From High-level Component-Based Models to Distributed Implementations[*]

Borzoo Bonakdarpour
Department of Electrical and Computer engineering
University of Waterloo
200 University Avenue West
Waterloo, Canada, N2L3G1
borzoo@ecemail.uwaterloo.ca

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

Mohamad Jaber
VERIMAG, Centre Équation
2 avenue de Vignate
38610, GIÈRES, France
mohamad.jaber@imag.fr

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

Joseph Sifakis
VERIMAG, Centre Équation
2 avenue de Vignate
38610, GIÈRES, France
joseph.sifakis@imag.fr

ABSTRACT

Although distributed systems are widely used nowadays, their implementation and deployment is still a time-consuming, error-prone, and hardly predictive task. In this paper, we propose a methodology for producing automatically efficient and correct-by-construction distributed implementations by starting from a high-level model of the application software in BIP. BIP (Behavior, Interaction, Priority) is a component-based framework with formal semantics that rely on multi-party interactions for synchronizing components and dynamic priorities for scheduling between interactions.

Our methodology transforms arbitrary BIP models into Send/Receive BIP models, directly implementable on distributed execution platforms. The transformation consists of (1) breaking atomicity of actions in atomic components by replacing strong synchronizations with asynchronous Send/Receive interactions; (2) inserting several distributed controllers that coordinate execution of interactions according to a user-defined partition, and (3) augmenting the model with a distributed algorithm for handling conflicts between controllers. The obtained Send/Receive BIP models are proven observationally equivalent to the initial models. Hence, all the functional properties are preserved by construction in the implementation. Moreover, Send/Receive BIP models can be used to automatically derive distributed implementations. Currently, it is possible to generate stand-alone C++ implementations using either TCP sockets for conventional communication, or MPI implementation, for deployment on multi-core platforms. This method is fully implemented. We report concrete results obtained under different scenarios (i.e., partitioning of interactions and choice of algorithm for distributed conflict resolution).

[*]This is an extended version of the paper appeared in ACM/IEEE International Conference on Embedded Software (EMSOFT'10). This work is sponsored by the COMBEST European project.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems[Distributed applications] ; D.1.3 [**Programming Techniques**]: Concurrent Programming—*Distributed programming, Parallel programming* ; D.2.13 [**Software Engineering**]: Reusable Software—*Reuse models*; D.4.7 [**Operating Systems**]: Organization and Design—*Real-time and embedded systems*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*Logic of programs*; I.2.2 [**Artificial Intelligence**]: Automatic Programming—*Program transformation*

General Terms

Theory, Design, Languages, Reliability, Performance

Keywords

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

1. INTRODUCTION

Analysis and design of computing systems often starts with developing a high-level model of the system. Constructing models is beneficial, as designers can abstract away implementation details and validate the model with respect to a set of intended requirements through different techniques such as formal verification, simulation, and testing. However, deriving a *correct* implementation from a model

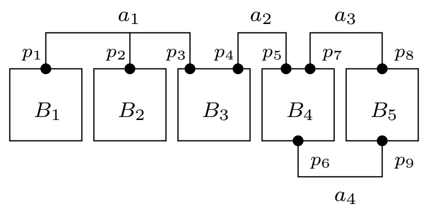


Figure 1: A simple BIP model with conflicts.

is always challenging, since adding implementation details involves many subtleties that can potentially introduce errors to the resulting system. In the context of distributed systems, these subtleties are amplified significantly because of inherently concurrent, non-deterministic, and non-atomic structure, as well as the occurrence of unanticipated physical and computational events such as faults. Thus, it is highly advantageous if designers can somehow derive implementations in a systematic and ideally automated correct fashion from high-level models. It is, nonetheless, unclear how to transform an abstract model (where atomicity is assumed through global state semantics and distribution details are omitted via employing high-level synchronization primitives) into a real distributed implementation.

In this paper, we present a novel method for automatically transforming high-level models in BIP [4] into distributed implementations. The BIP (Behavior, Interaction, Priority) language is based on a semantic model encompassing composition of heterogeneous components. The *behavior* of atomic components is described as an automaton or Petri net extended by data and functions given in C++. Transitions of the automaton or Petri net are labeled by port names and functions computing data transformations when they are executed. If a transition of the Petri net can be executed, we say that the associated port is *enabled*. BIP uses a composition operator for obtaining composite components from a set of atomic components. The operator is parameterized by a set of *interactions* between the composed components. One such interaction type is *rendezvous*, which is *enabled* if all of its participating ports are enabled. The execution of interactions and local code of components are orchestrated by a sequential *scheduler*.

In order to understand the subtleties of transforming a BIP model into a distributed implementation, consider the BIP model in Figure 1. In this model, atomic components $B_1 \cdots B_5$ are synchronized by four rendezvous interactions $a_1 \cdots a_4$. In sequential models, interactions are executed atomically by a single scheduler. To the contrary, introducing concurrency and distribution to this model requires the implementation to deal with more complex issues:

- (*Partial observability*) Suppose interaction a_1 (and, hence, components $B_1 \cdots B_3$) is being executed. 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. In such a case, a distributed scheduler must be designed, so that concurrent execution of interactions does not introduce behaviors that were not allowed by the high-level model.

- (*Resolving conflicts*) Suppose interactions a_1 and a_2 are enabled simultaneously. Since these interactions share component B_3, they cannot be executed concurrently. We call such interactions *conflicting*. Obviously, a distributed scheduler must ensure that execution of conflicting interactions is mutually exclusive.

- (*Performance*) On top of correctness issues, a real challenge is to ensure that a transformation does not add considerable overhead to the implementation. After all, one crucial goal of developing distributed and parallel systems is to exploit their computing power.

We address the issue of partial observability by breaking the atomicity of execution of interactions, so that a component can execute unobservable actions once the corresponding interaction is being executed [3]. Resolving conflicts leads us to solving the *committee coordination problem* [7], 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 [7]. Conflicts between interactions are resolved by reducing the problem to the dining or drinking philosophers problems [6], where each manager is mapped onto a philosopher. Bagrodia [1] proposes an algorithm where message counts are used to solve synchronization and exclusion is ensured by using a circulating token. In a follow-up paper [2], Bagrodia modifies the solution in [1] by using message counts to ensure synchronization and reducing the conflict resolution problem to dining or drinking philosophers. Also, Perez et al [11] propose another approach that essentially implements the same idea using a lock-based synchronization mechanism.

As Bagrodia notes [2], a family of solutions to the committee coordination problem is possible, ranging over fully centralized to fully decentralized ones, depending upon the mapping of sets of committees to the managers. Thus, augmenting a transformation with different families of solutions results in different distributed implementations of the initial BIP model. We expect that each class of solutions exhibits advantages and disadvantages and, hence, fits a specific type of applications on a target architecture and platform. Although the algorithms in [1, 2, 7] provide us with different families of solutions, transforming a high-level model into a concrete distributed implementation involves other details that have not been taken into account. Examples include preserving functional properties of the original model, computations associated with interactions and components, data transfer, level of concurrency, fairness, fault-tolerance, efficiency, and performance. These issues can significantly change the dynamics and performance of a distributed implementation and each deserves rigorous research beyond the algorithms and preliminary simulations in [1, 2, 7]. We believe we currently lack a deep understanding of the impact of these issues and their correlation in transforming high-level models into concrete distributed implementations. For example, existing transformation methods are either not flexible in generating different levels of distribution [5], inefficient [8], or require the designer to explicitly specify communication elements of the distributed implementation [12].

Contributions. With this motivation, in this paper, we propose a generic framework for transforming high-level BIP models into a distributed implementation that allow parallelism between components as well as parallel execution of non-conflicting interactions by embedding a solution to the committee coordination problem. To the best of our knowledge, this is the first instance of such a transformation (the related work mentioned above only focus on impossibility results, abstract algorithms, and in one instance [2] simulation of an algorithm). Our method utilizes the following sequence of transformations preserving *observational equivalence*:

1. First, we transform the given 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:
   (a) The *components layer* consists of a transformation of behavioral components in the original model.
   (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. This layer consists of a set of components, each hosting a user-defined subset of interactions from the original BIP model.
   (c) The *reservation protocol* resolves conflicts requested by the interaction protocol. The reservation protocol 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 reduction to distributed dining philosophers.
2. Then, we transform the 3-layer BIP model into C++ code that employs TCP sockets for communication.

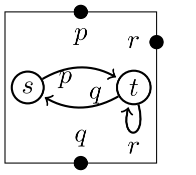
We also conduct a set of experiments to analyze the behavior and performance of the generated code using different scenarios (i.e., different partitioning schemes and choice of committee coordination algorithm). Our experiments clearly show that each scenario is suitable for a different topology, size of the distributed system, communication load, and of course, the 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 our BIP to BIP transformation. In Section 5, we show the correctness of our transformation. Section 6 presents the results of our experiments. Finally, in Section 7, we make concluding remarks and discuss future work.

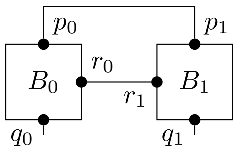
2. BASIC SEMANTIC MODELS OF BIP

In this section, we present operational *global state* semantics of BIP. BIP is a component framework for constructing systems by superposing three layers of modeling: Behavior, Interaction, and Priority. Since the issue of priorities is irrelevant to this paper, we omit it.

Atomic Components We define *atomic components* as transition systems with a set of ports labeling individual transitions. These ports are used for communication between different components.



(a) Atomic component



(b) Composite component

Figure 2: BIP composite component

Definition 1 (Atomic Component) *An* atomic component *B is a labeled transition system represented by a triple $(Q, P, \rightarrow)$ where Q is a set of* states*, P is a set of* communication ports*, $\rightarrow \subseteq Q \times P \times Q$ is a set of* possible transitions*, each labeled by some port.*

For any pair of states $q, q' \in Q$ and a port $p \in P$, we write $q \xrightarrow{p} q'$, iff $(q, p, q') \in \rightarrow$. When the communication port is irrelevant, we simply write $q \rightarrow q'$. Similarly, $q \xrightarrow{p}$ means that there exists $q' \in Q$ such that $q \xrightarrow{p} q'$. In this case, we say that p is *enabled* in state q.

In practice, atomic components are extended with variables. Each variable may be bound to a port and modified through interactions involving this port. We also associate a guard and an update function to each transition. A guard is a predicate on variables that must be true to allow the execution of the transition. An update function is a local computation triggered by the transition that modifies the variables. Figure 2(a) shows an atomic component B, where $Q = \{s, t\}$, $P = \{p, q, r\}$, and $\rightarrow = \{(s, p, t), (t, q, s), (t, r, t)\}$.

Interaction For a given system built from a set of n atomic components $\{B_i = (Q_i, P_i, \rightarrow_i)\}_{i=1}^n$, we assume that their respective sets of ports are pairwise disjoint, i.e., for any two $i \neq j$ from $\{1..n\}$, we have $P_i \cap P_j = \emptyset$. We can therefore define the set $P = \bigcup_{i=1}^n P_i$ of all ports in the system. An *interaction* is a set $a \subseteq P$ of ports. When we write $a = \{p_i\}_{i \in I}$, we suppose that for $i \in I$, $p_i \in P_i$, where $I \subseteq \{1..n\}$.

Similar to atomic components, BIP extends interactions by associating a guard and a transfer function to each of them. Both the guard and the function are defined over the variables that are bound to the ports of the interaction. The guard must be true to allow the interaction. When the interaction takes place, the associated transfer function is called and modifies the variables.

Definition 2 (Composite Component) *A* composite component *(or simply* component*) is defined by a composition operator parameterized by a set of interactions $\gamma \subseteq 2^P$. $B \overset{def}{=} \gamma(B_1, \ldots, B_n)$, is a transition system $(Q, \gamma, \rightarrow)$, where $Q = \bigotimes_{i=1}^n Q_i$ and $\rightarrow$ is the least set of transitions satisfying the rule*

$$\frac{a = \{p_i\}_{i \in I} \in \gamma \qquad \forall i \in I : q_i \xrightarrow{p_i}_i q_i' \qquad \forall i \notin I : q_i = q_i'}{(q_1, \ldots, q_n) \xrightarrow{a} (q_1', \ldots, q_n')}$$

The inference rule says that a composite component $B = \gamma(B_1, \ldots, B_n)$ can execute an interaction $a \in \gamma$, iff for each port $p_i \in a$, the corresponding atomic component B_i can execute a transition labeled with p_i; the states of components that do not participate in the interaction stay unchanged. Figure 2(b) illustrates a composite component $\gamma(B_0, B_1)$, where each B_i is identical to component B in Figure 2(a) and $\gamma = \{\{p_0, p_1\}, \{r_0, r_1\}, \{q_0\}, \{q_1\}\}$.

3. THE 3-LAYER ARCHITECTURE

In this section, we describe the overall architecture of our BIP source-to-source transformation. 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 semantics of the global state model. This leads to the notion of structural *conflicts* between interactions.

Definition 3 *Let $\gamma(B_1, \ldots, B_n)$ be a BIP model. We say that two interactions $a_1, a_2 \in \gamma$ are* conflicting *iff either:*

- *they share a common port p; i.e., $p \in a_1 \cap a_2$, or*
- *there exist an atomic component $B_i = (Q_i, P_i, \rightarrow_i)$, a state $q \in Q_i$, and two ports $p_1, p_2 \in P_i$ such that (1) $p_1 \in a_1$, (2) $p_2 \in a_2$, and (3) $q \xrightarrow{p_1} \wedge\, q \xrightarrow{p_2}$.*

As discussed in the introduction, handling conflicting interactions in a BIP model running by a sequential scheduler is quite straightforward. However, in a distributed setting, detecting and avoiding conflicts are not trivial. Thus, our target BIP model in a transformation should have the following three properties: (1) preserving the behavior of each atomic component, (2) preserving the observational behavior of interactions, and (3) resolving conflicts in a distributed manner. Since several distributed algorithms exist in the literature for conflict resolution, we design our framework, so that it provides appropriate interfaces with minimal restrictions. 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 using conventional communication primitives (e.g., TCP sockets or MPI).

Definition 4 *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 in B^{SR} into three sets P_s, P_r, P_u that are respectively the set of* send-ports, receive-ports, *and* unary interaction ports*, such that:*

- *Each interaction $a \in \gamma^{SR}$, is either a Send/Receive interaction $a = (s, r_1, r_2, \ldots, r_k)$ with $s \in P_s$ and $r_i \in P_r$, or, a unary interaction $a = \{p\}$ with $p \in P_u$.*
- *If s is a port in P_s, then there exists one and only one Send/Receive interaction $(s, r_1, r_2, \ldots, r_k) \in \gamma^{SR}$ where 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 $(s, r_1, \ldots, r_k)$ 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.*

We design our target BIP model based on the three tasks identified above, where we incorporate one layer for each

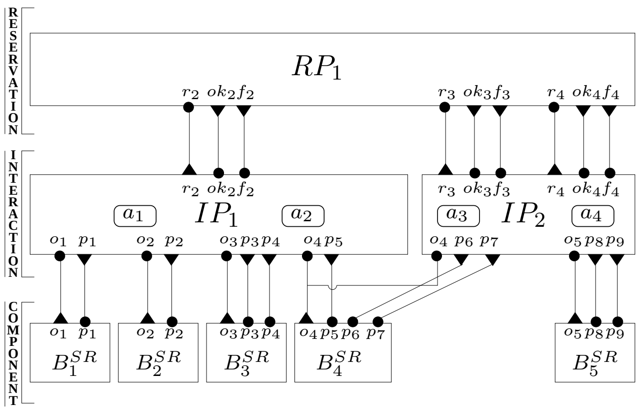


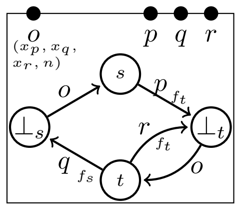
Figure 3: 3-layer model of Figure 1.

task. We use the high-level BIP model in Figure 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. Our 3-layer architecture consists of the following layers.

Components Layer. Atomic components in the high-level model are placed in this layer with the following additional ports per component. The send-port o that shares the list of enabled ports in the component with the upper layer. Also, for each port p in the original component, we include a receive-port p through which the component is notified to execute the transition labeled by p once the upper layers resolve conflicts and decide on which components can execute on what port. The bottom layer in Figure 3 includes components illustrated in Figure 1.

Interaction Protocol. This layer consists of a set of components each hosting a set of interactions in the high-level model. Conflicts between interactions included in the same component are resolved by that component locally. For instance, interactions a_1 and a_2 (resp. a_3 and a_4) of Figure 1 are grouped into component IP_1 (resp. component IP_2) in Figure 3. Thus, the conflict between a_1 and a_2 (resp. a_3 and a_4) is handled locally in IP_1 (resp. IP_2). To the contrary, the 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 forms the top layer of our model. The interaction protocol also evaluates the guard of each interaction and executes the code associated with an interaction that is selected locally or by the upper layer. The interface between this layer and the component layer provides ports for receiving enabled ports from each component (i.e., port o) and notifying the components on permitted port for execution.

Reservation Protocol. This layer accommodates an algorithm that solves the committee coordination problem. For instance, the external conflicts between interactions a_2 and a_3, and, interactions a_2 and a_4 are resolved by the central component RP_1 in Figure 3. We emphasize that the structure of components in this layer solely depends upon the augmented conflict resolution algorithm. Incorporating a centralized algorithm results in one component RP_1 as



$$f_s = \begin{cases} x_p := true \\ x_q := false \\ x_r := false \\ n++ \end{cases}$$

$$f_t = \begin{cases} x_p := false \\ x_q := true \\ x_r := true \\ n++ \end{cases}$$

Figure 4: SR atomic component of Figure 2(a).

illustrated in Figure 3. Other algorithms (as will be discussed in Subsection 4.3), such as ones that use a circulating token [1] or dining philosophers [2, 7] result in different structures. The interface between this layer and the Interaction Protocol involves ports for receiving request to reserve an interaction (labeled r) and responding by either success (labeled *ok*) or failure (labeled f).

4. TRANSFORMING BIP INTO 3-LAYER BIP

In this section, we describe our technique for transforming a high-level BIP model into a 3-layer BIP model in detail. Construction of the three layers are described in Subsections 4.1, 4.2, and 4.3 respectively. Finally, we describe cross-layer interactions in Subsection 4.4.

4.1 Transformation of Atomic Components

We now present how we transform an atomic component B from a given 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. Enabled ports are specified by a set of Boolean variables. These variables are modified by a *port update* function. The function evaluates each variable when reaching a new state. When the upper layers select an interaction involving B^{SR} for execution, B^{SR} is notified by a response sent on the port chosen. We also include a *participation number* variable n in B^{SR}, which counts the number of interactions that B^{SR} has participated in.

Since each response triggers an internal computation, following [3], we split each state s into two states, namely, s itself and a *busy state* $\perp_s$. 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 5 *Let $B = (Q, P, \rightarrow)$ be an atomic component. The corresponding Send/Receive atomic component is $B^{SR} = (Q^{SR}, P^{SR}, \rightarrow^{SR})$ with the additional variables X, such that:*

- *$Q^{SR} = Q \cup Q^{\perp}$, where $Q^{\perp} = \{\perp_s \mid s \in Q\}$.*
- *$P^{SR} = P \cup \{o\}$, where the set of variables $X = \{x_p\}_{p \in P} \cup \{n\}$ are associated to* offer *port o. The* port update *function f_t modifies X as follows: it sets x_p to true if $t \xrightarrow{p}$, to false otherwise, and increments n.*
- *For each transition $(s, p, t) \in \rightarrow$, we include the following two transitions in $\rightarrow^{SR}$: $(\perp_s, o, s)$ and $(s, p, \perp_t)$. The transition $(s, p, \perp_t)$ triggers the function f_t.*

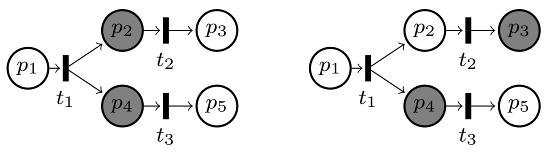


Figure 5: A simple Petri net

Figure 4 illustrates transformation of the component in Figure 2(a) into its corresponding Send/Receive component.

4.2 Interaction Protocol

Given a high-level BIP model $B = \gamma(B_0 \cdots B_n)$, one parameter to our transformation is a partition of interactions $\gamma_1, \ldots, \gamma_m$. Partitioning of interactions is a means for the designer to enforce load-balancing and improving 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 a set of non-conflicting interactions (either locally or by the help of the Reservation Protocol), and (3) executing the selected interactions in γ_i by notifying the corresponding atomic components. For instance, in Figure 3, we have two classes: $\gamma_1 = \{a_1, a_2\}$ (hosted by component IP_1) and $\gamma_2 = \{a_3, a_4\}$ (hosted by component IP_2). In this section, for simplicity of reasoning about correctness, we construct the behavior of an *IP* component by a Petri net.

Definition 6 *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.*

We represent a Petri net as an oriented bipartite graph $G = (L \cup T, E)$. Places are represented by circular vertices and transitions are represented by rectangular vertices. The set of oriented edges E is the union of the edges $\{(l, \tau) \in L \times T \mid l \in {}^\bullet\tau\}$ and the edges $\{(\tau, l) \in T \times L \mid l \in \tau^\bullet\}$.

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. Upon the execution of τ, tokens in input places ${}^\bullet\tau$ are removed and output places in $\tau^\bullet$ are marked. Formally, let $\longrightarrow_S$ 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 by a labeled transition system $(2^L, P, \longrightarrow_S)$.

Figure 5 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 left Petri net is obtained by executing transition t_1. The right Petri net shows the resulting state of the left Petri net when transition t_2 is fired.

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 3. 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 referred to the Reservation Protocol. For instance, in Figure 3, 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 hosts them. For instance, in Figure 3, 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. Note, however, that if IP_1 chooses a_2, then it has to request its reservation from RP_1, as it is in conflict with a_3 and a_4 externally.

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. We refer to Figure 6 as a concrete example for construction of the Petri net of component IP_1 in Figure 3.

Places. We include 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 waits 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 the Boolean variables associated to the ports of the sender), IP_j moves from w_i to rcv_i. In Figure 6, since components $B_1 \cdots B_4$ are involved in interactions hosted 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 Figure 6, 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 Figure 6, 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. The value of this variable is equal to the value of the same variable in the most recent offer received from the corresponding component. Also, for each component B_i involved in interactions of γ_j, we include an integer n_i that stores participation number of B_i. The set of ports of IP_j is the following:

- For each component B_i involved in interactions of γ_j, we include an offer port o_i. Each port o_i updates the values of variables n_i and x_p for each port p exported by B_i. In Figure 6, 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 response port p. In Figure 6, ports $p_1 \cdots p_5$ correspond to the ports that form interactions a_1 and a_2.

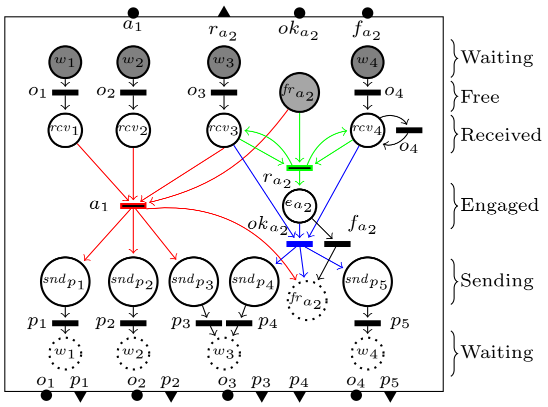


Figure 6: Component IP_1 in Figure 3.

- For each interaction $a \in \gamma_j$ that is in external conflict, we include reservation ports r_a, ok_a, and f_a. If $a = \{p_i\}_{i \in I}$, the port r_a is associated to the variables $\{n_i\}_{i \in I}$, where I is the set of components involved in interaction a. In Figure 6, ports r_{a_2}, ok_{a_2}, and f_{a_2} represent 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 Figure 6, 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:

- 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 Figure 6 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 $\bigwedge_{i \in I} x_{p_i}$ 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. Execution of this transition results in moving the token from a free place to an engaged place. In Figure 6, 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 $(\{rcv_i\}_{i \in I} \cup \{e_a\}, ok_a, \{snd_{p_i}\}_{i \in I} \cup \{fr_a\})$. Upon execution of this transition, the token from the engaged place moves to the free place and the tokens from received move to sending places for informing the corresponding components. Transition ok_{a_2} in Figure 6 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 Figure 6 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, but 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 $\bigwedge_{i \in I} x_{p_i}$ and moves the tokens from receiving to sending places. Tokens from $fr_{a'}$ places ensure that no internally conflicting interaction requested a reservation. The transition labeled by a_1 in Figure 6 falls in this category.

- Finally, for each component B_i exporting p, we include the transitions (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 Figure 6.

4.3 Reservation Protocol

As discussed earlier, the main task of the Reservation Protocol is to ensure that externally conflicting interactions are executed mutually exclusive. The Reservation Protocol can be implemented using any algorithm that solves the committee coordination problem. Our design of Reservation Protocol allows employing any such algorithm with minimal restrictions.

We adapt a variation of the idea of the message-count technique from [2] as a minimal restriction to ensure that our design makes *progress* (see Lemma 2) and it does not interfere with exclusion algorithms. This technique is based on counting the number of times that a component interacts. Each component keeps a counter n which indicates the current number of participation of the component in interactions. The Reservation Protocol ensures that each participation number is used only once. That is, each component takes part in only one interaction per transition. To this end, in the Reservation Protocol, for each component B_i, we keep a variable N_i which stores the latest number of participation 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.

Now, since the structure and behavior of the Reservation Protocol components depend on the employed algorithm, we only specify an abstract set of minimal restrictions of this layer as follows:

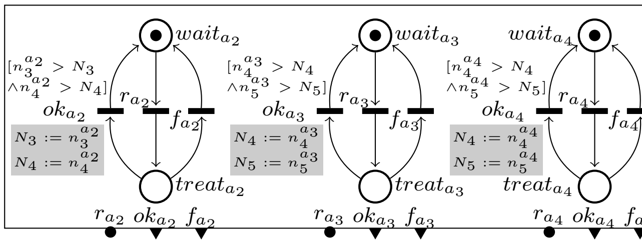


Figure 7: A centralized Reservation Protocol for Figure 3.

- For each component B_i, 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 N_i variables are updated in case of positive response.

- Each r_a message should be acknowledged by exactly one ok_a or f_a message.

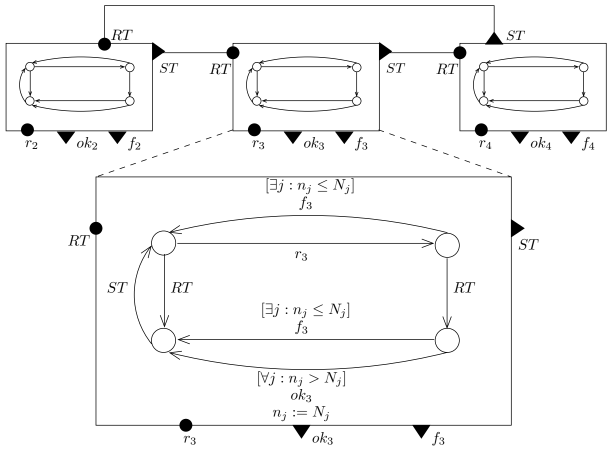
- Each component of the Reservation Protocol should respect the message-count properties described above.

4.3.1 *Centralized Implementation*

Figure 7 shows a centralized Reservation Protocol for the model in Figure 3. In fact, the component in Figure 7 is the component RP_1 in Figure 3. A reservation request, for instance, r_{a_2}, contains fresh 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 state, 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 are atomic in BIP, if transition ok_{a_2} is fired, it modifies variables N_i atomically (i.e., 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 [1], where we add one reservation component per externally conflicting interaction. Figure 8 shows the respective components for the model presented in Figure 3. Exclusion is ensured using a circulating token carrying N_i variables; i.e., the component that owns the token compares the value of the received n_i variables with the N_i variables from the token. If they are greater, an *ok* message is sent to the component that handles that interaction and the N_i values on the token are updated. 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 level of distribution



RT
ST
r_2
ok_2
f_2
r_3
ok_3
f_3
r_4
ok_4
f_4
$[\exists j : n_j \leq N_j]$ f_3
r_3
RT
$[\forall j : n_j > N_j]$ ok_3 $n_j := N_j$

Figure 8: Token-based Reservation Protocol for the BIP models in Figures 1 and 3.

at the reservation layer. We denote this implementation by TR.

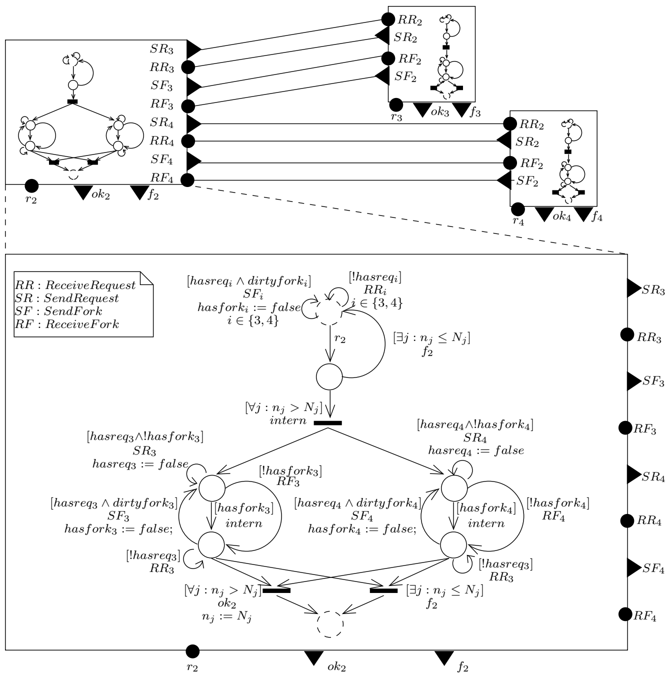
4.3.3 *Implementation Based on Dining Philosophers*

A third choice of Reservation Protocol algorithm is an adaption of the hygienic solution to the dining philosophers problem presented in [2,7]. Its Send/Receive BIP implementation is presented in Figure 9. Similar to token ring, each externally conflicting interaction is handled by a separate component. If two interactions are conflicting, the two corresponding components share a fork carrying N_i variables corresponding to the atomic components causing the conflict. In order to positively respond to a reservation request, a component has to fetch all forks shared with its neighbors. Then, it compares participation numbers received from the reservation request with the numbers in forks and responds accordingly. After such a response, the forks become dirty. Finally, the component sends the forks if it is asked to do so. We denote this implementation by DP.

4.4 Cross-Layer Interactions

In this subsection, we define the interactions of our 3-layer model. Following Definition 4, we construct Send/Receive interactions by specifying which one is the sender. Given a BIP model $\gamma(B_1 \cdots B_n)$, a partition $\gamma_1 \cdots \gamma_m$, and the obtained Send/Receive components $B_1^{SR} \cdots B_n^{SR}$, interaction protocol components $IP_1 \cdots IP_m$, and Reservation Protocol components $RP_1 \cdots RP_k$, we construct the Send/Receive interactions γ^{SR} according to Definition 4 as follows:

- For each component B_i, γ^{SR} contains a multicast connector formed by all ports o_i, where B_i is the sender.
- For each Interaction Protocol component IP_j and port p in IP_j, we include a binary interaction, such that port p of IP_j is the sender, and, port p of the corresponding component in the components layer is the receiver.
- For each interaction a that is in external conflict, γ^{SR} contains an interaction between r_a ports, such that the Interaction Protocol is the sender and Reservation Protocol is the receiver. Likewise, γ^{SR} contains interactions between ok_a and f_a ports.



SR_3
RR_3
SF_3
RF_3
SR_4
RR_4
SF_4
RF_4
RR_2
SR_2
RF_2
SF_2
r_2
ok_2
f_2
r_3
ok_3
f_3
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 $i \in \{3,4\}$
r_2
$[\exists j : n_j \leq N_j]$ f_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
$[hasreq_3 \wedge dirtyfork_3]$ 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
$[!hasreq_3]$ RR_3
$[\forall j : n_j > N_j]$ ok_2 $n_j := N_j$
$[\exists j : n_j \leq N_j]$ f_2

Figure 9: Dining philosophers-based Reservation Protocol for the BIP models in Figures 1 and 3.

Note that these interactions do not depend on the Reservation Protocol. The entire model obtained is denoted B_{RP}^{SR}, B_{TR}^{SR} or B_{DP}^{SR} following the embedded Reservation Protocol. The interactions between the three layers of our running example are presented in Figure 3. The send-ports are graphically denoted by triangles and receive-ports by bullets.

5. CORRECTNESS

In Subsection 5.1, we show that our 3-layer model meets the constraints of the Send/Receive model specified in Section 3. In Subsection 5.2, we prove that a BIP model is observationally equivalent with the BIP model obtained by the transformation of Section 4. Finally, we prove the correctness of models embedding different implementations of Reservation Protocol in Subsection 5.3.

5.1 Compliance with Send-Receive Models

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

PROOF. The send-ports and receive-ports are clearly determined in subsection 4.4 and respect the syntax presented in the two first points of definition 4. We now prove the third point, that is whenever a send-port is enabled, all its associated receive-ports are enabled.

Between the 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 ok or fail interaction takes place we switch back to the initial configuration.

Between components and Interaction Protocol layers, for all interactions involving component B_i, it is sufficient to consider only the places w_i, r_i and s_p for each port p exported by B_i in 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 might be active since it would require one of the token 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 an a place s_p, it comes either from an a or an ok_a labeled-transition. 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, the ok_a was 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. ■

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

Definition 7 (Weak Simulation) *A weak simulation over A and B, denoted $A \subset B$, is a relation $R \subseteq Q_A \times Q_B$, such that we have $\forall (q,r) \in R,\ a \in P :\ q \xrightarrow{a}_A q' \implies \exists r' : (q', r') \in R \wedge r \overset{\beta^* a \beta^*}{\rightarrow}_B r'$ 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} are observationally equivalent. We consider the correspondence between actions of B and B^{SR} as follows. For each interaction $a \in \gamma$, where γ is the set of interactions of B, we associate either the binary interaction ok_a or the unary interaction a, depending upon existence of an external conflict. All other interactions (offer, response, reserve, fail) are unobservable and denoted 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 components layer and the Interaction Protocol (namely offer and response), and β_2 actions that are communications between the Interaction Protocol and and Reservation Protocol (namely reserve and fail). We denote q^{SR} a state of B^{SR} and q a state of B. A state of B^{SR} 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 component 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_i^{SR} 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 ordering of β_1 action does not change the final state. Thus $[q]^{SR}$ is unique. ■

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} is in a stable state, for each couple (i,j), such that B_i is involved in interactions handled by IP_j, we have $B_i.n_i = IP_j.n_i > RP.N_i$.*

PROOF. When in stable state, all offers have been sent, thus the participation numbers in Interaction Protocol correspond to those in components $B_i.n_i = IP_j.n_i$.

Initially, for each component B_i, $RP.N_i = 0$ and $B_i^{SR}.n_i = 1$ thus the property holds. The N_i variables in Reservation Protocol are updated on a ok transition, using values provided by the Interaction Protocol, that is by the components. We show that after each ok_a transition, the property still holds. For each component B_i^{SR} participant in a, it holds that $B_i^{SR}.n_i = RP.N_i$ after the offer. Then, the response transitions increments participation numbers in components such that in the next stable state $B_i^{SR}.n_i > RP.N_i$. For components $B_{i'}$ not participating in a, by induction we have $B_i^{SR}.n_{i'} > RP.N_{i'}$ and only participation numbers in components can be incremented. ■

Since we need to take into account participation numbers n_i, we introduce an intermediate centralized 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 set of states and transitions labeled by the same ports, they are observationally equivalent. (They are even strongly bisimilar.)

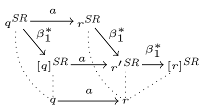
Lemma 3 *$B \sim B^n$.*

PROOF. We say that two states (q, q^n) of B and B^n are equivalent if they have the same control states. This defines a bisimulation. ■ We are now ready to state and prove our central result.

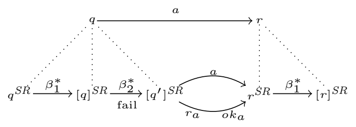
Proposition 2 *$B^{SR} \sim B^n$.*

PROOF. We define a relation R between the states Q^{SR} of B^{SR} and the 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 $(r^{SR}, q) \in R$.



(b) Point (ii)



(c) Point (iii)

Figure 10: Proof of observational equivalence

(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 $[q]^{SR} = [r]^{SR}$, either 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^{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 a or r_a transition in Interaction Protocol, with the participation numbers n_i. 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 fr_a place is empty.

At stable state $[q]^{SR}$, the 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 state q^{SR} and $[q]^{SR}$, which implies $q \xrightarrow{a}$.

Since the β_1 actions needed to reach the state $[q]^{SR}$ did not interfere with action a, we can replay them from r^{SR} to reach a state r'^{SR}, as shown on figure 10. The state r'^{SR} is not stable because of response and offers that can take place in each component participant in B_i. Executing these actions brings the system in state $[r']^{SR}$ which is clearly equivalent to r, and by point (i) we have $(r^{SR}, r) \in R$.

(iii) In figure 10, we show the different actions and states involved in this part. From q^{SR}, we reach $[q]^{SR}$ by doing β_1 actions. Then we execute all possible fail interactions (that 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, the interaction a is enabled, else the sequence r_a ok_a can be executed since lemma 2 ensures that guard of ok_a is true. In both cases, the interaction corresponding to a brings the system in 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 $(r^{SR}, r) \in R$. ■

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. We also proposed two other implementations, respectively, token-ring TR and dining philosophers DP. 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. 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.

Nevertheless, we can prove an 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 they do not enforce progress. But, they play a technical role in proving the correctness of our approach. The following proposition establishes the relation between the different implementations of the Reservation Protocol.

Proposition 3 *(i)* $RP \sim TR^{(w)} \sim DP^{(w)}$
(ii) $TR \subset TR^{(w)}$, $DP \subset DP^{(w)}$.

Let us denote by B_X^{SR} the 3-layer model obtained from the initial system B and embedding algorithm X in the Reservation Protocol. Also, let us denote $Tr(B)$ the set of all possible traces of observable actions allowed by an execution of B. The following proposition states the correctness of our implementation.

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

Proof. (i) The leftmost equivalence is 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) The trace inclusions follows from the simulations $TR \subset TR^{(w)}$ respectively $DP \subset DP^{(w)}$ ■

6. EXPERIMENTAL RESULTS

In this section, we present the results of our experiments. Our implementation automatically generates C++ code from the 3-layer BIP model developed in Sections 3 and 4, where Send/Receive interactions are implemented by TCP sockets primitives. Code generation involves generating a stand-alone executable for each Send/Receive component in each layer of the 3-layer BIP model. The code of each component simulates its automaton or Petri net using the technique presented in [5].

In the following, we denote each experiment scenario by (i, X), where i is the number of interaction partitions and X is the choice among the three Reservation Protocols described in Subsection 4.3 (i.e., RP, TR, or DP). For the case

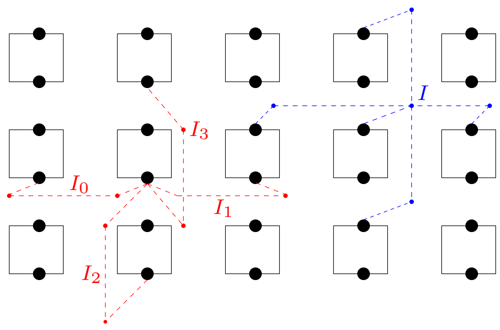


Figure 11: Partial BIP model for diffusing computations.

where partitioning of interactions results in having no external conflicts and, hence, requiring no reservation component, we use the symbol '$-$' to denote an empty Reservation Protocol. All experiments in this section 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 partitioning may result in significantly different performance.

For our experiments, we model a simplified version of Dijkstra-Scholten termination detection algorithm for diffusing computations [9] in BIP. *Diffusing computation* is the task of propagating a message across a distributed system; i.e., a wave that starts from an initial node and diffuse to all processes in a distributed system. Diffusing computation has numerous applications such as traditional distributed deadlock detection and reprogramming of 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 is complete once all its children are complete. In this setting, when the root is complete, termination is detected.

Our BIP model has $n \times m$ atomic components (see Figure 11 for a partial model). Each component participates in two types of interactions: (1) four binary rendezvous interactions (e.g., $I_0 \cdots I_3$) 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., I) for the completion wave, as each parent has to wait for all its children to complete.

Our first set of experiments is on a 6×4 torus. We apply different partitioning scenarios as illustrated in Figure 12. Figure 13(a) shows the time needed for 100 rounds of 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 of $(2, -)$, all interactions of the propagation wave are grouped into one

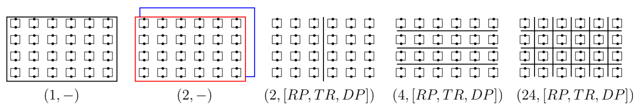


Figure 12: Different scenarios for diffusing computations.

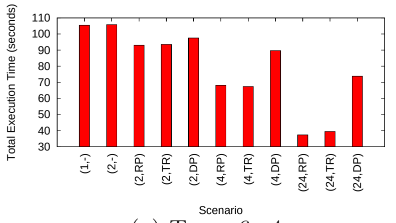
component of the Interaction Protocol and all interactions related to the completion wave are grouped into the second component. Such grouping does not allow parallel execution of interactions. This is the main reason that the performance of $(1, -)$ and $(2, -)$ are the worst in Figure 13(a).

Next, we group all interactions involved in components $1 \cdots 12$ into one component and the rest in a second component of the Interaction Protocol. This constitutes experiments $(2, RP)$, $(2, TR)$, and $(2, DP)$. Such partitioning allows more parallelism during propagation and completion waves, as an interaction in the first partition can be executed in parallel with an interaction in the second partition[1]. This is why the performance of $(2, RP/TR/DP)$ is better than $(1, -)$ and $(2, -)$. Now, since almost all propagation interactions conflict with each other and so do all completion interactions, in case of the dining philosophers algorithm, the conflict graph is not dense. Hence, a small of number decisions can be made in a local neighborhood of philosophers. It follows that the performance of $(2, TR)$ is quite competitive with $(2, DP)$. It can also be seen that $(2, RP)$ performs as good as $(2, TR)$ and $(2, DP)$. This is due to the fact that there exist only two partitions, which results in a low number of reservation requests.

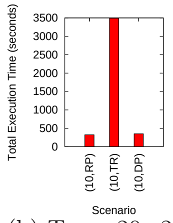
Figure 13(a) also shows the same type of experiments with 4 and 24 partitions. Similar to the case of two partitions, the performance of TR and RP for 4 and 24 partitions compete with each other. However, RP and TR outperform DP. This is due to the fact that in case of DP, each philosopher needs to acquire 4 forks, which requires considerable communication. On the other hand, 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 in case of a 6×4 torus is not high enough to overcome the communication volume.

In the next experiment, following the lesson learned from the tradeoff between communication volume and parallelism, we design a scenario where we exploit the fact that each reservation component in DP resolves conflict through communicating with its neighboring components. This is not the case in TR. Thus, we consider a 20×20 torus. As can be seen in Figure 13(b), the performance of DP is significantly better than TR. This is solely because when we have a large number of components, in TR, the token has to travel a long way in order to allow parallel execution of interactions. On the contrary, in DP, the Reservation Protocol components act in their local neighborhood and although more communication is needed, it allows better concurrency and, hence, higher simultaneous execution of interactions. We expect that by increasing the size of the torus, DP outperforms RP as well.

[1]Execution of each interaction involves 10ms suspension of the corresponding component in the Interaction Protocol to perform and I/O command.



(a) Torus 6×4



(b) Torus 20×20

Figure 13: Performance of termination detection in diffusing computation in different scenarios.

We conclude this section by stating the main lesson learned from our experiments:

> *Different partitioning schemes 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.*

7. CONCLUSION

We focused on developing a generic framework for automated transformation of high-level BIP models in terms of a set of components glued by *rendezvous* interactions into distributed implementations. In a distributed setting, implementation of a multi-party rendezvous results in solving the *committee coordination problem* [7], where a set of professors are organized in a set of committees and two committees can meet concurrently only if they have no professor in common; i.e., they are not *conflicting*. Our transformation consists of two steps. First, it 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 after resolving conflicts either locally or by the help of the third layer, and (3) the *Reservation Protocol* resolves conflicts in a distributed fashion. The Reservation Protocol implements a committee coordination algorithm and our design allows employing any such algorithm. The second step of our transformation takes the intermediate three-layer BIP model as input and generates C++ executables using TCP sockets for communication. We reported the lessons learned through conducting several experiments using different algorithms in the Reservation Protocol and partitioning schemes. As predicated, there is no silver bullet to automate code generation of distributed applications. Hence, designers must have access to a formal framework and a rich library of algorithms such as the ones presented in this paper to develop correct and yet efficient distributed applications automatically.

For future work, we are considering several research directions. An important extension is to allow the Reservation Protocol to incorporate different algorithms for conflict resolution simultaneously, 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 and distributed independent set for better understanding of tradeoffs between parallelism, load balancing, and network traffic. Another important line of research is to study the overhead of our transformation where communication cost is crucial such as in peer-to-peer and large sensor networks. Finally, given the recent advances in the multi-core technology, we plan to customize our transformation for multi-core platforms.

8. REFERENCES

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

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

[3] A. Basu, 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.

[4] 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.

[5] 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.

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

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

[8] 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.

[9] E. W. Dijkstra and C. S. Scholten. Termination detection for diffusing computations. *Information Processing Letters*, 11(1):1–4, 1980.

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

[11] 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.

[12] 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.