HAL
open science

Facilitating the Implementation of Distributed Systems with Heterogeneous Interactions

Salwa Kobeissi, Adnan Utayim, Mohamad Jaber, Yliès Falcone

To cite this version:

Salwa Kobeissi, Adnan Utayim, Mohamad Jaber, Yliès Falcone. Facilitating the Implementation of Distributed Systems with Heterogeneous Interactions. IFM 2018 - 14th International Conference on integrated Formal Methods, Sep 2018, Maynooth, Ireland. pp.1-19. hal-01868748

HAL Id: hal-01868748

https://inria.hal.science/hal-01868748

Submitted on 5 Sep 2018

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

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Facilitating the Implementation of Distributed Systems with Heterogeneous Interactions

Salwa Kobeissi[1], Adnan Utayim[2], Mohamad Jaber[2], Yliès Falcone[3]

[1] University of Strasbourg, Inria, ICube Laboratory, Strasbourg, France
salwa.kobeissi@inria.fr
[2] American University of Beirut, Computer Science Department
{mmu00, mj54}@aub.edu.lb
[3] Univ. Grenoble Alpes, Inria, LIG, F-38000 Grenoble, France
ylies.falcone@univ-grenoble-alpes.fr

Abstract. We introduce HDBIP an extension of the Behavior Interaction Priority (BIP) framework. BIP is a component-based framework with a rigorous operational semantics and high-level and expressive interaction model. HDBIP extends BIP interaction model by allowing heterogeneous interactions targeting distributed systems. HDBIP allows both multiparty and direct send/receive interactions that can be directly mapped to an underlying communication library. Then, we present a correct and efficient code generation from HDBIP to C++ implementation using Message Passing Interface (MPI). We present a non-trivial case study showing the effectiveness of HDBIP.

1 Introduction

Developing correct and reliable distributed systems is challenging mainly because of the complex structures of the interactions between distributed processes. On the one hand, the use of abstract interaction models may simplify the development process but may deteriorate the performance of the generated implementation. On the other hand, the use of low-level primitives makes modeling error prone and time consuming. Although different frameworks [3,15] exist to model interactions between distributed processes, building correct, reliable and scalable distributed systems is still challenging and a hardly predictive task.

In this paper, we introduce HDBIP an extension of the Behavior, Interaction, and Priority (BIP) framework. BIP is a component-based framework used to model heterogeneous and complex systems. BIP has an expressive interaction model [5] that handles synchronization and communication between processes/components. Using only multiparty interactions simplifies the modeling of distributed barriers with local non-determinism, by automatically generating controllers to handle conflicts [6]. Nonetheless, restricting the language to only multiparty interactions affects the performance of the distributed implementations for instance to model a simple asynchronous send/receive primitive. In that case, the implementation requires an explicit buffer component/process. As such, this allows the creation of extra processes that are not needed. This extra buffer is practically duplicated as system buffers are usually provided by the low-level communication libraries (e.g., MPI). Moreover, it is

required to use multiparty interactions to connect the send primitives and the receive primitives with the explicit buffers. As such, those connections may introduce conflicts between themselves and between multiparty interactions, which may drastically affect the performance of the the distributed implementation.

This paper introduces HDBIP, which allows the modeling of both multiparty and asynchronous send receive interactions in an elegant way. Moreover, we provide an efficient code generation that allows by-construction to directly execute the send receive interactions with no need to create the extra buffers and instead use the system buffers. We show the effectiveness of HDBIP on distributed two-phase commit protocol. We mainly compare with respect to BIP the execution time and the lines of code needed.

The remainder of this paper is structured as follows. Section 2 presents the existing BIP framework. Section 3 introduces HDBIP, an extension of BIP. Section 4 defines how it is possible to generate efficient implementations from HDBIP along with the arguments supporting correctness of the generated implementation. In Section 5, we evaluate the performance of HDBIP by comparing it to BIP. Section 6 presents related work. Finally, Section 7 draws some conclusions and presents future work.

2 Behavior Interaction Protocol (BIP) Framework

The Behavior Interaction Priority (BIP) framework [3] offers high-level synchronization primitives that simplify system development and allow for the generation of both centralized and distributed implementations from high-level models. It consists of three layers: Behavior, Interaction and Priority. *Behavior* is expressed by Labeled Transition Systems (LTS) describing atomic components extended with data and C functions. Moreover, transitions of atomic components are labeled with ports that are exported for communication/synchronization with other components. *Interaction* models the synchronization and communication between ports of atomic components. *Priority* specifies scheduling constraints on interactions.

2.1 Atomic Components

Let us consider a set of local variables X.

Definition 1 (Port). *A port is a tuple $\langle p, X_p \rangle$ where p is an identifier and $X_p \subseteq X$ is a set of exported local variables. A port is referred to by its identifier.*

Definition 2 (Atomic component - Syntax). *An atomic component is a tuple $\left\langle P, L, T, X, \{g_\tau\}_{\tau \in T}, \{f_\tau\}_{\tau \in T} \right\rangle$, such that:*

- *$\langle P, L, T \rangle$ is an LTS over a set of ports P, L is a set of control locations, and $T \subseteq L \times P \times L$ is a set of transitions;*
- *X is a finite set of variables;*
- *Every transition $\tau \in T$ has a guard g_τ (a predicate over X), and a function $f_\tau \in \{x := f^x(X) \mid x \in X\}^*$, triggered by this transition, that updates the values of variables in X.*

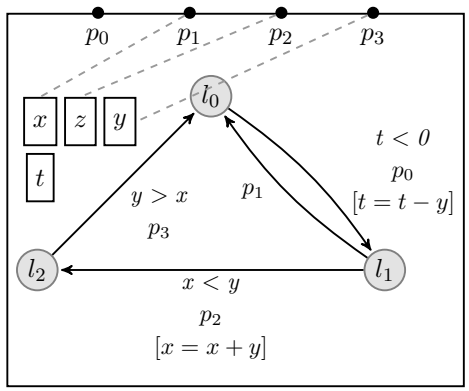


Fig. 1: Atomic component in BIP

A transition $\tau = \langle l, p, l' \rangle \in T$, where l (resp. l') is the source (resp. destination) of τ. p is the label of τ used as an interface to synchronize with other components. Moreover, a transition can be augmented with a guard g_τ and a function f_τ, thus defined as $\tau = \langle l, p, g_\tau, f_\tau, l' \rangle$. The port attached to a transition is said to be *enabled* only if the guard of the transition g_τ holds.

Definition 3 (Atomic component - semantics). *The semantics of atomic component* $\langle P, L, T, X, \{g_\tau\}_{\tau \in T}, \{f_\tau\}_{\tau \in T} \rangle$ *is the LTS* $\langle Q, P, T_0 \rangle$*, where:*

- $Q = L \times [X \to \text{Data}] \times (P \cup \{\texttt{null}\})$*;*
- $T_0 = \{\langle \langle l, v, p \rangle, p'(v_{p'}), \langle l', v', p' \rangle \rangle \in Q \times P \times Q \mid \exists \tau = \langle l, p', l' \rangle \in T : g_\tau(v) \wedge v' = f_\tau(v/v_{p'})\}$*, where* $v_{p'} \in [X_{p'} \to \text{Data}]$.

A configuration/state of an atomic component is a triple $\langle l, v, p \rangle \in Q$ where $l \in L$, $v \in [X \to \text{Data}]$ is a valuation of variables in X, and $p \in P$ is the port of the last-executed transition (or `null` otherwise, i.e., in case of the initial configuration). The evolution $\langle l, v, p \rangle \overset{p'(v_{p'})}{\to} \langle l', v', p' \rangle$, where $v_{p'}$ is a valuation of the variables in $X_{p'}$, is possible if there exists a transition $\langle l, p', g_\tau, f_\tau, l' \rangle$, s.t. p' is enabled or $g_\tau(v) = \texttt{true}$. Valuation v is modified to $v' = f_\tau(v/v_{p'})$.

We use the dot notation to denote the elements of an atomic component B. For instance, we refer to its set of ports as $B.P$, its set of locations as $B.L$ and its set of local variables as $B.X$.

Figure 1 depicts an atomic component B. B has four ports p_0, p_1, p_2 and p_3 and four local variables x, y, z and t. Port p_1 exports variable x, p_2 exports z, and p_3 exports y. In addition, B has three locations ℓ_0, ℓ_1 and ℓ_2 with initial location ℓ_0. Each transition between locations has a $guard$, a $port$ and an update function or the computation to be applied. For example, the transition between locations ℓ_1 and ℓ_2 is labeled by port p_2 and guarded by $x < y$ and applies computation $(x = x + y)$ when executed. When this transition is executed, the value of z exported by p_2 is changed according to the valuation received through p_2.

2.2 Composite Components

We consider a set of atomic components $\{B_i\}_{i\in I}$ with $I \subseteq [1,n]$ and $B_i = \langle P_i, L_i, T_i, X_i, \{g_\tau\}_{\tau\in T_i}, \{f_\tau\}_{\tau\in T_i}\rangle$, where atomic components have disjoint sets of locations, variables and ports, i.e., for all $i,j \in I$ such that $i \neq j$, $L_i \cap L_j = \emptyset$, $P_i \cap P_j = \emptyset$ and $X_i \cap X_j = \emptyset$. We denote the set of all ports (resp. locations, variables) of a composite component by $P = \bigcup_{i\in I} P_i$ (resp. $L = \bigcup_{i\in I} L_i$, $X = \bigcup_{i\in I} X_i$). Atomic components synchronize and exchange data through interactions.

Definition 4 (Interaction). *An interaction is defined as a tuple $a = \langle P_a, G_a, F_a\rangle$, where:*

- *P_a is a non-empty set such that $P_a \subseteq P$, and, for every $i \in I$ $|P_i \cap P_a| \leq 1$, i.e., an interaction a consists of at most one port of every atomic component in B;*
- *G_a is a guard over valuation of X_a, where X_a are the variables attached to ports P_a; and*
- *F_a is an update function over the valuation of X_a.*

We denote the ports associated in an interaction a as $P_a = \{p_i\}_{i\in I}$ where i is the identification index of the atomic component because at most one port of every atomic component can be included in the same interaction. Moreover, an interaction can include variables that are denoted as $X_a = \bigcup_{p\in P_a} X_p$. The updated value of X_{p_i}, transferred to B_i as an interaction outcome, after projecting the update function F_a is denoted as F_{a_i}.

Definition 5 (Composite component). *A composite component C consists in applying a set of interactions γ to a set of distinct atomic components $\{B_i\}_{i\in I}$ with $I \subseteq [1,n]$. Therefore, a composite component C is defined as $\gamma(\{B_i\}_{i\in I})$*

Figure 2 shows an example of a composite component $C = \gamma(\{B_1, B_2, B_3\})$ where B_1, B_2 and B_3 are atomic components, and $\gamma = \{a_1, a_2, a_3, a_4, a_5\}$.

Definition 6 (Semantics of composite components). *A state q of composite component $C = \gamma(\{B_1, \ldots, B_n\})$ is an n-tuple $\langle q_1, \ldots, q_n\rangle$ where $q_i = \langle l_i, v_i, p_i\rangle$ is a state of B_i. The semantics of C is an LTS $S_c = \langle Q, \gamma, \longrightarrow\rangle$, where:*

- *$Q = B_1.Q \times \ldots \times B_n.Q$;*
- *γ is the set of all possible interactions; and*
- *$\longrightarrow$ is the least set of transitions satisfying the following rule:*

$$\frac{\exists a \in \gamma : a = \langle \{p_i\}_{i\in I}, G_a, F_a\rangle \quad G_a(v(X_a)) \qquad \forall i \in I :\ q_i \xrightarrow{p_i(v_i)}_i q'_i \wedge v_i = F_{a_i}(v(X_a)) \qquad \forall i \notin I :\ q_i = q'_i}{\langle q_1, \ldots, q_n\rangle \xrightarrow{a} \langle q'_1, \ldots, q'_n\rangle}$$

X_a is the set of variables attached to the ports of a, v is the global valuation. F_{a_i} is the projection of F to the variables of p_i yielding to the valuation v_{p_i} of the variables in X_i exported by p_i.

The above rule means that whenever all the ports of an interaction a are enabled and the guard corresponding to a, $(G_a(v(X_a)))$ holds, a is enabled. One enabled interaction

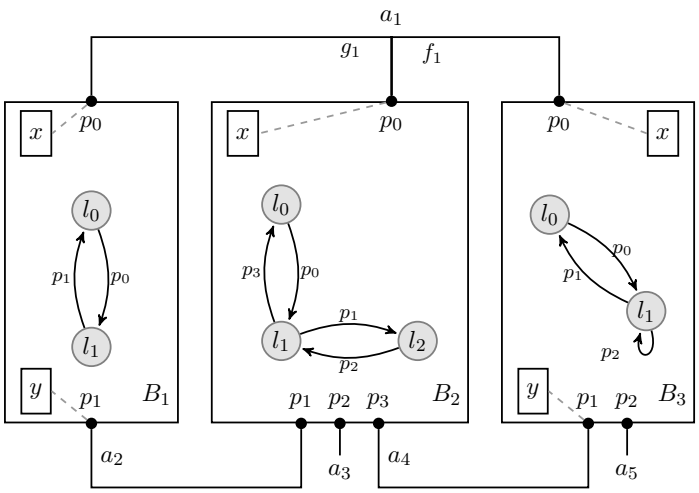


Fig. 2: Composite component in BIP

is selected and the state of the components whose ports are involved in the interaction a changes by executing location function and moving to the next set of locations. The state of the components that are not involved in this interaction remain unchanged. A straightforward implementation of this semantics can be realized by a centralized engine that allows the execution of one enabled interaction at a time. Note, practically, it is also possible to concurrently execute independent interactions (which do not share components), while preserving the above semantics.

Figure 2 represents a composite component $\mathcal{C}$ made up of three components $atomic = \{B_1, B_2, B_3\}$ by applying a set of five interactions $\gamma = \{a_1, a_2, a_3, a_4, a_5\}$. For instance, interaction a_1 is enabled when all of its involved ports, $i.e.$, $B_1.p_0$, $B_2.p_0$ and $B3.p_0$, are enabled and its corresponding guard g_1 holds. But, in this example, the ports are not associated with guards which means that by default all ports are enabled. Assuming that guard of a_1 holds, this interaction is said to be enabled. In case it is selected to execute, its function f_1 is also applied upon its execution. Furthermore, upon the execution of a_1, transitions $\langle B_1.l_0, B_1.p_0, B_1.l_1\rangle$, $\langle B_2.l_0, B_2.p_0, B_2.l_1\rangle$, $\langle B_3.l_0, B_3.p_0, B_3.l_1\rangle$ will, also, execute for their ports are involved in a_1.

2.3 Distributed Implementation - Send/Receive BIP

A high-level BIP model can be transformed into a distributed implementation to achieve parallelism between components and interactions [6]. To do so, a BIP model is transformed into its equivalent send/receive BIP. Send/receive BIP consists of three layers: (1) an atomic components layer that consists of atomic components transformed to interact with the upper layer to execute multiparty interactions; (2) an interaction layer that consists of components responsible to execute interactions; (3) a conflict resolution layer that is responsible to forbid the concurrent execution of two conflicting interactions (to preserve the semantics of the initial model). The

obtained model consists of transforming multiparty interactions into send/receive communication protocols. More precisely, each transition of atomic components is split in two transitions: (1) send offering, which sends the enabled ports to the components (that are handling the interactions corresponding to enabled ports) in the interaction layer; (2) receive, which waits for an acknowledgment from the interaction layer to execute the selected port. As such, the interaction protocol collects all enabled ports and determines what are the enabled interactions. As the interaction layer consists of several components handling different interactions, it is possible that two conflicting interactions are marked to be enabled by different components of the interaction layer, which may lead to the concurrent execution of two conflicting interactions. To remedy this, the interaction layer consults first with the conflict resolution, which is responsible for handling conflicts between interactions. Note, two interactions are said to be conflicting iff either: (1) there is a common port involved in them, or (2) if they include two distinct ports belonging to the same component where those ports are the label of two distinct transitions outgoing from the same source location.

Remark. Implementing a system with multiparty interactions requires solving potential conflicts, which is addressed in [6] for systems without priorities and in [7] for systems with priorities. Independently, we focus on those interactions that can be realized by asynchronous send/receive communication over multiparty interaction. For the sake of simplicity, and without lost of generality we consider systems without priorities.

3 Heterogeneous Distributed BIP - `HDBIP`

BIP uses multiparty interactions to model communication and synchronization between components, which is expressive enough to model any communication or synchronization primitives [5]. Nonetheless, modeling a simple asynchronous send/receive primitive requires to (1) explicitly create components representing buffers; (2) create intermediate schedulers to coordinate the execution of the interactions. This may drastically affect the performance of the generated distributed implementations. To overcome this, we introduce `HDBIP` that combines both multiparty and direct asynchronous send/receive (DASR) interactions. This simplifies the modeling of distributed systems and allows for efficient code generation. For instance, implementing DASR primitives can benefit from the underlying primitives such as system buffers and does not require to create extra components for scheduling with other interactions (i.e., conflict-resolution) or for buffer modeling. The components composing the `HDBIP` model are known as partially asynchronous (PA) atomic components.

3.1 `HDBIP` Syntax

A PA atomic component $B^\star$ is a regular BIP atomic component where transitions are labeled with three types of ports: ordinary, direct send and direct receive: (1) ordinary ports are the same to those defined in BIP; (2) direct send ports are used to model asynchronous direct communication with receive ports. Hereafter, we represent ordinary, direct send and direct receive ports, by black circle, blue rectangle and red diamond, respectively.

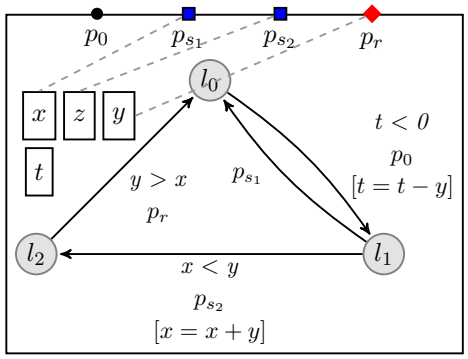


Fig. 3: Atomic component in HDBIP

Definition 7 (Partially Asynchronous Atomic component). *A PA atomic component $B^\star$ is tuple $\langle B, t\rangle$ where:*

- *B is an atomic component;*
- *$t : P \rightarrow \{ordinary, send, receive\}$ is a function that maps ports to their types.*

Figure 3 depicts a PA atomic component $B^\star$ in HDBIP. $B^\star$ has four ports p_0, p_{s_1}, p_{s_2}, and p_r and four local variables x, y, z, and t. Port p_{s_1} exports x, p_{s_2} exports z and p_r exports y. In addition, B has three locations ℓ_0, ℓ_1 and ℓ_2 with initial location ℓ_0. Hereafter, we consider a set of PA atomic components $\{B_i^\star\}_{i\in I}$, where $\forall i \in I$, $B_i^\star = \langle B_i, t_i\rangle$. Let $P_o = \bigcup_{i\in I} B_i^\star.P_o$ (resp. P_s, P_r, P) denotes the set of all the ordinary (resp. direct send, direct receive, all) ports. Moreover, without loss of generality, we assume that from any location, the outgoing transitions can be labeled with both ordinary and send or receive ports (i.e., either only ordinary ports or a mix of send or receive ports). This allows to efficiently generate distributed implementation and makes the interaction model not ambiguous to the developers of the atomic components. Note that the transitions requirements hold in the PA component depicted in Figure 3 .

We distinguish two types of interactions: (1) ordinary; and (2) DASR. Ordinary interaction is the same as regular BIP interaction, i.e., allows to model multiparty interaction. Hence, it connects ordinary ports. DASR interaction allows to model asynchronous send receive interaction and connects a sender port of a component to receiver ports of different components.

Definition 8 (Ordinary Interaction). *An ordinary interaction a is defined by the tuple $\langle P_a, G_a, F_a\rangle$ where:*

- *$P_a \subseteq P$ is a non-empty set such that $P_a \subseteq P_o$ and, $\forall i \in I$, $|B_i^\star.P \cap P_a| \leq 1$; and*
- *G_a and F_a are the guard and the function of the ordinary interaction, the same as the ones defined in the BIP interaction.*

Definition 9 (DASR Interaction). *A DASR interaction a is defined by P_a where:*

- *$P_a \subseteq P$, with $|P_a| > 1$, is a set such that $|P_a \cap P_s| = 1$, $|P_a \cap P_o| = 0$, $|P_a \cap P_r| > 0$ and, $\forall i \in I, |P_i \cap P_a| \leq 1$;*

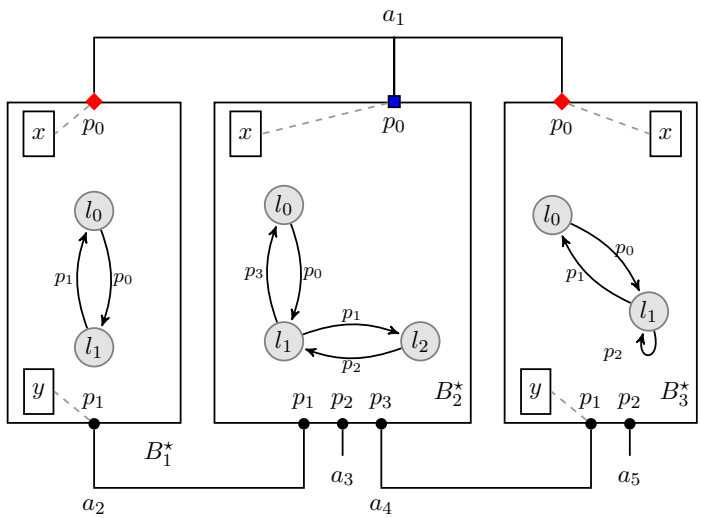


Fig. 4: Composite component in HDBIP

- *all ports have the same type; (3) its guard is always hold, however, its send port can have a local guard;*
- *its function allows only for data transfer of data attached to the sender port to the data attached to the receiver ports.*

Note that a send port can only participate in one DASR interaction, whereas a receive port can participate in several DASR interactions.

Definition 10 (Partially asynchronous composite component). *A PA composite component $C^\star$ denoted by $\gamma^\star(\{B_i^\star\}_{i\in I})$ consists of a set of atomic components $\{B_i^\star\}_{i\in I}$ composed by applying a set of ordinary and DASR interactions $\gamma^\star$*

Given a PA composite component $\gamma^\star(\{B_i^\star\}_{i\in I})$ where $B_i^\star = \langle B_i, t_i\rangle$ for all $i \in I$, we define:

- $type : \gamma^\star \to \{ordinary, sendreceive\}$ is a function that maps interactions to their types;
- $\gamma_o = \{a \in \gamma^\star \mid type(a) = ordinary\}$ the set of all ordinary interactions; and
- $\gamma_{sr} = \{a \in \gamma^\star \mid type(a) = sendreceive\}$ the set of all DASR interactions.

Clearly, $\gamma^\star = \gamma_o \cup \gamma_{sr}$ and $\gamma_o \cap \gamma_{sr} = \emptyset$. Figure 4 depicts a PA composite component made up of a set of three PA components $B^\star = \{B_1^\star, B_2^\star, B_3^\star\}$ by applying a set of five interactions $\gamma^\star = \{a_1, a_2, a_3, a_4, a_5\}$, where only a_1 is a DASR interaction while the rest (a_2, a_3, a_4 and a_5) are ordinary interactions. a_1 is a DASR interaction because it consists of a direct send port $B_2^\star.p_0$ and the receive ports $B_1^\star.p_0$ and $B_3^\star.p_0$. a_1 is said to be a valid DASR interaction because it does not include any ordinary port. Moreover, $B_2^\star.p_0$ cannot participate in further interactions.

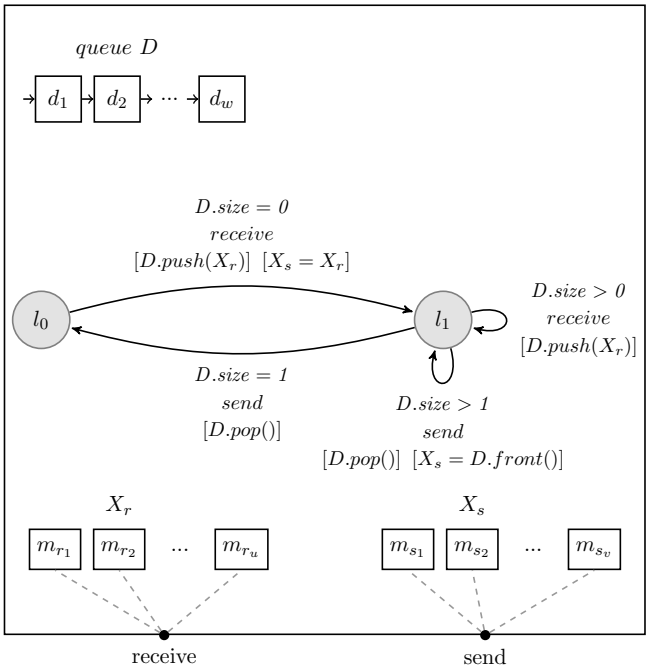


Fig. 5: Buffer component

3.2 HDBIP Semantics

We define the semantics of a PA composition component $C^\star$ by transforming it into its equivalent BIP model $C = [\![C^\star]\!]$. The transformation consists of the following steps: (1) create buffer atomic components; (2) create interactions connecting send and receive ports with buffer components.

Creating Buffer components. We first create a buffer component Bu_p^i for every receive port $p \in \bigcup_{i \in I} B_i^\star.Pr$. Bu^i is an atomic component where: (1) $Bu.X = X_s \cup X_r \cup D$ such that D is a queue that can hold data of the same type of the port, $X_s = \{x_s \mid x \in p.X\}$ is the set of received variables that correspond to port p, and $X_r = \{x_r \mid x \in p.X\}$ is the set of send variables that correspond to port p; (2) $Bu.P = \{send, receive\}$, where $send$ port exports the set of variables X_s, and $receive$ port exports the set X_r; (3) $Bu.L = \{l_0, l_1\}$, where l_0 is the initial location; (4) $Bu.T = \{\tau_1, \tau_2, \tau_3, \tau_4\}$ such that $\tau_1 = \langle l_0, receive, l_1\rangle$, $\tau_2 = \langle l_1, receive, l_1\rangle$, $\tau_3 = \langle l_1, send, l_1\rangle$ and $\tau_4 = \langle l_1, send, l_0\rangle$. The guards of transitions are predicates over the queue D and its size. Assuming the queue size can be denoted as $D.size$, guard g_1 of τ_1 is $D.size = 0$, g_2 of τ_2 is $D.size > 0$, g_3 of τ_3 is $D.size > 1$, and, finally, g_4 of τ_4 is $D.size = 1$. Yet, the size of the queue is not determined by the size of the message received, but by the number of messages received. The functions, on transitions including the port receive, from l_0 to l_1, involve adding the values of X_r (as one list)

to the list D and updating the values of X_s to that of X_r, whereas, from l_1 to l_1, the values of X_r are only added to D. Initially, in τ_1, X_s is updated to the values of the first received message. Thus, the functions, on transitions including the port send, from l_1 to l_1, involve removing data from the list D first, then updating the values of X_s to be the oldest list of values received and pushed to D. On the other hand, from l_1 to l_0, only the last list of values in D is removed emptying D. The set of all buffers for all receive ports in $C^\star$is denoted by $BU = \bigcup_{i \in I}\{Bu_p^i \mid p \in B_i^\star.P \wedge t_i(p) = receive\}$. Note that port $receive$ of the buffer is always enabled, i.e., its guard is `true`, whereas port $send$ is enabled when there are messages to be sent, i.e., the internal queue is not empty. Figure 5 shows an example of a buffer component that corresponds to port $p[X_r]$ (port p exporting a set of variables X_r).

Integration. We now are ready to define the semantics of a partially asynchronous composite component $C^\star$ as follows: (1) create a buffer component for each receive port; (2) append ordinary interactions; (3) for each DASR interaction we create one interaction connecting the send port of the DASR interaction to the receive ports of the buffers that correspond to the receive ports of the DASR interaction, and we create one binary interaction for each receive port, which is connected to send port of its corresponding buffer. Finally, all send and receive ports in `HDBIP` become ordinary.

Definition 11 (PA composite component semantics). *Given a partially asynchronous composite component $C^\star$=$\gamma^\star(\{B_i^\star\}_{i \in I})$, its semantics is defined by the transformation into a regular BIP system $C = [\![C^\star]\!]$, such that $C = \gamma(\{B_i\}_{i \in I} \cup BU)$ where:*

- *B_i is the atomic component that corresponds to $B_i^\star$ by removing labeling of the ports;*
- *BU is the set of buffers created for each receive port in $C^\star$;*
- *γ is the set of interactions applied to the set of atomic components $\{B_i\}_{i \in I} \cup BU$ such that $\gamma = \gamma_o \cup \gamma_s \cup \gamma_r$ where, γ_o is the set of all ordinary interactions in $C^\star$, $\gamma_s = \bigcup_{a \in \gamma_{sr}}\{(P_a, \texttt{true}, \texttt{identity}) \mid P_a = \{a.send\} \cup \bigcup_{r \in a.recvs}\{Bu_p^i.recv\}\}$ is the set of interactions between each direct send port in interaction a and the corresponding buffer receive port, and $\gamma_r = \bigcup_{p \in P_r}\{(P_a, \texttt{true}, \texttt{identity}) \mid p \in B_i^\star.P \wedge P_a = \{p, Bu_p^i.send\}\}$ is the set of interactions between each receive port and its corresponding send buffer port.*

Figure 6 shows how the `HDBIP` model presented in Figure 4 is transformed to its equivalent BIP model. All the PA atomic components $B_1^\star$,$B_2^\star$ and $B_3^\star$ are transformed to their equivalent BIP versions (ignoring ports types) B_1, B_2 and B_3 respectively. For every direct receive port ($B_1^\star.p_0$, $B_3^\star.p_0$) we added a corresponding buffer component in the BIP model. Then, the DASR interaction, in the `HDBIP` model, from the send port $B_2^\star.p_0$ to $B_1^\star.p_0$ and $B_3^\star.p_0$ is replaced by an interaction involving $B_2.p_0$ and the port receive of each of the buffer components corresponding to the receive ports of the `HDBIP` model. Additionally, DASR replacement includes adding other interactions involving the port send of every buffer component and its corresponding previous receive port. For this example, we included two interactions: (1) involving $B_1.p_0$ and $Buffer1.send$, (2) $B_3.p_0$ and $Buffer2.send$.

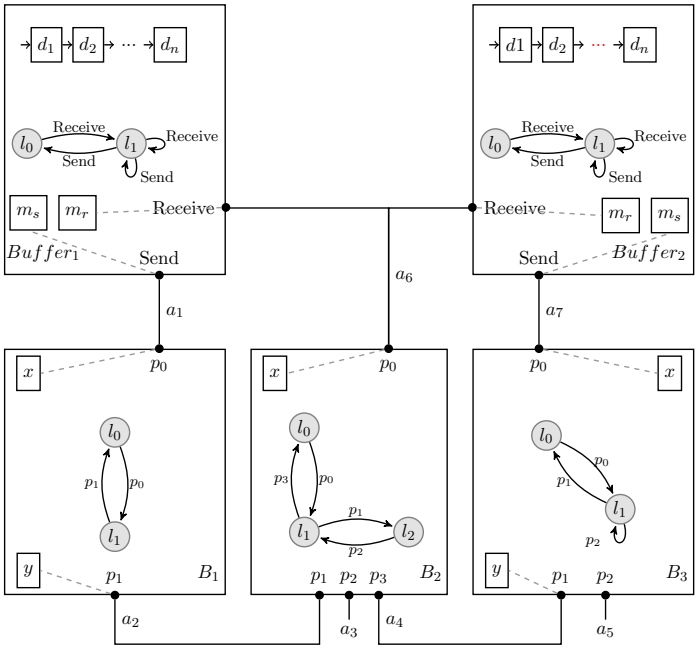


Fig. 6: Transformation of HDBIP composite in Figure 4 to BIP

4 Efficient Code Generation

Given an HDBIP system, it is possible to transform it to a regular BIP (i.e., consisting only of regular ports) and use the code generation provided by BIP (three-layer model). However, this may lead to the generation of inefficient implementations mainly because of: (1) the buffer components that correspond to receive ports will be replaced with actual threads or processes; (2) interactions between send/receive ports and the buffer components will be mixed with the multiparty interactions and will be added to the interaction protocol components; hence, their execution requires communication between base components, interaction protocols and possibly with conflict resolution components in case of conflicts. Although using HDBIP simplifies the development process by automatically generating buffer components and the corresponding communications, a naive implemenention would impose an additional overhead due to the extra communication as well as the creation of unnecessary threads/processes to represent the buffer components. Therefore, we introduce an efficient code generation that allows to avoid the creation of buffer components and the communication with the interaction and conflict resolution layers. To do so, we first transform PA atomic components of HDBIP system by splitting (following [6]) the transitions labeled with ordinary ports into two transitions to interact and receive notifications from the interaction protocol components, respectively. As for the transitions labeled with send and receive ports are not split and kept unchanged. Figure 7 presents an example of the transformation of a PA atomic component into its equivalent

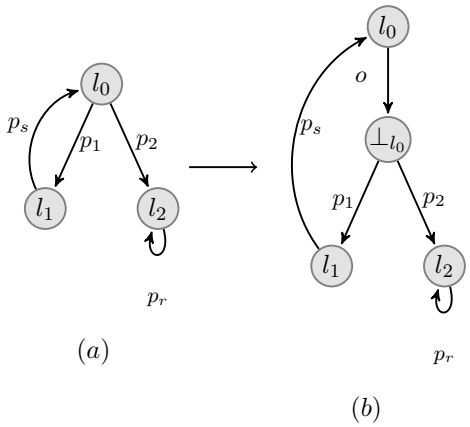


Fig. 7: HDBIP transitions transformation in 3-layer model

PA send/receive atomic component. Second, we generate the three layer send/receive model by only creating components in the interaction layer for ordinary interactions. DASR interactions are not integrated with the interaction layer and remain in the transformed model.

4.1 Correctness

The aim of the proof is to show that the efficient code generation is equivalent to the one provided by transforming HDBIP into regular BIP. The proof consists of two independent steps: (1) preservation of the buffer components; (2) no need for conflicts handling.

Preservation of the buffer components. Our code generation produces C++ implementation that uses MPI for the communication between threads/processes. As MPI has its internal system buffer, sending a message to a specific receive port (i.e., labeled with the name of the receive port) is implicitly added to the system buffer of MPI with the corresponding label. As such, there is no need to create buffer components.

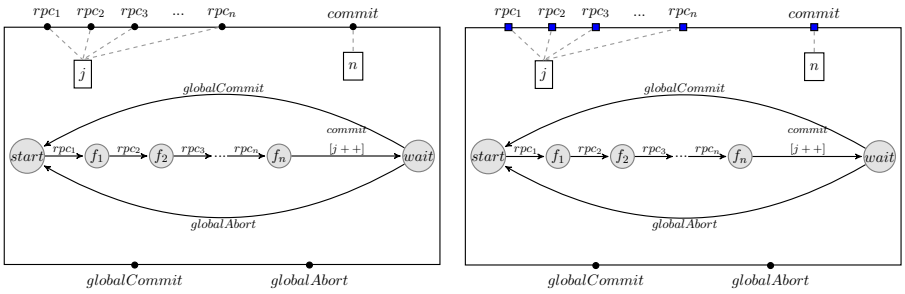
No need for conflict handling. In the equivalent BIP model obtained from HDBIP (Definition 11), conflicts may occur between: (1) only ordinary interactions; (2) ordinary and DASR interactions; (3) only direct/send interactions. Recall that our efficient code generation only requires to integrate ordinary interactions into interaction and conflict resolution layer, whereas DASR interactions are kept unchanged in the 3-layer send/receive model. As such, conflicts between only ordinary interactions are resolved by the interaction protocol and the conflict resolution protocol layers in the usual way (Section 2). As from any state, the outgoing transitions can be labeled with either ordinary ports or send/receive ports, it is not possible to get conflicts between ordinary and direct send/receive interactions. Regarding direct send/receive

interactions, a conflict may arise between two interactions that either involve: (1) a common direct receive port; (2) a common direct send port; (3) two ports of the same component that are the labels of two outgoing transitions from a same state. As for the (1) the execution of the receive port allows the buffer component to remain in state l_1 (see Figure 5). As such, even in the case of two concurrently-executing interactions connected to the same receive port, the final state will still belong to the state space of the semantics of the transformed regular BIP. As for (2) a direct send port can be connected to only one interaction. As for (3) we consider several cases either: (3a) the two ports are send ports, then the component will pick one of the two ports and execute the corresponding send; (3b) the two ports are receive ports, then the component can execute the corresponding receive port that has a message on its buffer, that is; (3c) one port is send and another is receive, in order to avoid deadlock of the execution, we consider giving priority to send port if its guard is enabled, otherwise, we can safely wait until a message on one of the receive ports is available. Consequently, in all the cases a conflict can be resolved locally.

5 Performance Evaluation

We evaluate the execution times and the number of lines of code in `HDBIP` versus BIP on distributed two-phase commit protocol [13]. Two-phase commit is a consensus protocol used to commit or abort a distributed transaction. A distributed transaction consists of a sequence of operations applied to several processes/participants. The system consists of n resource managers (participant of the transaction) rm_1, rm_2, $\ldots$, rm_n and a transaction manager tm. Executing a distributed transaction consists of the following steps: (1) the client sends a begin transaction message to tm; (2) client executes the operations of the transaction on its participants (resource managers); (3) client sends a commit transaction message to tm; (4) tm starts running two-phase commit protocol by sending a vote request message to all the resource managers; (5) each resource manager has the ability to commit or abort the transaction by sending local commit or local abort; (6) tm receives all the votes and broadcasts global commit to all resource managers if it has received a local commit from all the resource managers, otherwise it broadcasts global abort message; finally (7) depending on the receive message a resource manager either aborts or commits the transaction. For the sake of simplicity, we omit the handling of crash/recovery and timeouts that are handled by running specific termination protocols and by assuming the existence of persistent storage to keep track of the logs.

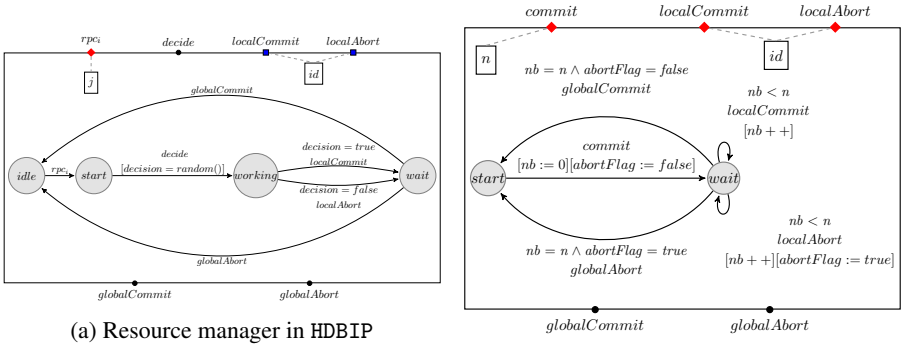
We provide two implementations of two-phase commit protocol using standard BIP and `HDBIP`. Figures 8a and 8b show the atomic components of the clients in standard BIP and `HDBIP`, respectively. It mainly initiates the transaction by calling remote procedure calls on the resource managers accompanied with the current transaction id j. It then notifies tm through the port $commit$ and waits for the reception of the global decision. In case of standard BIP all ports are ordinary. In `HDBIP` only $globalAbort$ and $globalCommmit$ ports are ordinary as they require a global agreement (multiparty interaction), and all the other remaining ports are send ports.



(a) Client component in BIP

(b) Client component in HDBIP

Fig. 8: Client component in BIP and HDBIP



(a) Resource manager in HDBIP

(b) Transaction manager in HDBIP

Fig. 9: Resource and transaction managers in HDBIP

The behavior of the resource manager rm and transaction manager tm in HDBIP are shown in Figures 9a and 9b (in regular BIP, we have the same behavior but all ports are ordinary). Each rm starts the transaction by executing the function. Then, a decision is made to abort or commit the transaction. Accordingly, it either synchronizes with tm with the port $localCommit$ or $localAbort$. tm collects all the responses and synchronizes with all the resource managers as well as the clients to globally commit or abort the transaction.

Figures 10 and 11 show the composite component of the whole system in regular BIP and HDBIP, respectively. Recall that in regular BIP all buffers should be explicitly modeled with components and all ports are ordinary ports. In HDBIP the design is much simpler as buffer components will be implicitly replaced by the system buffers during code generation.

Efficiency. We compare the execution times of the distributed implementations generated from BIP and the one generated from HDBIP. Note that in case of HDBIP the direct send receive interactions are treated in a special way and are not integrated with the regular code generation of multiparty interactions. We consider two different

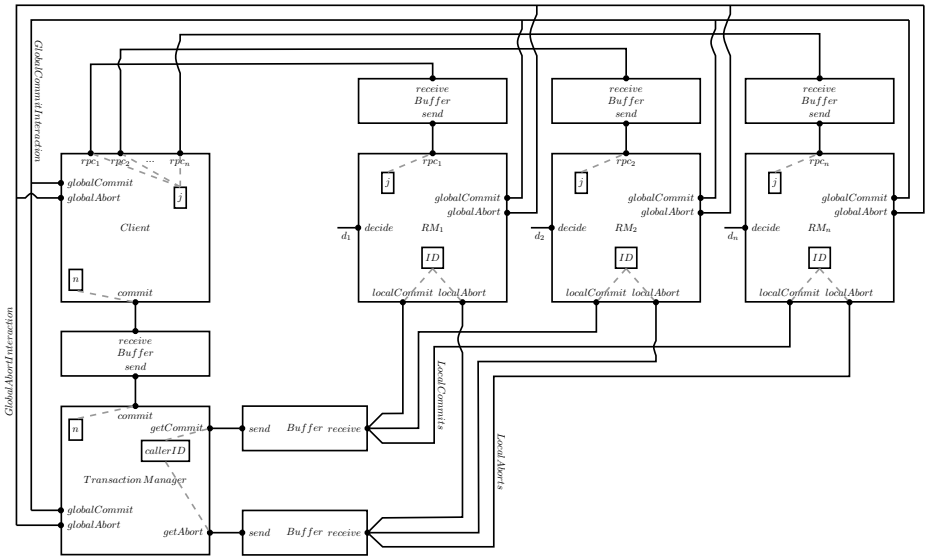


Fig. 10: Two-phase commit in BIP

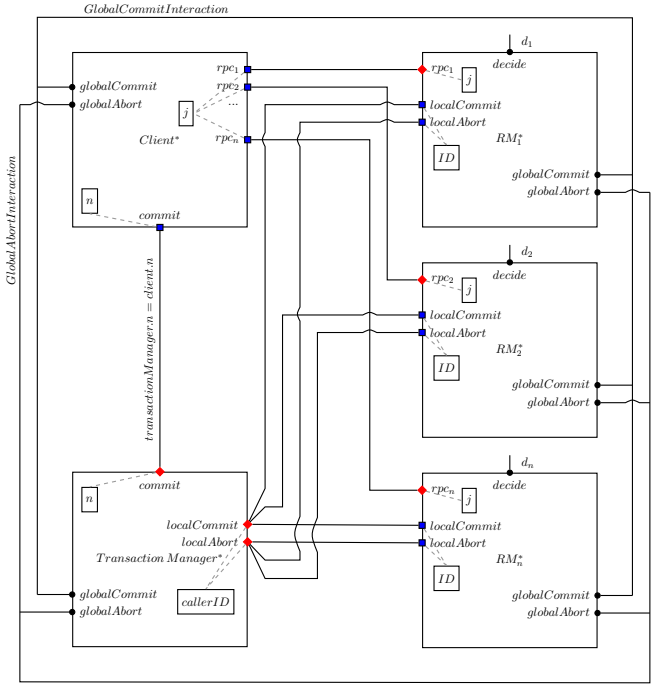
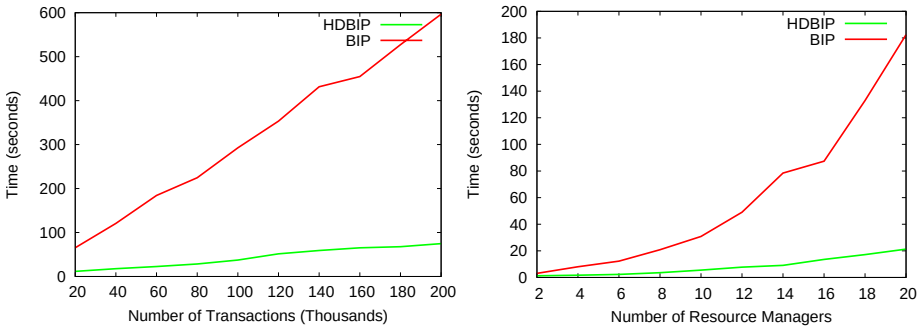


Fig. 11: Two-phase commit in HDBIP

scenarios by varying the number of resource managers and the number of transactions. For both scenarios, we consider a cluster of four Linux machines (64-bit Ubuntu 16.04), each with 8 cores, Intel Core i7-6700 processor, and 32 GB memory. In the first scenario, we vary the number of transactions from 20,000 to 200,000 by a step of 20,000 and we fix the number of resource managers to be 10. In the second scenario, we vary the number of resource managers from 2 to 20 by a step of 2, and we fix the number of transactions to be 10,000. Figures 12a and 12b show the execution times of theses scenarios for both implementations, respectively. In both scenarios, it is clear that the implementation of `HDBIP` drastically outperforms regular BIP. This is mainly due to the extra messages exchanged in case of the regular BIP with the buffer components, and the multiparty interactions between the buffer components. In case of `HDBIP`, we can still execute multiparty interactions, however, direct send receive can be directly executed with no need to create message buffer and benefit from the system buffers that are already available.



(a) Scenario 1 (b) Scenario 2

Fig. 12: Performance evaluation of two-phase commit

Lines of code (LOC). Using `HDBIP` requires less LOC than BIP as there is no need to create (1) the buffer component type and the corresponding instances; (2) the interactions between send/receive ports and the buffer components. For instance, modeling two-phase commit in case of 10 resource managers, requires 280 LOC in case of `HDBIP` and 390 LOC in case of BIP.

6 Related Work

In [11], a method is introduced to automatically generate correct asynchronously communicating processes starting from a global communication protocol. Unlike our model, the proposed method considers a simple communication model where each message has a unique sender and receiver. As such, modeling multiparty interactions requires to explicitly defining the communication protocol and conflict resolution handling, which is time consuming and error prone.

Session types [4,15,8,17,12] model interactions between distributed processes, and are based on the following methodology: (1) interactions are described as a *global protocol* between processes; (2) *Local protocols* are synthesized by projecting global protocol to local processes; (3) implementation of local processes; (4) type-checking of local types with respect to local processes. The design methodology of session type has major drawbacks: (1) there is a huge gap between design and implementation; (2) the design flow includes redundancy (global protocol, local protocol, process implementation), which is error prone; (3) there is no clear separation between communication and computation in local processes.

LASP [16] is a programming model designed to facilitate the development of reliable and large-scale distributed computing. It combines ideas from deterministic data-flow programming and conflict-free replicated data types (CRDTs). However, LASP is tailored to consistency over replicated data types. It would be interesting to integrate LASP with `HDBIP` to support fault-tolerance in `HDBIP`.

Other industrial frameworks simplify the development of large scale distributed systems such as AzureBot [1]. However, using such frameworks modeling communication models and synchronization are too abstract, which does not allow the expressiveness of explicit communication models. Moreover, AzureBot supports only applications written in C# and hosted in the Azure cloud platform.

Some recent research efforts tackle correctness-preserving code generation from models to asynchronously communicating systems. For example, in AlbertBBM16,HenrioR16, they introduce a formal translation from abstract behavioral specification (ABS) to object-oriented implementation, where [2] (resp. [14]) specifically targets parallel (resp. distributed) systems. However, the underlying communication model of ABS does not support multiparty interactions but only asynchronous calls.

7 Conclusion and Perspectives

We introduce a rigorous model to facilitate the development of correct, efficient and scalable distributed systems. In particular, `HDBIP` allows both multiparty and asynchronous send/receive primitives. Moreover, our method (1) uses the primitives provided by the underlying systems such as system buffers; and (2) makes a clear separation, which is correct-by-construction, between multiparty interactions and asynchronous send/receive interactions; which allow the generation of efficient distributed implementations

For future work, we first consider to develop a source-to-source transformation from session types to `HDBIP`. This would avoid code redundancy of the methodology provided by session types. Moreover, we consider using other primitives provided by the underlying library (e.g., MPI) such as barriers in order to support efficient implementation of multiparty interactions. We also work on extending `HDBIP` to support fault tolerance. We also consider to leverage the asynchronous send/receive communication primitive to improve the efficiency of the runtime verification [10] and enforcement [9] of component-based systems.

Acknowledgment. The authors acknowledge the support of the University Research Board (URB) at American University of Beirut and the ICT COST (European Cooperation in Science and Technology) Action IC1402 Runtime Verification beyond Monitoring (ARVI).

References

1. Agarwal, D., Prasad, S.K.: Azurebot: A framework for bag-of-tasks applications on the azure cloud platform. 2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (2013). https://doi.org/10.1109/ipdpsw.2013.261
2. Albert, E., Bezirgiannis, N., de Boer, F.S., Martin-Martin, E.: A formal, resource consumption-preserving translation of actors to haskell. In: Logic-Based Program Synthesis and Transformation - 26th International Symposium, LOPSTR 2016, Edinburgh, UK, September 6-8, 2016, Revised Selected Papers. pp. 21–37 (2016)
3. Basu, A., Bensalem, S., Bozga, M., Combaz, J., Jaber, M., Nguyen, T.H., Sifakis, J.: Rigorous component-based system design using the bip framework. IEEE Software **28**(3), 41–48 (2011)
4. Bejleri, A., Yoshida, N.: Synchronous multiparty session types. Electr. Notes Theor. Comput. Sci. **241**, 3–33 (2009). https://doi.org/10.1016/j.entcs.2009.06.002, `http://dx.doi.org/10.1016/j.entcs.2009.06.002`
5. Bliudze, S., Sifakis, J.: The algebra of connectors - structuring interaction in BIP. IEEE Trans. Computers **57**(10), 1315–1330 (2008). https://doi.org/10.1109/TC.2008.26, `https://doi.org/10.1109/TC.2008.26`
6. Bonakdarpour, B., Bozga, M., Jaber, M., Quilbeuf, J., Sifakis, J.: A framework for automated distributed implementation of component-based models. Distributed Computing **25**(5), 383–409 (2012). https://doi.org/10.1007/s00446-012-0168-6, `http://dx.doi.org/10.1007/s00446-012-0168-6`
7. Bonakdarpour, B., Bozga, M., Quilbeuf, J.: Model-based implementation of distributed systems with priorities. Design Autom. for Emb. Sys. **17**(2), 251–276 (2013). https://doi.org/10.1007/s10617-012-9091-0, `https://doi.org/10.1007/s10617-012-9091-0`
8. Bonelli, E., Compagnoni, A.B.: Multipoint session types for a distributed calculus. In: Trustworthy Global Computing, Third Symposium, TGC 2007, Sophia-Antipolis, France, November 5-6, 2007, Revised Selected Papers. pp. 240–256 (2007)
9. Falcone, Y., Jaber, M.: Fully automated runtime enforcement of component-based systems with formal and sound recovery. STTT **19**(3), 341–365 (2017). https://doi.org/10.1007/s10009-016-0413-6, `https://doi.org/10.1007/s10009-016-0413-6`
10. Falcone, Y., Jaber, M., Nguyen, T., Bozga, M., Bensalem, S.: Runtime verification of component-based systems in the BIP framework with formally-proved sound and complete instrumentation. Software and System Modeling **14**(1), 173–199 (2015). https://doi.org/10.1007/s10270-013-0323-y, `https://doi.org/10.1007/s10270-013-0323-y`
11. Farah, Z., Ait-Ameur, Y., Ouederni, M., Tari, K.: A correct-by-construction model for asynchronously communicating systems. Int. J. Softw. Tools Technol. Transf. **19**(4), 465–485 (Aug 2017)
12. Gay, S.J., Vasconcelos, V.T., Ravara, A., Gesbert, N., Caldeira, A.Z.: Modular session types for distributed object-oriented programming. In: Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid,

Spain, January 17-23, 2010. pp. 299–312 (2010). https://doi.org/10.1145/1706299.1706335, http://doi.acm.org/10.1145/1706299.1706335

13. Gray, J., Lamport, L.: Consensus on transaction commit. ACM Trans. Database Syst. **31**(1), 133–160 (Mar 2006). https://doi.org/10.1145/1132863.1132867, http://doi.acm.org/10.1145/1132863.1132867
14. Henrio, L., Rochas, J.: From modelling to systematic deployment of distributed active objects. In: Coordination Models and Languages - 18th IFIP WG 6.1 International Conference, COORDINATION 2016, Held as Part of the 11th International Federated Conference on Distributed Computing Techniques, DisCoTec 2016, Heraklion, Crete, Greece, June 6-9, 2016, Proceedings. pp. 208–226 (2016)
15. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. In: Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008. pp. 273–284 (2008). https://doi.org/10.1145/1328438.1328472, http://doi.acm.org/10.1145/1328438.1328472
16. Meiklejohn, C., Van Roy, P.: Lasp: A language for distributed, coordination-free programming. In: Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming. pp. 184–195. PPDP '15, ACM, New York, NY, USA (2015). https://doi.org/10.1145/2790449.2790525, http://doi.acm.org/10.1145/2790449.2790525
17. Vallecillo, A., Vasconcelos, V.T., Ravara, A.: Typing the behavior of software components using session types. Fundam. Inform. **73**(4), 583–598 (2006), http://iospress.metapress.com/content/82bf1qafeel5g8n4/