Automated Conflict-Free Distributed Implementation of Component-Based Models

Borzoo Bonakdarpour Marius Bozga Mohamad Jaber
Jean Quilbeuf Joseph Sifakis

VERIMAG, Centre Équation, 2 avenue de Vignate, 38610, Gières, France

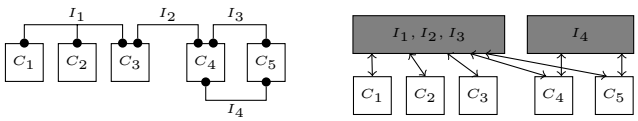
***Abstract*—We propose a method for generating distributed implementations from high-level models expressed in terms of a set of components glued by rendezvous interactions. The method is a 2-phase transformation preserving all functional properties. The first phase is a source-to-source transformation from global state to a partial state model (to relax atomicity). This transformation replaces multi-party rendezvous interactions by send/receive primitives managed by a set of automatically generated distributed schedulers. These schedulers are conflict-free by construction in the sense that they do not require communication in order to safely execute interactions of the high-level model. In the second phase, from the transformed model in phase one, we generate C++ distributed code using either TCP sockets or MPI to implement send/receive primitives. Our method is fully implemented in a tool for automatic generation of distributed applications. We present experimental results using different case studies.**

Keywords. Component-based modeling, Automated transformation, Distributed systems, BIP, Correct-by-construction.

I. Introduction

As computing systems become more complex, the need for their decentralization is becoming increasingly apparent in many domains. Examples include super-computers for increasing computing power, the world-wide-web for sharing information, distributed fault-tolerant systems for gaining robustness and reliability, and complex networks of sensors and actuators for coping with hostile physical environments in deeply embedded systems or the so-called cyber-physical systems. Although many languages and techniques have been proposed in distributed computing (e.g., [4], [12], [14], [15], [18]), constructing correct distributed applications is still time-consuming, error-prone, and hardly predictive. Thus, it is highly desirable to develop new methodologies that enable us to automatically construct correct distributed applications by starting from high-level models without getting involved in low-level code development.

In this paper, we focus on the BIP formalism as our high-level modeling language. BIP (Behavior, Interaction, Priority) is a component-based framework with formal semantics relying on multi-party interactions for synchronizing components and dynamic priorities for scheduling between interactions. Figure 1(a) shows a simple BIP model, where *components* $C_1, \ldots, C_5$ interact via rendezvous *interactions* $I_1, \ldots, I_4$. An interaction is formed by a set of *ports* graphically denoted by bullets. In BIP, the behavior of each component is described by



(a) A simple high-level model (b) Distributed version of Figure 1(a)

Fig. 1. Abstract overview of the transformation

a Petri net or automaton extended by data, whose transitions are labeled by ports. Whenever a transition is possible, we say that the associated port is *enabled*. A rendezvous interaction is enabled when all its participating ports are enabled. A sequential implementation for BIP models is obtained by using a centralized Scheduler that has consistent view of the global state of the system. The Scheduler orchestrates the behavior of components by repeating the following straightforward operations:

1) It computes the set of enabled interactions at the current global state.
2) It selects one interaction among this set and executes the associated computation.
3) It calls sequentially each component involved in the selected interaction to execute the corresponding transitions and waits for their completion.

Unlike the sequential setting, deriving a distributed implementation from a high-level model such as the one in Figure 1(a) is not a trivial task. For instance, the system must respect the global state semantics although it works in a distributed setting where components do not have a global view of the system. Moreover, suppose that interactions I_1 and I_2 are enabled simultaneously. Since these interactions share component C_3, they cannot be executed concurrently. We call such interactions *conflicting*. Obviously, a distributed scheduler must ensure that conflicting interactions are mutually exclusive.

The problem of distributed conflict resolution is known as the *committee coordination problem* [11], where a set of professors organize themselves in different committees and two committees that have a professor in common cannot meet simultaneously. Several solutions to this problem have been proposed. Some use a particular set of manager processes to schedule interactions (e.g., [11]). Bagrodia [2] presents a solution based on message counting to detect enabledness of interactions in managers and a circulating token to ensure mutual exclusion. In a later paper [3], he replaces the token-

based solution by a reduction to the dining (or drinking) philosophers problem [10]. In [19], Perez et al present another solution using a lock-based mechanism instead of counters.

Contributions. Although the related work provides us with solutions to conflict resolution, dealing with real distributed implementations, requires better understanding of the problem beyond abstract algorithms and simulations. In this paper, we propose a novel method to transform high-level BIP models into distributed implementations that allows parallelism between components as well as parallel execution of non-conflicting interactions. Our method utilizes the following sequence of transformations:

1) First, we transform the given BIP model into another BIP model that (1) operates in partial state semantics (to relax atomicity), and (2) expresses multi-party interactions in terms of asynchronous message passing (send/receive primitives). This is obtained by replacing transitions in atomic components by a request/response mechanism.
2) We insert a set of distributed Schedulers, each handling a subset of conflicting interactions, such that each conflict is local to a Scheduler. In Figure 1(a), we suppose that I_4 does not conflict with any other interaction because of components inner structure. Thus, the transformed model (see Figure 1(b)) contains two Schedulers. Since the interactions of the first Scheduler do not conflict with those of the second one, they are conflict-free. Thus, the two Schedulers can resolve conflicts independently; i.e., no communication between them is required. In this context, maximum parallelism is achieved when each Scheduler contains only one interaction. However, the number of Schedulers depends on the structure of the model. In the worst case, there is only one Scheduler handling all interactions.
3) We transform the intermediate BIP model into actual C++ code that employs either TCP sockets or the Message Passing Interface (MPI) [14] for communication.

We conduct a set of experiments to analyze the behavior and performance of the generated code. Our experiments show that depending upon the structure of the model, the distributed code generated using our method exhibits little overhead. We also illustrate that in some cases, the performance of the generated code is competitive with the performance of hand-written code developed using MPI.

We also present two types of optimizations. First, we consider a transformation from BIP models directly to MPI code for cases where there is no conflict between interactions. Our experiments show that the performance of automatically generated MPI code is almost identical to the hand-written optimized code. Since most popular MPI applications fall in this category, we argue that this transformation assists developers of parallel and multi-core applications to start development from high-level BIP models and avoid getting involved in low-level synchronization details. The second optimization is for cases where distributed conflicts are more complex and adding distributed schedulers on top of MPI libraries adds considerable overhead. Our solution to this case involves a merging mechanism of computing and scheduling processes.

We emphasize that all our transformations preserve *observational equivalence*; i.e., the semantics of the high-level model are not modified during our transformations. In fact, our findings are opposite of the work in [13], where the author argues remote rendezvous should not be supported by Ada because of tremendous complexity and difficulties in semantics. Observational equivalence shows that the our derived distributed implementations are correct-by-construction, which makes our approach different from solutions such as in [8], where distribution is studied only from a practical point of view. In addition to the issue of semantics, our experiments also show insignificant overhead at runtime.

We also note that the problem studied in this paper is of different nature from the related work in distributed versions of Ada. Most of the work in this area focuses on task distribution in heterogeneous multi-processor platforms [17] and client-server implementation of remote rendezvous [8], where the issue of distributed conflicts are unimportant.

Organization. In Section II, we present the global state sequential operational semantics of BIP. Then, in Section III, we present our BIP to BIP transformation. Section IV describes transformation of the intermediate send/receive BIP model into C++ distributed code. Section V presents the results of our experiments. Our optimizations are discussed in Section VI. Finally, in Section VII, we make concluding remarks and discuss future work.

II. Basic Semantic Models of BIP

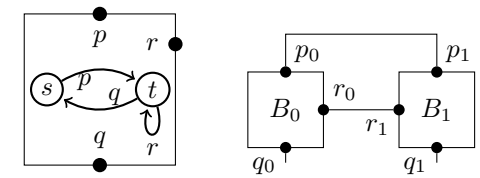
In this section, we present operational *global state* semantics of BIP. BIP is a component framework for constructing systems by superposing three layers of modeling: Behavior, Interaction, and Priority. Since the issue of priorities is irrelevant to this paper, we omit it.

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.

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



(a) Atomic component (b) Composite component

Fig. 2. BIP composite component

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 $I \subseteq \{1..n\}$.

As for atomic components, real 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 $\gamma \subseteq 2^P$. $B \stackrel{def}{=} \gamma(B_1, \ldots, B_n)$, 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 $\gamma = \{\{p_0, p_1\}, \{r_0, r_1\}, \{q_0\}, \{q_1\}\}$.

III. Transformation from High-Level BIP to Send/Receive BIP

As mentioned in the introduction, the first step of our solution is an intermediate transformation from a high-level BIP model into a message passing BIP model. More specifically, we transform a composite component $B = \gamma(B_1, \ldots, B_n)$ in global state semantics with multi-party interactions into another BIP composite component B^{SR} in partial state semantics that only involves binary "*Send/Receive*" interactions. To this end, we transform each atomic component B_i into an atomic Send/Receive component B_i^{SR} (described in Subsection III-A). We also add a set of atomic components $S_1^{SR}, \ldots, S_m^{SR}$ that act as schedulers. First, we describe how we build a centralized scheduler in Subsection III-B. We construct the interactions between Send/Receive components in Subsection III-C. Finally, we replace the centralized scheduler by a set of distributed schedulers in Subsection III-E.

Definition 3. *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 $a = (s, r)$ with $s \in P_s$ and $r \in P_r$, 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 receive-port r, such that $(s, r) \in \gamma^{SR}$. We say that r is the receive-port associated to s.*
- *If (s, r) is a send/receive interaction in γ^{SR} and s is enabled at some global state of B^{SR}, then r is also enabled at that state.*

Notice that the second condition requires that only one component can receive a "message" sent by another component. The last condition ensures that every Send/Receive interaction can take place as soon as the sender is enabled, *i.e.* the sender can send the message immediately.

A. Transformation of Atomic Components

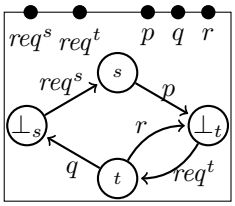
Let B_i be an atomic component. We now present how we transform B_i into a Send/Receive atomic component B_i^{SR} that is capable of communicating with the scheduler. There are two types of Send/Receive interactions: *request* and *response*. A request interaction from component B_i^{SR} informs the scheduler that B_i^{SR} is ready to interact through a set of enabled ports. When the scheduler selects an interaction involving B_i^{SR} for execution, it notifies the component by a response interaction that includes the port chosen.

Definition 4. *Let $B_i = (Q_i, P_i, \rightarrow_i)$ be an atomic component and s be a state in Q_i. The* request *associated to s is the set of ports $req^s = \{p \in P_i | s \xrightarrow{p}_i\}$. We denote the set of all requests from B_i by $REQ_i = \{req^s | s \in Q_i\}$.*

Since each response triggers an internal computation, following [5], 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_i into B_i^{SR}.

Definition 5. *Let $B_i = (Q_i, P_i, \rightarrow_i)$ be an atomic component. The corresponding Send/Receive atomic component is $B_i^{SR} = (Q_i^{SR}, P_i^{SR}, \rightarrow_i^{SR})$, where*

- *$Q_i^{SR} = Q_i \cup Q_i^{\perp}$, where $Q_i^{\perp} = \{\perp_s | s \in Q_i\}$.*



$req^s = \{p\}$
$req^t = \{q, r\}$

Fig. 3. Send/Receive atomic component of Figure 2-a.

- $P_i^{SR} = P_i \cup REQ_i$ *where, as we will see later,* P_i *are receive-ports and* REQ_i *are send-ports.*
- *For each transition* $(s, p, t) \in \rightarrow_i$*, we include the following two transitions in* $\rightarrow_i^{SR}$*:* $(s, p, \perp_t)$ *and* $(\perp_t,\ req^t, t)$*.*

Figure 3 illustrates transformation of the component in Figure 2-a into its corresponding Send/Receive component. Since there are two states in B, we have two request ports in B^{SR}: one for the request $req^s = \{p\}$ and one for the request $req^t = \{q, r\}$.

B. Building the Scheduler Component

In order to implement interactions, we add a new atomic component S, called the *scheduler component*. This component receives request messages sent by the Send/Receive atomic components. Based on the request messages received, the scheduler calculates the set of enabled interactions and selects one of them for execution. Then, it sends a response to each component involved in the selected interaction, so that they start their internal computations. We define the scheduler component as 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 $\{(l, \tau) \in L \times T | l \in {}^\bullet\tau\}$ and the edges $\{(\tau, l) \in T \times L | l \in \tau^\bullet\}$.

We depict the state of a Petri net by *marking* some 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 ${}^\bullet\tau$ 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 4 shows an example of a Petri net in two successive markings. This Petri net 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 right figure shows the resulting state of the left Petri net when transition t_2 is fired.

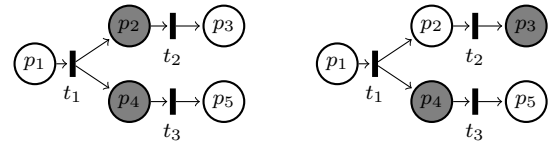


Fig. 4. An example of a simple Petri net

Intuitively, the Petri net that defines a scheduler component is constructed as follows. We associate a token with each request. This token circulates through three types of places: *waiting* places, *received* places, and *response* places. A transition from a waiting place to a received place occurs when a request is received. The set of marked received places determines the received requests and, thus, the enabled interactions. Transitions from received places to response places correspond to interactions. The execution of an interaction transition collects the required tokens in received places and puts them in appropriate response places. A transition from a response place to a waiting place sends the corresponding response.

Let $a = \{p_i\}_{i \in I}$ be an interaction. We say that a set of requests $\{req_i\}_{i \in I}$ *enables* a iff $\forall i \in I,\ p_i \in req_i$, that is, if for each port in a, there is one request of the set that provides this port. For each set of requests that enables a, we add a transition from the received to response places. Definition 7 formalizes the construction of the scheduler.

Definition 7. *Let* $B = \gamma(B_1, \ldots, B_n)$ *be a BIP composite component,* $REQ = \bigcup_{i=1}^n REQ_i$ *be the set of all requests and* $RES = \bigcup_{i=1}^n P_i$*, where* P_i *is the set of ports of* B_i*, be the set of all responses. We define the* centralized scheduler S *as a Petri net* (L, P, T) *where:*

- *The set* L *of places is the union of the following:*
  1) *The set* $\{w_{req} | req \in REQ\}$ *of* waiting places*.*
  2) *The set* $\{r_{req} | req \in REQ\}$ *of* received places*.*
  3) *The set* $\{s_{p.req} | req \in REQ, p \in req\}$ *of* response places*.*
- *The set* P *of ports is* $RES \cup REQ \cup \gamma$*, which are respectively send-ports, receive-ports and unary ports.*
- *The set* T *of transitions consists of the following:*
  1) *(waiting to received) For each request* $req \in REQ$*,* T *contains the request transition* (w_{req}, req, r_{req})*,*
  2) *(received to response) For each interaction* $a \in \gamma$ *and each set of requests* $\{req_j\}_{j \in J}$ *that enables* a*,* T *contains the transitions* $(\{r_{req_j}\}_{j \in J}, a, \{s_{p_j.req_j}\}_{j \in J})$*, where* $\forall j \in J,\ \{p_j\} = req_j \cap a$*.*
  3) *(response to waiting) For each request* $req \in REQ$*,* T *contains the set of response transitions* $\{(s_{p.req}, p, w_{req}) | p \in req\}$*.*

Figure 5 depicts the scheduler constructed for the composite component presented in Figure 2. The dotted places are the waiting places redrawn here for the sake of readability. Initially, all waiting places contain a token. In the depicted state, we assume that both request req_0^s and req_1^s have been received. Then, the execution of transition $p_0 p_1$ is possible and

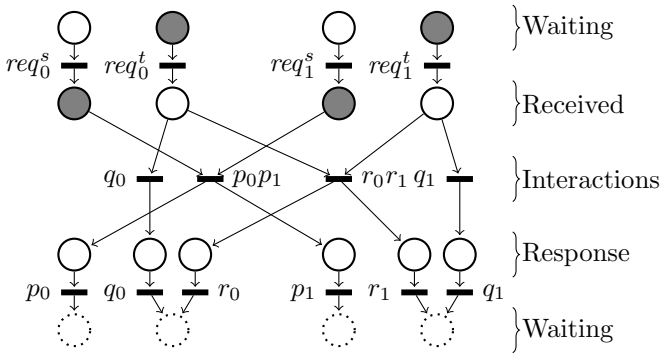


Fig. 5. A scheduler component for Figure 2

.

brings the tokensto response places. Then, these tokens return to their initial places by sending the responses p_0 and p_1.

C. Interactions between Send/Receive Atomic Components and the Scheduler

The next step of our transformation is to construct the set γ^{SR} of interactions between Send/Receive atomic components and the scheduler. To avoid confusion between ports of the scheduler and atomic components, we prefix the ports that appear in the scheduler by "S :" and we leave the ports of atomic components as they are.

Definition 8. *Let $B = \gamma(B_1, \ldots, B_n)$ be a composite component, $B_1^{SR}, \ldots, B_n^{SR}$ be the corresponding Send/Receive atomic components, and S be the scheduler constructed for B. The set of interactions γ^{SR} is the union of the following:*

- *The set of all request interactions from components to scheduler $\{(req, S{:}req) | req \in REQ\}$,*
- *The set of all response interactions from scheduler to components $\{(S{:}p,\ p) | p \in RES\}$, and*
- *The set of all unary interactions $\{\{S : a\} | a \in \gamma\}$ corresponding to interaction transitions in the scheduler.*

Observe that by construction of γ^{SR}, request ports are send-ports in atomic components and receive-ports in the scheduler S. Likewise, response ports are send-ports in the scheduler and receive-ports in atomic components. Unary ports of the scheduler (that are labeled by original interactions from γ) remain unary interactions.

Figure 6 shows the Send/Receive composite component by transforming the composite component in Figure 2-b. We use arrows to denote the direction of communications. For the sake of clarity, we have omitted the prefixes for naming the scheduler ports. Non-connected ports of the scheduler are unary interactions, that is interactions not subject to synchronization constraints.

D. Correctness

In order to prove correctness, we first show that the composite component B^{SR} that we have built thus far is a well-formed Send/Receive component. In particular, we have to verify that for each Send/Receive interaction, a receive-port is enabled when its corresponding send-port becomes enabled. Then, we

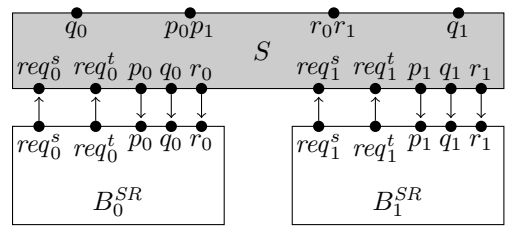


Fig. 6. A composite Send/Receive component

show that the composite component B^{SR} is observationally equivalent to the composite component B.

A state of the composite component B is given by the n-tuple $s = (s_1, s_2, \ldots, s_n)$ where s_i is the state of the component B_i. A state of the composite component B^{SR} also takes into account the state of the Scheduler which is described by its marking, denoted m. Thus, we will denote the state of B^{SR} by $s^{\perp} = (s_1^{\perp}, s_2^{\perp}, \ldots, s_n^{\perp}, m)$. We also denote by Q the set of all possible states of the composite component B and by Q^{SR} the set of all possible states of B^{SR}.

Lemma 1. *Let B^{SR} be the Send/Receive transformation of B. Then, for each Send/Receive interaction $(s, r) \in \gamma^{SR}$, whenever a send-port s becomes enabled, the associated receive-port r is already enabled.*

Proof: Intuitively, this property holds since each component starts listening to any response by the time it sends a request. Dually, the scheduler starts listening again to any request as soon as it sends the corresponding response.

Let B_i^{SR} be a Send/Receive atomic component. We show that all Send/Receive interactions involving B_i^{SR} meets the statement of the lemma. We abstract the state of S by considering only the information related to B_i and S. We distinguish the following cases, according to the state $(s_i^{\perp}, m)$:

i) $s_i^{\perp} = \perp_{s_0}, m \supseteq \{w_{req^s} | s \in Q_i\}$, where $\perp_{s_0}$ is a state of B_i^{SR}, and m contains all places w_{req^s} associated to requests from B_i^{SR}. The send-port req^{s_0} is enabled as well as the receive-port $S : req^{s_0}$. Thus, the property holds for the initial configuration, and in general for configurations of this form. Moreover, by executing this request interaction, we fall into the second situation.

ii) $s_i^{\perp} = s_0, m \supseteq \{r_{req^{s_0}}\} \cup \{w_{req^s} | s \in Q_i, s \neq s_0\}$. From this configuration, no send-port is enabled.

iii) $s_i^{\perp} = s_0, m \supseteq \{s_{p.req^s}\} \cup \{w_{req^s} | s \in Q_i, s \neq s_0\}$. Such a configuration is reached whenever the scheduler executes an interaction involving B_i^{SR}. In this state, since send-port p is enabled in S and receive-port p is enabled in B_i^{SR}, the corresponding response interaction is enabled. Moreover, by executing this response interaction, this case is reduced to the first case. ■

We now define *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 [16], where β-transitions are considered unobservable. The same

definition for atomic and composite BIP components trivially follows.

Definition 9 (Weak Simulation)**.** *A weak simulation over A and B is a relation $R \subseteq Q_A \times Q_B$ such that we have $\forall (q,r) \in R,\ a \in P:\ q \xrightarrow{a}_A q' \implies \exists r':\ (q',r') \in R \wedge r \xrightarrow{\beta^* a \beta^*}_B r'$ and $\forall (q,r) \in R:\ q \xrightarrow{\beta}_A q' \implies \exists r':\ (q',r') \in R \wedge r \xrightarrow{\beta^*}_B r'$*

A weak bisimulation over A and B is a relation R such that R and R^{-1} are 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 the context of our problem, observable events are interactions of the high-level BIP model, that is, unary interactions in the Send/Receive model, and, unobservable events are Send/Receive interactions, that is, binary interaction. Let us fix some notations. Let $s^\perp, t^\perp \in Q^{SR}$ be two states of B^{SR} and $a \in \gamma^{SR}$ be an interaction such that $s^\perp \xrightarrow{a}_{SR} t^\perp$. We rewrite $s^\perp \xrightarrow{\beta}_{SR} t^\perp$ if a is a Send/Receive interaction, otherwise a is a unary interaction and is observable in B^{SR}. It can be shown that the relation $\xrightarrow{\beta}_{SR}$ is terminating and confluent. Formally, for any state $s^\perp$, there is a unique state $[s^\perp]$ that can be reached by executing all the possible Send/Receive interactions, after a finite number of steps, that is, $s^\perp \xrightarrow{\beta^*}_{SR} [s^\perp]$ and $[s^\perp] \not\xrightarrow{\beta}_{SR}$.

Lemma 2. *Let B be a composite component and B^{SR} be its Send/Receive version. B and B^{SR} are observationally equivalent when hiding all Send/Receive interactions in B^{SR}.*

Proof : We define the relation $R = \{(s, s^\perp) \in Q \times Q^{SR} | \forall 1 \leq i \leq n \quad : [s^\perp]_i = s_i\}$. It can be shown that R is an observational equivalence as follows. Let $s, t \in Q$ be some states of B, $s^\perp, t^\perp \in Q^{SR}$ be some states of B^{SR}, and $a \in \gamma$ an interaction. It follows that:

i) If $(s, s^\perp) \in R$ and $s^\perp \xrightarrow{\beta}_{SR} t^\perp$, then $(s, t^\perp) \in R$.
ii) If $(s, s^\perp) \in R$ and $s^\perp \xrightarrow{a}_{SR} t^\perp$,f then $\exists t \in Q$ such that $s \xrightarrow{a} t$ and $(t, t^\perp) \in R$
iii) If $(s, s^\perp) \in R$ and $s \xrightarrow{a} t$ then $\exists t^\perp \in Q^{SR}$, such that $s^\perp \xrightarrow{\beta^* a} t^\perp$ and $(t, t^\perp) \in R$

All these conditions can be checked depending on the structure of the state in a similar way to [5]. ■

E. Decentralized Scheduler

The idea behind decentralization is to decompose the centralized scheduler component into a set of "disjoint" scheduler components. Let $S = (L, P, T)$ be a centralized scheduler. A decomposition of $S = \bigcup_{j=1}^m S_i$, is a set of 1-safe Petri nets $S_i = (L_i, P_i, T_i)$ such that $L = \bigcup_{j=1}^m L_j$, $P = \bigcup_{j=1}^m P_j$ and $T = \bigcup_{j=1}^m T_j$. We say that a decomposition is *disjoint* if both $L = \bigcup_{j=1}^m L_j$ and $T = \bigcup_{j=1}^m T_j$ are disjoint unions.

Reconsider the Petri net depicted in Figure 5. As shown in Figure 7, it can be decomposed into two disjoint Petri nets, the gray one and the black one. Thus, we build one scheduler for each of these Petri nets. Observe that such decomposition can

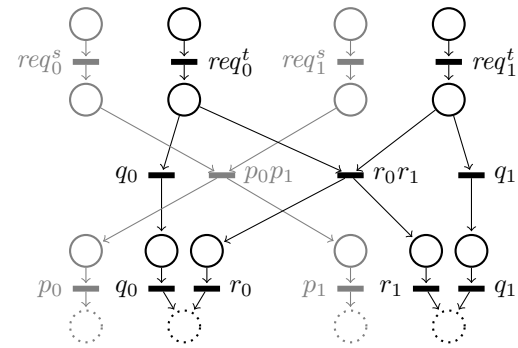


Fig. 7. Splitting the scheduler

be automatically achieved through a simple syntactic check by computing the transitive closure of each connected component in the scheduler developed in Subsection III-B.

Note that any decomposition of the transitions or places is not valid. Some places and transitions cannot be disconnected. For instance, in Figure 7, transitions r_0r_1 and q_1 are in the same Scheduler (the black one) because they have a common input place. Any decomposition separating r_0r_1 and q_1 cannot be disjoint, since the common input place would be duplicated, and then, the global state semantics will be violated.

Since the overall structure of the system changes, we need to redefine the Send/Receive interactions. Let first consider the situation of request ports $req \in REQ$. Since there is only one req labeled transition in the centralized scheduler S, there is only one decentralized scheduler S_i that contains this transition and the associated port req. We denote this port by $S_{j_{req}}:\ req$. The situation of response ports $p \in RES$ is different. The same response port $p \in RES$ can label multiple transitions in S, thus there might be more than one scheduler S_j that triggers the port p. If the response port p is contained in the decentralized scheduler S_j, we denote it $S_j : p$. The formal definition is provided below.

Definition 10. *Let $\gamma^{SR}(B_1^{SR}, \ldots, B_n^{SR}, S)$ be a Send/Receive composite component and $S_1, \ldots, S_m$ be a disjoint decomposition of S. The set of interactions γ_2^{SR} is the union of the following :*

- *The set of all requests from components to schedulers $\{(req, S_{j_{req}} : req) | req \in REQ\}$*
- *The set of all responses from schedulers to components $\{(S_j : p,\ p) | p \in P_j\}$*
- *The set of all unary interactions $\{\{S : a\} | a \in \gamma\}$*

Then we define the decentralized Send/Receive version of B, denoted $B_2^{SR} = \gamma_2^{SR}(B_1^{SR}, \ldots, B_n^{SR}, S_1, \ldots, S_m)$. Figure 8 presents the decentralized version of the composite component originally presented in Figure 2. The gray Petri net from Figure 7 is S_1 and the black one is S_2.

Theorem 1. *B_2^{SR} is observationally equivalent to B^{SR}.*

Proof : The centralized scheduler S in B^{SR} is the union of the decentralized schedulers $S_1, \ldots, S_m$. Thus, we can say that a state of B^{SR} and a state of B_2^{SR} are equivalent if the marked places are the same. This relation is an observational

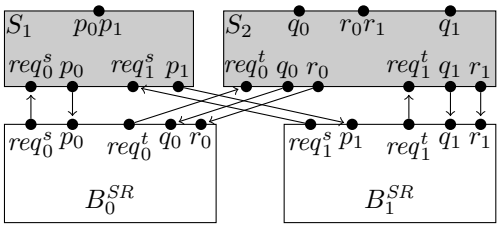


Fig. 8. A Send/Receive composite component with Decentralized Scheduler

equivalence since the marked places enable the same interactions in both models. ■

IV. Transforming Send/Receive BIP into C++

In this section, we describe how we generate pseudo C++ code for a scheduler and a Send/Receive BIP atomic component. Notice that since the behavior of these components are formalized as Petri nets, we only present generation of C++ code for a Petri net whose transitions are labeled by send-ports, receive-ports, or unary port (see C++ Pseudo Code 1).

Initially, each component creates a TCP socket and establishes reliable connections with all components that it needs to interact (Lines 1-2). These interactions and their corresponding physical connections are determined according to the complete Send/Receive BIP model and a *configuration file*. This file specifies the IP address and TCP port number of all components for final deployment. We assign one Boolean variable to each place of the given Petri net, which shows whether or not the place contains the token. Thus, the initial state of the Petri net is determined by an initial assignment of these variables (Line 3).

After initializations, the code enters an infinite loop that executes the transitions of the Petri net as follows. For each step, 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 11-15). Actual emission of data is performed by an invocation of the TCP sockets system call send() in Line 7. Once data transmission or an internal computation is completed, tokens are removed from input places and put in output places of the corresponding transitions (Lines 8 and 13).

Finally, if no send-port is enabled and all internal computations are completed, execution stops and waits for messages from other components (Line 17). Once one of the sockets contains a new message, the component resumes its execution and receives the message (Line 18). Note that based on the structure of Send/Receive components and schedulers developed in Section III, it is straightforward to observe that our code avoids creating deadlocks by giving priority to send-ports and unary-port. Moreover, sending messages before doing internal computation triggers receiver components waiting for a response and increases parallelism.

V. Experimental Results

We have implemented and integrated the transformations described in Sections III and IV in the BIP toolset. The tool

C++ Pseudo Code 1 Petri net

Input: A Petri net of a Send/Receive BIP component and a configuration file.
Output: C++ code that implements the given Send/Receive Petri net

```
    // Initializations
 1: CreateTCPSocket();
 2: EstablishConnections();
 3: PrepareInitialState();

 4: while true do
 5:     // Handling send-ports and internal computations
 6:     if there exists an enabled transition labeled by a send-port then
 7:         send(...);
 8:         PrepareNextState();
 9:         continue;
10:     end if
11:     if there exists an enabled transition labeled by a unary port then
12:         DoInternalComputation();
13:         PrepareNextState();
14:         continue;
15:     end if

16:     // Handling receiving messages
17:     select(...);
18:     recv(...);
19:     PrepareNextState();
20: end while
```

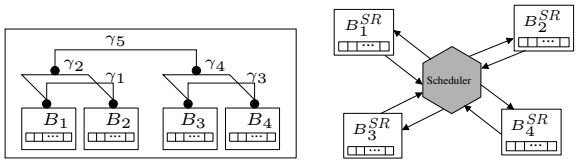
takes a composite BIP model in the global state semantics and a network configuration file as input and generates the corresponding C++ executable for each atomic component and scheduler. Each executable can be run independently on a different machine or a processor core.

We now present the results of our experiments for two sorting algorithms often used as parallel computing benchmarks. The structure and behavior of the two benchmarks are considerably different in terms of conflicting interactions, number of schedulers, and the required computation and communication times. All experiments in this section are run on (single or dual-core) 2.2 GHz Intel machines running under Debian Linux connected through a dedicated 100 Mb/s Ethernet network. We consider five different configurations: $1c$, $2c$, $2c'$, $4c$ and $4c'$, which denote respectively, one single-core machine, one dual-core machine, two single-core machines, two dual-core machines, and four single-core machines.

Moreover, for each experiment we compare the performance of the BIP generated code against a handwritten MPI program, implementing the same sorting algorithm and deployed on the same configuration.

A. Network Sorting Algorithm

We consider 2^n atomic components, each of them containing an array of N items. The goal is to sort all the items, so that the items in the first component are smaller than those of the second component and so on. Figure 9-a shows the high-level model of the Network Sorting Algorithm [1] for $n = 2$ using incremental and hierarchical composition of



(a) High-level BIP model (b) Send/Receive BIP model

Fig. 9. Network Sorting Algorithm.

components[1]. The atomic components $B_1 \ldots B_4$ are identical. Each atomic component computes independently the minimum and the maximum values of its array. Once this computation completes, interaction γ_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 γ_2 gets enabled. This interaction exports the minimum of B_1 and the maximum of B_2 to interaction γ_5. The same principle is applied in components B_3 and B_4 and interactions γ_3 and γ_4. Finally, interaction γ_5 works in the same way as interaction γ_1 and swaps the minimum and the maximum values, if they are not correctly sorted. Notice that all interactions in Figure 9-a are in conflict and, hence, our transformation constructs a single scheduler that encompasses all these interactions (see Figure 9-b), and which cannot be decomposed. Moreover, let us remark that the handwritten MPI program has an identical structure, that is several components and one scheduler to deal with communications. We run two sets of experiments for $n = 1$ (2 atomic components) and $n = 2$ (4 atomic components).

Case $\mathbf{n = 1}$. We consider three different configurations: $1c$, $2c$ and $2c'$. For $1c$, we use a single-core machine, which runs the two atomic components and the scheduler. For $2c$, we use one dual-core machine, where each core runs an atomic component and one of the cores runs also the scheduler. The component distribution is similar for $2c'$, except that the cores now are in different machines.

The results are reported in Table I for arrays of size $k \times 10^4$ elements, for $k = 20, 40, 80, 160$. In general, the generated BIP code outperforms the equivalent MPI program. For instance, the execution time for sorting an array of size 80×10^4, for the configuration $2c$ is: 669 seconds for MPI, and 600 seconds for BIP. Moreover, the difference is more important for an array of size 160×10^4, for the configuration $2c'$: 3090 seconds for MPI and only 2601 seconds for BIP. As expected, in the configuration $2c'$, we gain less speedup compared to $2c$, both for MPI and BIP, because of the network communication overhead (for this example the number of messages sent by each component is equal to the size of the array $\times 2$). Furthermore, for the configuration $1c$, we notice an important overhead due to context switching between processes which appears to be more significant in the case of MPI.

[1]We note that a composite component obtained by composition of a set of atomic components (as described in Section II) can be composed with other components in a hierarchical and incremental fashion using the same operational semantics. It is also possible to flatten a composite component and obtain a non-hierarchical one [9].

k	MPI (handwritten)			C++/Socket (generated)		
	$1c$	$2c$	$2c'$	$1c$	$2c$	$2c'$
20	118	40	60	105	34	100
40	497	157	198	409	133	256
80	1936	669	764	1526	600	758
160	8259	2833	3090	5819	2343	2601

TABLE I
PERFORMANCE OF NSA ($n = 1$).

Case $\mathbf{n = 2}$. Again, we consider three configurations: $1c$, $4c$, and $4c'$. For $1c$, we use one single-core machine, where the four atomic components run along with the scheduler. For $4c$, we use two dual-core machines and place each atomic component on a different core. The scheduler is placed arbitrarily on one of the cores. For $4c'$, the distribution of components and scheduler is similar to $4c$.

The results are reported in Table II for arrays of size $k \times 10^4$ elements, for $k = 20, 40, 80, 160$. We remark that the MPI program outperforms the corresponding BIP program. As can be seen in Table II the execution time for sorting an array of size 160×10^4, for the configuration $4c$ is: 2775 seconds for handwritten MPI, and 4621 seconds for BIP. This overhead is essentially due to communication. The number of messages exchanged is now four times bigger than for the case $n = 1$ and MPI provides a more efficient implementation for communication.

B. Bitonic Sorting

Bitonic sorting [7] is one of the fastest sorting algorithms suitable for distributed implementation in hardware or in parallel processor arrays. A sequence is called *bitonic* if it is initially nondecreasing then it is nonincreasing. 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 totally ordered sequence. We provide an implementation of the bitonic sorting algorithm in BIP using four atomic components, each one handling one part of the array. These components are connected as shown in the Figure 10. The six connectors are non conflicting. Hence, we use six schedulers for the distributed implementation. In this example each component sends only three messages, each one containing its own array.

We run experiments for three configurations: $1c$, $4c$, and $4c'$. For $1c$, we use one single-core machine, where the four atomic components along with the schedulers run. For $4c$, we use two dual-core machines and place each atomic component

k	MPI (handwritten)			C++/Socket (generated)		
	$1c$	$4c$	$4c'$	$1c$	$4c$	$4c'$
20	224	70	107	217	168	217
40	808	176	240	795	392	502
80	3239	655	789	3071	1792	1264
160	12448	2775	3217	11358	4621	3726

TABLE II
PERFORMANCE OF NSA ($n = 2$).

k	MPI (handwritten)			C++/Socket (generated)			MPI (direct transformation)		
	$1c$	$4c$	$4c'$	$1c$	$4c$	$4c'$	$1c$	$4c$	$4c'$
20	80	14	14	96	23	24	57	16	15
40	327	59	60	375	96	100	222	58	57
80	1368	240	240	1504	390	397	880	227	225
160	5605	1007	958	6024	1539	1583	3540	952	909

TABLE III
PERFORMANCE OF BITONIC SORTING ALGORITHM.

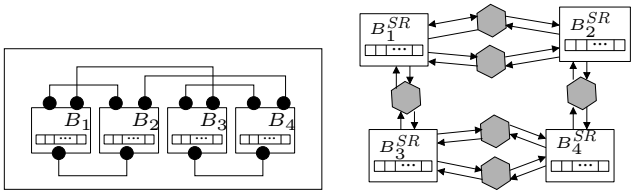
on a different core. We also distribute the schedulers over the four cores, such as to reduce the network communication overhead. For $4c'$, we use the same distribution for components and schedulers. The results are reported in Table III for arrays of size $k \times 10^4$ elements, and $k = 20, 40, 80, 160$. As can be seen in Table III the overall performance of MPI and BIP implementations are quite similar. For example, the execution time for sorting an array of size 80×10^4, for the configuration $4c$ is: 240 seconds for MPI, and 390 seconds for BIP. The overhead induced by the schedulers appears in the differences of performance between handwritten MPI and generated BIP code.

VI. OPTIMIZATIONS

In this section, we present two techniques that aim at reducing the overhead introduced by schedulers and amplified by high-level communication libraries such as MPI. These techniques reduce the number of components generated by the transformation presented in Section III. The first concerns cases, where no interactions are conflicting. This is the case for most parallel computing algorithms and in particular MPI applications such as bitonic sorting, matrix multiplication, tree adder, and the Linpack algorithm for solving linear systems (cf. Subsection VI-A). The second technique can be applied when all interactions that are handled by a scheduler share a common component (cf. Subsection VI-B).

A. Direct Transformation to MPI

Consider again the gray part of Figure 7. It corresponds to the scheduler S_1 in Figure 8 managing only one interaction, namely p_0p_1. This scheduler is only active (i.e., executing interaction code) when both B_0^{SR} and B_1^{SR} have sent a request to S_1 and are waiting for a response. Otherwise, this scheduler is waiting. Thus, we do not have parallel computation between S_1 and the participants in the interaction. This scenario demonstrates a scheduler that is acting only as proxy between two components and does not run in parallel with other schedulers and components.



(a) High-level BIP model (b) Send/Receive BIP model

Fig. 10. Bitonic Sorting Algorithm.

In general, let S be a scheduler that handles only interaction I. In this case, we can augment one of the participating components in I with S. We call such a component the *master* component. When the master component is ready to take part in I, instead of sending a request to S, it starts listening to requests from other components. When all requests from other participants in I have been sent, the master component executes the interaction code for I. Then, it sends responses to all other participants in I and continues its own execution. Using this technique, we reduce the number of components without losing any parallelism. Moreover, we remove the communication overhead between the master component and the scheduler. We have implemented this method within a BIP to MPI transformer. We use MPI collective communication primitives (`Gather` and `Scatter`) instead of Send/Receive to transfer data. The performance of this transformation for the bitonic sorting is shown in Table III. Observe that the automatically generated code outperforms the hand-written code slightly. This is due the fact that we used collective communications in generated MPI code, whereas the handwritten code used only send/receive primitives.

B. Merging

This technique is applied to the intermediate Send/Receive model developed in Section III. We generalize our observation in Subsection VI-A as follows: let S be a scheduler and B^{SR} be a Send/Receive component, such that each interaction handled by S involves B^{SR}. Hence, B^{SR} and S cannot run in parallel. If B^{SR} is running, then S has to wait for a request from B^{SR} and cannot execute any interaction. If S is computing, then B^{SR} has committed to an interaction in S and is waiting for the response from S and, hence, not running.

Since B^{SR} and S cannot run in parallel, we can merge them into one component without losing any parallelism. We obtain this result by using composition techniques as in [9]. More precisely, given two components and their interactions, we build their composition as a component whose behavior is expressed as a Petri net. We apply this technique to the bitonic sorting example, where each scheduler is responsible for one interaction involving two components (as shown in Figure 10). We merge each scheduler with one of these components. We obtain a BIP Send/Receive model containing four components.

Using this technique, we generated (1) C++/Socket code as described in Sections III and IV, and (2) MPI code by starting from Send/Receive BIP. The latter is implemented by simply replacing TCP sockets `send` and `receive` primitives by corresponding MPI primitives. The performance of case $4c$ (2 dual-core machines) configuration is shown in Table

k	S/R BIP		Merged S/R BIP	
	Socket	MPI	Socket	MPI
20	23	63	24	24
40	96	271	96	96
80	390	964	391	394
160	1539	4158	1548	1554

TABLE IV
THE IMPACT OF MERGING ON SEND/RECEIVE MODELS

IV. Observe that the performance of the C++/Socket code is approximately identical in both cases. This is because socket operations are interrupt-driven. Thus, if a component is waiting for a message, it does not consume CPU time. On the other hand, MPI uses active waiting, which results in CPU time consumption when the scheduler is waiting. Since we have two cores for five processes, the MPI code generated from the original Send/Receive model is much slower than the socket code. Nevertheless, as it appears in the table, reducing the number of components to one per core by merging allows the MPI code to reach the same speed as in the C++/socket implementation.

VII. CONCLUSION

In this paper, we proposed a novel method for transforming high-level models in BIP [6] into distributed implementations. The BIP (Behavior, Interaction, Priority) language is based on a semantic model encompassing composition of heterogeneous components. Our transformation consists of three steps: (1) we transform the given BIP model into another BIP model that operates in partial state semantics and expresses multi-party interactions in terms of asynchronous message passing using send/receive primitives, (2) we construct a set of Schedulers each executing a subset of conflicting interactions, and (3) we transform the intermediate BIP model into actual C++ distributed code that employs TCP sockets or MPI primitives for communication. We showed that our transformation preserves the *correctness* of the high-level model. We also provided two ways of optimizing the generated code. The first one generates directly efficient MPI code when there are no conflicts between interactions. The second one consists in merging components that could not run in parallel, thus reducing the number of components but not the parallelism.

We presented a set of experiments that validate the effectiveness of our approach in achieving parallelism regardless of the platform and architecture. Our experiments illustrated that depending upon the structure of the model, the distributed code generated using our methods exhibits little communication overhead. We also showed that in some cases, the performance of the generated code is competitive with the performance of hand-written code developed using the Message Passing Interface (MPI).

For future work, we plan to pursue several directions. One direction is introducing the notion of time in distributed semantics of BIP. Providing timing guarantees in a distributed setting has always been a challenge and BIP is not an exception. Another avenue to explore is to build a library of transformations based on different solutions to the conflict resolution problem. For instance, one can reduce our problem to distributed graph matching, distributed independent set, and distributed clique. These approaches would construct a wide range of designs for the distributed Scheduler, each appropriate for a particular application domain and platform. Thus, another future task is to identify a mapping from each transformation to an application domain and platform. Of course, a central issue that needs to be rigorously studied for each type of transformation and target language or platform is performance analysis and communication overhead. We are also working on a generic formal framework where different transformations can be applied in a plug-and-play manner.

REFERENCES

[1] M. Ajtai, J. Komlós, and E. Szemerédi. Sorting in $c \log n$ parallel steps. *Combinatorica*, 3(1):1–19, 1983.

[2] 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.

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

[4] J. G. P. Barnes, editor. *Ada 95 Rationale, The Language, The Standard Libraries*, volume 1247 of *Lecture Notes in Computer Science*. Springer, 1997.

[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] M. Bayassi, H. Bitteur, J.-F. Jézéquel, and P. Legrain. A practical use of the ada rendez-vous paradigm in distributed systems. In *Ada-Europe international conference on Ada*, 1992.

[9] 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.

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

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

[12] CORBA. `http://www.corba.org/`

[13] A. Gargaro. Towards distributed programming paradigms in Ada 9X. In *Washington Ada symposium on Ada*, 1993.

[14] W. Gropp, E. Lusk, and R. Thakur. *Using MPI-2: Advanced Features of the Message Passing Interface.* MIT Press, 1999.

[15] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

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

[17] H. Nishida, T. Itoh, and R. Nakayama. Distribution of Ada tasks onto a heterogeneous environment. In *TRI-Ada*, pages 155–165, 1991.

[18] OpenMP. `http://openmp.org/`

[19] 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.