' is obtained by applying f_τ after updating variables X_p exported by p by the values v''_p). In this case, we say that p is *enabled* in state (m, v).

In Definition 3, we introduce an intermediate valuation v''_p to parametrize the transition labeled by the port p. Indeed, executing a transition labeled by p modifies the values of variables in X_p through port p. Modified values are the outcome of an interaction, as defined in Subsection 2.2.

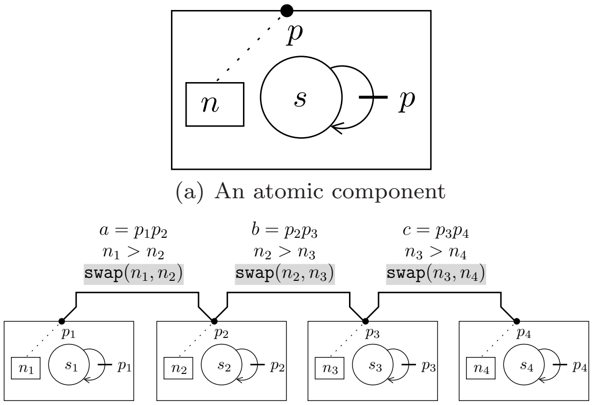
2.2 Composite Components

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

Definition 4 (Interaction) Let $\{B_i = (L_i,\ P_i,\ T_i,\ X_i,\ \{X_p\}_{p\in P_i},\ \{g_\tau\}_{\tau\in T_i},\ \{f_\tau\}_{\tau\in T_i})\}_{i=1}^n$ be a composite component. An *interaction* a between the atomic components is a triple (P_a, G_a, F_a), where

- $P_a \subseteq P$ is a nonempty set of ports, that contains at most one port of every component, that is, $|P_i \cap P_a| \leq 1$, for all i, $1 \leq i \leq n$. We denote by $X_a = \bigcup_{p\in P_a} X_p$ the set of variables available to a.
- $G_a : \mathbf{X_a} \rightarrow \{\mathit{True}, \mathit{False}\}$ is a guard.
- $F_a : \mathbf{X_a} \rightarrow \mathbf{X_a}$ is an update function.

By definition, P_a uses at most one port of every component. Therefore, we simply denote $P_a = \{p_i\}_{i\in I}$, where $I \subseteq \{1 \dots n\}$ contains the indices of the components involved in a and for all $i \in I, p_i \in P_i$. We denote by F_a^i the projection of F_a on X_{p_i}. This projection corresponds to the values transmitted to atomic component B_i as the outcome of the interaction.



(a) An atomic component

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

Fig. 4 Atomic and composite components in BIP

Definition 5 (Composite Component) We denote by $B \overset{def}{=} \gamma(B_1, \dots, B_n)$ the composite component obtained by applying a set of interactions γ to the set of atomic components $B_1, \dots, B_n$.

Definition 6 Let $\gamma(B_1, \dots, B_n)$ be a composite component, where $B_i = (L_i, P_i, T_i, X_i, \{X_p\}_{p\in P_i}, \{g_\tau\}_{\tau\in T_i}, \{f_\tau\}_{\tau\in T_i})_{i=1}^n$ and $S_{B_i} = (Q_i, P_i, \underset{B_i}{\longrightarrow})$ is the semantics of B_i. The semantics of $\gamma(B_1, \cdots, B_n)$ is a transition system $(Q, \gamma, \underset{\gamma}{\longrightarrow})$, where $Q = \Pi_{i=1}^n Q_i$, and $\underset{\gamma}{\longrightarrow}$ is the least set of transitions satisfying the rule:

$$\frac{\begin{array}{c} a = (\{p_i\}_{i\in I}, G_a, F_a) \in \gamma \\ G_a(\{v_{p_i}\}_{i\in I}) \qquad \forall i \notin I.\ (m_i, v_i) = (m'_i, v'_i) \\ \forall i \in I.\ (m_i, v_i) \underset{B_i}{\xrightarrow{p_i(v''_{p_i})}} (m'_i, v'_i), v''_{p_i} = F_a^i(\{v_{p_i}\}_{i\in I}) \end{array}}{((m_1, v_1), \dots, (m_n, v_n)) \underset{\gamma}{\xrightarrow{a}} ((m'_1, v'_1), \dots, (m'_n, v'_n))}$$

where for each $i \in I$, v_{p_i} denotes the valuation v_i restricted to the variables of X_{p_i}. Note that $\{v_{p_i}\}_{i\in I}$ defines a valuation on X_a and therefore we may apply the guard and the update function to it.

This rule says that a composite component $B = \gamma(B_1, \dots, B_n)$ can execute an interaction $a \in \gamma$ *enabled* in state $((m_1, v_1), \dots, (m_n, v_n))$, iff (1) for each $p_i \in P_a$, 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 on the variables exported by the ports participating in interaction a. Execution of interaction a triggers the function F_a which modifies the variables of the components exported by ports p_i. The new values obtained, encoded in the valuation v''_{p_i}, are then processed by the components' transitions. The state of a component that does not participate in the interaction remains unchanged. We say that an interaction $a \in \gamma$ is *enabled* at state $q \in Q$ if there exists state $q' \in Q$ such that $q \xrightarrow[\gamma]{a} q'$.

Example 3 Fig. 4(b) illustrates a composite component $\gamma(B_1, \cdots, B_4)$, where each B_i is identical to component B in Fig. 4(a). The set γ of interactions is $\{a, b, c\}$, where $a = (\{p_1, p_2\}, n_1 > n_2, \texttt{swap}(n_1, n_2))$ and function `swap` swaps the values of its arguments. Interactions b and c are defined in a similar fashion, that is, $b = (\{p_2, p_3\}, n_2 > n_3, \texttt{swap}(n_2, n_3))$ and $c = (\{p_3, p_4\}, n_3 > n_4, \texttt{swap}(n_3, n_4))$. 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 values of $n_1 \cdots n_4$, so that n_1 contains the smallest value and n_4 contains the largest value.

3 The 3-Layer Architecture

In this section, we describe the overall architecture of the source-to-source transformation in BIP. Since we target a distributed setting, we assume concurrent execution of interactions. However, if two interactions are simultaneously enabled, they cannot always run in parallel without breaking the semantics of the initial global state model. This leads to the notion of *conflict* between interactions. Intuitively, two interactions are conflicting if they share a port or they are using conflicting ports of the same component.

Definition 7 Let $\gamma(B_1, \ldots, B_n)$ be a BIP model. We say that two interactions $a_1 = (P_1, G_1, F_1)$, $a_2 = (P_2, G_2, F_2)$, where $a_1, a_2 \in \gamma$, are *conflicting* iff either:

- they share a common port p; i.e., $p \in P_1 \cap P_2$, or
- there exists in an atomic component $B_i = (L_i, P_i, T_i, X_i, \{X_p\}_{p \in P_i}, \{g_\tau\}_{\tau \in T_i}, \{f_\tau\}_{\tau \in T_i})$, a location $l \in L_i$, two transitions $\tau_1 = ({}^\bullet\tau_1, p_1, \tau_1^\bullet), \tau_2 = ({}^\bullet\tau_2, p_2, \tau_2^\bullet) \in T$, and two ports $p_1, p_2 \in P_i$, such that (1) $p_1 \in P_1$, (2) $p_2 \in P_2$, and (3) $l \in {}^\bullet\tau_1 \cap {}^\bullet\tau_2$.

As discussed in the introduction, handling conflicting interactions in a BIP model executed by a centralized Engine is quite straightforward. However, in a distributed setting, detecting and avoiding conflicts is not trivial. Thus, our target transformed BIP model should have the following three properties:

1. preserving the behavior of each atomic component,
2. preserving the behavior of interactions modulo some observation criterion, and
3. resolving conflicts in a distributed manner.

The target BIP model is designed based on the three properties identified above. It consists of three layers ensuring the three properties respectively. Moreover, we require that interactions in the target model are of the form *Send/Receive* with one sender and multiple receivers. Such interactions can be implemented by using conventional communication primitives (e.g., TCP sockets, MPI primitives, or shared memory). Finally, we provide generic and minimal interfaces for the third layer for conflict resolution, which can be implemented using existing distributed algorithms.

Definition 8 We say that $B^{SR} = \gamma^{SR}(B_1^{SR}, \ldots, B_n^{SR})$ is a *Send/Receive* BIP composite component iff we can partition the set of ports of B^{SR} into three sets P_s, P_r, and P_u that are respectively the sets of *send-ports*, *receive-ports*, and *unary interaction ports*, such that:

- Each interaction $a = (P_a, G_a, F_a) \in \gamma^{SR}$, is either (1) a Send/Receive interaction with $P_a = (s, r_1, r_2, \ldots, r_k)$, $s \in P_s$, $r_1, \ldots, r_k \in P_r$, $G_a = true$ and F_a copies the variables exported by port s to the variables exported by ports $r_1, r_2, \ldots, r_k$, or, (2) a unary interaction $P_a = \{p\}$ with $p \in P_u$, $G_a = true$, F_a is the identity function.
- If s is a port in P_s, then there exists one and only one Send/Receive interaction $a = (P_a, G_a, F_a) \in \gamma^{SR}$ with $P_a = (s, r_1, r_2, \ldots, r_k)$ and all ports $r_1, \ldots, r_k$ are receive-ports. We say that $r_1, r_2, \ldots, r_k$ are the receive-ports associated to s.
- If $a = (P_a, G_a, F_a)$ with $P_a = (s, r_1, \ldots, r_k)$ is a Send/Receive interaction in γ^{SR} and s is enabled at some global state of B^{SR}, then all its associated receive-ports $r_1, \ldots, r_k$ are also enabled at that state.

Definition 8 defines a class of BIP models for distributed implementation based on asynchronous message passing. In such systems, communication is sender-triggered, in the sense that a message is emitted by the sender, regardless the availability of receivers. The third property of the definition, requires that all receivers are ready to receive whenever the sender may send a message. This ensures that the sender is never blocked and triggers the Send/Receive interaction.

We use the BIP model in Fig. 1 as a running example throughout the paper to describe the concepts of our transformation. We assume that interaction a_1 is in conflict with only interaction a_2, and, interactions a_2, a_3, and a_4 are in pairwise conflict. In all Figures depicting Send/Receive BIP components we represent send-ports using triangles and receive-ports using bullets. Our architecture consists of the following three

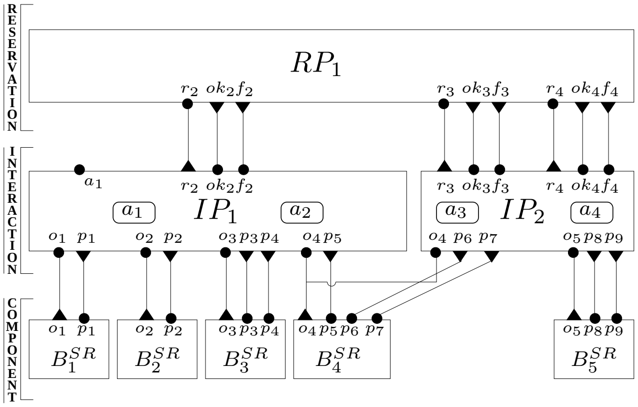


Fig. 5 3-layer model of Fig. 1.

layers.

Components Layer. Atomic components in the given BIP model are placed in this layer with the following additional ports per component. The send-port o (stands for *offer*) sends to the upper layer messages containing the list of enabled ports in the component. Also, for each port p in the original component, we include a receive-port p through which the component is notified from the upper layer to execute a transition labeled by p. In Fig. 5, the bottom layer depicts the modified components illustrated in Fig. 1.

Interaction Protocol Layer. This layer consists of a set of components, each in charge of execution of a set of interactions in the initial BIP model. Assignment of interactions to components is defined by a partition of interactions, where each class is handled by a different component in this layer. Conflicts between interactions executed by the same component are resolved by that component locally. For instance, interactions a_1 and a_2 (resp. a_3 and a_4) of Fig. 1 are grouped into component IP_1 (resp. component IP_2) in Fig. 5. Thus, conflicts between a_1 and a_2 (resp. a_3 and a_4) are handled locally in IP_1 (resp. IP_2). On the contrary, conflicts between a_2 and either a_3 or a_4 have to be resolved using an external algorithm that solves the committee coordination problem. Such an algorithm is implemented by the upper layer of our model. An Interaction Protocol also evaluates the guard and executes the update function associated with an interaction that is selected locally or by the upper layer. The interface between this layer and the component layer provides ports for receiving *offers* from each component and sending *responses* to the components on permitted ports for execution.

Reservation Protocol Layer. This layer accommodates an algorithm that solves the *committee coordination problem* [15]. This problem is as follows:

> *"Professors in a certain university have organized themselves into committees. Each committee has an unchanging membership roster of one or more professors. From time to time a professor may decide to attend a committee meeting; it starts waiting and remains waiting until a meeting of a committee of which it is a member is started. All meetings terminate in finite time. The restrictions on convening a meeting are as follows: (1) meeting of a committee may be started only if all members of that committee are waiting, and (2) no two committees may convene simultaneously, if they have a common member. The problem is to ensure that (3) if all members of a committee are waiting, then a meeting involving some member of this committee is convened."*

It is straightforward to observe that the committee coordination problem can resolve conflicts in our models, where professors represent components and committees represent interactions. For instance, the external conflicts between interactions a_2 and a_3, and, interactions a_2 and a_4 are resolved by this layer (component RP_1) in Fig. 5. We emphasize that this layer is fully determined by the conflict resolution algorithm chosen and the externally conflicting interactions. Incorporating a centralized algorithm results in one component RP_1 as illustrated in Fig. 5. Other algorithms (as will be discussed in Subsection 4.3), such as circulating token [3] or dining philosophers [4, 15] result in different structures. The interface between this layer and the Interaction Protocol involves ports for receiving request to reserve an interaction (labeled r for reserve) and responding by either success (labeled ok) or failure (labeled f for fail).

We note that when multiple interactions are enabled, the choice for executing interactions is nondeterministic. This non-determinism depends on the order of messages travelling in an actual distributed system as well as the choice of algorithm for solving the committee coordination problem.

4 Transforming BIP into 3-Layer Send/Receive-BIP

In this section, we describe the method for automated transformation of a BIP model into a 3-layer *Send/Receive*-BIP model in detail. This transformation takes a BIP model, a partition of its interactions, and a committee coordination algorithm as input, and generates another BIP model as described in Section 3. Specifically,

- atomic components in the input BIP model are automatically transformed into atomic Send/Receive components (see Subsection 4.1),
- the partition of interactions, determines the components in the Interaction Protocol layer and the interface between these components and the Atomic Components layer (see Subsection 4.2),
- the choice of the input committee coordination algorithm determines the Reservation Protocol layer and its interface with the Interaction Protocol (see Subsection 4.3).

All these transformations are fully automated. Furthermore, the interaction partition and the committee coordination algorithm determine the level of distribution of the transformed model. For instance, the architecture in Fig. 5 results in a distributed implementation, where one centralized component manages all conflicts. Notice that, an interaction cannot be handled by two components in the Interaction Protocol. That is, only one component executes the interaction to avoid inconsistencies. Hence, in general, obtaining a fully distributed implementation such as the one shown in Fig. 2(c) is not trivial. However, if an interaction is used only for the purpose of synchronization (i.e., the interaction has an identity update function), then a fully distributed solution is possible using committee coordination algorithms that allow full distribution (e.g., [11, 26]).

4.1 Atomic Components Layer

For the sake of simplicity and clarity, we present the transformation for atomic components such that Petri nets are finite state automata. That is, each transition has a single source place and a single target place.

We transform an atomic component B of a BIP model into a Send/Receive atomic component B^{SR} that is capable of communicating with the Interaction Protocol in the 3-layer model. As mentioned in Section 3, B^{SR} sends *offers* to the Interaction Protocol that are acknowledged by a *response*. An offer includes the set of enabled ports of B^{SR} at the current state through which the component is ready to interact. For each port p of the transformed component B^{SR}, we introduce a Boolean variable x_p. This variable is modified by a *port update* function when reaching a new state: the variable x_p is then set to true if the corresponding port p becomes enabled, and to false otherwise. When the upper layer selects an interaction involving B^{SR} for execution, B^{SR} is notified by a response sent on the port chosen. We also include in B^{SR} a *participation number* variable n which counts the number of interactions that B^{SR} has participated in. This number is used by the Reservation Protocol to ensure conflict resolution. That is, the number of times that from each state a component has taken part in an interaction.

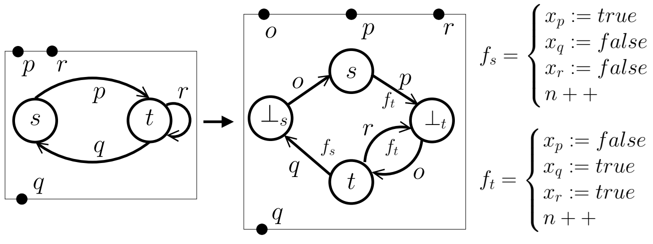

p
r
p
r
s
t
q
q
o
p
r
o
s
p
f_t
$\perp_s$
$\perp_t$
r
f_s
f_t
q
t
o
q
$f_s = \begin{cases} x_p := true \\ x_q := false \\ x_r := false \\ n++ \end{cases}$
$f_t = \begin{cases} x_p := false \\ x_q := true \\ x_r := true \\ n++ \end{cases}$

Fig. 6 Transformation of atomic component.

Since each notification from the Interaction Protocol triggers an internal computation in a component, following [5], we split each place s into two places, namely, s itself and a *busy place* $\perp_s$. Intuitively, reaching $\perp_s$ marks the beginning of an unobservable internal computation. We are now ready to define the transformation from B into B^{SR}.

Definition 9 Let $B = (L, P, T, X, \{X_p\}_{p\in P}, \{g_\tau\}_{\tau\in T}, \{f_\tau\}_{\tau\in T})$ be an atomic component. The corresponding Send/Receive atomic component is $B^{SR} = (L^{SR}, P^{SR}, T^{SR}, X^{SR}, \{X_p^{SR}\}_{p\in P}, \{g_\tau\}_{\tau\in T^{SR}}, \{f_\tau\}_{\tau\in T^{SR}})$ with the additional variables X, such that:

- $L^{SR} = L \cup L^{\perp}$, where $L^{\perp} = \{\perp_s \mid s \in L\}$.
- $X^{SR} = X \cup \{x_p\}_{p\in P} \cup \{n\}$, where each x_p is a new Boolean variable, and n an integer called *participation number*.
- $P^{SR} = P \cup \{o\}$, where the *offer* port o exports the variables $X_o^{SR} = \{n\} \cup \bigcup_{p\in P}(\{x_p\} \cup X_p)$, that is the participation number, the new Boolean variables and the variables associated to each port. For all other ports $p \in P$, we define $X_p^{SR} = X_p$.
- For each location $s \in L$, we include an offer transition $\tau_s = (\perp_s, o, s)$ in T^{SR}. The guard g_{τ_s} is true and the update function f_{τ_s} is the identity function.
- For each transition $\tau = (s, p, t) \in T$, we include a response transition $\tau_p = (s, p, \perp_t)$ in T^{SR}. The guard g_{τ_p} is true. The function f_{τ_p} first applies the original update function f_τ, then increments n and finally updates the Boolean variables:

$$\text{for all } r \in P \quad x_r := \begin{cases} g_{\tau'} & \text{if } \exists \tau' = (t, r, t') \in T \\ \textit{false} & \text{otherwise} \end{cases}$$

Fig. 6 illustrates the transformation of a component into its corresponding Send/Receive component.

4.2 Interaction Protocol Layer

Consider a composite component $B = \gamma(B_1 \cdots B_n)$ and a partition of the set of interactions $\gamma = \gamma_1 \cup \ldots \cup \gamma_m$.

This partition allows the designer to enforce load-balancing and improve the performance of the given model when running in a distributed fashion. It also determines whether or not a conflict between interactions can be resolved locally. We associate each class γ_j of interactions to an Interaction Protocol component IP_j that is responsible for (1) detecting enabledness by collecting *offers* from the components layer, (2) selecting one non-conflicting interaction (either locally or assisted by the Reservation Protocol), and (3) executing the selected interaction in γ_j and sending responses to the corresponding atomic components. For instance, in Fig. 5, we have two classes: $\gamma_1 = \{a_1, a_2\}$ (handled by component IP_1) and $\gamma_2 = \{a_3, a_4\}$ (handled by component IP_2).

Since components of the Interaction Protocol deal with interactions of the original model, they need to be aware of conflicts in the original model as defined in Definition 7. We distinguish two types of conflicting interactions according to a given partition:

- *External:* Two interactions are externally conflicting if they conflict and they belong to different classes of the partition. External conflicts are resolved by the Reservation Protocol. For instance, in Fig. 5, interaction a_2 is in external conflict with interactions a_3 and a_4.
- *Internal:* Two interactions are internally conflicting if they conflict, but they belong to the same class of the partition. Internal conflicts are resolved by the Interaction Protocol within the component that handles them. For instance, in Fig. 5, interaction a_1 is in internal conflict with interaction a_2. If component IP_1 chooses interaction a_1 over a_2, no further action is required. On the contrary, if IP_1 chooses a_2, then it has to request its reservation from RP_1, as it is in external conflict with a_3 and a_4.

The Petri net that defines the behavior of an Interaction Protocol component IP_j handling a class γ_j of interactions is constructed as follows. Fig. 7 illustrates the construction of the Petri net of component IP_1 in Fig. 5.

Places. The Petri net has three types of places:

- For each component B_i involved in interactions of γ_j, we include *waiting* and *received* places w_i and rcv_i, respectively. IP_j remains in a waiting place until it receives an offer from the corresponding component. When an offer from component B_i is received (along with the fresh values of its variables), IP_j moves from w_i to rcv_i. In Fig. 7, since components $B_1 \cdots B_4$ are involved in interactions handled by IP_1 (i.e., a_1 and a_2), we include waiting places $w_1 \cdots w_4$ and received places $rcv_1 \cdots rcv_4$.
- For each port p involved in interactions of γ_j, we include a *sending* place snd_p. The response to an offer with $x_p = true$ is sent from this place to port p of the component that has made the offer. In Fig. 7, places $snd_{p_1} \cdots snd_{p_5}$ correspond to ports $p_1 \cdots p_5$ respectively, as they form interactions hosted by IP_1 (i.e., a_1 and a_2).
- For each interaction $a \in \gamma_j$ that is in external conflict with another interaction, we include an *engaged* place e_a and a *free* place fr_a. In Fig. 7, only interaction a_2 is in external conflict, for which we add places e_{a_2} and fr_{a_2}.

Variables and ports. For each port p involved in interactions of γ_j, we include a Boolean variable x_p and a local copy of the variables X_p exported by p. Values of these variables are updated whenever an offer is received from the corresponding component. Also, for each component B_i involved in an interaction of γ_j, we include an integer n_i that stores the participation number of B_i. The set of ports of IP_j is the following:

- For each component B_i involved in an interaction of γ_j, we include a receive-port o_i, to receive offers. Each port o_i exports the variables n_i, x_p and the variables X_p associated to each port p of B_i. In Fig. 7, ports $o_1 \cdots o_4$ represent offer ports for components $B_1 \cdots B_4$.
- For each port p involved in interactions of γ_j, we include a send-port p, which exports the set of variables X_p. In Fig. 7, ports $p_1 \cdots p_5$ correspond to the ports that form interactions a_1 and a_2.
- For each interaction $a \in \gamma_j$ that is in external conflict with some other interaction, we include ports r_a (reserve a, send-port), ok_a (success in reserving a, receive-port), and f_a (failure to reserve a, receive-port). If $a = \{p_i\}_{i \in I}$, the port r_a exports the variables $\{n_i\}_{i \in I}$, where I is the set of indices of components involved in interaction a. In Fig. 7, ports r_{a_2}, ok_{a_2}, and f_{a_2} are used for solving the external conflict of a_2 with interactions a_3 and a_4.
- For each interaction $a \in \gamma_j$ that is not in external conflict, we include a unary port a. In Fig. 7, we include unary port a_1, as a_1 is only in internal conflict with a_2.

Transitions. IP_j performs two tasks: (1) receiving offers from components in the lower layer and responding to them, and (2) requesting reservation of an interaction from the Reservation Protocol in case of an external conflict. The following set of transitions of IP_j performs these two tasks:

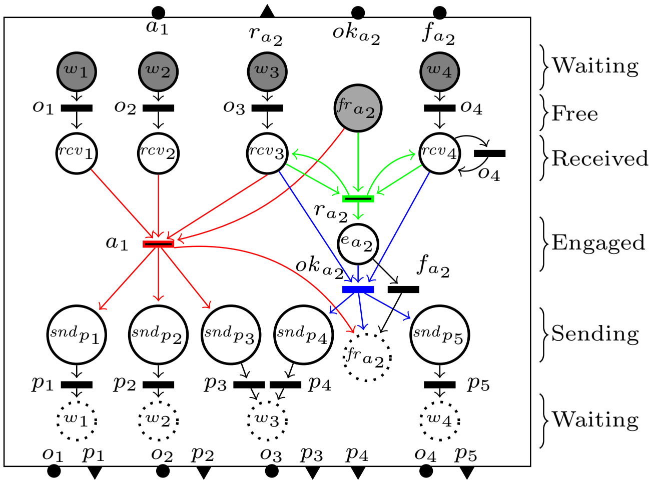


Fig. 7 Component IP_1 in Fig. 5.

- In order to receive offers from a component B_i, we include transition (w_i, o_i, rcv_i). If B_i participates in an interaction not handled by IP_j, we also include transition (rcv_i, o_i, rcv_i) to receive new offers when B_i takes part in such an interaction. Transitions labeled by $o_1 \cdots o_4$ in Fig. 7 are of this type.
- Requesting reservation of an interaction $a \in \gamma_j$ that is in external conflict is accomplished by transition $(\{rcv_i\}_{i \in I} \cup \{fr_a\}, r_a, \{rcv_i\}_{i \in I} \cup \{e_a\})$, where I is the set of components involved in interaction a. This transition is guarded by the predicate $\bigwedge_{i \in I} x_{p_i} \wedge G_a$ which ensures enabledness of a. Notice that this transition is enabled when the token for each participating component is in its corresponding receive place rcv_i and the guard G_a of the interaction a is true. Execution of this transition results in moving the token from the free place fr_a to the engaged place e_a. In Fig. 7, transition r_{a_2} is of this type, and is guarded by $x_{p_4} \wedge x_{p_5}$.
- For the case where the Reservation Protocol responds positively, we include the transition $(\{rcv_i\}_{i \in I} \cup \{e_a\}, ok_a, \{snd_{p_i}\}_{i \in I} \cup \{fr_a\})$. The execution of this transition triggers the function F_a of the interaction a, and then, the token in the engaged place moves to the free place and the tokens in received places move to sending places to inform the corresponding components. Transition ok_{a_2} in Fig. 7, occurs when interaction a_2 is successfully reserved by the Reservation Protocol.
- For the case where the Reservation Protocol responds negatively, we include the transition (e_a, f_a, fr_a). Upon execution of this transition, the token moves from the engaged place to the free place. Transition f_{a_2} in Fig. 7, occurs when the Reservation Protocol fails to reserve interaction a_2 for component IP_1.
- For each interaction $a = \{p_i\}_{i \in I}$ in γ_j that has only internal conflicts, let A be the set of interactions that are in internal conflict with a, and are externally conflicting with other interactions. We include the transition $(\{rcv_i\}_{i \in I} \cup \{fr_{a'}\}_{a' \in A}, a, \{snd_{p_i}\}_{i \in I} \cup \{fr_{a'}\}_{a' \in A})$. This transition is guarded by the predicate $\bigwedge_{i \in I} x_{p_i} \wedge G_a$ and moves the tokens from receiving to sending places. Tokens at $fr_{a'}$ places ensure that no internally conflicting interaction requested a reservation. This transition triggers the function F_a. The transition labeled by a_1 in Fig. 7 falls in this category.
- Finally, for each component B_i exporting p, we include a transition (snd_p, p, w_i). This transition notifies component B_i to execute the transition labeled by port p. These are transitions labeled by $p_1 \cdots p_5$ in Fig. 7.

4.3 Reservation Protocol Layer

The Reservation Protocol ensures that externally conflicting interactions are executed in mutual exclusion. This can be ensured by employing an algorithm that solves the committee coordination problem. Our design characterizes the Reservation Protocol by its generic interface with the Interaction Protocol and thus allows employing different algorithms with minimal restrictions.

We adapt the message-count technique from [4]. This technique is based on counting the number of times that a component interacts. This number is recorded as the participation number n, in every atomic component. The Reservation Protocol ensures that, for a fixed value of the participation number, each component takes part in only one interaction. To this end, for each component B_i, the Reservation Protocol has a variable N_i which stores the latest value of the participation number n_i of B_i. Whenever a reserve message r_a for interaction $a = \{p_i\}_{i \in I}$ is received by the Reservation Protocol, the message provides a set of participation numbers $(\{n_i^a\}_{i \in I})$ for all components involved in a. If for each component B_i, the participation number n_i^a is greater than N_i, then the Reservation Protocol acknowledges successful reservation through port ok_a and the participation numbers in the Reservation Protocol are set to values sent by the Interaction Protocol. On the contrary, if there exists a component whose participation number is less than or equal to what Reservation Protocol has recorded, then the corresponding component has already participated for this number and the Reservation Protocol replies failure via port f_a.

Since the structure and behavior of the Reservation Protocol components depend on the employed commit-

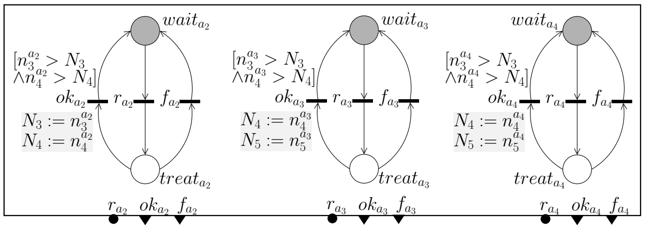


Fig. 8 A centralized Reservation Protocol for Fig. 5.

tee coordination algorithm, we only specify an abstract set of minimal restrictions of this layer as follows:

- For each component B_i connected to an interaction with external conflicts, the Reservation Protocol maintains a variable N_i indicating the last participation number reserved for B_i.
- For each interaction $a = \{p_i\}_{i \in I}$ handled by the Reservation Protocol, we include three ports: r_a, ok_a and f_a. The receive-port r_a accepts reservation requests containing fresh values of variables n_i^a. The send-ports ok_a and f_a accept or reject the latest reservation request, and the variables N_i are incremented in case of positive response.
- Each r_a message should be acknowledged by exactly one ok_a or f_a message.
- Each component of the Reservation Protocol should respect the message-count properties described above.

4.3.1 Centralized Implementation

Fig. 8 shows a centralized Reservation Protocol for the model in Fig. 5 (i.e., component RP_1 in Fig. 5). A reservation request, for instance, r_{a_2}, contains fresh value of variables $n_3^{a_2}$ and $n_4^{a_2}$ (corresponding to components B_3 and B_4). The token representing interaction a_2 is then moved from place $wait_{a_2}$ to place $treat_{a_2}$. From this place, the Reservation Protocol can still receive a request for reserving a_3 and a_4 since $wait_{a_3}$ and $wait_{a_4}$ still contain a token. This is where message-counts play their role. The guard of transition ok_{a_2} is $(n_3^{a_2} > N_3) \wedge (n_4^{a_2} > N_4)$ where N_i is the last known used participation number for B_i. Note that since execution of transitions is atomic in BIP, if transition ok_{a_2} is fired, it modifies variables N_i before any other transition can take place. We denote this implementation by RP.

4.3.2 Token Ring Implementation

Another example of a Reservation Protocol is inspired by the token-based algorithm due to Bagrodia [3], where we add one reservation component for each externally conflicting interaction. We denote by TR_a the component that corresponds to interaction a. Fig. 9 shows the components of a token-based reservation protocol for the model presented in Fig. 5 and depicts the behavior of component TR_{a_3}. Mutual exclusion is ensured using a circulating token carrying N_i variables. Initially, all components are in place *wait*, except the one that owns the token which is in place *ready*. From place *wait*, a component waits to receive a reservation request, and then, it moves to place *req*. From this place, it is possible to respond to the reservation request by a *fail* message before receiving the token, if there exists a variable n_i whose value is less or equal to the last known value of the corresponding N_i. This is correct, since the value of N_i can only increase. Otherwise, the component waits to receive the token. After receiving the token (i.e., place *token*), the component compares the values of n_i variables with the values of N_i variables from the token. If $n_i > N_i$ for all i, then an *ok* message is sent to the component that handles the requested interaction and increases the values of the variables N_i. Otherwise, a *fail* message is sent. Subsequently, the reservation component releases the token via port ST, which is received by the next component via port RT. Obviously, this algorithm allows a better degree of parallelism at the reservation protocol layer. We denote this implementation, that is the set of components TR_a for all a that are externally conflicting, by TR.

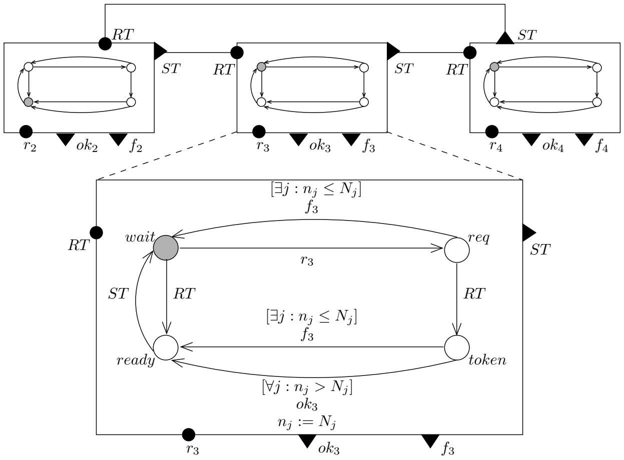


Fig. 9 Token-based Reservation Protocol for the BIP models in Fig. 1 and Fig. 5.

4.3.3 Implementation Based on Dining Philosophers

A third choice of Reservation Protocol algorithm is an adaptation of the hygienic solution to the Dining Philosophers problem presented in [4, 15]. In this problem, a number of philosophers are placed at the ver-

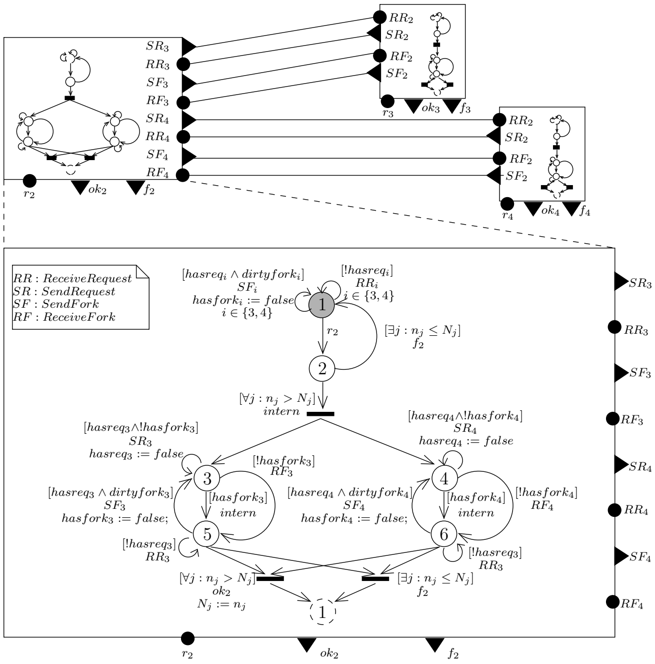

SR_3
RR_3
SF_3
RF_3
SR_4
RR_4
SF_4
RF_4
r_2
ok_2
f_2
RR_2
SR_2
RF_2
SF_2
r_3
ok_3
f_3
RR_2
SR_2
RF_2
SF_2
r_4
ok_4
f_4
$RR : ReceiveRequest$
$SR : SendRequest$
$SF : SendFork$
$RF : ReceiveFork$
$[hasreq_i \wedge dirtyfork_i]$ SF_i $hasfork_i := false$ $i \in \{3,4\}$
$[!hasreq_i]$ RR_i $i \in \{3,4\}$
1
r_2
$[\exists j : n_j \leq N_j]$ f_2
2
$[\forall j : n_j > N_j]$ $intern$
$[hasreq_3 \wedge !hasfork_3]$ SR_3 $hasreq_3 := false$
$[hasreq_4 \wedge !hasfork_4]$ SR_4 $hasreq_4 := false$
$[!hasfork_3]$ RF_3
3
4
$[hasreq_3 \wedge dirtyfork_3]$ SF_3 $hasfork_3 := false;$
$[hasfork_3]$ $intern$
$[hasreq_4 \wedge dirtyfork_4]$ SF_4 $hasfork_4 := false;$
$[hasfork_4]$ $intern$
$[!hasfork_4]$ RF_4
5
6
$[!hasreq_3]$ RR_3
$[!hasreq_3]$ RR_3
$[\forall j : n_j > N_j]$ ok_2 $N_j := n_j$
$[\exists j : n_j \leq N_j]$ f_2
1
SR_3
RR_3
SF_3
RF_3
SR_4
RR_4
SF_4
RF_4
r_2
ok_2
f_2

Fig. 10 Dining philosophers-based Reservation Protocol for the BIP models in Fig. 1 and Fig. 5.

tices of a finite undirected graph with one philosopher at each vertex. An edge between two vertices models adjacent philosophers. A philosopher is in one of the following three states: (1) *thinking*: the philosopher is not requesting to eat (i.e., using a critical resource); (2) *hungry*: the philosopher is requesting to eat; or (3) *eating*: the philosopher is eating (i.e., is in critical section). Philosophers can eat for only a finite amount of time; i.e., they cannot remain in the critical section indefinitely. Adjacent philosophers share a common fork that represents the shared critical resource. A philosopher can eat if she has all the forks from its neighbors. Solutions to the Dining Philosophers problem ensure the following properties: (1) *mutual exclusion*: adjacent philosophers do not eat simultaneously; (2) *fairness*: every hungry philosopher eventually eats; (3) *symmetry*: all philosophers execute the same code (they only differ by their initial places); (4) *concurrency*: non-neighbors can eat simultaneously; and (5) *boundedness*: the number of messages that are in transit between any pair of philosophers, as well as their size, are bounded.

The corresponding Send/Receive BIP implementation is presented in Fig. 10. Similarly to the token ring implementation, each externally conflicting interaction a is handled by a separate component DP_a. If two interactions are conflicting, the two corresponding components share a fork carrying N_i variables corresponding to the atomic components causing the conflict. A fork is either *clean* or *dirty*. Initially, all forks are dirty and are located around philosophers. We construct a precedence graph H, where philosophers are vertices and the direction of arcs of this graph are determined as follows. The direction of an arc between two philosophers u and v is from u to v (i.e., u has precedence over v) if and only if one of the following conditions hold: (1) u holds the fork shared by u and v, and the fork is clean, (2) v holds the fork and the fork is dirty, or (3) the fork is in transit from v to u. In practice, this can be done, by giving a unique identifier to each philosopher and if two philosophers u and v share a fork, then the philosopher with the higher identifier retains the fork initially. Obviously, H is acyclic by construction. A fork gets cleaned only when a philosopher is willing to release it to another philosopher. Moreover, we introduce a *request* for each fork. A philosopher that intends to acquire a fork f, needs to send the corresponding request to the philosopher that currently holds the fork. Initially, every fork f and request for f are held by different philosophers (i.e., $hasfork_i \oplus hasreq_i = true$, where $\oplus$ denotes the exclusive-or operator).

In Fig. 10, initially a component is in place 1 (thinking). From this place, the component either receives a fork request, sends a fork, or receives a reservation request. After receiving a reservation request (r_2), it moves to place 2 (hungry). From this place, first, the component compares participation numbers n_i received from the reservation request with its corresponding numbers N_i. If there exists a participation number n_i in the reservation request which is less than or equal to its current number N_i, the component responds *fail* directly before receiving all the forks. This is due to the fact that N_i can only increase. Otherwise, the component has to acquire all the forks shared with its neighbors. Thus, the state of the component moves to places 3 and 4. Note that, in general, for each conflicting interaction, the Petri net needs to have two places for negotiating the corresponding fork (here, locations 3 and 5 are used for negotiation with component DP_{a_3} and places 4 and 6 are used for negotiation with component DP_{a_4}). From place 3, the component moves to place 5, if it has the fork. Otherwise, it sends a request for the corresponding fork and waits for its reception. After receiving all the forks (locations 5 and 6), the component compares participation numbers (n_i) received from the reservation request with its current numbers (N_i) and responds accordingly. After responding, the forks become dirty. Moreover, at these places, it is possible to receive request in order to release a dirty fork if the philosopher is asked to do so. We denote this implementation, that is the set of components DP_a for all a that are externally conflicting, by DP.

4.4 Cross-Layer Interactions

In this subsection, we define the interactions of our 3-layer model. Following Definition 8, we introduce Send/Receive interactions by specifying only the sender. Given a BIP model $\gamma(B_1 \cdots B_n)$, a partition $\gamma_1 \cdots \gamma_m$ of γ, the transformation gives a Send/Receive BIP model $\gamma^{SR}(B_1^{SR}, \ldots, B_n^{SR}, IP_1, \ldots, IP_m, X)$ where X can be any set of the Reservation Protocol components RP, TR, or DP. We define the Send/Receive interactions of γ^{SR} as follows:

- For each component B_i^{SR}, let $IP_{j_1}, \ldots, IP_{j_l}$ be the Interaction Protocol components handling interactions involving B_i^{SR}. We include in γ^{SR} the *offer interaction* $(B_i^{SR}.o, IP_{j_1}.o_i, \ldots, IP_{j_l}.o_i)$.
- For each port p in component B_i^{SR} and for each Interaction Protocol component IP_j handling an interaction involving p, we include in γ^{SR} the *response interaction* $(IP_j.p, B_i^{SR}.p)$.
- For each Interaction Protocol component IP_j handling an interaction a that is in external conflict, we include in γ^{SR} the *reserve interaction* $(IP_j.r_a, X.r_a)$. Likewise, we include in γ^{SR} the *ok interaction* $(X.ok_a, IP_j.ok_a)$ and the *fail interaction* $(X.f_a, IP_j.f_a)$.

Since the interface of the Reservation Protocol is the same for the 3 versions, they all have the same interaction with the Interaction Protocol layer. The transformed model obtained in that manner is denoted by B_{RP}^{SR}, B_{TR}^{SR} or B_{DP}^{SR} depending on the embedded Reservation Protocol. The interactions between the three layers of our running example are depicted in Fig. 5.

5 Correctness

In Subsection 5.1, we show that the 3-layer model is indeed a Send/Receive model as defined in Section 3. In Subsection 5.2, we prove that the initial high-level BIP model is observationally equivalent to the Send/Receive BIP model obtained by the transformation of Section 4. Finally, we prove correctness of models embedding different implementations of Reservation Protocol in Subsection 5.3.

5.1 Compliance with Send-Receive Model

We need to show that receive-ports of B_{RP}^{SR} are enabled whenever the corresponding send-ports are enabled. This holds since communications between two successive layers follow a request/acknowledgement pattern. Whenever a layer sends a request, it enables the receive-port to receive acknowledgement and no new request is sent until the first one is acknowledged.

Proposition 1 *Given a BIP model B, the model B_{RP}^{SR} obtained by transformation of Section 4 meets the properties of Definition 8.*

Proof The send-ports and receive-ports determined in subsection 4.4 respect the syntax presented in the two first properties of Definition 8. We now prove the third property, that is whenever a send-port is enabled, all its associated receive-ports are enabled.

Between Interaction Protocol and Reservation Protocol layers, for reserve, ok and fail interactions related to $a \in \gamma$ it is sufficient to consider places fr_a and e_a in the Interaction Protocol layer, $wait_a$ and $treat_a$ in the Reservation Protocol layer. Initially, the configuration is $(fr_a, wait_a)$ from which only the send-port r_a in Interaction Protocol might be enabled, and the receive-port r_a is enabled. If the r_a interaction takes place, we reach the configuration $(e_a, treat_a)$, in which only send-ports ok_a and f_a in Reservation Protocol might be enabled, and the associated receive-ports in Interaction Protocol are enabled. Then, if either an ok or a fail interaction takes place, we switch back to the initial configuration.

Between components and Interaction Protocol layers, for all interactions involving a component B_i, it is sufficient to consider only the places w_i, r_i and s_p for each port p exported from B_i to the Interaction Protocol. Whenever one of the places w_i or r_i is enabled in each Interaction Protocol component, the property holds for the o_i interaction. In this configuration, no place s_p can be active since this would require one of the tokens from a w_i or a r_i, thus no send port p is enabled.

If there is an Interaction Protocol component such that the token associated to B_i is in a place s_p, it comes either from a transition labelled by a or ok_a. In the first case, no other interaction involving B_i can take place; otherwise, it would be externally conflicting with a. In the second case, according to the Reservation Protocol, ok_a is given for the current participation number in the component B_i and no other interaction using this number will be granted. Thus in all cases, there is only one active place s_p with p exported by B_i. The response can then take place and let the components continue their execution. □

This proof ensures that any component ready to perform a transition labeled by a send-port will not be blocked by waiting for the corresponding receive-ports. In other terms, it proves that any Send/Receive interaction is initiated by the sender. This proof can be adapted for B^{SR}_{TR} and B^{SR}_{DP} by remarking that in both cases each reservation interaction is acknowledged by exactly one interaction (either ok or fail).

5.2 Observational Equivalence between Original and Transformed BIP Models

We recall the definition of *observational equivalence* of two transition systems $A = (Q_A, P \cup \{\beta\}, \rightarrow_A)$ and $B = (Q_B, P \cup \{\beta\}, \rightarrow_B)$. It is based on the usual definition of weak bisimilarity [28], where β-transitions are considered unobservable. The same definition is trivially extended for atomic and composite BIP components.

Definition 10 (Weak Simulation) A *weak simulation* from A to B, denoted $A \subset B$, is a relation $R \subseteq Q_A \times Q_B$, such that $\forall (q,r) \in R,\ a \in P :$ $q \xrightarrow{a}_A q' \implies \exists r' :\ (q',r') \in R \wedge r \overset{\beta^* a \beta^*}{\rightarrow}_B r'$ and $\forall (q,r) \in R :\ q \xrightarrow{\beta}_A q' \implies \exists r' :\ (q',r') \in R \wedge r \xrightarrow{\beta^*}_B r'$

A weak bisimulation over A and B is a relation R such that R and R^{-1} are both weak simulations. We say that A and B are *observationally equivalent* and we write $A \sim B$ if for each state of A there is a weakly bisimilar state of B and conversely. In this subsection, our goal is to show that B and B^{SR}_{RP} are observationally equivalent. We consider the correspondence between actions of B and B^{SR}_{RP} as follows. To each interaction $a \in \gamma$ of B, we associate either the binary interaction ok_a or the unary interaction a of B^{SR}_{RP}, depending upon the existence of an external conflict. All other interactions of B^{SR}_{RP} (offer, response, reserve, fail) are unobservable and denoted by β.

We proceed as follows to complete the proof of observational equivalence. Amongst unobservable actions β, we distinguish between β_1 actions, that are communication interactions between the atomic components layer and the Interaction Protocol (namely offer and response), and β_2 actions that are communications between the Interaction Protocol and Reservation Protocol (namely reserve and fail). We denote by q^{SR} a state of B^{SR}_{RP} and q a state of B. A state of B^{SR}_{RP} from where no β_1 action is possible is called a *stable state*, in the sense that any β action from this state does not change the state of the atomic components layer.

Lemma 1 *From any state q^{SR}, there exists a unique stable state $[q]^{SR}$ such that $q^{SR} \xrightarrow{\beta_1^*} [q]^{SR}$.*

Proof The state $[q]^{SR}$ exists since each Send/Receive component B^{SR}_i can do at most two β_1 transitions: receive a response and send an offer. Since two β_1 transitions involving two different components are independent (i.e. do not change the same variable or the same place), the same final state is reached independently of the order of execution of β_1 actions. Thus $[q]^{SR}$ is unique. □

The above lemma proves the existence of a well-defined stable state for any of the transient states reachable by the distributed model B^{SR}_{RP}. This stable state will be used later to define our observational equivalence. We now show a property of the participation numbers. Let $B.n$ mean ‘the variable n that belongs to component B’.

Lemma 2 *When B^{SR}_{RP} is in a stable state, for each pair (i,j), such that B_i is involved in interactions handled by IP_j, we have $B^{SR}_i.n_i = IP_j.n_i > RP.N_i$.*

Proof At a stable state, all the offers have been sent, thus the participation numbers in Interaction Protocol are equal to the corresponding participation numbers of the components: $B^{SR}_i.n_i = IP_j.n_i$.

Initially, for each component B_i, $RP.N_i = 0$ and $B^{SR}_i.n_i = 1$ thus the property holds. The variables N_i in Reservation Protocol are updated upon execution of ok_a transition, using values provided by the Interaction Protocol, that is by the components. Thus, in the unstable state reached immediately after a ok_a transition, we have $B^{SR}_i.n_i = RP.N_i$ for each component B^{SR}_i participant in a. Then, the response transition increments participation numbers in components so that in the next stable state $B^{SR}_i.n_i > RP.N_i$. For components $B_{i'}$ not participating in a, by induction we have $B^{SR}_i.n_{i'} > RP.N_{i'}$ and only participation numbers in atomic components can be incremented. □

Lemma 2 shows that the participation numbers propagate in a correct manner. In particular, at any stable state the reservation protocol has only previously used values and all components of the interaction protocol layer have the freshest values, that is the same as in the atomic components.

Since we need to take into account participation numbers n_i, we introduce an intermediate high-level model B^n. This new model is a copy of B that includes in each atomic component an additional variable n_i which is incremented whenever a transition is executed. As B and B^n have identical sets of states and transitions labeled by the same ports, they are observationally equivalent. (They are even strongly bisimilar.)

Lemma 3 $B \sim B^n$.

Proof We say that two states (q, q^n) of B and B^n are equivalent if they have the same state when removing participation numbers from B^n. Since these participation numbers do not change the behavior of the model (i.e. they only count the number of transitions), the above equivalence defines a bisimulation. □

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

Proposition 2 $B_{RP}^{SR} \sim B^n$.

Proof We define a relation R between the set of states Q^{SR} of B_{RP}^{SR} and the set of states Q of B^n as follows: $R = \{(q^{SR}, q) \mid \forall i \in I : [q]_i^{SR} = q_i\}$ where q_i denotes the state of B_i^n at state q and $[q]_i^{SR}$ denotes the state of B_i^{SR} at state $[q]^{SR}$. The three next assertions prove that R is a weak bisimulation:

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

(i) If $q^{SR} \xrightarrow{\beta} r^{SR}$, either β is a β_1 action and $[q]^{SR} = [r]^{SR}$, or β is a β_2 action which does not change the state of component layer and does not enable any send-port.

(ii) The action a in B_{RP}^{SR} is either a unary interaction a or a binary interaction ok_a. In both cases, $a = \{p_i\}_{i \in I}$ has been detected to be enabled in IP_j by the tokens in received places and the guard of the transitions labeled by a or ok_a in Interaction Protocol. We show that a is also enabled at state $[q]^{SR}$:

- If a has only local conflicts, no move involving B_i can take place in another Interaction Protocol, and no β_1 move involving B_i can take place in IP_j, since a is enabled.
- If a is externally conflicting, no move involving B_i has taken place in another Interaction Protocol (otherwise ok_a would not have been enabled), nor in IP_j since the place fr_a is empty.

At the stable state $[q]^{SR}$, Lemma 2 ensures that $IP_j.n_i = B_i^{SR}.n_i$. Following the definition of R, we have $B_i.n_i = B_i^{SR}.n_i$ when B^n is at state q. Thus a is enabled with the same participation numbers at state q and in IP_j at states q^{SR} and $[q]^{SR}$, which implies $q \xrightarrow{a}$.

Since interactions β_1 lead to state $[q]^{SR}$, they do not interfere with action a. We can replay them from r^{SR} to reach a state r'^{SR}, as shown on Fig. 11. State r'^{SR} is not stable because of β_1 actions that take place in components participating in a (that is, receiving response and sending offer). Executing these actions brings the system to state $[r']^{SR}$ which is equivalent to r, and by point (i) we have $(r^{SR}, r) \in R$.

(iii) In Fig. 12, we show the different actions and states involved in this condition. From q^{SR}, we reach $[q]^{SR}$ by doing β_1 actions. Then we execute all possible fail interactions (these are β_2 actions), so that all fr_a places are empty, to reach a state $[q']^{SR}$. At this state, if a has only local conflicts, interaction a is enabled; else the sequence r_a ok_a can be executed since Lemma 2 ensures that the guard of ok_a is true. In both cases, the interaction corresponding to a brings the system to state r^{SR}. From this state, the responses corresponding to each port of a are enabled, and the next stable state $[r]^{SR}$ is equivalent to r, thus $(r^{SR}, r) \in R$. □

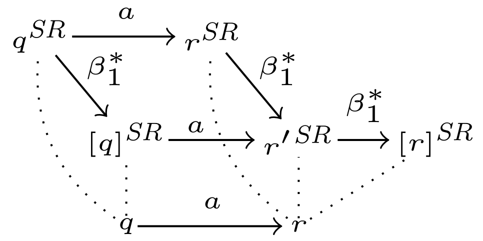


Fig. 11 Proof of observational equivalence – point (ii)

5.3 Interoperability of Reservation Protocol

As mentioned in Subsection 4.3, the centralized implementation RP of the Reservation Protocol can be seen as a specification. Recall that, we chose two other implementations in Subsection 4.3, respectively, token-ring TR and dining philosophers DP that can be embeded in our framework as reservation protocol as well. However, these implementations are not observationally equivalent to the centralized implementation. More precisely, the centralized version defines the most liberal implementation: if two reservation requests a_1 and a_2 are received, the protocol may or may not acknowledge them, in a specific order. This general behavior is not implemented neither by the token ring nor by the dining philosophers implementations, which are focused on ensuring progress. In the case of token ring, the response may depend on the order the token travels through the components. In the case of dining philosophers, the order may depend on places and the current status of forks.

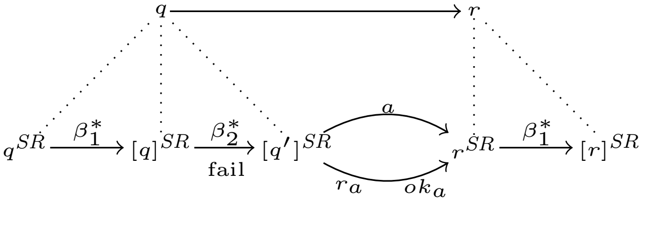


Fig. 12 Proof of observational equivalence – point (iii)

Nevertheless, we can prove observational equivalence if we consider *weaker* versions of the above implementations. More precisely, for the token ring protocol, consider the weaker version $TR^{(w)}$ which allows to release the token or provide a fail answer regardless of the values of counters. Likewise, for the dining philosophers protocol, consider the weaker version $DP^{(w)}$, where forks can always be sent to neighbors, regardless of their status and the values of counters. Clearly, a weakened Reservation Protocol is not desirable for a concrete implementation since it do not enforce progress. But, it is an artifact for proving the correctness of our approach. The following proposition establishes the relation between the different implementations of the Reservation Protocol.

Proposition 3 $RP \sim TR^{(w)} \sim DP^{(w)}$

Proof The observable actions are request, ok and fail messages, namely r_a, ok_a and f_a. The unobservable actions are token passings for $TR^{(w)}$ and forks exchange for $DP^{(w)}$.

Given a state s^{TR} of the TR protocol or a state s^{DP} of the DP protocol, we construct a state s^{RP} for the centralized protocol as follows. For each interaction $a \in \gamma$:

- If a request for a is pending in s^{TR} or s^{DP}, the equivalent state s^{RP} is defined such that the place $treat_a$ contains a token. Otherwise, the place $wait_a$ contains a token.
- For each component B_i involved in a, we set the participation number n_i^a associated to the pending requests in s^{RP} to the value of n_i held in the component managing interaction a, that is TR_a or RP_a.

Moreover, we set the last used participation number N_i in s^{RP} to the value of variable N_i stored on the token in s^{TR} or to the value of variable N_i stored on the forks in s^{DP}.

In $TR^{(w)}$ and $DP^{(w)}$, it is clear that any unobservable action does not change the associated state s^{RP}.

Weakening makes the above relation an observational equivalence. Indeed, whenever an action (either ok_a, f_a or r_a) is possible at s^{RP}, then by moving the token, (resp. the forks), we can always reach a state s^{TR} (respectively s^{DP}) where this action is possible as well. Reciprocally, if an action is possible in either s^{TR} or s^{DP} then, in the equivalent state s^{RP}, the same action is allowed. □

Recall that we denote B_X^{SR} the 3-layer model obtained from the initial system B and that embeds the Reservation Protocol X, which ranges over RP, TR and DP. Also, let us denote by $Tr(B)$ the set of all possible traces of observable actions allowed by an execution of B. We now show the correctness of the implementation, by using the weak implementations of TR and DP. Since the real implementations of these protocols restrict the behaviour of their weak version, we only state correctness of our implementation with respect to the original model, by showing that the traces of our implementation are included in the traces of the original model.

Proposition 4 *(i)* $B \sim B_{RP}^{SR} \sim B_{TR^{(w)}}^{SR} \sim B_{DP^{(w)}}^{SR}$ *(ii)* $Tr(B) \supseteq Tr(B_{TR}^{SR})$ *and* $Tr(B) \supseteq Tr(B_{DP}^{SR})$.

Proof (i) The leftmost equivalence is a consequence of Lemma 3 and Proposition 2. The other equivalences come from Proposition 3 and the fact that observational equivalence is a congruence with respect to parallel composition. (ii) Trace inclusions come from the fact that any trace of TR (respectively DP) is also a trace of $TR^{(w)}$ (respectively $DP^{(w)}$). □

6 BIP into Distributed Implementations: The BIP2Dist Tool-Chain

We have implemented the presented transformations in the BIP2Dist tool[3]. Fig. 13 illustrates the complete design flow for generating different types of distributed implementations from a given BIP model. The tool is written in *Java*. In this section, we discuss the design choices and features of the BIP2Dist.

1. Starting from a hierarchical BIP model, first we *flatten* [13] the hierarchy of components and connectors. Hierarchical models are normally obtained using incremental addition of components and interactions to an existing model, while respecting the operational semantics presented in Section 2. Component flattening replaces the hierarchy on components by a set of hierarchically structured connectors applied on atomic components. An example of hierarchically structured connector is shown in Fig. 27. Connector flattening computes for each hierarchically structured connector an equivalent flat connector.
2. From the flattened model that consists only of atomic components and flat connectors, we generate a 3-layer Send/Receive BIP model by choosing a user-specified partition of interactions and a Reservation Protocol.
3. From the 3-layer Send/Receive BIP model, a designer may *merge* sets of components implemented on the same processor. Such merging (described

[3] Information about the tool is provided at `http://www-verimag.imag.fr/dbip`.

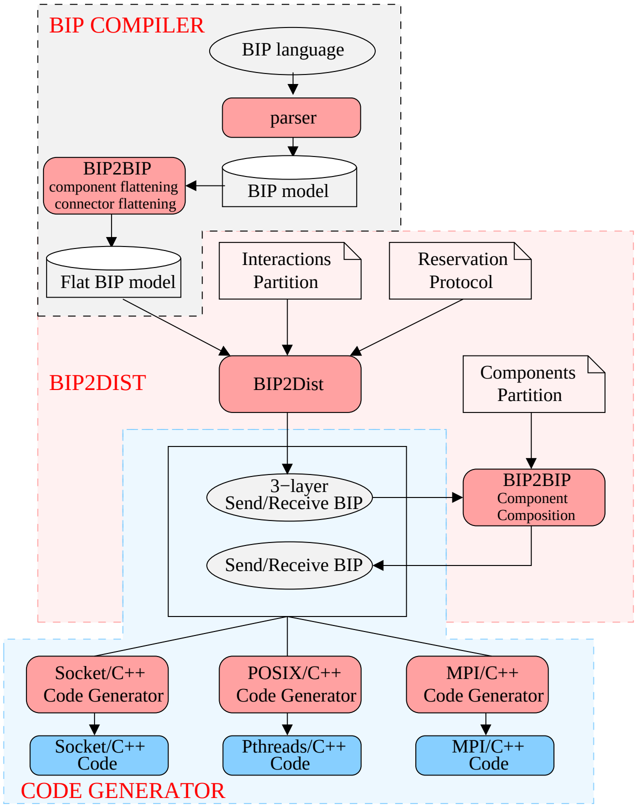


Fig. 13 BIP2Dist toolset: General architecture

in Subsection 6.1) is defined by a mapping from Send/Receive components into processors.

4. Finally, from the obtained model, we generate C++ code by employing communication mechanisms supported by the following platforms: (1) TCP sockets, (2) POSIX (shared memory) threads, or (3) the Message Passing Interface (MPI) (described in Subsection 6.2).

6.1 Merging Components

Given a user-defined partition of components of the 3-layer Send/Receive model, merging consists in generating for each block of the partition a single component which is strongly bisimilar to their product. Component merging is formally defined in [12]. Merging components allows significant performance gains. For instance, when we generate MPI code, significant overhead is due to context switching between processes and merging can potentially avoid such context switches. Fig. 14 shows the Send/Receive model obtained by merging components RP_1 and IP_2 and components IP_1 and B_1^{SR}.

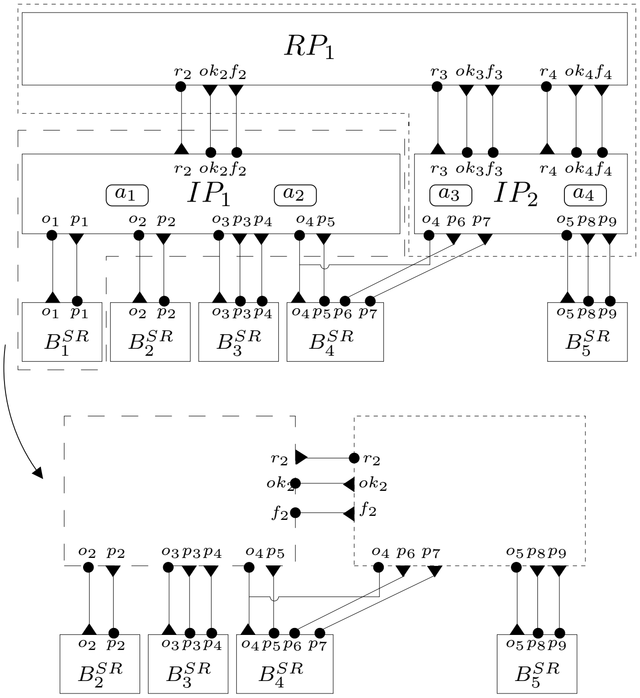


Fig. 14 Merging together components B_1^{SR} and IP_1, and RP_1 and IP_2.

6.2 Transformation from Send/Receive BIP into C++

We describe the principle of generation of C++ code from a Send/Receive BIP component. The code structure is depicted in Fig. 15 and involves the following steps.

First, we initialize connections with respect to the platform communication primitives library (Line 2). For instance, in case of TCP sockets, this step establishes connection-oriented stream sockets between components that need to send and receive messages to each other. We assign one Boolean variable to each place of Petri net of the Send/Receive component. The value of these variables shows whether or not each place contains a token. Thus, the initial state of the Petri net is determined by an initial assignment of these variables (Line 3).

The code scans the list of all possible transitions and gives priority to transitions that are labeled by a send-port (Lines 6-10) or unary ports of the given Petri net (Lines 12-16). Actual emission of data is performed by an invocation of the function `send()` in Line 7. Once data transmission or an internal computation is completed, tokens are removed from input places and put to output places of the corresponding transitions (Lines 8 and 14).

Finally, if no send-port is enabled and all internal computations are complete, then execution stops and waits for messages from other components (Line 18).

```
 1: // Initialization
 2: InitializeConnections();
 3: PrepareInitialState();

 4: while true do
 5:    // Send messages
 6:    if there exists an enabled send-port then
 7:       send(...);
 8:       PrepareNextState();
 9:       goto line 4;
10:    end if

11:    // Internal computation
12:    if there exists an enabled unary port then
13:       DoInternalComputation();
14:       PrepareNextState();
15:       goto line 4;
16:    end if

17:    // Receive messages
18:    select(...);
19:    recv(...);
20:    PrepareNextState();
21: end while
```

Fig. 15 Code for execution of Send/Receive Petri nets

Once a message is received, the component resumes its execution (Line 19).

This code generation scheme gives priority to send actions over unary (local computation) and receiving actions. Sending messages before doing internal computation triggers components waiting for a response so as to preserve parallelism. Regarding the TCP implementation, we use the send(), select(), and receive() primitives for sending, waiting, and receiving messages respectively. Regarding the MPI implementation, we use the following communication primitives MPI_Send(), MPI_ISend(), MPI_Recv(), and MPI_IRecv(). The logic of MPI- and socket-based implementations follows exactly the presented principle for code generation.

We provide a multi-threaded implementation as well. We use shared memory, mutexes, and semaphores offered by the POSIX library for implementing send, receive, and select primitives. More precisely, for each atomic component, we create a shared-memory FIFO buffer, a semaphore, and a mutex. The communication primitives used in the code are implemented as follows:

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

7 Simulations

We conducted simulations to study the impact of different choices of reservation protocol and partition of interactions. We emphasize that we distinguish between simulations (results in this section) and an experiments (results in Section 8). In distributed systems, the execution of a task or network communication may take a considerable amount of time depending on the underlying platform. Thus, we provide simulations by adding communication delays and computation times to take into account the dynamics of different target platforms. Unlike simulations, all parameters and results in the experiments are determined by real platform characteristics.

We denote each simulation scenario by (i, X), where i is the number of Interaction Protocol components and X is one among the three Reservation Protocols described in Subsection 4.3 (i.e., *RP*, *TR*, or *DP*). For the cases where partition of interactions results in having no external conflicts and, hence, requires no conflict resolution, we use the symbol '$-$' to denote absence of Reservation Protocol. The scenarios considered in this Section are referred to as 'simulations', because we consider a large number of processes running on a limited number of stand-alone machines. Thus, we model communication delays by temporarily suspending communicating processes.

All simulations are conducted on five quad-Xeon 2.6 GHz machines with 6GB RAM running under Debian Linux and connected via a 100Mbps Ethernet network. Our aim is to show that different conflict resolution algorithms and partitions may result in significantly different performance. In Subsection 7.1, we present simulation results for a distributed diffusing computation algorithm. In Subsection 7.2, we describe results for a distributed transportation system.

7.1 Diffusing Computation

We model a simplified version of Dijkstra-Scholten termination detection algorithm for diffusing computations [17] in BIP. *Diffusing computation* consists in propagating a message across a distributed system; i.e., a wave starts from an initial node and diffuses to all processes in a distributed system. Diffusing computation has numerous applications including traditional distributed deadlock detection and reprogramming of

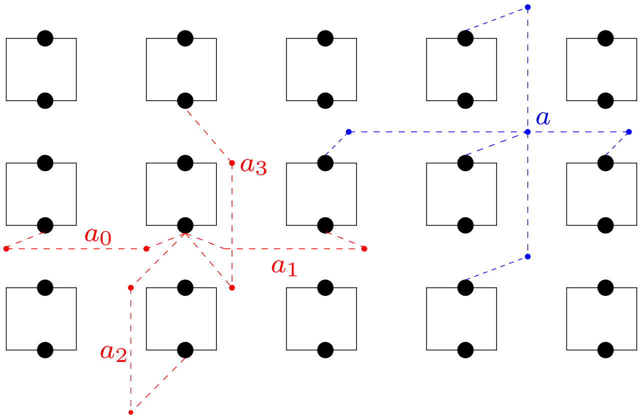
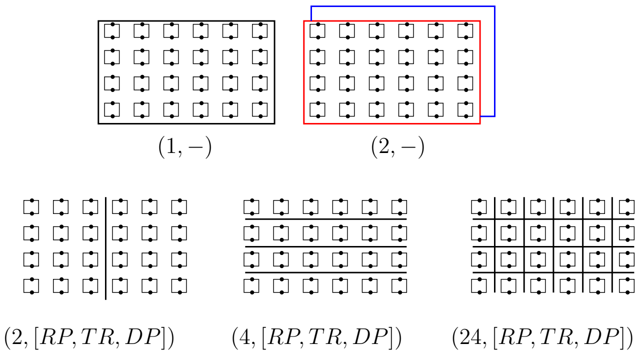


Fig. 16 Partial BIP model for diffusing computations.


$(1, -)$ $(2, -)$

$(2, [RP, TR, DP])$ $(4, [RP, TR, DP])$ $(24, [RP, TR, DP])$

Fig. 17 Different scenarios for diffusing computations.

modern sensor networks. One challenge in diffusing computation is to detect its termination. In our version, we consider a torus (wrapped around grid) topology for a set of distributed processes, where a spanning tree throughout the distributed system already exists. Each process has a unique parent and the root process is its own parent. Termination detection is achieved in two phases: (1) the root of the spanning tree possesses a message and initiates a *propagation wave*, so that each process sends the message to its children; and (2) once the first wave of messages reaches the leaves of the tree, a *completion wave* starts, where a parent completes once all its children have completed. When the root has completed, termination is detected.

The BIP model has $n \times m$ atomic components (see Fig. 16 for a partial model). Each component participates in two types of interactions: (1) four binary rendezvous interactions (e.g., $a_0 \cdots a_3$) to propagate the message to its children (as in a torus topology, each node has four neighbors and, hence, potentially four children), and (2) one 5-ary rendezvous interaction (e.g., a) for the completion wave, as each parent has to wait for all its children to complete. Finally, in order to make our simulations realistic, we require that execution of each interaction involves $10ms$ suspension of the corresponding component in the Interaction Protocol.

7.1.1 Influence of Partition

Our first set of simulations is for a 6×4 torus. We used different partitions as illustrated in Fig. 17. Fig. 18 shows the time needed for 100 rounds for detecting termination of diffusing communication for each scenario. In the first two scenarios, the interactions are partitioned, so that all conflicts are internal and, hence, resolved locally by the Interaction Protocol. In case $(2, -)$, all interactions of the propagation wave are grouped into one component of the Interaction Protocol and all interactions related to the completion wave are grouped into the second component. Such grouping does not allow parallel execution of interactions. This is the main reason for poor performance of $(1, -)$ and $(2, -)$ shown in Fig. 18.

Other scenarios group all interactions involved in components $1 \cdots 12$ into one component and the remaining interactions in a second component of the Interaction Protocol. These provide simulations $(2, RP)$, $(2, TR)$, and $(2, DP)$. Such partitions allow more parallelism during propagation and completion waves, as an interaction in the first class can be executed in parallel with an interaction in the second class. Recall that execution of each interaction involves $10ms$ of suspension of the corresponding component in the Interaction Protocol. This is why $(2, RP/TR/DP)$ outperforms $(1, -)$ and $(2, -)$. As almost all propagation interactions conflict with each other and so do all completion interactions, the conflict graph is dense. Hence, to make a decision within DP, each philosopher needs to grab a high number of forks, which entails a lot of communication. Thus, the performance of $(2, TR)$ is slightly better than $(2, DP)$. It can also be seen that $(2, RP)$ performs as well as $(2, TR)$ and $(2, DP)$. This is due to the fact that we have only two classes, which results in a low number of reservation requests.

Fig. 18 also shows the same type of simulations for 4 and 24 partition classes. As for two partitions, TR and RP for 4 and 24 partition classes have comparable performance. However, RP and TR outperform DP. This is due to the fact that for DP, each philosopher needs to acquire the forks corresponding to all conflicting interactions, which requires a considerable amount of communication. On the contrary, TR does not require as much communication, as the only task it has to do is releasing and acquiring the token. Moreover, the level of parallelism in DP for a 6×4 torus is not high enough to cope with the communication volume.

Following our observations regarding tradeoff between communication volume and parallelism, we design a scenario illustrating the advantages of DP. Recall that each component in DP resolves conflicts through communication involving only its neighboring components. This is not the case for TR, since the token has to travel through a large number of components. For a 20×20 torus, as can be seen in Fig. 19, DP outperforms TR. This is solely because, in TR, the token travels along the ring of components and enables interactions sequentially. On the contrary, in DP, the Reservation Protocol components act in their local neighborhood and although more communication is needed, a higher level of concurrency is possible and, hence, more interactions can be enabled simultaneously. We expect for increasing size of the torus, DP outperforms RP as well.

7.1.2 Influence of Communication Delays and Execution Times

We now study the influence of communication delays and execution times of functions attached to interactions. We simulate different environments by adding execution times and communication delays. The idea is to provide some guidelines on whether one should use a coarse or fine grain partition and which protocol to choose, depending on the application software and the architecture.

Note that adding execution times within atomic components will slow down the system regardless of the partition and committee coordination algorithm. The response time to a notification is the time needed for the reception of the notification, parallel execution of transitions in atomic components and emission of offer messages. Since we have parallelism between components, the response time to a notification is determined by the response time of the slowest component and does not depend on the partition of interactions or the reservation protocol used. Therefore, we do not model execution times on atomic components transitions, nor the communication delay between components layer and interaction protocol layer.

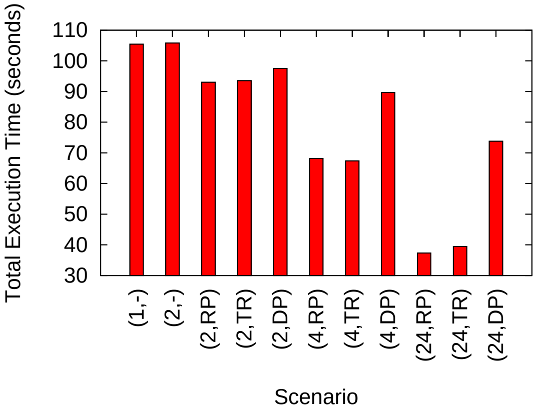


Fig. 18 Performance of termination detection in diffusing computation in different scenarios for torus 6×4.

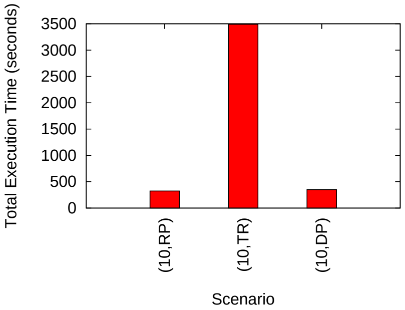


Fig. 19 Performance of termination detection in diffusing computation in different scenarios for torus 20×20.

We consider three different environments, each of them defined by the following parameters:

- t_{inter} is the execution time for an interaction.
- $t_{IP \leftrightarrow RP}$ is the communication delay between the Interaction Protocol and Reservation Protocol layers.
- $t_{RP \leftrightarrow RP}$ is the communication delay between components inside the Reservation Protocol layer.

In the first environment, we assume $t_{inter} = 10ms$ for an interaction execution time, as for the previous simulation, and no communications delay ($t_{IP \leftrightarrow RP} = t_{RP \leftrightarrow RP} = 0ms$). In the second environment, we still assume the same execution time and we add a delay of $t_{IP \leftrightarrow RP} = 10ms$ for communication between Interaction Protocol and Reservation Protocol layers. In the third environment, we assume slower processors with $t_{inter} = 100ms$ interaction execution time. Furthermore, we assume $t_{IP \leftrightarrow RP} = 10ms$ for communications between Interaction Protocol and Reservation Protocol and $t_{RP \leftrightarrow RP} = 1ms$ for communications inside the Reservation Protocol.

For each of these environments, we executed a different scenario of diffusing computation built on a 5×5 grid. We used three different partitions of the 5×5 torus: a centralized one, a partition with 5 Interaction Protocol components (similar to the one with 4 Interaction Protocol components depicted in Fig. 17), and the fully decentralized one, with 25 Interaction Protocol components. The total execution time of these scenarios in the three environments described above, are shown in Fig. 20, Fig. 21, and Fig. 22 respectively.

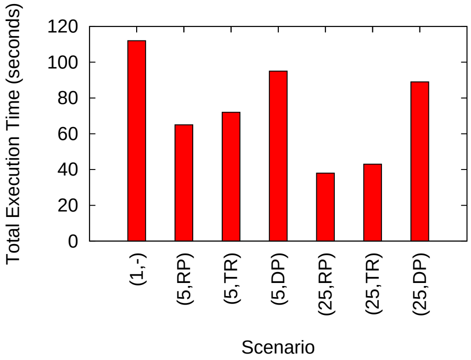


Fig. 20 Simulation of a diffusing computation on a 5 × 5 torus. $t_{inter} = 10ms$, $t_{IP \leftrightarrow RP} = 0ms$, $t_{RP \leftrightarrow RP} = 0ms$

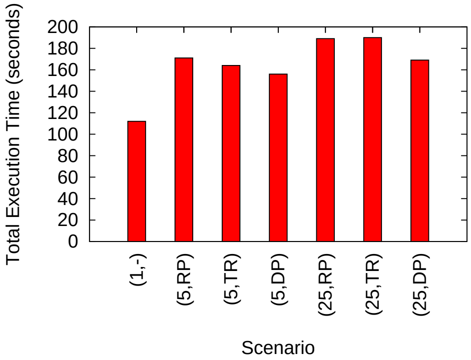


Fig. 21 Simulation of a diffusing computation on a 5 × 5 torus, $t_{inter} = 10ms$, $t_{IP \leftrightarrow RP} = 10ms$, $t_{RP \leftrightarrow RP} = 0ms$

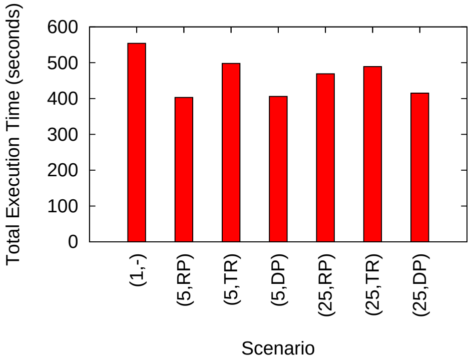


Fig. 22 Simulation of a diffusing computation on a 5 × 5 torus, $t_{inter} = 100ms$, $t_{IP \leftrightarrow RP} = 10ms$, $t_{RP \leftrightarrow RP} = 1ms$

Notice that in the first environment, the best performance is achieved for the most decentralized partition, as in Fig. 18, since there are no communication delays. In the second environment, where we made the assumption that one communication and one interaction execution require the same time, the best performance is obtained for the centralized solution. In this case, the communication between interaction protocol and reservation protocol is too expensive compared to the speedup obtained by having interactions running in parallel. Finally, in the third environment, where the communication costs 10 times less than executing an interaction, the solution with 5 interaction protocol components gives the best performance, except for the token ring protocol. Indeed, the bottleneck is the time needed for the token to cycle through all reservation protocol components. Having more interaction protocol components allows to have more pending reservations and thus diminishing the time between two granted reservations. Increasing the communication time $t_{RP \leftrightarrow RP}$ (experiments not shown in this paper) penalizes token ring much more than dining philosophers.

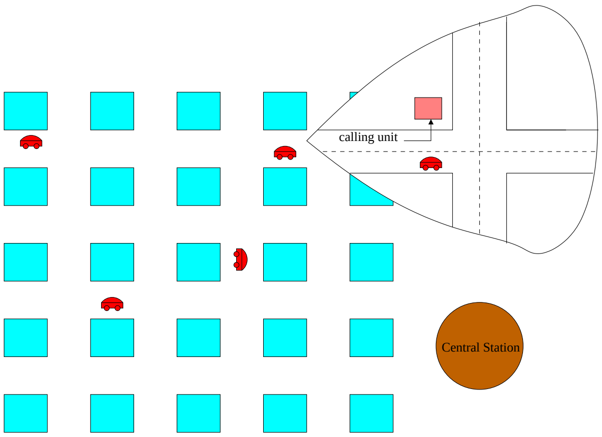


Fig. 23 Utopar transportation system.

7.2 Utopar Transportation System

Utopar is an industrial case study proposed in the context of the European Integrated Project SPEEDS[4]. Utopar is an automated transportation system managing requests for transportation. The system consists of a set of autonomous vehicles, called U-cars, a centralized automatic control (Central-Station), and calling units (see Fig. 23).

We model a simplified version of Utopar in BIP. The overall system architecture is depicted in Fig. 24. It is a composition of an arbitrary (but fixed) number of components of three different types: U-Cars, Calling-Units, and Central-Station. The Utopar system interacts with external users (passengers). Users are also represented as components, however, their behavior is not explicitly modeled.

The overall behavior of the system is obtained by composing the behavior of the components using the following set of interactions:

[4] http://www.speeds.eu.com/

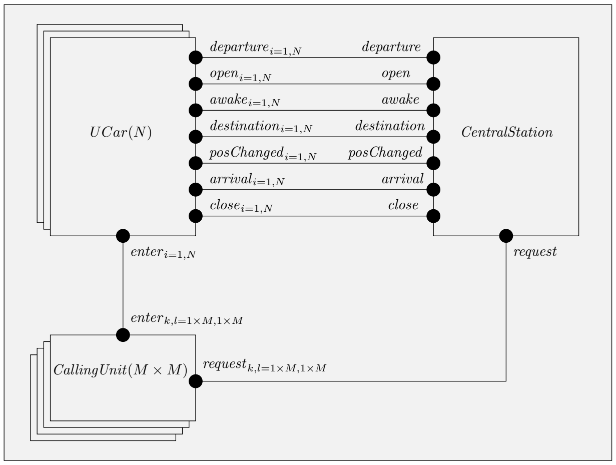


Fig. 24 A BIP model for Utopar system.

- *awake*: handling awake calls of cars by Central-Station;
- *request*: handling car requests by users at Calling-Units;
- *destination*: handling destination requests by users seating within U-Cars;
- *enter*: handling the step on (resp. off) of users into U-Cars;
- *departure*: handling departure commands issued by Central-Station towards U-Cars;
- *posChanged*, *arrival*: information provided by moving U-Cars towards Central-Station;
- *open*, *close*: handling the opening/closing of U-Cars doors, while parked at Calling-Units.

Our first set of simulations consists of $25 = 5 \times 5$ calling units and 4 cars. For each calling unit, we group all the interactions it is involved, in one Interaction Protocol component. Moreover, for each car, we group all the interactions connecting the car with CentralStation in the same Interaction Protocol component. Thus, we obtain 29 Interaction Protocol components. Using this partition, we generate the corresponding 3-layer Send/Receive model for the three Reservation Protocols. We simulate the target platform as follows. We consider that there exists a machine on each calling unit. Moreover, each machine is connected to its four neighbours and the communication time between two neighbour machines is 1ms.

We generate the corresponding C++ executables as follows. We merge each calling unit with its associated Interaction Protocol component and map the resulting code into the machine located on the calling unit. In a similar way, we merge each car with its associated Interaction Protocol component and map the resulting code into an arbitrary machine, such that two different cars are located in different machines. Regarding CentralStation, we map its code into the central (midmost) machine. Regarding the Reservation Protocol, in the case of RP, the best choice is to map its code into the central machine as well. In the case of TR and DP protocols, we map the code of each component in the same machine as the Interaction Protocol component communicating with it.

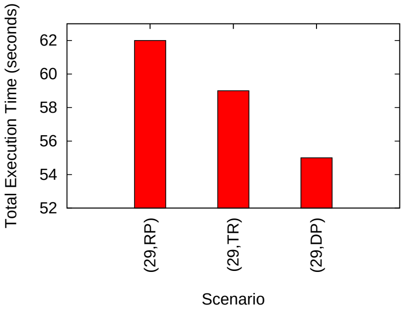


Fig. 25 Performance of responding 10 requests per calling unit in Utopar System in different scenarios for 5×5 calling units and 4 cars.

Fig. 25 shows the time needed for responding to 10 requests by each calling unit. Clearly, $(29, DP)$ outperforms $(29, TR)$ and $(29, RP)$. This is due to the overhead of communications for the case of TR and RP. More precisely, regarding RP the overhead is due to the communication between the components of Interaction Protocol layer and Reservation Protocol layer, since the centralized Reservation Protocol is placed in the central machine. Regarding TR, the overhead is due to communications between Reservation Protocol which depend on the number of components in this layer. To the contrary, in DP, the Reservation Protocol components act in their local neighborhood although more communication is needed.

Fig. 26 also shows the same type of simulations by taking 4 cars and $49 = 7 \times 7$ calling units. Performance becomes worse for TR since the token has to travel a

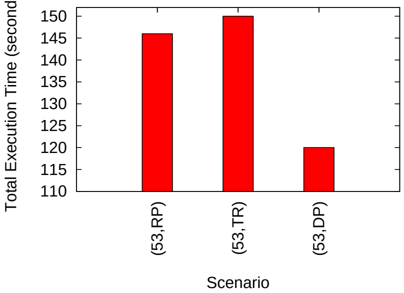


Fig. 26 Performance of responding 10 requests per calling unit in Utopar System in different scenarios for 7×7 calling units and 4 cars.

long way through the components of the Reservation Protocol layer.

We conclude this section by stating the main lesson learned from our simulations. Different partitions and choice of committee coordination algorithm for distributed conflict resolution, suit different topologies and settings although they serve a common purpose. Designers of distributed applications should have access to a library of algorithms and choose the best according to parameters of the application.

8 Running Experiments

In this section, we present the results of our experiments on two popular parallel sorting algorithms: (1) network sorting algorithm (see Subsection 8.1), and (2) bitonic sorting (see Subsection 8.2). Unlike simulations in Section 7, all results in this section are determined by physical computation and communication times.

All experiments are conducted on quad-Xeon 2.6 GHz machines with 6GB RAM running under Debian Linux and connected via a 100Mbps Ethernet network. We show that our method allows evaluation of parallel and multi-core applications modeled in BIP. Moreover, we show that different mergings of components may result in significantly different performance.

8.1 Network Sorting Algorithm

We consider 2^n atomic components, each containing an array of N items. The goal is to sort the items, so that all the items in the first component are smaller than those of the second component and so on. Fig. 27 shows a BIP model of the Network Sorting Algorithm [1] for $n = 2$ using incremental and hierarchical composition of components. The atomic components $B_1 \dots B_4$ are identical. Each atomic component computes independently the minimum and the maximum values of its array. Once this computation completes, interaction a_1 compares the maximum value of B_1 with the minimum value of B_2 and swaps them if the maximum of B_1 is greater than the minimum of B_2. Otherwise, the corresponding arrays are correctly sorted and interaction a_2 gets enabled. This interaction exports the minimum of B_1 and the maximum of B_2 to interaction a_5. The same principle is applied to components B_3 and B_4 and interactions a_3 and a_4. Finally, interaction a_5 works in the same way as interaction a_1 and swaps the minimum and the maximum values, if they are not correctly sorted.

All interactions in Fig. 27 are in conflict. We choose to construct a single Interaction Protocol component IP_1 that encompasses all these interactions (see Fig. 28). We try different merging schemes in order to study the degree of parallelism for each scenario. More precisely, we run experiments for five configurations 1c, 2c, 3c, 4c, and 5c. For 1c, we *merge* all components to obtain a single component, as described in Subsection 6.1. For 2c, we merge components $[B_1^{SR}, B_2^{SR}, IP_1]$ and $[B_3^{SR}, B_4^{SR}]$, obtaining two Send/Receive components. For 3c, we merge components $[B_1^{SR}, IP_1]$, $[B_2^{SR}, B_3^{SR}]$, and $[B_4^{SR}]$, obtaining three Send/Receive components. For 4c, we merge components $[B_1^{SR}, IP_1]$, $[B_2^{SR}]$, $[B_3^{SR}]$, and $[B_4^{SR}]$, obtaining four Send/Receive components. Finally, for 5c we do not merge components, hence, we have five Send/Receive components (see Fig. 28).

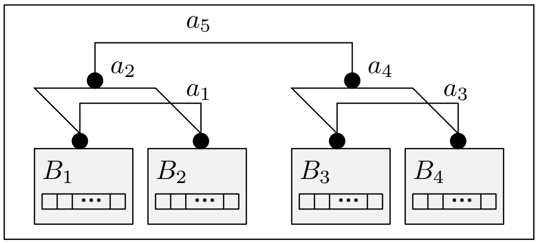


Fig. 27 A BIP model for Network Sorting Algorithm.

Table 1 shows performance of the automatically generated C++ code using TCP sockets and POSIX threads. The implementation using POSIX slightly outperforms TCP sockets. This is due to the fact that the number of messages exchanged per component is huge, making the socket-based implementation slower, as it requires network communication. Performance clearly depends on the size of the input array as well. Also, notice that configuration 5c outperforms all the other configurations. This is due to the fact that the computational load in this configuration is much higher than the communication load.

8.2 Bitonic Sorting

Bitonic sorting [7] is one of the fastest sorting algorithms suitable for distributed implementations or in

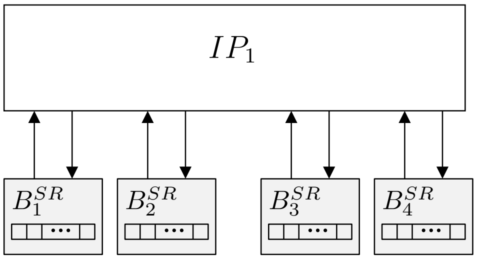


Fig. 28 3-layer Send/Receive BIP model for Network Sorting Algorithm.

	$1c$	$2c$	$3c$	$4c$	$5c$
k	C++/POSIX (Shared Memory)				
1	1.78	1.61	0.93	0.73	**0.53**
5	9.72	8.75	5.08	4.21	**2.88**
10	21.52	19.42	11.21	9.42	**6.52**
50	191.14	171.12	98.07	82.96	**63.43**
k	C++/Socket				
1	1.8	1.94	1.27	1.07	**0.83**
5	9.81	9.44	5.85	4.91	**3.53**
10	21.7	20.55	12.56	10.03	**7.51**
50	191.47	177	104.1	86.17	**65.84**

Table 1 Performance (execution time in seconds) of NSA ($n = 2$), where $k \times 10^3$ is the size of array for sorting.

parallel processor arrays. A sequence is called *bitonic* if it is initially non-decreasing, then it is non-increasing. The first step of the algorithm consists in constructing a bitonic sequence. Then, by applying a logarithmic number of bitonic merges, the bitonic sequence is transformed into a totally ordered sequence. We provide an implementation of the bitonic sorting algorithm in BIP using four atomic components, each handling one quarter of the array. These components are connected as shown in Fig. 29.

The six interactions are non-conflicting. Moreover, interactions a_1, a_2, and a_3 cannot run in parallel. The same holds for interactions a_4, a_5, and a_6. Thus, to obtain maximal parallelism between interactions, it is sufficient to create only two components for the Interaction Protocol layer. The first component IP_1 handles interactions a_1, a_2, and a_3 and the second component IP_2 handles interactions a_4, a_5, and a_6. Furthermore, since all interactions are non-conflicting, there is no need for Reservation Protocol. According to this partition of interactions, we obtain the 3-layer Send/Receive BIP model shown in Fig. 30. In this example, each component sends only three messages containing the array of values.

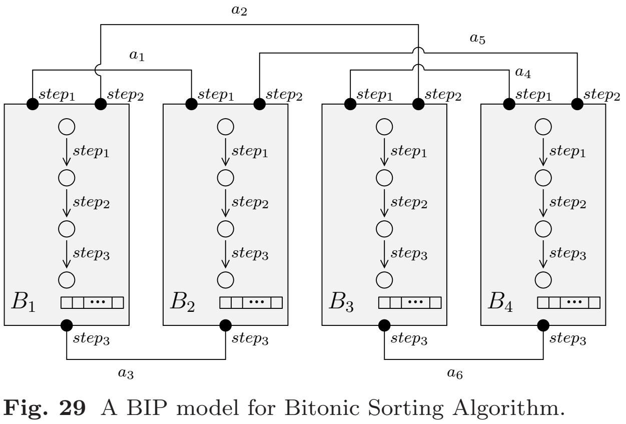


Fig. 29 A BIP model for Bitonic Sorting Algorithm.

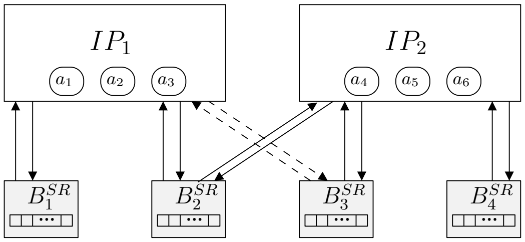


Fig. 30 3-layer Send/Receive BIP model for Bitonic Sorting Algorithm ($6c$).

We choose different merging schemes to study the degree of parallelism with respect to each configuration. More precisely, we run experiments for four configurations $1c$, $2c$, $4c$, and $6c$ (see Fig. 31). For $1c$, we merge all components in one component. For $2c$, we merge components $[B_1^{SR}, B_2^{SR}, IP_1]$ and $[B_3^{SR}, B_4^{SR}, IP_2]$, obtaining two components. For $4c$, we merge components $[B_1^{SR}, IP_1]$, $[B_2^{SR}]$, $[B_3^{SR}]$, and $[B_2^{SR}, IP_2]$, obtaining four components. Finally, for $6c$ we do not merge components, hence, we have six Send/Receive components (see Fig. 30).

Table 2 shows performance of the automatically generated C++ code using TCP sockets, POSIX threads (shared memory), and MPI. Clearly, the configuration $6c$ outperforms the other configurations for TCP sockets and for POSIX threads. Furthermore, the overall performance of these implementations is quite similar. This is due to the fact that, in contrast to the previous example, the number of messages exchanged per component is small. More precisely, each component performs three steps in order to obtain a totally ordered sequence. Each step requires a binary synchronization and leads to one message exchange between an atomic component and the Interaction Protocol. On the other hand, at each step the amount of computation per atomic component is huge with respect to the communication time.

For the MPI implementation, configuration $4c$ outperforms the other configurations. This is due to the fact that MPI uses active waiting, which entails CPU time consumption when a component is waiting. The MPI code consisting of four processes is therefore the best fitting the four cores available on the target machine and yields best performances.

Moreover, Table 2 shows performance of the handwritten C++ code using MPI collective communication primitives (e.g., Gather and Scatter) instead of Send/Receive to transfer data. We notice that the best performance of automatically generated C++ code (obtained in configuration $6c$) is comparable to the performance of handwritten code (and run on configuration $4c$).

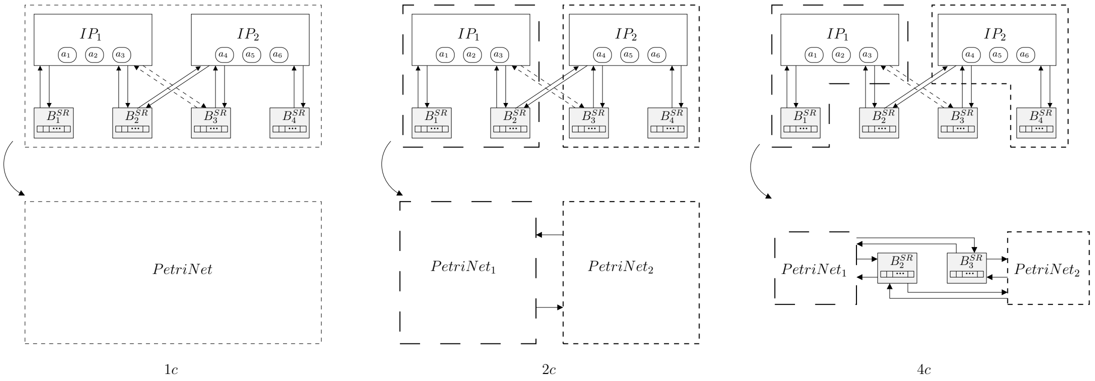


Fig. 31 Merging schemes applied on Send/Receive BIP model of Bitonic Sorting Algorithm.

	1*c*	2*c*	4*c*	6*c*
k	C++/POSIX (Shared Memory)			
1	0.02	**0.01**	**0.01**	**0.01**
10	1.8	0.96	0.75	**0.54**
50	44	23.57	18.37	**12.04**
100	178.42	94.71	73.22	**48.1**
k	C++/Socket			
1	**0.02**	**0.02**	0.34	0.27
10	1.8	1	1.01	**0.75**
50	44	24.1	18.57	**12.3**
100	178.42	95.32	74.01	**48.7**
k	MPI			
1	**0.26**	0.8	1.15	1.16
10	1.93	2.06	**1.92**	2.03
50	44.85	24.04	**19.47**	23.08
100	179.4	95.83	**74.59**	85.37
k	Handwritten			
1			1.54	
10			2.01	
50			13.47	
100			49.67	

Table 2 Performance (execution time in seconds) of bitonic sorting ($n = 2$), where $k \times 10^3$ is the size of array for sorting.

The main observation from these experiments is that determining adequate component merging and communication primitives depends on (1) the topology of the system with respect to communication delays, and (2) the computational load of the system, and (2) the target architecture on which the system is deployed.

9 Related Work

In this section, we report on the work related to automated code generation for distributed systems from high-level models. We first discuss solutions to the *committee coordination problem* in Subsection 9.1. Then, we present frameworks for automatic generation of distributed code in Subsection 9.2.

9.1 Algorithms for Solving the Committee Coordination Problem

As mentioned in the introduction, resolving distributed conflicts in the context of our framework leads us to solving the *committee coordination problem* [15], where a set of professors organize themselves in different committees. Two committees that have a professor in common cannot meet simultaneously. The original distributed solution to the committee coordination problem assigns one *manager* to each interaction [15]. Conflicts between interactions are resolved by reducing the problem to the dining or drinking philosophers problems [14], where each manager is mapped onto a philosopher. Bagrodia [3] proposes an algorithm where message counts are used to solve synchronization and exclusion is ensured by a circulating token. In a follow-up paper [4], Bagrodia modifies the solution in [3] by using message counts to ensure synchronization and reducing the conflict resolution problem to dining or drinking philosophers. Also, Perez et al [30] propose another approach that essentially implements the same idea using a lock-based synchronization mechanism.

In [23], Kumar proposes an algorithm that replaces the managers by tokens, one for each interaction, traversing the processes. An interaction executes whenever the corresponding token manages to traverse all involved processes. Another solution without managers has been provided in [22] based on a randomized algorithm. The idea is that each process randomly chooses an enabled interaction and sends its choice to all other processes. If some process detects that all participants in an interaction have chosen it, then the interaction is executed. Otherwise, this procedure gets restarted.

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

9.2 Automated Generation of Distributed Code

LOTOS [21] is a specification language based on process algebra, that encompasses multiparty interactions. In [34], the authors describe a method of executing a LOTOS specification in a distributed fashion. This implementation is obtained by constructing a tree at runtime. The root is the main connector of the LOTOS specification and its children are the subprocesses that are connected. A synchronization between two processes is handled by their common ancestor. This approach is not suitable for BIP where there is no 'main' interaction. Also, the idea of a parent to be responsible for ensuring synchronization makes solutions more centralized than distributed with greater communication overhead.

Another framework that offers automatic distributed code generation is described in [33]. The input model consists of composition of I/O automata [25], from which a Java implementation using MPI for communication is generated. The model, as well as the implementation, can interact with the environment. However, connections between I/O automata (binary synchronization) are less expressive than BIP interactions, as described in [8]. Indeed, to express an n-ary rendezvous between n I/O automata, one would need to add an automaton in charge of synchronization, which is not needed in BIP. Another difference with our work is that the designer has to provide some function that resolves non-determinism. Finally, the framework in [33] requires the designer to specify low-level elements of a distributed system such as channels and schedulers.

Finally, [31] provides a distributed implementation method for the Reo framework [2]. In this framework, components are black boxes with input and output ports synchronized by data-flow connectors. A distributed implementation is obtained by deploying connectors, according to a user-defined partition, on a set of generic engines, implemented in Scala. At execution, a consensus algorithm between all the engines chooses a subset of connectors to be executed, based on the set of currently enabled ports. Unlike BIP interactions, dataflow connectors in Reo do not provide support for guard conditions and arbitrary data transfer functions. Moreover, the consensus algorithm used enforces a global agreement between all engines, whereas in the 3-layer BIP decisions are taken independently by each engine, or by communication with the Reservation Protocol.

10 Conclusion

We presented a framework for automated transformation of BIP models into distributed implementations. Our transformation, first, takes as input a BIP model and generates another BIP model which contains components glued by *Send/Receive* interactions in the following three layers: (1) the *components* layer consists of a transformation of behavioral atomic components in the original model, (2) the *Interaction Protocol* detects enabledness of interactions of the original model and executes them, and (3) the *Reservation Protocol* resolves conflicts among interactions in a distributed fashion by employing a solution to the committee coordination problem. The second step of our transformation takes the intermediate 3-layer BIP model as input and generates C++ executables using TCP sockets, MPI primitives, or POSIX threads.

We reported our observations through conducting several simulations and experiments using different algorithms in the Reservation Protocol and partitions. Design of efficient and correct distributed systems depends on a large variety of parameters and choices. Designers must be provided with a rigorous method and rich libraries of algorithms such as the ones presented in this paper, to derive correct and yet efficient distributed implementations.

For future work, we are considering several research directions. An important extension is to allow the Reservation Protocol to incorporate different algorithms for concurrent conflict resolution, so that each set of conflicting interactions within the same system is handled by the most appropriate algorithm. In this context, we are also planning to explore other algorithms, such as solutions to distributed graph matching [18] and distributed independent set problems [24] for better understanding the tradeoffs between parallelism, load balancing, and communication overhead. Another important line of research is to study the overhead introduced by our transformations where communication cost is crucial, such as in peer-to-peer and large sensor networks.

Another interesting problem is to develop techniques for generating an efficient Interaction Protocol partition automatically rather than requiring the user to provide it as input to our transformation. Finally, given the recent advances in the multi-core technology, we plan to customize our transformation for multi-core platforms as well by using non-blocking data structures (e.g., transactional memory [20, 32]) rather than interprocess communication data structures. Thus, one can investigate whether it is possible to devise more effective techniques to achieve correct-by-construction process synchronization in multi-core settings.

References

1. M. Ajtai, J. Komlós, and E. Szemerédi. Sorting in c log n parallel steps. *Combinatorica*, 3(1):1–19, 1983.
2. F. Arbab. Reo: a channel-based coordination model for component composition. *Mathematical. Structures in Comp. Sci.*, 14:329–366, June 2004.
3. R. Bagrodia. A distributed algorithm to implement n-party rendevouz. In *Foundations of Software Technology and Theoretical Computer Science, Seventh Conference (FSTTCS)*, pages 138–152, 1987.
4. R. Bagrodia. Process synchronization: Design and performance evaluation of distributed algorithms. *IEEE Transactions on Software Engineering (TSE)*, 15(9):1053–1065, 1989.
5. A. Basu, P. Bidinger, M. Bozga, and J. Sifakis. Distributed semantics and implementation for systems with interaction and priority. In *Formal Techniques for Networked and Distributed Systems (FORTE)*, pages 116–133, 2008.
6. A. Basu, M. Bozga, and J. Sifakis. Modeling heterogeneous real-time components in BIP. In *Software Engineering and Formal Methods (SEFM)*, pages 3–12, 2006.
7. K. E. Batcher. Sorting networks and their applications. In *AFIPS '68 (Spring): Proceedings of the April 30–May 2, 1968, spring joint computer conference*, pages 307–314, 1968.
8. S. Bliudze and J. Sifakis. A notion of glue expressiveness for component-based systems. In *Concurrency Theory (CONCUR)*, pages 508–522, 2008.
9. B. Bonakdarpour, M. Bozga, M. Jaber, J. Quilbeuf, and J. Sifakis. From high-level component-based models to distributed implementations. Technical Report TR-2010-9, VERIMAG, March 2010.
10. B. Bonakdarpour, M. Bozga, and J. Quilbeuf. Automated distributed implementation of component-based models with priorities. In *ACM International Conference on Embedded Software (EMSOFT)*, pages 59–68, 2011.
11. B. Bonakdarpour, S. Devismes, and F. Petit. Snap-stabilizing committee coordination. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 231–242, 2011.
12. M. Bozga, M. Jaber, and J. Sifakis. Source-to-source architecture transformation for performance optimization in BIP. In *Symposium on Industrial Embedded Systems (SIES)*, pages 152–160, 2009.
13. M. Bozga, M. Jaber, and J. Sifakis. Source-to-source architecture transformation for performance optimization in BIP. *IEEE Transactions on Industrial Informatics*, 5(4):708–718, 2010.
14. K. M. Chandy and J. Misra. The drinking philosophers problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 6(4):632–646, 1984.
15. K. M. Chandy and J. Misra. *Parallel program design: a foundation.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1988.
16. O. M. Cheiner and A. A. Shvartsman. Implementing an eventuallyserializable data service as a distributed system building block. In *Principles Of Distributed Systems (OPODIS)*, pages 9–24, 1998.
17. E. W. Dijkstra and C. S. Scholten. Termination detection for diffusing computations. *Information Processing Letters*, 11(1):1–4, 1980.
18. Z. Galil, S. Micali, and H. N. Gabow. An o(ev log v) algorithm for finding a maximal weighted matching in general graphs. *SIAM J. Comput.*, 15(1):120–130, 1986.
19. G. Gössler and J. Sifakis. Composition for component-based modeling. *Science of Computer Programming*, 55(1-3):161–183, 2005.
20. M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *ISCA*, pages 289–300, 1993.
21. ISO/IEC. *Information Processing Systems – Open Systems Interconnection: LOTOS, A Formal Description Technique Based on the Temporal Ordering of Observational Behavior*, 1989.
22. Y.-J. Joung and S. A. Smolka. Strong interaction fairness via randomization. *IEEE Trans. Parallel Distrib. Syst.*, 9(2):137–149, 1998.
23. D. Kumar. An implementation of n-party synchronization using tokens. In *ICDCS*, pages 320–327, 1990.
24. M. Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM J. Comput.*, 15(4):1036–1053, 1986.
25. N. Lynch. *Distributed Algorithms.* Morgan Kaufmann Publishers, San Mateo, CA, 1996.
26. F. Manne, M. Mjelde, L. Pilard, and S. Tixeuil. A new self-stabilizing maximal matching algorithm. *Theoretical Computer. Scence.*, 410(14):1336–1345, 2009.
27. R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, 1980.
28. R. Milner. *Communication and concurrency.* Prentice Hall International (UK) Ltd., Hertfordshire, UK, 1995.
29. T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541 –580, apr 1989.
30. 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.
31. J. Proença. *Synchronous Coordination of Distributed Components.* PhD thesis, Faculteit der Wiskunde en Natuurwetenschappen, May 2011.
32. N. Shavit and D. Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, 1997.
33. J. A. Tauber, N. A. Lynch, and M. J. Tsai. Compiling IOA without global synchronization. In *Symposium on Network Computing and Applications (NCA)*, pages 121–130, 2004.
34. G. von Bochmann, Q. Gao, and C. Wu. On the distributed implementation of lotos. In *FORTE*, pages 133–146, 1989.