

AMERICAN UNIVERSITY OF BEIRUT

BIP-PLUS FOR DERIVING EFFICIENT DISTRIBUTED
IMPLEMENTATIONS OF COMPONENT-BASED
DESIGN

by

SALWA LOTFI KOBEISSI

A thesis

submitted in partial fulfillment of the requirements
for the degree of Master of Science
to the Department of Computer Science
of the Faculty of Arts and Sciences
at the American University of Beirut

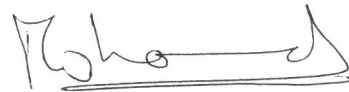
Beirut, Lebanon
August 17, 2016

AMERICAN UNIVERSITY OF BEIRUT

BIP-PLUS FOR DERIVING EFFICIENT DISTRIBUTED
IMPLEMENTATIONS OF COMPONENT-BASED DESIGN

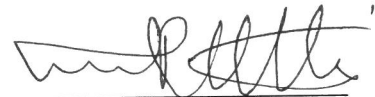
by
SALWA LOTFI KOBEISSI

Approved by:



Dr. Mohamad Jaber, Assistant Professor
Computer Science

Advisor



Dr. Paul Attie, Associate Professor
Computer Science

Member of Committee



Dr. Shady Elbassuoni, Assistant Professor
Computer Science

Member of Committee

Date of thesis defense: August 17, 2016

ACKNOWLEDGEMENTS

I owe my deepest gratitude to my advisor, Prof. Mohamad Jaber, for his great supervision, assistance and availability through out my journey.

Also, I am very thankful for Mhd Adnan Utayim for assisting me in implementing the case studies required for my thesis.

Additionally, I would like to extend my gratitude and gratefulness for those who have made my achievement possible. I am thankful for my brothers Sameh and Mohammad, who have been my supporters and motivators to pursue my goals in life, and for my brother Oussama, who has been my role model since day one and who has guided me to the right way reaching my aims. I am grateful for my father Lotfi, from whom I inherit my perseverance and determination to reach the summit of my goals, and for my mother Alia who has taught me to create light out of darkness, to dream when it was even impossible to and to be courageous enough to go and seek for my dreams through the bumpy roads I had. Finally, I am very thankful for my husband to be, Mohamed, who has been my number one supporter, since we met, as he has accompanied me, step by step, through my way to success and has made it simple for me to keep going with his love, care and faith in me.

Finally, I shall mention that the thesis has received funding from the University Research Board (URB) at AUB.

AN ABSTRACT OF THE THESIS OF

Salwa Lotfi Kobeissi for Master of Science
Major: Computer Science

Title: BIP-plus for Deriving Efficient Distributed Implementations of Component-based Design

Developing correct and reliable distributed systems is challenging mainly because of the complex structures of the interactions between distributed processes. Interactions can be modeled using either low-level primitives (e.g., MPI - Message Passing Interface) or high-level synchronization-primitives (e.g., BIP - Behavior Interaction Priority). Using the latter simplifies the development since it helps abstracting away the implementation details and validating the model w.r.t. a particular set of requirements. Nonetheless, abstraction reduces expressiveness of the interaction model. Consequently, generating efficient distributed implementations becomes very challenging due to the gap between the interaction model and the underlying platform or libraries. In this thesis, we propose BIP-plus which is an extension of the BIP framework to combine abstraction and expressiveness in a rigorous way. BIP is a component based framework with a rigorous operational semantics and high-level interaction model. We extend the interaction model by allowing both multi-party interactions and direct send-receive interactions that could be directly mapped to the underlying platform. Then, we define a correct (w.r.t. original model) and efficient code generation. We present two non-trivial case studies that show the effectiveness of our method: Two Phase Commit and distributed Support Vector Machines. The experimental results show that, in both problems, the distributed implementation, obtained by our proposed model, outperforms its equivalent implementation generated by the original BIP model.

Contents

ACKNOWLEDGEMENTS	v
ABSTRACT	vi
LIST OF FIGURES	ix
LIST OF ABBREVIATIONS	xi
1 INTRODUCTION	1
1.1 Problem Definition	1
1.2 Our Approach	3
1.3 Thesis Organization	4
2 BEHAVIOR INTERACTION PRIORITY (BIP)	5
2.1 Behavior Interaction Protocol (BIP) Framework	5
2.1.1 Atomic Components	5
2.1.2 Interactions and Composite Components	8
2.1.3 Priorities	10
2.1.4 Engine Protocol	11
2.1.5 BIP Toolchain	11
2.2 Transformation of BIP Models to Distributed Implementations	13
2.2.1 Interactions Conflicts	13
2.2.2 3-Layer Send/Receive BIP Model	14
3 THE BIP-PLUS FRAMEWORK	18
3.1 Overview	19
3.2 Syntax	19
3.2.1 Partially Asynchronous Atomic Components	19
3.2.2 Partially Asynchronous Composite Components	21
3.3 Semantics: Transformation from BIP-Plus to BIP	23
3.3.1 Buffer Components	24
3.3.2 Transforming Partially Asynchronous Composite Component to a Composite Component	27
3.4 Distributed Implementations from BIP-Plus	30
3.4.1 Distributed Implementation through the Intermediate Buffered-BIP Model	30
3.4.2 Direct Distributed Implementation from a BIP-Plus Model	31
3.5 Proof of Correctness of BIP-Plus Distributed Implementation	36
3.5.1 Preservation of Buffer Components Role	39
3.5.2 Preservation of IP Role for Buffers' Interactions Selection	40
3.5.2.1 Execution of Buffer's "receive" Interactions	41
3.5.2.2 Execution of Buffer's "send" Interactions	41
3.5.3 Conflicts Handling	42

3.5.3.1	Conflicts Between Ordinary Interactions	43
3.5.3.2	Conflicts Between Ordinary and DSR Interactions	43
3.5.3.3	Conflicts Between DSR Interactions	43
4	BIP-PLUS TOOL IMPLEMENTATION	49
4.1	BIP-Plus Tool Implementation From the BIP tool	49
4.2	Middle-End Modifications	49
4.3	Back-End Modification	50
5	BENCHMARKS	52
5.1	Two Phase Commit (2PC)	53
5.1.1	BIP Implementation of 2PC Protocol	55
5.1.2	BIP-Plus Implementation of 2PC Protocol	58
5.1.2.1	Experimental Results	62
5.2	Support Vector Machines (SVM)	69
5.2.1	BIP Implementation of (SVM)	70
5.2.2	BIP-Plus Implementation of (SVM)	74
5.2.2.1	Experimental Results	76
6	RELATED WORKS	80
6.1	Session Types	80
6.2	Grid Component Model (GCM)	81
6.2.1	Fractal	81
6.2.2	GCM	81
6.2.3	ProActive/GCM	82
6.3	Correct-by-construction model for asynchronously communicating systems	82
6.4	LASP	83
6.5	AzureBOT	83
6.6	FlowPools	84
7	CONCLUSION AND FUTURE WORKS	85
7.1	Conclusion	85
7.2	Future Works	86
	REFERENCES	87

List of Figures

2.1	Atomic Component Example	7
2.2	Composite Component Example	10
2.3	BIP Tool-Chain	12
2.4	Conflicting Interactions Example	14
2.5	Conflicting Interactions Example	14
2.6	BIP Transitions Transformation in 3-Layer Model Example	15
2.7	3-Layer S/R BIP Model	17
3.1	PA Atomic Component Example	20
3.2	PA Composite Component Example	23
3.3	The Buffer Component	27
3.4	Transformation from BIP ⁺ to BIP Model Example	29
3.5	3-Layer S/R Buffered-BIP Model Example	32
3.6	BIP ⁺ Transitions Transformation in 3-Layer Model	34
3.7	3-Layer S/R BIP ⁺ Model Example	35
3.8	BIP ⁺ Model Example	37
3.9	BIP with Buffers Model Equivalent to that of Figure 3.8	37
3.10	3-Layer S/R BIP ⁺ Model of 3.8	38
3.11	3-Layer S/R BIP Model of 3.11	38
3.12	BIP ⁺ Model with System Buffers Example	39
3.13	3-Layer S/R BIP ⁺ Model with System Buffers Example	40
3.14	Many Senders and one Receiver Conflict Example	45
3.15	Conflicting Interactions Involving Direct Send Ports	46
3.16	Conflicting Interactions Involving Direct Receive Ports	47
5.1	Benchmarks transformation.	54
5.2	Client.	56
5.3	Resource Manager _{<i>i</i>}	57
5.4	Transaction Manager.	57
5.5	Composite Component.	58
5.6	Client*.	59
5.7	Resource Manager _{<i>i</i>} *.	60
5.8	Transaction Manager*.	60
5.9	Composite Component*.	61
5.10	2PC Time Performance with Respect to Number of Transactions . . .	63
5.11	2PC Time Performance with Respect to Number of Resource Man- agers	64
5.12	2PC Time Performance with Respect to Number of Machines	66
5.13	2PC Time Performance with Respect to Number of Machines with Selective Distribution	67
5.14	2PC Time Performance with Respect to Number of Machines Com- parison.	68
5.15	Preprocessor.	71

5.16	Weak Learner.	71
5.17	Strong Learner.	72
5.18	Composite Component.	73
5.19	Preprocessor*.	74
5.20	Weak Learner*	75
5.21	Strong Learner*	75
5.22	Composite Component*.	76
5.23	SVM Time Performance with Respect to Number of Preprocessors and Weak Learners.	78
5.24	SVM Time Performance with Respect to Number of Preprocessors and Weak Learners.	79

LIST OF ABBREVIATIONS

MPI	Message Passing Interface
BIP	Behavior Interaction Priority
2PC	Two Phase Commit
SVM	Support Vector Machines
TCP	Transmission Control Protocol
S/R	Send/Receive
CL	Components Layer
IPL	Interaction Protocol Layer
RPL	Reservation Protocol Layer
PA	Partially Asynchronous
DS	Direct Send
DR	Direct Receive
DSR	Direct Send-Receive

Chapter 1

INTRODUCTION

Contents

1.1 Problem Definition	1
1.2 Our Approach	3
1.3 Thesis Organization	4

1.1 Problem Definition

The need of parallel and distributed implementations is growing extremely fast, especially with High-performance computing (HPC), Big Data Analytics and Internet of Things. However, Developing correct distributed systems is notoriously a difficult task. This is mainly due to their complex structures that consist of complex interactions between distributed processes.

Although, different frameworks exist to model interactions between distributed processes, building correct, reliable and scalable distributed systems is still a time-consuming, error-prone and hardly predictive task.

Depending on the used framework, interactions can be modeled using either low-level (e.g. MPI - *Message Passing Interface*) or high-level synchronization primitives (e.g., BIP - *Behavior Interaction Priority* [1, 2, 3]).

Modeling interactions using low-level primitives allows obtaining an expressive and efficient implementations. Nevertheless, the development process imposes much effort, burden and complication on programmers, and reliability verification of these models is considered as a challenging task.

On the other hand, modeling interactions using high-level primitives drastically simplifies the development process, as developers can abstract away implementation details and validate the model with respect to a set of intended requirements. However, once the abstract model is validated, deriving correct and efficient implementation from it is always challenging, since adding implementation details involves many subtleties that can potentially introduce errors into the resulting system [4]. Moreover, abstraction reduces expressiveness of the interaction model.

Developing distributed systems and applications from high-level models is much more desirable than getting involved into the development details of low-level code [5]. This drives us, in this thesis, to choose and focus on a high-level model from which distributed applications can be produced. Also, we will work towards exploring this model's limitations and trying to tackle them.

The high-level model that is utilized, discussed and enhanced is BIP.

BIP (Behavior, Interaction, Priority) framework [1, 2, 3] is a component-based framework with formal operational semantics. Coordination between components is achieved by using multiparty interactions and dynamic priorities for scheduling interactions.

BIP allows the generation of correct-by-construction distributed implementations from a high-level model [6, 5, 4]. The code generation is split into two phases: (1) transform high-level model into asynchronous Send/Receive model; (2) generate distributed implementation from asynchronous Send/Receive model.

Additionally, scalable distributed implementations are, also, supported by BIP. This is due to the fact that BIP is a component-based framework consisting in decomposing problems into smaller blocks/components. On the other hand, decomposing large and complex problems into smaller blocks is the one of the most rational techniques to tackle these problems.

However, the proposed approach, BIP, has the following drawbacks: First, the code generation does not take into account the execution platform. Second, the high-level model provides only multiparty interactions. For instance, if process A wants to asynchronously send a data or signal to process B , this requires to create a buffer component to store the received message or signal, and hence the code generation: (1) does not distinguish between this buffer component and a normal component process, which introduces an overhead to the generated code; and (2) does not use the execution platform or the underlying operating system that may provide efficient communication primitives.

Therefore, a BIP model is a high-level model from which simple, correct-by-construction and scalable distributed implementations and applications can be derived. Nevertheless, this communication model is not expressive enough and does not use the underlying platform making it not efficient as desired in some implementations, *e.g.*, the case of asynchronous data transfer between processes.

For this reason, we are driven to tackle the limitations of the BIP framework. We aim to attain a communication primitive that makes it possible for programmers to simply construct scalable, reliable and more efficient distributed system while maintaining the model’s expressiveness which preserves the programmers’ ability to control more the aspects of their implementation.

1.2 Our Approach

We target an extension of the BIP framework (BIP-plus) that combines both low-level and high-level communication primitives. We aim at extending the BIP communication primitives by combining the high-level model and a direct send-receive model in an elegant way. This allows to use send and receive primitives at the high-level model.

A send allows a process to send data or signals to another one and resumes execution, *i.e.*, it is an asynchronous send. On the other hand, receive allows a process to wait until there is a message that can be received by this particular process, *i.e.*, it is asynchronous yet a blocking receive.

Our proposed direct send-receive communications (data transfer) are expressed, in a BIP-plus model, through interactions. However, they are not multiparty interactions. They do not impose synchronization between processes, they make use of the underlying platform (*e.g.*, system buffer), and they exactly express and are generated to equivalent low-level asynchronous send and receive primitives when the final code generation of the distributed implementation is obtained.

In order to obtain a rigorous communication model and to allow efficient code generation, we separate the original high-level primitives from the new ones. That is, at any execution point, a process may either call send and receive primitives or high-level primitives, but not a combination of those at the same execution point of one process. Moreover, a send can connect to one or more receives through the same interaction allowing sending data from a specific process to one or more processes (multicast). But, multiple sends are not allowed to participate in the same interaction. On the other hand, a receive can be connected to multiple sends through multiple interactions.

Combining high-level and direct send-receive primitives confers numerous advantages. First, the code generation will be guided to take into account the underlying platform. Second, the code generation produces a correct implementation, which

is semantically equivalent to the original model. Third, the communication model becomes more expressive while maintaining the possibility of using abstract and high-level primitives.

1.3 Thesis Organization

The thesis consists of 7 chapters. Chapter 1 presents the problem, introduces the context, and gives an overview about our proposed approach. Chapter 2 introduces some preliminaries related to the BIP framework and the 3-layer architecture for distributed implementations of BIP models. Chapter 3 presents the BIP-plus framework, the transformation from the BIP-plus model to its equivalent BIP model, the derivation of distributed implementations from a BIP-plus model and the correctness proof of our proposed approach for obtaining distributed implementations. Chapter 4 shows how the BIP-plus tool, responsible for generating distributed code from a BIP⁺ model, is implemented. Chapter 5 presents experimental results on two non-trivial case studies. Chapter 6 presents the related works. Finally, Chapter 7 draws some conclusions and presents future works.

Chapter 2

BEHAVIOR INTERACTION PRIORITY (BIP)

Contents

2.1	Behavior Interaction Protocol (BIP) Framework	5
2.1.1	Atomic Components	5
2.1.2	Interactions and Composite Components	8
2.1.3	Priorities	10
2.1.4	Engine Protocol	11
2.1.5	BIP Toolchain	11
2.2	Transformation of BIP Models to Distributed Implementations	13
2.2.1	Interactions Conflicts	13
2.2.2	3-Layer Send/Receive BIP Model	14

2.1 Behavior Interaction Protocol (BIP) Framework

Behavior Interaction Priority (BIP) framework [1, 2, 3] offers high-level synchronization primitives that simplify the process of system development and allow the generation of distributed implementation from high-level models.

This framework consists of three layers: Behavior, Interaction and Priority. *Behavior* is expressed by Labeled Transition Systems (LTS) describing a set of atomic components, identified by communication ports and extended with data and C functions. *Interaction* is described by interaction/communication between atomic components building-up composite components. *Priority* is specified by scheduling strategies on interactions between atomic components.

2.1.1 Atomic Components

An atomic component B is an LTS associated with a set of local variables X . These variables range over a particular domain Data. Interactions, including data transfer/exchange and synchronization, take place between these components through *ports*.

Definition 1 (Port). A port p in B is defined by an identifier p and X_p which is a set of local variables exported by this port such that $X_p \subseteq X$. Port p stands for the tuple $\langle p, X_p \rangle$.

Definition 2 (Atomic component). An atomic component B is defined by a tuple $\langle P, L, T, X, \{g_\tau\}_{\tau \in T}, \{f_\tau\}_{\tau \in T} \rangle$, such that:

- $\langle P, L, T \rangle$ is an LTS over a set of ports P in B : 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,
- For every transition $\tau \in T$, there are g_τ which is the guard of τ , a boolean condition or a predicate over X , and a function $f_\tau \in \{x := f^x(X) \mid x \in X\}^*$ which is a computation, triggered by this particular transition τ , that updates or reassigns variables in X .

A transition $\tau = \langle l, p, l' \rangle \in T$ is defined by $l \in L$ being the source of this transition, by $l' \in L$ being the destination of this transition and p which is a port responsible for the interaction with another component. Also, the transition can be associated with a guard g_τ , and a function f_τ making $\tau = \langle l, p, g_\tau, f_\tau, l' \rangle$. Such transitions are only executed if g_τ is true.

The set of all transitions $T = \{\tau_j\}_{j \in J}$ with $J = [1, m]$.

Moreover, there is a set of variables utilized in a transition defined as $\text{var}(f_\tau) = \{x \in X \mid x := f^x(X) \in f_\tau\}$.

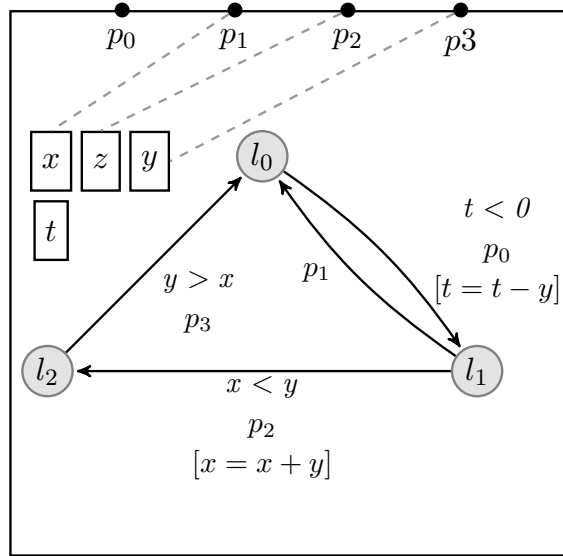
Definition 3 (Semantics of atomic components). The semantics of atomic component $B = \langle P, L, T, X, \{g_\tau\}_{\tau \in T}, \{f_\tau\}_{\tau \in T} \rangle$ is defined by a LTS $S_b = \langle Q, P, T_0 \rangle$, where:

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

A configuration is a triple $\langle l, v, p \rangle \in Q$ where $l \in L$, $v \in [X \rightarrow \text{Data}]$ is a valuation of variables in X , and $p \in P$ is the port of the last-executed transition (or null otherwise). The evolution $\langle l, v, p \rangle \xrightarrow{p'(v_{p'})} \langle l', v', p' \rangle$, 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 $g_\tau(v) = \mathbf{true}$. Valuation v is modified to $v' = f_\tau(v/v_{p'})$.

In other words, for every transition executed from a particular source to a destination location, the atomic components have some of their variables values changed. This means that at a certain location l , there is a valuation v of some local variables in X . This valuation is previously got by executing a transition to location l as its destination through a port p . Thus, at each location, the atomic components have updated values for their list of variables. A transition from l , as a source location, to another location l' , as destination, through port p' can only execute if its guard over the variables valuation obtained at l holds. upon execution of this transition in case the guard is true, a new valuation v' is obtained after receiving new data for the variables exported by port p' (valuation v'_p of X'_p) by the corresponding interaction executed (defined in definition 4) and applying the computation of this executed transition through p' on some variables in X . The resultant valuation of variable in X on l' is $v' = f_\tau(v/v_{p'})$.



B_1

Figure 2.1: Atomic Component Example

Example 1 (Atomic Component). Figure 2.1 represents an atomic component B . B has four ports $\{p_0, p_1, p_2, p_3\}$ and four local variables $\{x, y, z, t\}$. Port p_1 exports variable x , p_2 exports z , and p_3 exports y . In addition, B has a set of three locations l_0, l_1 and l_2 with the 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 whose source location is ℓ_1 , destination location is ℓ_2 and labeled by port p_2 , its guard is $x < y$ and its function to be applied is $(x = x + y)$ when the transition is executed. When $x < y$ is true, i.e., the guard holds for this particular transition, p_2 is said to be enabled, thus this transition can be executed. Supposing this transition is chosen to perform, z the variable exported by p_2 , will be equal to its valuation received through p_2 . Then, moving from the source ℓ_1 to ℓ_2 , the function $x = x + y$ will be executed.

2.1.2 Interactions and Composite Components

Given a set of distinct atomic components $\{B_i\}_{i \in I}$ with $I \subseteq [1, n]$, such that for $i \in I$, $B_i = \langle P_i, L_i, T_i, X_i, \{g_\tau\}_{\tau \in T_i}, \{f_\tau\}_{\tau \in T_i} \rangle$, we have, for all $i \in I$ and $j \in I$ 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 of a composite component as $P = \bigcup_{i \in I} P_i$, the set of all locations as $L = \bigcup_{i \in I} L_i$ and the set of variables as $X = \bigcup_{i \in I} X_i$. The atomic components B_1, \dots, B_n collaborate through interactions as they are able to synchronize with each other or exchange data.

Given an atomic component B_i for some $i \in I$, we can, also, use dot notation or the index i as a superscript to point to the elements of this component. For instance, we can refer to its set of ports P_i as $B_i.P$ or P^i , its set of locations L_i as $B_i.L$ or L^i and the set of local variables X_i as $B_i.X$ or X^i .

Definition 4 (Interaction). Given the set of atomic components $\{B_i\}_{i \in I}$ with $I \subseteq [1, n]$, an interaction between them is defined by $a = \langle P_a, G_a, F_a \rangle$, where:

- P_a is a non-empty set such that $P_a \subseteq P$, and, for every $i \in I$ $|P_i \cap P_a| \leq 1$, meaning that 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 such that the interaction can take place if this guard is true.
- 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 is built up of a set of distinct atomic components $\{B_i\}_{i \in I}$ with $I \subseteq [1, n]$ and by applying a set of interactions γ to this set of atomic components. Therefore, a composite component C is defined as $\gamma(\{B_i\}_{i \in I})$

For $I = [1, n]$, $C = \gamma(\{B_1, \dots, B_n\})$.

Figure 2.2 shows an example of a composite component $B = \{B_1, B_2, B_3\}$ where B_1 , B_2 and B_3 are atomic components.

Definition 6 (Semantics of composite components). A state q of composite component $C = \gamma(\{B_1, \dots, B_n\})$ is an n -tuple $\langle q_1, \dots, q_n \rangle$ 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:

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

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

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 .

Whenever all the ports included in an interaction a are enabled (*i.e.*, their guards hold), and the guard corresponding to a , ($G_a(v(X_a))$) holds, a can be launched. Accordingly, the state of the components whose ports are involved in the interaction a changes. This change in state of these components is due to the new valuation of variables exported by their ports involved in a in addition to applying the functions of transitions labelled by these ports. On the other hand, the state of components which are not involved in this interaction remain unchanged. Although many interactions may be ready to start at the same time, only one interaction is possible at a time selected by a centralized engine.

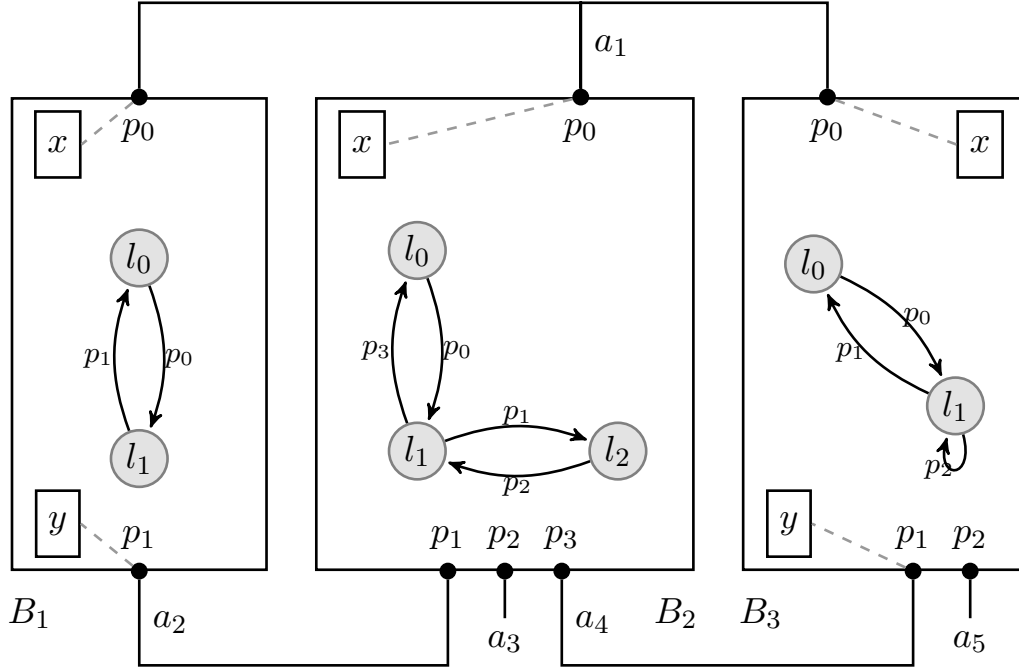


Figure 2.2: Composite Component Example

Example 2 (Composite Component). Figure 2.2 represents a composite component C made up of three components $atomic = \{atomic_1, atomic_2, atomic_3\}$ 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., p_0 , p_1 and p_2 , 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 .

2.1.3 Priorities

There is a possibility to have multiple interactions enabled at the same time. In this case, we may prefer to have some certain interactions deterministically selected rather than other enabled interactions to execute. For this sake, a priority model was proposed to order the enabled interactions given priority values. Eventually, the model selects the enabled interaction having the greatest priority to execute.

However, we are not going to explore further the notion of priority for it is not

taken into account in our study. We assume that all enabled interactions have equal probability to execute (without a specified priority), and that selection decision is made non-deterministically.

2.1.4 Engine Protocol

There is a centralized engine whose role is to sequentially execute the interactions and transitions in the BIP model achieving a correct execution of the composite component according to the global operational semantics [6].

The engine and the atomic components of the model interact together as follows:

1. In every component, guards corresponding to transitions outgoing from a location reached at last are evaluated.
2. components send their enabled ports, those whose guards are true, to the engine.
3. The engine computes the set of all interaction whose execution is possible *i.e.*, their guards hold and their involved ports are enabled.
4. The engine selects non-deterministically one interactions among the enabled ones and executes it along with its update function which might allow data exchange between the involved components.
5. The Engine executes in a sequential fashion all the transitions whose ports participate in the executed interaction along with the transition's function.

2.1.5 BIP Toolchain

The BIP tool-chain is made up of many tools that are responsible for modeling, executing and validating the BIP models [1, 7]. Figure 2.3 displays the relationships between those tools. Mainly, the tool-chain consists of the following.

Front-end. The front-end includes *editor* in addition to a *compiler*, whose role is to describe a system in BIP language. Given the BIP description source, the BIP model is generated by the compiler. The BIP model complies with BIP meta-model built on top of the Eclipse Modeling Framework [8].

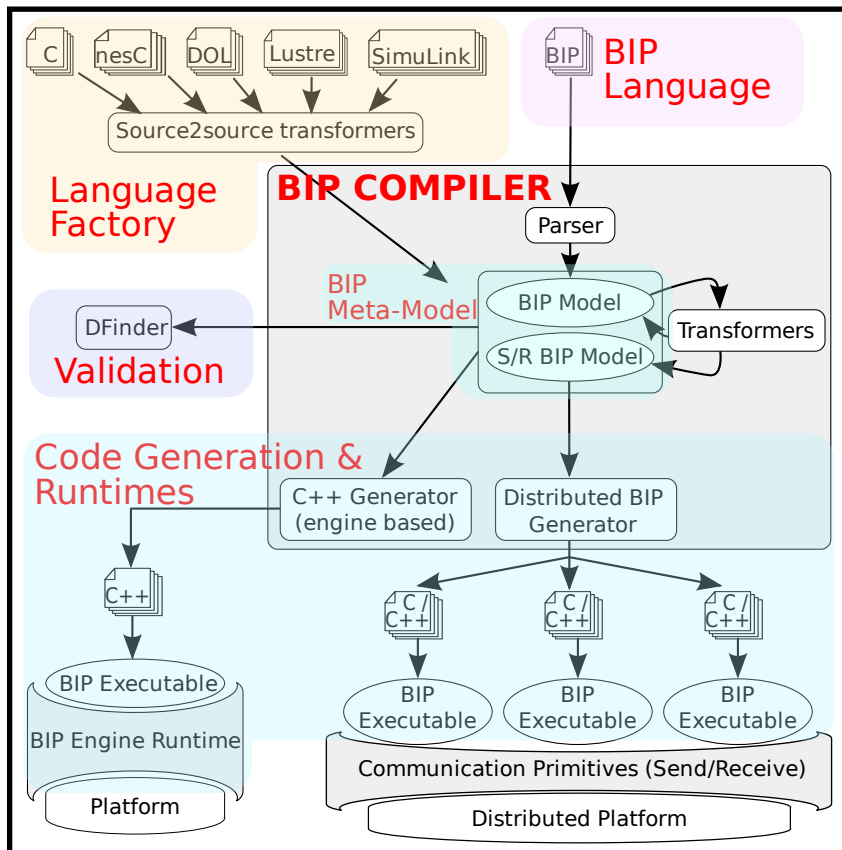


Figure 2.3: BIP Tool-Chain

Middle-end. The middle-end allows source-to-source transformations transforming either a BIP model to another BIP model for optimization goals, or a language different than BIP (e.g., C, Simulink... etc) to an equivalent BIP model.

Back-end. The back-end consists of many code generators that generate both sequential implementations (e.g., serial C++ code) and distributed implementations (e.g., multi-threaded C++ implementation, C++ code employing MPI or Sockets implementation) given a particular BIP model.

Validation. The validation module attain compositional verification by making use of DFinder [9]. This makes possible, for example, to check, for a particular BIP model, its deadlock-freedom and invariants. Additionally, statistical models can be derived, by utilizing the validation module, in order to examine BIP models.

2.2 Transformation of BIP Models to Distributed Implementations

The high-level BIP model can be transformed into a distributed implementation to achieve parallelism. In order to obtain an efficient distributed setting of a given model or composite component, interactions between components must be executed concurrently [6]. However, parallelism between components and interactions must respect the global state semantics of the model [5, 4].

2.2.1 Interactions Conflicts

Simultaneous execution of interactions can violate the global semantics of the initial model because of conflicts that take place between interactions [4].

Definition 7 (Conflict). Consider the composite component $C = \gamma(\{B_i\}_{i \in I})$ for $I \subseteq [1, n]$. The two interactions, in C , $a_1 = \langle P_{a_1}, G_{a_1}, F_{a_1} \rangle$ and $a_2 = \langle P_{a_2}, G_{a_2}, F_{a_2} \rangle$, with $a_1, a_2 \in \gamma$, are conflicting iff:

- $P_{a_1} \cap P_{a_2} \neq \emptyset$, or
- there exists some component B_i for some $i \in I$ and two transitions, having the same source location, $\tau_1 = \langle l, p_1, l' \rangle$ and $\tau_2 = \langle l, p_2, l'' \rangle$ such that $p_1 \in P_{a_1}$ and $p_2 \in P_{a_2}$ and $l, l', l'' \in L$, for $p_1, p_2 \in B_i.P$.

In other words, interactions are said to be conflicting iff there is a common port involved in all of them, or if they all include ports belonging to the same component such that these ports label transitions outgoing from the same source location.

Example 3 (Conflicting Interactions Example). Figures 2.4 and 2.5 represent composite components that include conflicting interactions. In Figure 2.4 the composite component is made up of a set of three atomic components $\{B_1, B_2, B_3\}$ and two interactions $\{a_1, a_2\}$ such that there is a conflict between a_1 and a_2 . They are conflicting because the port p in B_2 participates in both of a_1 and a_2 .

Similarly, the model in Figure 2.5 consists of three atomic components $\{B_1, B_2, B_3\}$ one which a set of two interactions $\{a_1, a_2\}$ is applied. a_1 and a_2 are conflicting because $B_2.p_1 \in a_1$ and $B_2.p_2 \in a_2$ are both ports of B_2 labeling transitions $\tau_1 = \langle \ell_0, p_1, \ell_1 \rangle$ and $\tau_2 = \langle \ell_0, p_2, \ell_2 \rangle$. In these two examples, if a_1 and a_2 are to be

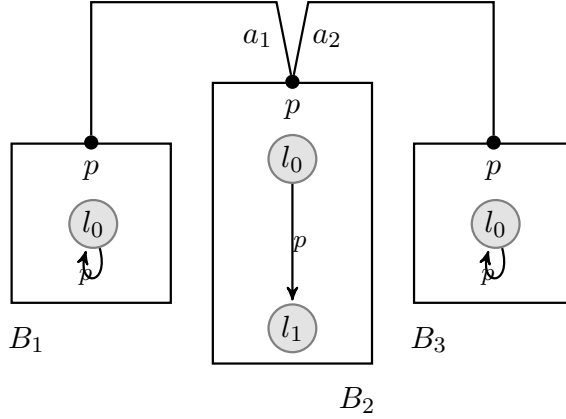


Figure 2.4: Conflicting Interactions Example

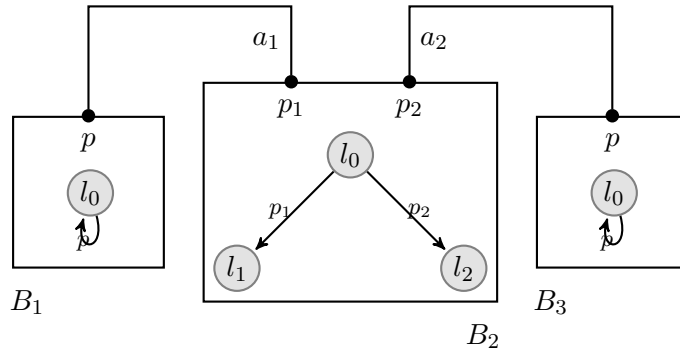


Figure 2.5: Conflicting Interactions Example

executed concurrently, the global state semantics of these models will be violated. Therefore, in both examples, we can either execute a_1 only, or a_2 although we are targeting a distributed implementation.

2.2.2 3-Layer Send/Receive BIP Model

In a BIP model, conflicts between interactions are handled in a centralized engine. However, in distributed implementations, resolving such conflicts is complicated which requires preserving atomic components, maintaining interaction behavior and handling these conflicts with a distributed approach. These requirements are fulfilled by three layers building up the transformed BIP model for distributed implementation [6, 5, 4]. Moreover, the components of the three layers interact with each other using asynchronous send and receive communication primitives. Thus, the obtained model after transformation is called a Send/Receive (SR) 3-layer BIP model. This 3-layer BIP model is an intermediate model from which, finally, distributed code is to be generated. The generated distributed code is a C++ code that uses TCP sockets of MPI [10].

The 3-layer architecture consists of the following layers:

1. **Components Layer (CL).** This layer includes the atomic components of the initial BIP model after adding, to each of them, more ports and modifying their transitions. Given the BIP composite component $C = \gamma \{B_i\}_{i \in I}$, for every $i \in I$, the atomic component, corresponding to B_i , to be placed in the first layer in distributed implementation, is B_i^{SR} , known as a Send/Receive atomic component.

Every component B_i^{SR} includes an offer-port o_i which is responsible for sending the list of enabled ports, in the component, to the upper layer. As for the original set of ports P_i , every $p \in P_i$, waits to be triggered and receive a response from the upper layer to execute the transition labeled by p .

As for transitions, for every location $l \in L_i$ in B_i , a new transition, $\langle l, o_i, \perp_l \rangle$, is introduced, to B_i^{SR} , called offer transition. This transition has l as the source and location \perp_l , corresponding to l , as the destination. It is executed by the offer port of B_i^{SR} . Moreover, every transition $\tau = \langle l, p, l' \rangle$, in B_i , with $l, l' \in L_i$ and $p \in P_i$, is replaced by $\langle \perp_l, p, l' \rangle$, in B_i^{SR} . Therefore, given a transition $\langle l, p, l' \rangle$ in B_i , it is equivalent to $\langle l, o, \perp_l \rangle$ followed by $\langle \perp_l, p, l' \rangle$ in B_i^{SR} .

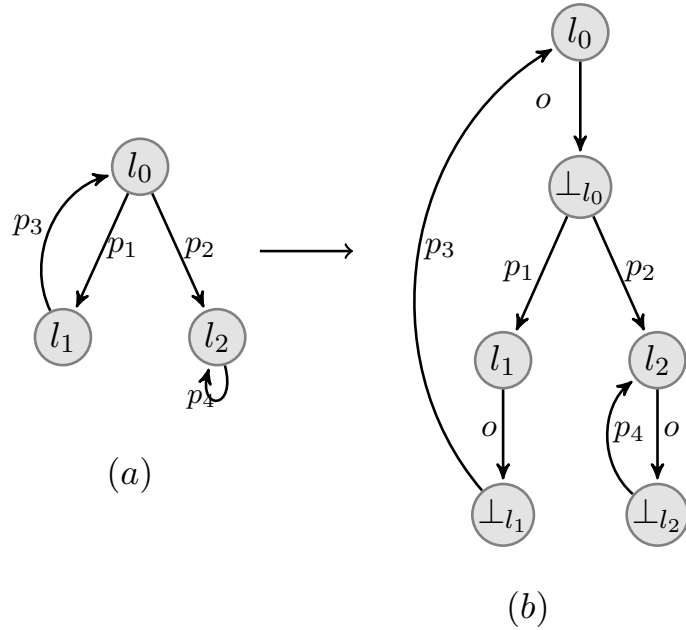


Figure 2.6: BIP Transitions Transformation in 3-Layer Model Example

Example 4 (BIP Transitions Transformation in 3-Layer Mode). Figure 2.6 presents an example, in (a), of an LTS in a BIP model, and shows, in (b), how it becomes after transforming the atomic component to an S/R component.

These modifications make it possible to utilize asynchronous communication primitives to transfer offers and responses between the CL and the upper layer.

2. **Interaction Protocol Layer (IPL).** This layer is built up by a set of atomic components to which interactions are assigned according to a particular partition. These components are responsible for interactions execution. The conflicts between the interactions assigned to one atomic component, are handled by this atomic component locally. However, conflicts between interactions that cannot be executed by the same component are to be resolved by an upper layer by sending a request to it for the conflicting interaction belonging to it. The IPL is in charge of guard evaluation and function execution of the interaction whose execution is permitted. This layer communicates with the Atomic components by receiving offers and sending responses corresponding to the selected interaction.
3. **Reservation Protocol Layer (RPL).** This layer allows to handle external conflicts between interactions, i.e., conflicting interactions handled by different components in the IPL. It can be implemented either using a (1) centralized implementation, (2) token-ring based implementation, (3) dining-philosopher based implementation, or (3) any distributed implementation of the committee coordination problem. This layer receives requests from the IPL about the externally conflicting interactions. Then, it responds back either: (1) by an "ok" message to the corresponding layer below for the interaction whose execution is allowed; or (2) by a "fail" message to the other components in which the interactions whose execution is not allowed at the moment.

The components of IPL and RPL communicate using asynchronous send-receive communication primitives.

Example 5 (3-Layer S/R BIP Model). Figure 2.7 shows the transformation of the composite component previously presented in figure 2.2 to its 3-layer S/R model. the CL consists of the S/R version of components of the initial BIP model, B_1^{SR} ,

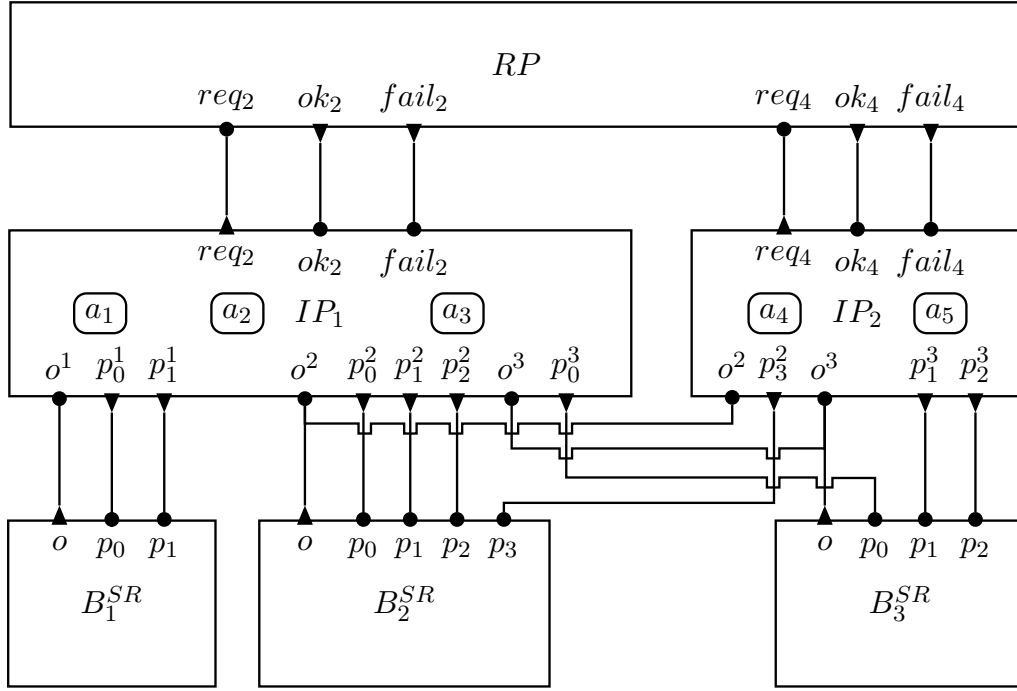


Figure 2.7: 3-Layer S/R BIP Model

B_2^{SR} and B_3^{SR} . The IPL consists of two components IP_1 and IP_2 such that interactions a_1 , a_2 and a_3 are assigned to IP_1 and a_4 and a_5 are assigned to IP_2 . Every one of these IP components must solve the conflicts that occur between the interactions that are local to it. That is, IP_2 is only responsible for handling conflicts that may occur between a_4 and a_5 while IP_1 is only responsible for handling conflicts between a_1 , a_2 and a_3 . In case there is a conflict between interactions belonging to different IP components, the RPL resolves it. For instance, in this example, there is a conflict between a_2 and a_4 (recheck figure 2.2), then both of the IP components to which they belong to must send a request to the upper layer, i.e., IP_1 sends request req_2 corresponding to a_2 , and IP_2 sends req_4 corresponding to a_4 . In turn, RPL picks one interaction to execute, by utilizing its algorithm, and responds back by ok_2 to IP_1 and $fail_4$ to IP_2 if the execution of a_2 is allowed, otherwise it responds with ok_4 to IP_2 and $fail_2$ to IP_1 .

Finally, this intermediate model of distributed implementation can be either transformed to a C/C++ code employing either MPI or TCP sockets for communication.

Chapter 3

THE BIP-PLUS FRAMEWORK

Contents

3.1	Overview	19
3.2	Syntax	19
3.2.1	Partially Asynchronous Atomic Components	19
3.2.2	Partially Asynchronous Composite Components	21
3.3	Semantics: Transformation from BIP-Plus to BIP	23
3.3.1	Buffer Components	24
3.3.2	Transforming Partially Asynchronous Composite Component to a Composite Component	27
3.4	Distributed Implementations from BIP-Plus	30
3.4.1	Distributed Implementation through the Intermediate Buffered-BIP Model	30
3.4.2	Direct Distributed Implementation from a BIP-Plus Model . .	31
3.5	Proof of Correctness of BIP-Plus Distributed Implementation	36
3.5.1	Preservation of Buffer Components Role	39
3.5.2	Preservation of IP Role for Buffers' Interactions Selection . . .	40
3.5.2.1	Execution of Buffer's "receive" Interactions	41
3.5.2.2	Execution of Buffer's "send" Interactions	41
3.5.3	Conflicts Handling	42
3.5.3.1	Conflicts Between Ordinary Interactions	43
3.5.3.2	Conflicts Between Ordinary and DSR Interactions . .	43
3.5.3.3	Conflicts Between DSR Interactions	43

3.1 Overview

BIP-plus (BIP⁺) is an extension of the BIP framework that makes use of, in addition to high communication primitives, send-receive low-primitives for modeling interactions.

BIP-plus can be defined as a component-based framework held by a mix of high level multi-party interactions for component synchronization purposes and direct send-receive interactions responsible for asynchronous data transfer between components making use of send receive primitives. The components composing the BIP⁺ model are known as partially asynchronous (PA) atomic components.

3.2 Syntax

3.2.1 *Partially Asynchronous Atomic Components*

A PA atomic component B^* is an LTS associated with a set of local variables X . These variables range over a particular domain Data. Interactions, including asynchronous data transfer/exchange and synchronization, take place between these components through *ports*. Ports are of three types: ordinary, direct send and direct receive.

Definition 8 (Partially Asynchronous Atomic component). A PA atomic component B^* is defined by the tuple $\langle B, t \rangle$ where:

- B is an atomic component,
- $t : P \rightarrow \{\text{ordinary}, \text{send}, \text{receive}\}$ is a function that maps ports to their types which helps distinguishing between three types of ports: ordinary, direct send and direct receive ports,

If, for some port $p \in P$, $t(p) = \text{ordinary}$, then p is the same as a port in BIP, and it is called ordinary port. If $t(p) = \text{send}$, then it is a direct send port, and if $t(p) = \text{receive}$, then it is a direct receive port.

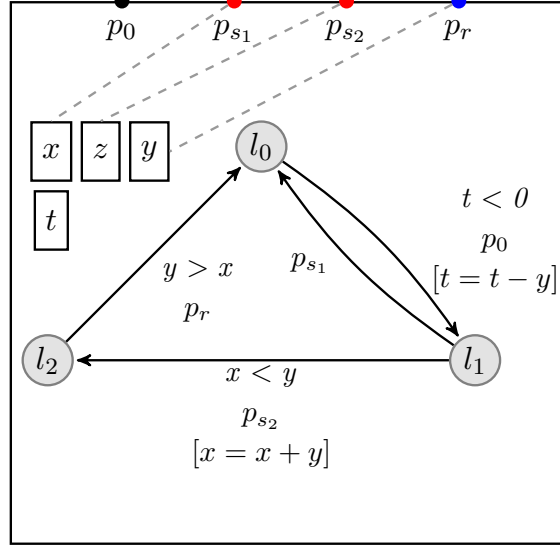
The set of ordinary ports is defined by $P_o = \{p \mid p \in P \wedge t(p) = \text{ordinary}\}$, that of send ports is defined by $P_s = \{p \mid p \in P \wedge t(p) = \text{send}\}$, and that of receive ports

is defined by $P_r = \{p \mid p \in P \wedge t(p) = \text{receive}\}$ such that $P_o \cup P_s \cup P_r = P$ and $P_o \cap P_s \cap P_r = \emptyset$.

The PA atomic component B^* must satisfy the following constraints concerning transitions:

Consider the set of transitions $T = \{\tau_j\}_{j \in J}$ with $J \subseteq [1, m]$ and $\tau_j = \langle l_j, p_j, l'_j \rangle$ for all $j \in J$.

For all $j \in J$ and $i \in J$ such that $i \neq j$, if $l_i = l_j$, the associated ports to these transitions, can be only either ordinary, direct send, direct receive, or a mix of direct send and receive ports.



B^*

Figure 3.1: PA Atomic Component Example

Example 6 (PA Atomic Component). Figure 3.1 depicts a PA atomic component B^* . B^* has four ports $\{p_0, p_{s_1}, p_{s_2}, p_r\}$ and four local variables $\{x, y, z, t\}$. Port p_1 exports 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 . To differentiate between the three types of ports, ordinary, direct send and direct receive, ports are represented by black, blue and red bullets, respectively. Thus, in this example, we have p_0 as an ordinary port, p_{s_1} and p_{s_2} as direct send ports and, finally, p_r as a direct receive port.

In this example, the transitions requirements are satisfied. Starting from a specific location, all the outgoing transitions have either only ordinary labelling ports or non-ordinary labelling ports (direct send and direct receive) e.g., from l_1 , the tran-

sitions to ℓ_0 and ℓ_2 are both labelled by the send ports p_{s_1} and p_{s_2} . Furthermore, the restrictions on transitions would be respected if we had any of the ports of these two transitions as a direct receive port.

3.2.2 Partially Asynchronous Composite Components

Consider the set of distinct PA atomic components $\{B_i^*\}_{i \in I}$ with $I \subseteq [1, n]$ and $\forall i \in I, B_i^* = \langle B_i, t_i \rangle$. For this set of PA atomic components, we have:

- $P_o = \bigcup_{i \in I} B_i^*.P_o$ is the set of all ordinary ports,
- $P_s = \bigcup_{i \in I} B_i^*.P_s$ is the set of all send ports,
- $P_r = \bigcup_{i \in I} B_i^*.P_r$ is the set of all receive ports,
- P the set of all ports, such that $P = P_o \cup P_s \cup P_r$.

Also, in order to refer to elements, of some PA atomic component, B_i^* for some $i \in I$, we can use the dot notation.

Definition 9 (Ordinary Interaction). Given a set of PA atomic components $\{B_i^*\}_{i \in I}$, 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, $\forall i \in I, |B_i^*.P \cap P_a| \leq 1$.
- 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.

The set of ports of an interaction a is denoted as $P_a = \{p_i\}_{i \in I} \subseteq P$ such that $I \subseteq [1, n]$ where i is the identification index of the PA atomic component because one port at most of each PA atomic component can be included in a .

Definition 10 (Direct Send-Receive (DSR) Interaction). Given a set of PA atomic components $\{B_i^*\}_{i \in I}$, a DSR interaction a is defined by $\langle P_a \rangle$ 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, $\forall i \in I, |P_i \cap P_a| \leq 1$. In other words, in a DSR interaction, the set of ports must include at most one port of every PA atomic component. Also, one included port must be a direct send port and the rest are direct receive ports.

- All $p \in P_a$, has the same type of the exported set of variables; that is the variables attached to one port, participating in the interaction a , must have the same types of the variables attached to every other port, in a , and in the same order
- The guard is excluded from its definition because the interaction must be always enabled when the direct send port is enabled. The DSR interactions is sender triggered.
- The function is excluded because, through this interaction, the values of variables exported by the send port are always copied to the values of variables exported by the receive ports participating in the same interaction; i.e., the function of DSR interaction is to send the values of variables from send port to receive port allowing data transfer between components.

The DSR interactions in said to be sender triggered. As in the ordinary interaction, the set of ports of a DSR interaction is $P_a = \{p_i\}_{i \in I} \subseteq P$ where i is the identification index of the PA atomic component of the included port.

An interaction can either be an ordinary interaction or send-receive interaction.

A direct send port can only participate in one DSR interactions i.e., $\forall a_1 = \langle P_{a_1} \rangle, a_2 = \langle P_{a_2} \rangle \in \gamma^*$, such that $a_1 \neq a_2, P_{a_1} \cap P_s \neq P_{a_2} \cap P_s$. On the contrary, a direct receive port can involve in multiple interactions.

Definition 11 (Partially Asynchronous Composite Component). A composite component, C^* , is built up of a set of distinct PA atomic components $\{B_i^*\}_{i \in I}$ with $I \subseteq [1, n]$ after applying a set of ordinary and send receive interactions γ^* to this set of components. We denote the composite component C^* as $\gamma^*(\{B_i^*\}_{i \in I})$

Given a PA composite component $C^* = \gamma^*(\{B_i^*\}_{i \in I})$ with $I \subseteq [1, n]$ and $B_i^* = \langle B_i, t_i \rangle$ for all $i \in I$, we define:

- $type : \gamma^* \rightarrow \{\text{ordinary}, \text{sendreceive}\}$ is a function that maps interactions to their types. if $a \in \gamma^*$ is an ordinary interaction, then $type(a) = \text{ordinary}$, and if $a \in \gamma^*$ is a DSR interaction, then $type(a) = \text{sendreceive}$.
- $\gamma_o = \{a \mid a \in \gamma^* \wedge type(a) = \text{ordinary}\}$ the set of all ordinary interactions.
- $\gamma_{sr} = \{a \mid a \in \gamma^* \wedge type(a) = \text{sendreceive}\}$ the set of all DSR interactions.

We have $\gamma^* = \gamma_o \cup \gamma_{sr}$ and $\gamma_o \cap \gamma_{sr} = \emptyset$.

The composite component C^* consisting of B_1^*, \dots, B_n^* and the set of interactions γ^* is noted $\gamma^*(\{B_1^*, \dots, B_n^*\})$.

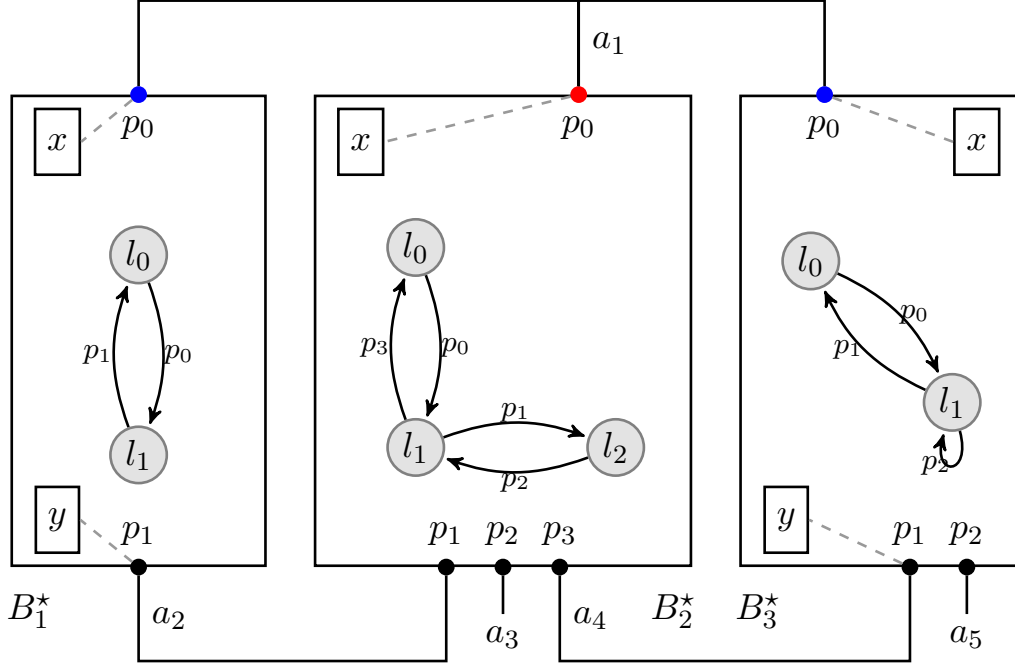


Figure 3.2: PA Composite Component Example

Example 7 (PA Composite Component). Figure 3.2 represents a PA composite component C^* made up of a set of three PA components $B^* = \{B_1^*, B_2^*, B_3^*\}$ by applying a set of five interactions $\gamma^* = \{a_1, a_2, a_3, a_4, a_5\}$. We have a_1 only as a DSR interaction while the rest (a_2, a_3, a_4 and a_5) are all ordinary interactions. a_1 is a DSR interaction because it consists of a direct send port $B_2^*.p_0$ and the receive ports $B_1^*.p_0$ and $B_2^*.p_0$. a_1 is said to be a valid DSR interaction because it does not include any ordinary port. Moreover, $B_2^*.p_0$ cannot participate in further interactions.

In order to define the semantics of a composite component in BIP^+ , we present its equivalent BIP model through a transformation process illustrated in the following section.

3.3 Semantics: Transformation from BIP-Plus to BIP

We define the operational semantics of BIP^+ by transforming it to ordinary model. However, building the latter with such a behavior needs adding implementation

details including many subtleties that can potentially introduce errors into the resulting system.

In the following section, we define the operational semantics of BIP⁺ by transforming it to BIP ordinary model.

3.3.1 Buffer Components

For every receive port $p \in \bigcup_{i \in I} B_i^*.Pr$ with $I \subseteq [1, n]$, we introduce a corresponding buffer Bu_p^i where i is the identification index of the PA atomic component to which the receive port p belongs and p is the port identifier. A buffer Bu is an atomic component where:

1. $Bu.X = X_s \cup X_r \cup D$ where,
 - D is variable of type queue
 - $X_s = \{x_s | x \in p.X\}$ that are the set of received variables that correspond to the variable of the port p ;
 - $X_r = \{x_r | x \in p.X\}$ that are the set of send variables that correspond to the variables of the port p
2. $Bu.P = \{send, receive\}$. Port send exports the set of variables X_s , and port receive 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 $\tau_4 = \langle l_1, send, l_0 \rangle$. These transitions are extended with guards and functions as follows:
 - 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 $queue.size > 0$, g_3 of τ_3 is $queue.size > 1$, and, finally, g_4 of τ_4 is $queue.size = 1$. 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 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^* is denoted as $BU = \bigcup_{i \in I} \{Bu_p^i \mid p \in B_i^*.P \wedge t_i(p) = receive\}$. We can, also, refer to the buffer of a receive port as $buffer(B_i^*.p)$ and use dot notation to refer to its contents.

Given a buffer component included in a composite component, where:

- its port *receive* is involved in an interaction a , having its guard as true by default, with a port p of another component such that interaction allows data transfer from p to *receive*.
- its port *send* is involved in an interaction a' , having its guard as true by default, with a port p' of another component such that this interaction allows data transfer from port *send* to p'

The buffer component will behave as follows:

1. Initially, the buffer component is at l_0 and D is empty satisfying g_1 of τ_1 labelled by "receive. Then, port *receive* is enabled, where "send" is not.
2. If p is enabled, then a can execute since *receive* is also enabled. However, if p' is enabled, a' cannot execute for *send* is not to prevent data sending from the buffer component to the other interacting one in case there are no messages previously received.
3. in case a is selected to execute, the transition τ_1 in the buffer is executed and it moved to location l_1 .

4. At this stage at l_1 , there is only one message saved in D and the data exported by port *send* is updated to have the values of the received message. Since D is not empty then port *receive* is enabled for the guard of transition (loop), from l_1 to l_1 , through this port holds. In addition, since the size of D is 1, *i.e.*, only one message, in total, is received by the buffer, port *send* is also enabled in transition from l_1 to l_0 . Thus, in case p and p' are enabled, any of the two interactions, a and a' , can execute, along with the transitions whose guards enable the buffer's ports, τ_2 and τ_4 . However, in case p is enabled and p' is not, then only a can execute, and in case p' is enabled and p is not, only a' can execute.
5. In case τ_4 , from l_1 to l_0 , is executed then the buffer reaches the initial state again, and all the previous steps may be repeated.
6. In case τ_2 , the loop from l_1 to l_1 , is executed, there will be more than a one message in D . Thus, both *send* and *receive* will be enabled. *send* is enabled because the guard of τ_3 holds. Eventually, in case p and p' are enabled, any of the two interactions, a and a' , can execute, along with the transitions whose guards enable the buffer's ports, *i.e.*, τ_2 and τ_3 .
7. Then, at l_1 , the buffer is willing to keep receiving messages through *receive* labeling τ_2 and adding them to D , and send messages through *send* labeling τ_3 , popping the sent messages put from D , until only one message is left. If one message is left, the steps starting step 4 will be repeated.

One important notice is that the port "receive" of the buffer is always enabled at any time. On the other hand, port "send" can only be enabled whenever there are messages to be sent, *i.e.*, the internal queue is not empty.

Figure 3.3 shows the buffer component, that corresponds to a port $p[X]$ (Port p exporting a set of variables X), and its constituents as presented previously in the beginning of this section.

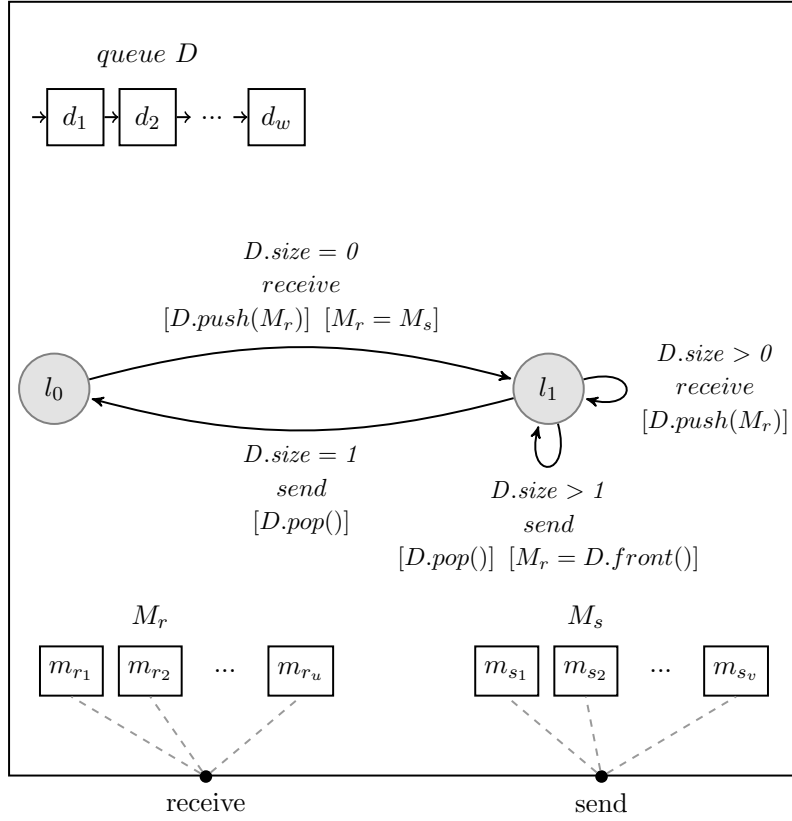


Figure 3.3: The Buffer Component

3.3.2 Transforming Partially Asynchronous Composite Component to a Composite Component

The partially synchronous composite component $C^* = \gamma^*(\{B_i^*\}_{i \in I})$, in BIP^+ , with $I \subseteq [1, n]$ and $B_i^* = \langle B_i, h_i \rangle$ for all $i \in I$, its semantics is defined as the semantics of the composite component C , in ordinary BIP, such that $C = \gamma(AC \cup BU)$ where:

- AC is a set of the atomic components
- BU is a set of buffers such that each buffer corresponds to a receive port in C^* ,
- γ is the set of interactions applied to the set of atomic components $AC \cup BU$ such that $\gamma = \gamma_o \cup \gamma_s \cup \gamma_r$ where:
 - γ_o is the set of all ordinary interactions in C^* ,
 - $RB_u = \{buffer(B_i^*.p).receive \mid B_i^*.p \in P_a \wedge t_i(p) = receive \wedge i \in I\}$ for some interaction $a \in \gamma_{sr}$, the set of receive ports of buffers, corresponding to the receive ports included in interaction a .

- $\gamma_s = \bigcup_{a \in \gamma_{sr}} \{\{B_i^*.p\} \cup RB_u \mid B_i^*.p \in P_a \wedge t_i(p) = send \wedge i \in I\}$ is the set of interactions between each direct send port in interaction a and the ports names *receive* of buffers corresponding to the receive ports, belonging to the same interaction a in C^* . Each of these interactions must allow copying values from the port, whose type was direct send in C^* , to port *receive* of every buffer component participating in the same interaction. Also, the guards of these interactions must always hold.
- $\gamma_r = \bigcup_{a \in \gamma_{sr}} \{\{B_i^*.p, buffer(B_i^*.p).send\} \mid B_i^*.p \in P_a \wedge t_i(p) = receive \wedge i \in I\}$ is the set of interactions between each receive port, included in an interaction, in C^* and the send port of the corresponding buffer. Each of these interactions must allow copying values from the port "send" of the buffer to the port, whose type was direct receive in C^* participating in the same interactions. Also, the guards of these interactions must always hold.

In other words, to convert a BIP⁺ model to its equivalent BIP model, the PA atomic components must be replaced by their matching atomic components. Moreover, buffer components corresponding to the direct receive ports of the BIP⁺ model must be added.

To glue these components together, ordinary interactions between PA atomic components of the BIP⁺ model remain unchanged; *i.e.*, they are all included in the BIP model such that they involve the same ports of the matching atomic components. As for the DSR interactions, they are intervened by the buffer components. Precisely, every DSR interaction a between two PA components is replaced by: (1) an interaction between the port, being the direct-send port in the BIP⁺ model involved in a , and the port *receive* of the buffer corresponding to each direct receive port, also, involved in a , and (2) interactions between each port, being the direct receive port in the BIP⁺ model participating in a , and the port *send* of the corresponding buffer.

The interactions in the composite component replacing DSR interactions of the PA composite component must allow data transfer and must have its guard set to true by default.

Definition 12 (Semantics of composite components). *A composite component, in*

BIP^+ , can be transformed to an equivalent component, in BIP , by adding the buffer components corresponding to every receive port and modifying the existing DSR interactions. Therefore, The operational semantics of a BIP^+ composite component is the same as semantics of its equivalent converted version in BIP . In order to get a deeper inside, recheck the semantics of composite components as defined in definition 6.

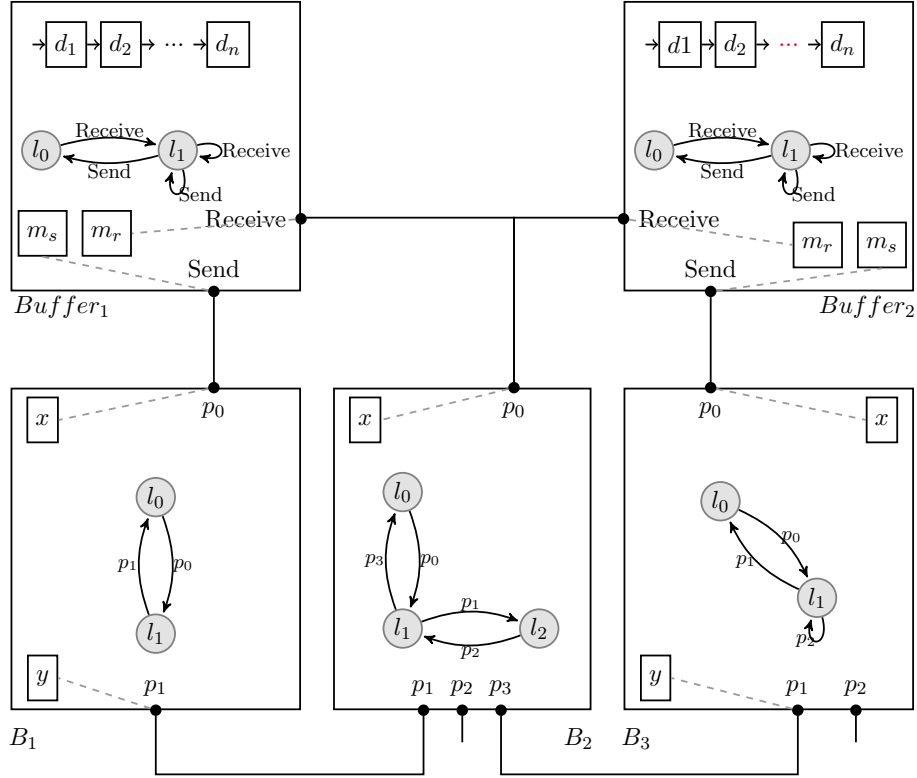


Figure 3.4: Transformation from BIP^+ to BIP Model Example

Example 8. Transformation from BIP^+ to BIP Example Figure 3.4 shows how the BIP^+ model presented in Figure 3.2 is transformed to its equivalent BIP model. All the PA atomic components B_1^* , B_2^* and B_3^* are transformed to their equivalent BIP versions (ignoring ports types) B_1 , B_2 and B_3 respectively. For every direct receive port ($B_1^*.p_0$, $B_3^*.p_0$) we added a corresponding buffer component in the BIP model. Then, the DSR interaction, in the BIP^+ model, from the send port $B_2^*.p_0$ to $B_1^*.p_0$ and $B_3^*.p_0$ 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 BIP^+ model. Additionally, DSR 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 $Buffer2.send$.

3.4 Distributed Implementations from BIP-Plus

We aim to derive correct and efficient distributed implementation given the high-level BIP⁺ model.

One obvious approach to derive distributed implementation from a particular BIP⁺ model is to:

1. transform it to a BIP equivalent model including the buffer components, and
2. get the 3-layer S/R model for the latter.

Although this approach simplifies the development process defining primitives to avoid explicitly implementing the buffer, it drastically affect the efficiency of the generated implementation due to the following reasons:

- We need to create one process/thread for each buffer
- Communication would not benefit from already existing system buffers

Consequently, another approach, to be adopted for the same purpose, is to transform directly the BIP⁺ model to its 3-layer S/R model by proposing a new transformation method.

In the following, we give a detailed description of the two implementations.

3.4.1 *Distributed Implementation through the Intermediate Buffered-BIP Model*

By following this approach, a BIP⁺ model must be converted to its equivalent BIP model with buffers, which can be named as the buffered-BIP Model. Then, this resultant model is transformed to a 3-layer S/R Model to derive correct distributed implementations for the initial high-level BIP⁺ Model. This approach can be referred to as: $BIP^+ Model \rightarrow Buffered - BIP Model \rightarrow 3 - Layer SR Buffered - BIP Model$.

Given the PA composite component $C^* = \gamma^* \{B_i^*\}_{i \in I}$ where, for all $i \in I$, $B_i^* = B_i, t$. Its corresponding composite component is $C = \gamma \{AC \cup_{i \in I} BU\}$ such that $AC = \{B_i\}_{i \in I}$ and BU is the set of buffers corresponding the receive ports in C^* .

Formally, the transformation of C^* , to its distributed implementations, is defined by $\langle C, Pt, A \rangle$ where:

- The components of C are transformed to S/R atomic components to be set in the CL, *i.e.*, for all $i \in I$, $B_i \in AC$ it modified to be B_i^{SR} .
- Pt is a partition of interactions in C in IPL, such that $Pt \subseteq 2^\gamma$ and for all $\alpha_1, \alpha_2 \in Pt$, $\alpha_1 \cap \alpha_2 = \emptyset$ and $\bigcup_{\alpha \in Pt} \alpha = \gamma$.
- A , it is an algorithm, specified in RPL, for solving the committee coordination problem, dining philosophers or circulation tokens.

Because of the first conversion, *i.e.*, from the BIP⁺ to buffered-BIP model, new components, the buffer components, are added to the system along with new interactions, to preserve equivalence between the two models. However, this induces overhead in the distributed implementation (3-layer S/R model), especially in bigger models where there are too many receive ports, as a result of adding more components (as many as the number of receive ports in the BIP⁺ model) to the CL and more interactions to the IPL. This overhead, in turn, prevents this implementation from being as efficient as desired.

This means, following this appeal may not be the best to acquire distributed applications from our proposed high-level model.

Example 9 (3-Layer S/R Buffered-BIP Model). Figure 3.5 presents the 3-layer S/R BIP Model corresponding to the buffered-BIP model in Figure 3.4 . This 3-layer Model, consequently, corresponds to the initial BIP⁺ composite component displayed in Figure 3.2 . In addition to the S/R version of the atomic components of the initial BIP⁺ model, B_1^{SR} , B_2^{SR} and B_3^{SR} , there are three new S/R components, $Bu_0^1^{SR}$ and $Bu_0^2^{SR}$, added to the CL corresponding to the buffer components of every receive port of the initial BIP⁺ model. Furthermore, the IPL includes more interactions than those of the initial BIP⁺ model.

3.4.2 Direct Distributed Implementation from a BIP-Plus Model

We propose, in this section, our approach to procure distributed implementations directly from the BIP⁺ model which is intended to be more efficient than the pre-

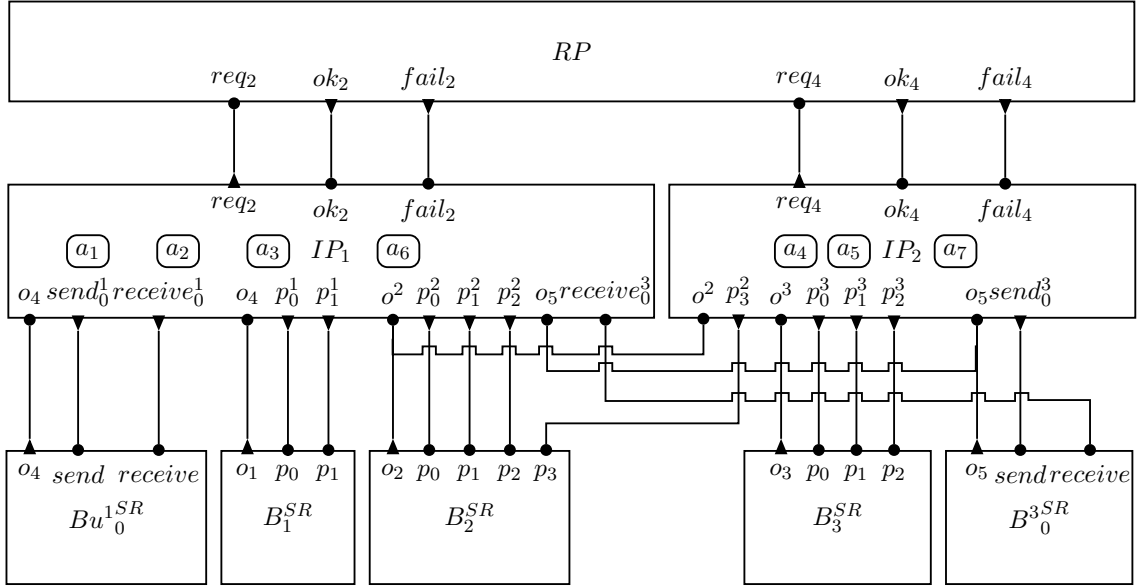


Figure 3.5: 3-Layer S/R Buffered-BIP Model Example

viously explored approach. This approach can be referred to as: BIP^+ Model \rightarrow 3-layer SR BIP^+ Model.

Given a transition $\tau = \langle l, p, l' \rangle$ in a PA atomic component $B^* = \langle P, L, T, X, \{g_\tau\}_{\tau \in T}, \{f_\tau\}_{\tau \in T}, t \rangle$, with $l, l' \in L$ and $p \in P$, we denote l , the source of τ as $\tau.src$, l' , the destination of τ , as $\tau.dest$ and p as $\tau.port$.

Definition 13 (Transformation from B^* to B^{*SR}). An S/R atomic component B^{*SR} , of B^* , is defined by the tuple $\langle P', L', T', X', \{g_\tau\}_{\tau \in T'}, \{f_\tau\}_{\tau \in T'} \rangle$, such that:

- $L' = L_{DSR} \cup L_\perp \cup L_{SR}$ is the set of locations, where:
 - L_{DSR} is the set of locations such that each one of these locations is a source of a transition labeled by a send port or receive port. We have $L_{DSR} = \{l \mid \exists \tau \in T \wedge (t(\tau.port) = send \vee t(\tau.port) = receive)\}$.
 - L_{SR} is the set of locations such that each one is a source of a transition labeled by an ordinary port $L_{SR} = \{l \mid \exists \tau \in T \wedge \tau.src = l \wedge (\tau.port) = ordinary\}$
 - L_\perp is the set of locations introduced to every location l in L_{SR} . We have, $L_\perp = \{\perp_l \mid \exists l \in L_{SR}\}$

- $X' = X \cup \{x_p\}_{p \in P} \cup n$ where x_p is a new boolean variable associated to every ordinary port $p \in P_o$ and n is called a participation number.
- $P' = P \cup o$ where o is an offer port.
- $T' = T_{DSR} \cup T_o \cup T_{SR}$ is the set of transitions where:
 - T_{DSR} is the set of direct send or receive transitions; i.e., labeled by a send or receive port in B^* . $T_{DSR} = \{\tau \mid \tau \in T \wedge (t(\tau.port) = send \vee t(\tau.port) = receive)\}$.
 - T_o is the set of transitions introduced to every location $l \in L_{SR}$ labeled by the offer port. We have $T_o = \{\langle l, o, \perp_l \rangle \mid \exists \tau \in T \wedge \wedge \tau.src = l \wedge t(\tau.port) = ordinary\}$. The guard of each of transition $\tau \in T_o$ is true and its function is the identity function.
 - T_{SR} is the set of response transitions, corresponding to transitions in B^* , labeled by ordinary ports. However, for every transition $\tau \in T_{SR}$, its source location belongs to L_o . We have, $T_{SR} = \{\langle \perp_l, p, l' \rangle \mid \exists \tau \in T \wedge \tau.dest = l' \wedge t(\tau.port) = ordinary\}$. The guard and the function of $\tau \in T_{SR}$ are the same as those of the original transition. These transitions, when executed, apply the function, followed by incrementing the participation number n followed by updating the boolean variable of the associated port.

Example 10 (3-Layer S/R BIP⁺ Model). Figure 3.6 presents an example, in (a), of an LTS in a BIP⁺ model, and shows, in (b), how it becomes after transforming the PA atomic component to an S/R PA component. We have p_1 and p_2 as ordinary ports, p_s as a send port and p_r as a receive port. It is well noticed that a new location \perp_{l_0} is only added for the location l_0 whose outgoing transitions are labelled with ordinary ports. The transition between \perp_{l_0} and l_0 is labeled with offer port. The other transitions are kept the same.

Definition 14 (Transformation from BIP⁺ to SR BIP⁺). Our proposed approach, as previously mentioned, is achieved by the direct transformation from the BIP⁺ model to an equivalent 3-layer S/R BIP⁺ model.

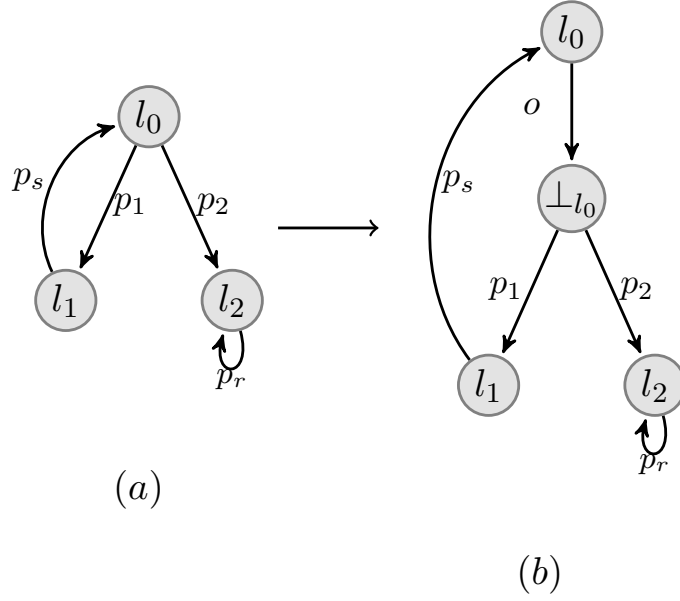


Figure 3.6: BIP⁺Transitions Transformation in 3-Layer Model

The transformation, of a PA composite component $C^* = \gamma^* \{B_i^*\}_{i \in I}$, is defined by the tuple $\langle C^*, Pt, A \rangle$, where:

- All PA atomic components, of C^* are transformed to S/R PA atomic components, i.e., for all $i \in I$, B_i^* is transformed to B_i^{*SR} .
- Pt is a partition of the ordinary interactions of C^*
- A is the algorithm, specified in RPL, for solving the committee coordination problem, dining philosophers or circulation tokens.
- All DSR interactions are kept direct between the S/R PA atomic components corresponding to the PA components of C^*

Consequently, the CL of the model is composed of S/R PA atomic components with their DSR interactions, IPL includes the partition of the ordinary interactions only, and, finally, the algorithm is chosen at RPL obtaining the 3-layer S/R BIP⁺ model.

Example 11 (3-Layer S/R BIP⁺ Model). Figure 3.7 represents the 3-layer S/R BIP⁺ model corresponding to the model displayed in Figure 3.1. It shows that the components layer is only composed of the S/R PA components corresponding to the PA components initially composing the model in Figure 3.1. As for the IPL, there

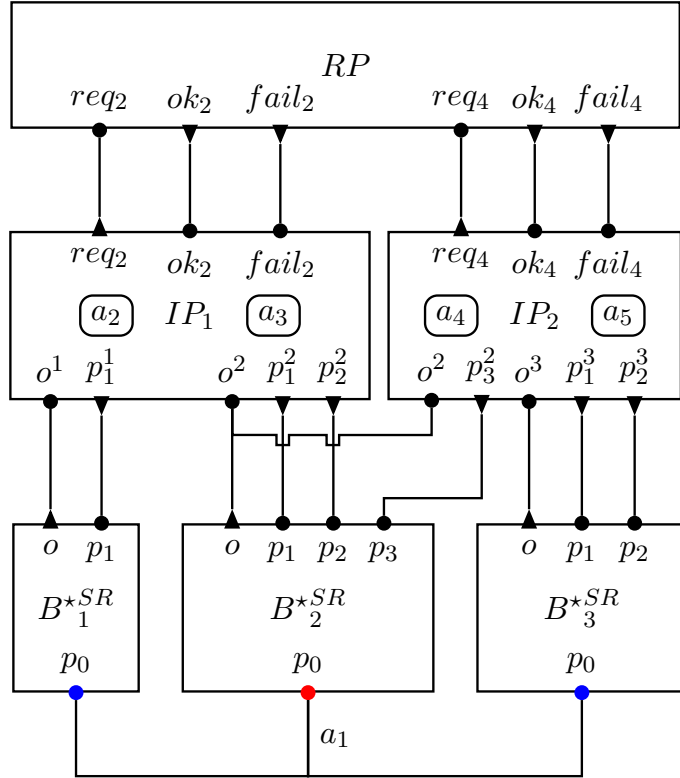


Figure 3.7: 3-Layer S/R BIP⁺ Model Example

are two components IP_1 and IP_2 . IP_1 is responsible for, locally, resolving conflicts of the ordinary interactions a_2 and a_3 where IP_2 is responsible for handling conflicts of a_4 and a_5 . As for RPL , it will resolve conflicts that may occur between either a_2 and a_4 , a_2 and a_5 , a_3 and a_4 , or a_3 and a_5 . Moreover, it shows that the DSR interaction a_1 is not assigned to any of the IPL components. a_1 is still a direct interaction between the components in CL .

This approach must guarantee efficient and correct distributed implementations of a high-level BIP⁺ model.

As mentioned before, the offer and response are implemented using asynchronous low level send and receive primitives. As for direct send and receive ports, they will be, also, implemented using the same asynchronous low-level primitives for we are targeting asynchronous direct data transfer between components. For this reason, to implement all of them, we must utilize low level communication primitives that make use of tags to differentiate between the many responses or messages received in a S/R BIP⁺ model to avoid any confusion and faulty data exchange.

Furthermore, since we target a sender triggered communication such that a message must be sent to the receiver when the sender is ready whether the receiver is

ready or not, the message must be preserved at the receiver side, so the receiver gets access to the message when the receiver is ready. For this purpose, the messages sent are stored in the system buffer to which the receiver must access. To make this possible, the communication primitives implementing our approach must also support access to the underlying platform.

3.5 Proof of Correctness of BIP-Plus Distributed Implementation

After we define the BIP⁺ framework in this chapter, we illustrate how a BIP⁺ model can be transformed to a BIP model, or, equivalently speaking, how a BIP⁺ model can be constructed as a BIP Model. This transformation allows obtaining distributed implementation of a particular BIP⁺ model by transforming the equivalent BIP model to its 3-layer S/R model.

A BIP model is equivalent to its corresponding 3-layer S/R model by weak bisimulation. Therefore, obtaining the distributed implementation for a BIP⁺ model through this way (BIP⁺ → BIP with buffers → 3-layer S/R BIP) is, also, proved to be correct. However, the main reason we do not go this way is the overhead delivered by adding buffers to the components layer (CL) and their associated interactions to the interaction protocol layer (IPL). For this reason, we propose another approach for generating distributed implementations of a BIP⁺ model through a direct derivation of the 3-layer S/R model from it (BIP⁺ → 3-layer S/R BIP⁺).

In this section, our aim is to prove that the obtained 3-layer S/R BIP⁺ model is equivalent to its relevant 3-layer S/R buffered-BIP model. As a result of proving them equivalent, the correctness of our proposed approach for deriving distributed implementations is guaranteed.

The proof consists of 3 independent steps. Consider the derivation of distributed implementations of a particular BIP⁺ model utilizing both approaches illustrated in chapter 4 section 3.4. As a result we obtain two models: (1) the 3-layer S/R buffered-BIP model, and (2) 3-layer S/R BIP⁺ model. Given the resultant 3-layer S/R buffered-BIP model, the first step proves that the role of buffers is preserved in the relevant 3-layer S/R BIP⁺ model. As for the second one, it proves that

execution of interactions in which the buffer components are involved do not need to be managed by the upper layers. Finally, given a 3-layer S/R BIP⁺ model, the third step proves how the existing conflicts are handled, or, otherwise, do not violate the global semantics.

For illustration purposes, we present an example, in Figure 3.8 , of a BIP⁺ model that includes many interactions possibilities sufficient to support our proof steps. Figure 3.9 is the equivalent BIP transformation of the model presented in Figure 3.8 .

In addition, we show both of the 3-layer S/R models as (1) derived directly from the BIP⁺ composite component of figure 3.8 and (2) derived from the equivalent BIP model of figure 3.9 in compiledfigures 3.10 and 3.11 respectively.

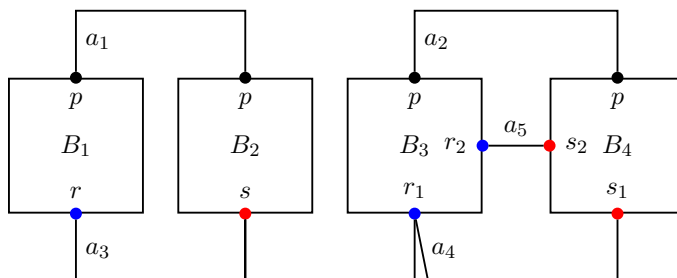


Figure 3.8: BIP⁺ Model Example

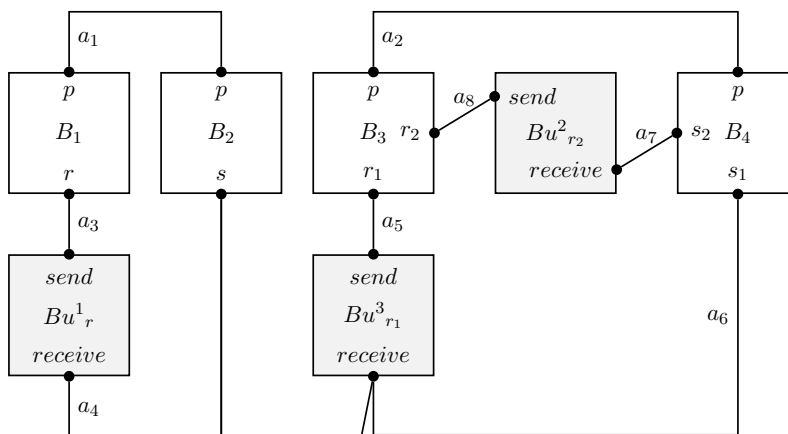


Figure 3.9: BIP with Buffers Model Equivalent to that of Figure 3.8

Example 12. For the BIP⁺ model in Figure 3.8 , we assume that $B_1^*.p$, $B_2^*.p$, $B_3^*.p$ and $B_4^*.p$ are ordinary ports (black bullets). On the other hand, $B_1^*.r_1$, $B_3^*.r_1$ and $B_3^*.r_2$ are direct receive ports (blue bullets), where $B_2^*.s_1$, $B_4^*.s_1$ and $B_4^*.s_2$ are direct

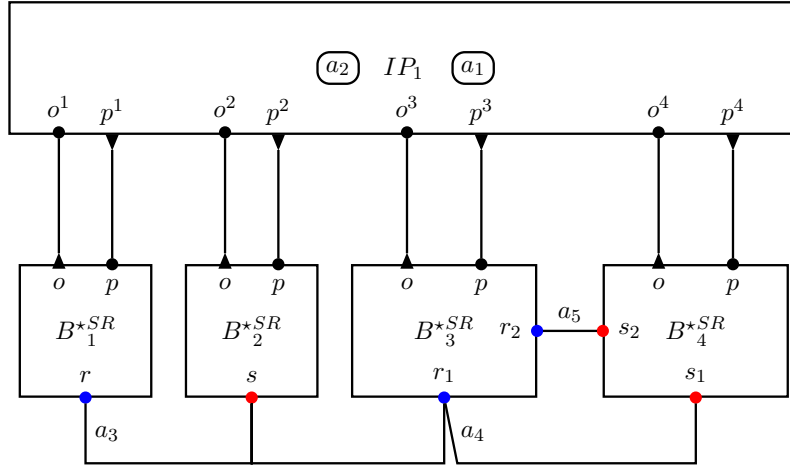


Figure 3.10: 3-Layer S/R BIP⁺ Model of 3.8

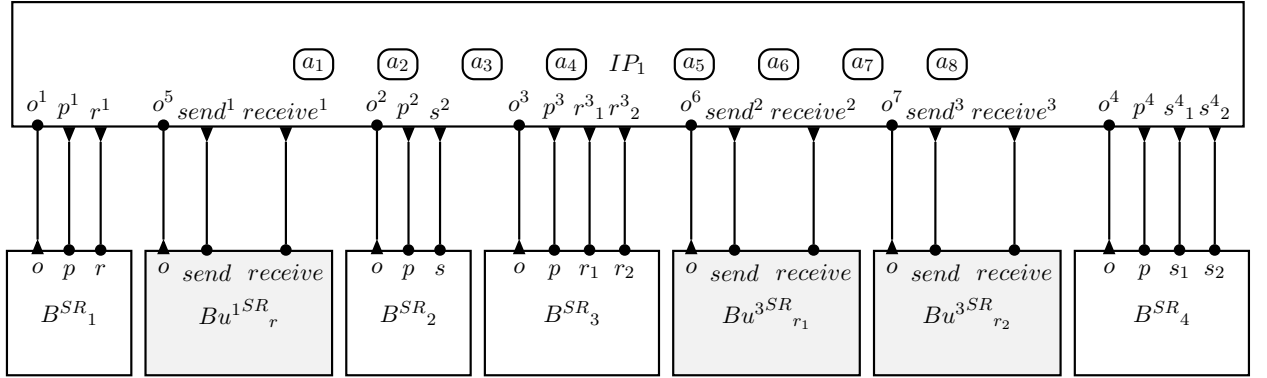


Figure 3.11: 3-Layer S/R BIP Model of 3.11

send ports (red). As for its equivalent BIP model with buffers, in Figure 3.9 , the buffer components are the light gray components for more emphasis.

As for the S/R BIP⁺ model in Figure 3.10 , the ordinary interactions a_1 and a_2 are assigned to the only component in IPL, IP_1 . On the other hand, the DSR interactions a_3 , a_4 and a_5 are all kept direct between the components of CL.

As for the S/R BIP model in Figure 3.10 , all interactions are assigned to the only component of IPL, IP_1 . In both examples, we use one component in the IPL to simplify them. Having one component in the IPL, delivers a 2-layer model instead of a 3-layer one. In IP_1 , non-offer ports superscript refers to the corresponding component's ID where the components' IDs are the subscripts of their names. similarly for offer ports corresponding to non-buffer components As for the rest offer ports, o^5 , o^6 and o^7 correspond to Bu^1SR_1 , Bu^3SR_1 and Bu^3SR_2 respectively.

3.5.1 Preservation of Buffer Components Role

In this section, we start with the first step of our proof. We prove that the buffer components' role in 3-layer S/R buffered-BIP model is preserved in the 3-layer BIP⁺ model.

The transformation from a BIP⁺ model to its corresponding BIP model requires adding the buffer components. However, these buffer components are not added from nowhere. We assume that they resemble system buffers and behave identically i.e., they receive messages from a sender in order and send them to a receiver, in the same order in which they have been previously received, when the receiver is willing to receive a message.

Furthermore, practically, the BIP⁺ model makes use of implicit buffers, the system buffers. In other words, the DSR interactions of the BIP⁺ composite components are intervened by system buffers at the level of every receive port, and they can be thought of as two interactions combined: (1) an interaction between the sending component and the system buffer, (2) an interaction between the system buffer and the receiving component.

In spite of having system buffers and buffer components equivalent, we have avoided using the latter and made use of the system buffers in order to reduce the overhead that buffer components create to the model and in order to utilize the underlying platform. Utilizing the underlying platform helps obtaining more efficient distributed implementations.

Consequently, we have proved that the buffer components role of the 3-Layer S/R model with buffers is preserved in the distributed implementation of the BIP⁺ model.

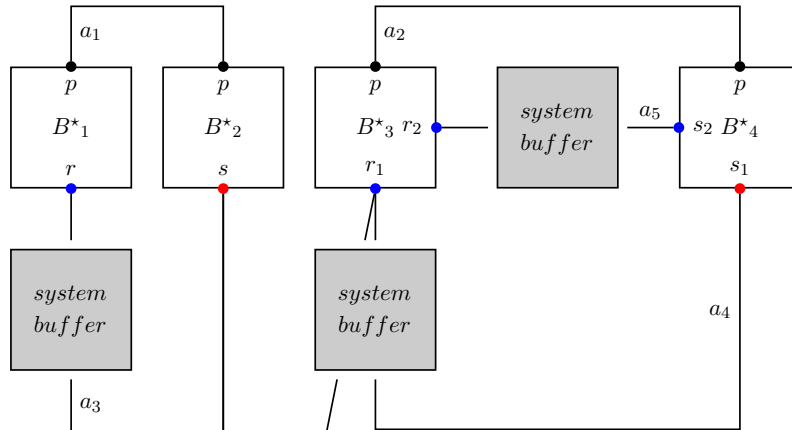


Figure 3.12: BIP⁺ Model with System Buffers Example

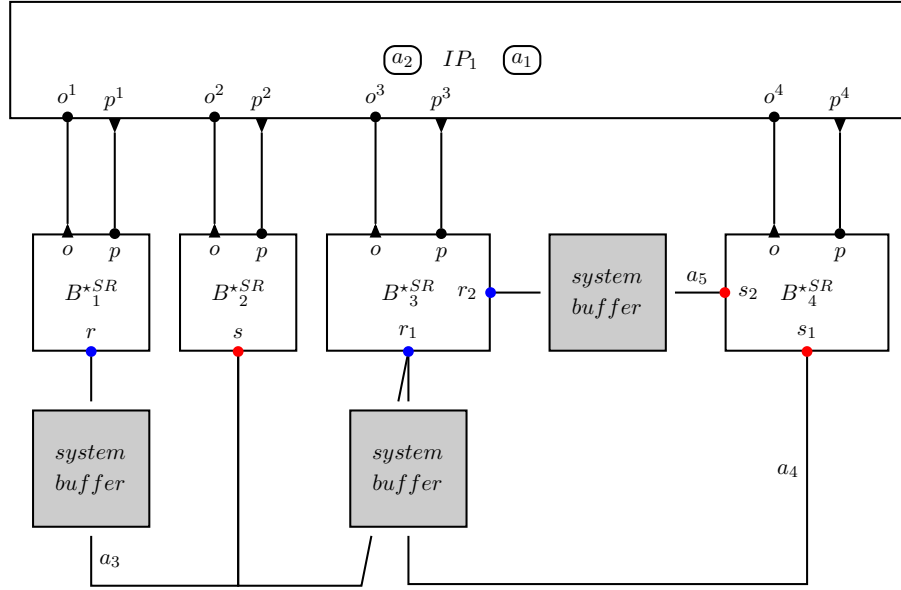


Figure 3.13: 3-Layer S/R BIP⁺ Model with System Buffers Example

Example 13. Figure 3.12 shows the BIP⁺ model example, of figure 3.8, including the system buffers. If any components send a message, through the direct send port, to another component receiving this message through a particular direct receive port such that these ports participate in the same DSR interaction, the message send will be stored in the system buffer accessible by the receiving component. For instance, when s_2 in B_4^* is enabled, B_4^* sends data through a_5 to r_2 B_3^* , the sent data will be stored in the system buffer until $B_3^*.r_2$ is enabled so it receives the messages from the system buffer. For more illustration we show the 3-Layer S/R model for it Figure 3.13 .

3.5.2 Preservation of IP Role for Buffers' Interactions Selection

In this section we discuss our second proof step. We prove that the role of the IP in selecting possible interactions to execute is preserved for interactions in which the buffer components' ports are involved without even passing through the upper layer.

For this purpose, we have to tackle two possibilities:

1. Executing interactions in which the buffer's *receive* is participating such that these interactions bypass the upper layer, and

2. Executing interactions in which the buffer's *send* is involved where these interactions bypass the upper layer.

3.5.2.1 Execution of Buffer's "receive" Interactions

Given the BIP model equivalent to BIP⁺, in the buffer component, the port "receive" is always enabled, i.e the buffer is always ready to receive messages. For this reason, the buffer component's port "receive" does not really affect the execution of interactions in which it is involved, but the other components involved ports do. This is because an interaction's execution is possible when all of its ports are enabled. Nevertheless, if we can assure that one of the ports is always enabled, we can avoid checking if it is activated every time, and it is enough to only check if the other participating ports are.

Additionally, the interactions in which the buffer component port "receive" is involved usually consist of:

- the buffer's port "receive", in addition to some other buffers' ports named "receive" in case of a DSR interaction involving multiple receivers, and
- a sending port of a certain non-buffer component.

Then, only if the involved port of the sending component is enabled, the interaction can be executed.

Hence, the execution of an interaction, involving a buffer's "receive", can be determined by one only involved port, the sending port, the one of the non-buffer component. To decrease the useless overhead in upper layer, the decision for executing such interactions can be taken at the level of the non-buffer participating component itself.

3.5.2.2 Execution of Buffer's "send" Interactions

The port "send" of the buffer is enabled when there is a message in its internal queue. As long as the queue of the buffer is not empty, the buffer component is willing to send its stored messages to another components that is willing to receive. The interactions that include the buffer's port "send", consist of:

- the buffer’s port ”send” and only one receiving port of a certain non-buffer component

So, these interactions are executed if both ”send” and the other receiving port are enabled which requires to make use of the upper layer that checks if all the ports of interactions are enabled. Fortunately, assuming that the buffer component resembles the system buffer, in the receiving component at the port involved in a DSR interaction, there is an access (through probing for instance) to the system buffer and there can be a local direct check if there is a message waiting to be received by this port. Therefore, the execution of these interactions does not need the upper layer for it can be decided, locally, at the receiving component’s side.

Finally, as discussed in step 1, assuming that buffer components in the BIP model resemble system buffers utilized in BIP⁺ models, we prove, in step 2, that DSR interactions do not need to pass through the upper layers and they are kept direct between the participating components and execution of these interactions can be determined at the level of the components.

Yet, we need to enforce our proof with a third step about conflict handling. This is due to the fact that not only is the upper layer an engine to get information about enabled ports of components and execute the enabled interactions, but also a conflict handler. We cannot avoid the upper layer without taking into consideration conflicts between interactions.

3.5.3 Conflicts Handling

Finally, we present the third step of our proof tackling the issue of conflicts. Conflicts may occur between interactions, in our proposed 3-layer architecture for the BIP⁺ model, for we choose to keep DSR interactions direct between components and not passing through the upper layer that resolves conflicts.

In order to prove that bypassing the upper layer by the DSR interactions do not violate the global semantics of the model, we are going to discuss conflict handling for all the following cases:

- Conflicts between only ordinary interactions
- Conflicts between ordinary and DSR interactions
- Conflicts between only DSR interactions

3.5.3.1 Conflicts Between Ordinary Interactions

The conflicts between the ordinary ports of a BIP⁺ model are resolved by the interaction protocol layer and the reservation protocol layers.

Example 14 (ordinary Interactions Conflict Handling). *The ordinary interactions, in the model of figure 3.10, a_1 and a_2 , are assigned to the component IP_1 of the IPL that resolves the conflicts between them.*

3.5.3.2 Conflicts Between Ordinary and DSR Interactions

Ordinary interactions do not conflict with DSR interactions. Back to the definition 8 of PA-atomic components in BIP⁺ in chapter 4, the outgoing ports from a particular state can only be either ordinary ports or direct send and receive ports.

Then, there exist no two interactions such that they both involve ports labeling transitions having the same source location where one of these ports is ordinary and the other is direct send or direct receive. Therefore, it is guaranteed that DSR interactions do not have conflicts with any of the ordinary interactions.

3.5.3.3 Conflicts Between DSR Interactions

Conflicting DSR interactions are interactions that either:

- involve a common direct receive ports, or
- involve ports that label transitions outgoing from the same source location

One note is that a direct send port cannot participate in more than one interaction in the whole PA composite component, so there is no way that conflicting interactions include a common receive port.

One Direct Receive Port in Multiple interactions In BIP⁺, executing concurrently multiple DSR interaction having a common receive port does not violate the global semantics.

To begin with the proof, we shall mention that, practically in BIP⁺, a receiving port, participating in many DSR interactions, gets the messages sent to it one at a time at each execution of a transition labeled by this port (completion of each of these interactions execution). Also, they are received in the same order in which they are saved at the system buffer even if senders sent them simultaneously.

When a BIP⁺ model with such a conflict is converted to its BIP model, the interactions that used to share a common receive port (causing the conflict) will, eventually, share the port "receive" of the buffer corresponding to the initial receive port involved in these conflicting interactions. Besides, the initial receive port will only be involved in one interaction with the port "send" of that buffer. Thus, the conflict, now, is because the buffer port "receive" is involved in several interactions.

Example 15 (Buffers port "receive" Conflicts). In Figure 3.8, a_3 and a_4 have port $B_{3.r_1}$ in common. When the model is transformed to the BIP version, we get, instead, a_4 and a_6 sharing the port "receive" of the buffer corresponding to $B_{3.r_1}$

So, we are going to prove that the way these conflicts are handled by the upper layer in the 3-layer S/R BIP model with buffers is preserved in the 3-layer BIP⁺S/R model although the upper layer is bypassed.

If interactions sharing the same port "receive" of a buffer are assigned to the Interaction Protocol Layer, the latter would wait until any of them is ready so it can be executed. But, when all of the interactions sharing the same port "receive" are ready, the upper layer must only pick one interaction out of them to be executed.

No matter which interaction is picked at a moment, the information about the non-executed interaction will be re-taken into consideration, in the upper layer, for another selection. Fortunately, since the buffer's port "receive" is always enabled, any one of the non-executed interactions can be selected and executed as long as there are ready interactions involving "receive". Until all these interactions are executed, the selection process by the upper layer will be repeated (since the buffer is always available for receiving messages), thus, all messages sent to the buffer will be received by it and saved in its internal queue in order.

This means that if multiple components are ready to send messages to a buffer, all of the messages are being sent successfully and received by the buffer one after the other, in a random order according to the selection of the upper layer.

Example 16 (Buffer port "receive" Conflicts Handling). Figure 3.14 shows an example of a buffer component interacting with two other components. This example

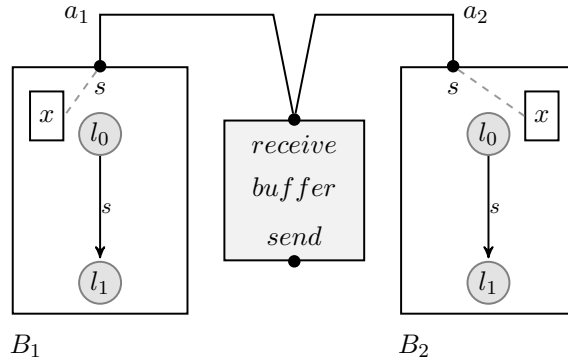


Figure 3.14: Many Senders and one Receiver Conflict Example

shows a conflict between the two interactions a_1 and a_2 for they share the port "receive" of the same buffer component. If this BIP model is transformed to its distributed implementation, the upper layer will receive information about ports of a_1 and a_2 , which are $B_1.s$, $B_2.s$ and "receive", (knowing that the latter does not affect the availability of the interactions as mentioned in step 1 for buffer is always available to receive messages). As shown in the figure, there are no guards labeling transitions through $B_1.s$ and $B_2.s$, so the ports are enabled and the interaction including them is, also, enabled. But, only one will be picked, either a_1 or a_2 . If a_2 is selected by the upper layer, $B_2.s$ is no longer enabled, but B_1 is still in the same state and its port s is still enabled. Since the buffer is always ready to receive, a_1 is ready to be executed and, since there are no more conflicting interactions assigned to the upper layer, it is surely selected.

In comparison with the BIP⁺ model, if multiple senders are sending to only one receiver consequently, their messages are all received by the system buffer in the order they arrived with. So, we maintained the upper layer's behavior to these conflicting interactions without really going through it.

Therefore, this conflict, if not handled by the upper layer, does not violate the global semantics of the model.

Conflicts between Ports of same Source Location Another conflict may exist, in a BIP⁺ model, between DSR interactions, when some interactions include ports label transitions having the same source location, such that these ports are either:

1. send ports
2. receive ports

3. send and receive ports

For the first case i.e the ports are of type send, we take as an example interactions a_4 and a_5 in figure 3.8, where the behavior of B_3 and B_4 enforces this conflict. For more illustration we present example of B_3 and B_4 where a_4 and a_5 are conflicting in figure 3.15 (ignoring the other model details that are out of interest for this case).

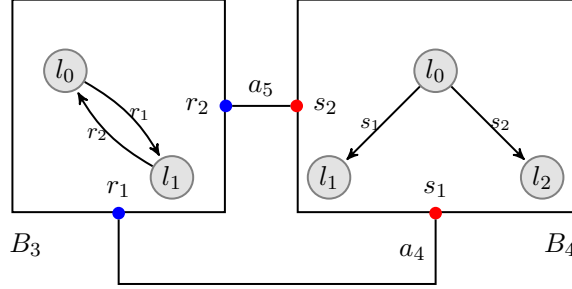


Figure 3.15: Conflicting Interactions Involving Direct Send Ports

When the model with this conflict is converted to its BIP implementation, these conflicts are not caused by the interactions including the receive ports of the receiving component anymore. Otherwise, they include the initial send ports of the sending component and the port "receive" of the buffer corresponding to the receiving ports, e.g a_6 and a_7 in figure 3.9. After getting the distributed implementation of this model, these conflicts are resolved by the upper layer.

However, in BIP⁺ model, we do not need to include these conflicting interactions in the upper layer. A PA-component can send a message through its sending port to the system buffer when this port is enabled at any time. Then, in case we have many send ports from a given state, and they are all enabled, in turn, the interactions involving these ports can be executed which causes the conflict. For this reason, we implement the function of the upper layer at the level of the state, in the component, from which several transitions labeled by send ports are outgoing. Now, from a given state, only one interaction involving one of its outgoing transitions' enabled ports is allowed to be executed. The decision of the interactions (between sending PA-component and system buffer) to be executed is randomly taken. For example, in figure 3.15, we have all ports of a_4 and a_5 enabled at state $B_4.l_0$, but either only a_4 or a_5 can be executed.

For the second case of conflicting interactions (i.e including receive ports), we take another example for the behavior of B_4 and B_5 as shown in figure 3.16.

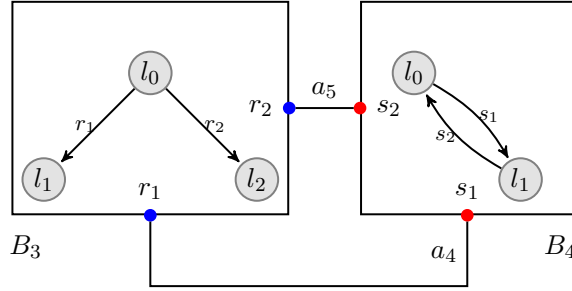


Figure 3.16: Conflicting Interactions Involving Direct Receive Ports

On the other hand, in this case, transforming the BIP^+ model to its BIP version, this conflict becomes between interactions including the receive ports of the receiving components and the port "send" of their corresponding buffer (e.g a_5 and a_8). The upper layer checks which of these interactions are ready for execution and selects only one to handle conflicts.

Nevertheless, in BIP^+ , we can avoid passing through the upper layer in this case. This is because in a PA-component as a particular state, we have access to the system buffer and can check if there is a message waiting to be received by a specific port. So, there is no need for the upper layer to check if the interaction is ready, as proved in step (2), as this can be done locally in the PA-component. As for conflict handling- somehow similar to case (1)- we implement the role of the upper layer inside the PA-component. For a given state, we check all the enabled receive ports and, then, check if there is a message waiting to be received, by each of these ports, in the system buffer. Finally, only one message is selected to be received by an enabled receiving port (only one interaction between system buffer and receiving port is executed). This way, we can say the role of the upper layer is implemented local to the component and precisely at the state level in which there are receiving ports. Therefore, these interactions (between system buffer and receiving port) do not need to pass through upper layer to handle the conflicts.

Finally, as for the third case, i.e the ports are a mix of send and receive ports, the conflict is tackled by giving a priority to the interaction involving the send port. In the PA-component at the state level (whose transitions' ports are involved in the conflicting interactions), as long as there is at least one send port, the random selection implemented for case (1) is processed. Otherwise, the selection process for case (2) is adopted.

Consequently, all conflicts between DSR interactions are tackled in the 3-layer

S/R BIP⁺ model without the need to utilize the IP and RP layers of the 3-layer S/R BIP model.

To sum up this chapter, We prove that our proposal for the 3-layer S/R BIP⁺ model is equivalent to 3-layer BIP model with buffers by preserving the behavior of the buffer components and upper layer although they are not explicitly used in our suggested model.

Chapter 4

BIP-PLUS TOOL IMPLEMENTATION

Contents

4.1	BIP-Plus Tool Implementation From the BIP tool	49
4.2	Middle-End Modifications	49
4.3	Back-End Modification	50

4.1 BIP-Plus Tool Implementation From the BIP tool

We have implemented the BIP⁺ tool by making use of the existing BIP tool applying modifications to integrate our proposed extension.

We modified two parts in the BIP tool, the middle-end and back-end. In the BIP tool, the middle-end, is responsible for generating a send/receive BIP model from the bip file. On the other hand, the back-end is responsible for generating C++ code with MPI, C++ serial code or sockets. In the back-end of the BIP⁺ tool, our only aim is to generate C++ code with MPI, so the part responsible for this generation, in the BIP tool, will be altered.

4.2 Middle-End Modifications

The middle-end of the BIP⁺ tool takes a BIP⁺ file as an input. After the user partitions ordinary interactions between components over the interaction protocol layer and specifies the scheduling algorithm to be adopted by the reservation protocol layer, the tool parses this file and generates the corresponding SR BIP⁺ file.

The code of this part has been modified to satisfy the theory mentioned earlier. In effect, the states whose outgoing transitions are labeled with non-ordinary ports (direct send or direct receive) must stay unchanged. As for the states with outgoing transitions labeled by ordinary ports, the corresponding busy states must be added and new transitions between these states must be created including the one labeled by the offer port. For this sake, we distinguish between 3 types of states. A state whose all outgoing transitions are labeled by ordinary ports is said to be an ordinary state, and the one whose all outgoing transitions are labelled by direct send ports

is called a send state, else it is said to be a receive state. The main modification applied to the BIP tool here is to check the type for every state before adding any new states or transitions. In case the state is not ordinary, the part responsible for creating new states and transitions and other details (at state level) relevant to a SR BIP model is skipped.

In addition, a DSR interaction must also be kept direct between components in the components layer unlike ordinary interactions that become intercepted by components from the interaction protocol layer according to the specified partition. For this reason, interactions checks are also made, in case a connector is a direct-send receive, it is kept as it is.

Furthermore, at this stage, the states and interactions are checked if they are valid (i.e a send port is used once in all DSR interactions, one send port is used in a DSR interaction, all transitions from a particular are enabled by ports of the same type).

The generated output, includes all the port, connector (interaction), atom and compound types of both the BIP⁺ and the SR BIP⁺ models. To prepare the input of the back-end part of the tool, the user must remove the atom and compound types of the old BIP⁺ model and keep the rest especially the connector(interaction) type of the DSR interactions.

4.3 Back-End Modification

The back-end part of the BIP⁺ tool takes, as an input, the output of the middle-end and generate C++ with MPI. In addition to the SR BIP⁺ file, it takes another one that includes the data types of exported variables/data by ports and their equivalent MPI type along with the size of this data. At this stage, all the interactions between components of layer are generated to an asynchronous (MPI) send receive primitives. The transferred messages between a sender and the corresponding receiver must have a matching tag. For the tag to be unique within a compound, the ports are given an identification number and the tag of the message is set to the ID of the receiver.

As for the DSR interaction, it is generated as also asynchronous send and receive MPI messages. The use of unique tag for each message helps eliminating any confusion between messages exchanged within the same component.

Many DSR interactions can be executed concurrently. However, from a given send state, if more than one send port are enabled, only one interaction involving

those enabled ports can be executed. Since there is no engine to handle this, a random selection of those port is implemented.

Although asynchronous primitives are used, the receiving party must wait until a message can be received through an enabled port at a certain state. Also, in case many receive ports are enabled at a receive state, and if there are messages ready to be received at some of them, random selection of a one receive port is made at this level. This ensures that only one message can be received by an enabled receive port.

Chapter 5

BENCHMARKS

Contents

5.1	Two Phase Commit (2PC)	53
5.1.1	BIP Implementation of 2PC Protocol	55
5.1.2	BIP-Plus Implementation of 2PC Protocol	58
5.1.2.1	Experimental Results	62
5.2	Support Vector Machines (SVM)	69
5.2.1	BIP Implementation of (SVM)	70
5.2.2	BIP-Plus Implementation of (SVM)	74
5.2.2.1	Experimental Results	76

To highlight the optimization that BIP⁺introduces, this chapter presents two applications that use buffering to carry-out asynchronous data transfer operations. Section 5.1 implements Two Phase Commit to solve the transaction commit problem, and section 5.2 implements Support Vector Machines to solve handwritten digit recognition. Since BIP⁺allows for PA send/receive, we start by providing a BIP design and implementation and then transform it to a more efficient model using BIP⁺PA send/receive primitives in order to make the comparison between BIP and BIP⁺.

The transformation is performed in the same way for both case studies presented. The starting BIP models conventionally follow these properties by construction:

- Buffering components used match the design specified in chapter 4 section 3.3.
- Each send port of a buffer is connected to exactly one interaction.
- Each receive port of a buffer is connected to at least one interaction.
- Each interaction connecting a buffer has exactly two ports
- No buffer component is connected to another buffer component

Part (a) of Figure 5.1 illustrates a BIP composite component that satisfies the aforementioned properties.

Following this convention, any buffered BIP model can be transformed to an equivalent BIP⁺ one by performing the following steps. The transformation is illustrated in Figure 5.1, where the starting BIP model is shown in (a), and the resulting BIP⁺ model is shown in (b):

- For each buffer component Bu let $\{a_1, a_2, \dots, a_n\}$ be the set of interactions connected to the port called receive of Bu , and $\{S_1, S_2, \dots, S_n\}$ be the atomic components respectively connected to these interactions via ports $\{S_1.p, S_2.p, \dots, S_n.p\}$. Then let a_r be the only interaction connected to the port called send of Bu , and R be the atomic component connected to this interaction through port $R.p$.
- Construct a BIP⁺ model matching the design of the BIP one, but with every port made ordinary.
- Remove every interaction a_i for $i \in [1, n]$ and transform each port $S_i.p$ into a *send* port
- Remove the interaction a_r and make port $R.p$ a *receive* port
- Remove Bu , and connect each port $S_i.p$ to $R.p$ using a send/receive interaction a_i^*

5.1 Two Phase Commit (2PC)

Transaction commit [11] is a consensus problem where different nodes, called resource managers $\{rm_1, rm_2, \dots, rm_n\}$ spanning a distributed system, have to reach consensus on a binary decision of whether to *commit/abort* a transaction. Each resource manager has the ability to locally *commit/abort* a transaction based on a local decision. In a fault-free system, the problem requires the global system to commit as a whole if each rm has locally committed, and that it aborts as a whole if any of the rm 's has locally aborted [11]. In case of global abort, locally-committed rm 's may perform roll-back steps to undo the effect of the last transaction [12]. To allow for this roll-back, resource managers keep necessary information in a log while they locally-commit the transaction.

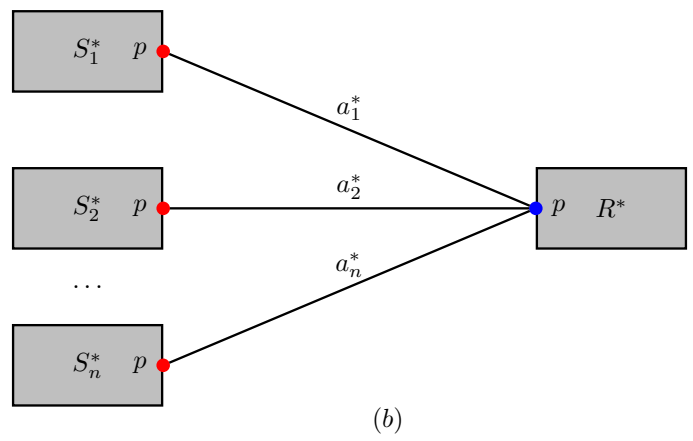
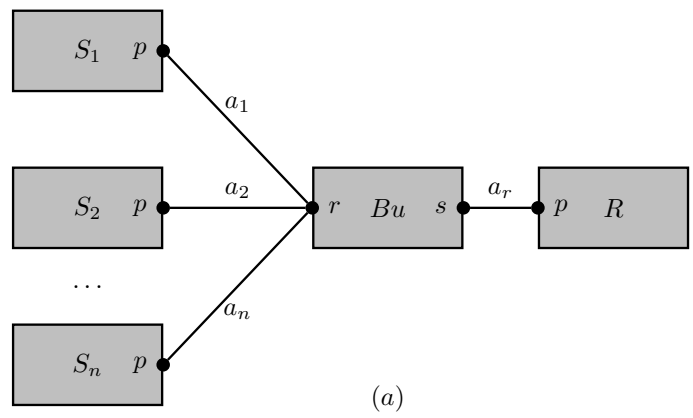


Figure 5.1: Benchmarks transformation.

Two phase commit protocol is a solution proposed by [13] to solve the transaction commit problem. It uses a transaction manager tm that coordinates between resource managers to ensure they all reach one global (final) decision regarding a particular transaction. The global decision, although made by the tm , is made upon the feedback it gets from resource managers after each of which has made its own local decision.

The protocol, running on a transaction t_j , uses a client c , a transaction manager tm and a non-empty set of resource managers $\{rm_1, rm_2, \dots, rm_n\}$ which are the active participants of transaction t_j . The protocol starts when c sends remote procedure calls rpc 's to all the participating rm 's. Then each rm_i makes its local decision $d_{ij} \rightarrow \{true, false\}$ regarding transaction t_j based on local criteria.

- $d_{ij} = true$ iff rm_i can locally-commit transaction t_j
- $d_{ij} = false$ iff rm_i cannot locally-commit transaction t_j

The resource manager then resides in a state that represents the decision it made. After all local decisions have been made and reported to tm , the latter makes a global decision $D_j = (d_{1j} \wedge d_{2j} \wedge \dots \wedge d_{nj})$ that all the system will agree upon. When D_j evaluates to true, the system will globally-commit as a whole, and it will abort as a whole when D_j evaluates to false.

In this implementation, the systems is assumed to be non-faulty, where all the nodes are expected to function correctly without any errors. Network connections between nodes are also assumed to be reliable and ensure the deliverance of non-duplicated messages within a bounded time limit. The network is not subject to be partitioned, and all storage devices (to hold the commit log) are stable and non-faulty. For the purpose of the benchmark, the system is implemented to do multiple transactions in a row, and all transactions have the same number and identity of participants.

5.1.1 BIP Implementation of 2PC Protocol

Each subsystem $\{c, tm, rm_1, rm_2, \dots, rm_n\}$ is represented by an atomic component in a BIP model where only strong synchronization(*rendezvous*) is used. Extra buffering components follow the design provided in Figure 3.3. The following describes the design of each atomic component:

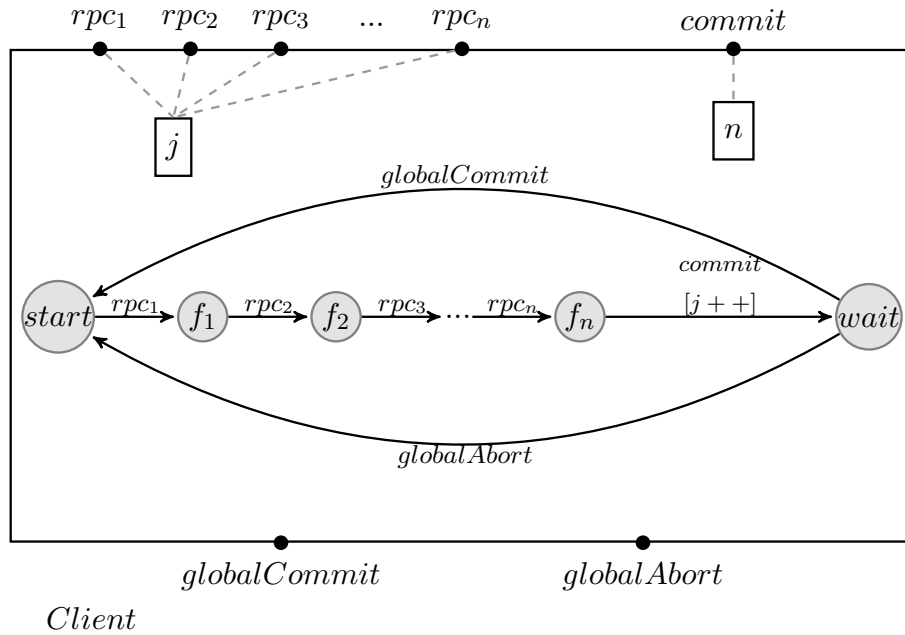


Figure 5.2: Client.

Figure 5.2 shows the client component c which initiates a transaction by sending remote procedure calls $(rpc_1, rpc_2, \dots, rpc_n)$ accompanied with the current transaction number j to $(rm_1, rm_2, \dots, rm_n)$ respectively. It then commits to tm notifying it of the number of participants in the current transaction (n) and *waits* for the reception of the global decision which will be later made by tm .

As shown in Figure 5.3, each rm starts the transaction by getting a remote procedure call from the client. The rm then makes a randomly generated decision based on a probability to locally-abort the current transaction. It then updates its local *decision* variable and notifies the tm accordingly. It stays in *wait* location until it hears back from tm whether to perform a global *commit/abort* for the current transaction.

Figure 5.4 shows the transaction manger. It gets triggered by the client to start coordinating a transaction. After receiving the number of participants, it *waits* to get the feedback from all rm 's updating its local variables according to each d_{ij} it receives. Upon receiving n local decisions, it evaluates D_j , and globally commits the transaction if all rm 's have locally-committed, or globally-aborts otherwise. This step is performed in synchrony with the transaction manager and every participating resource manager, which where all waiting for the global decision

Figure 5.5 shows the composite component which contains one instance of client component, one instance of transaction manager component, and n instances

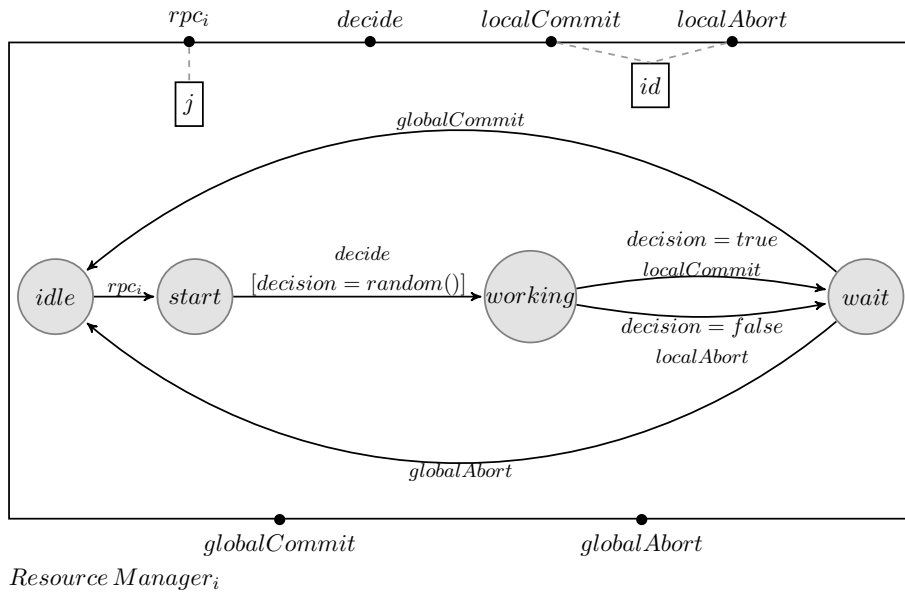


Figure 5.3: Resource Manager_i.

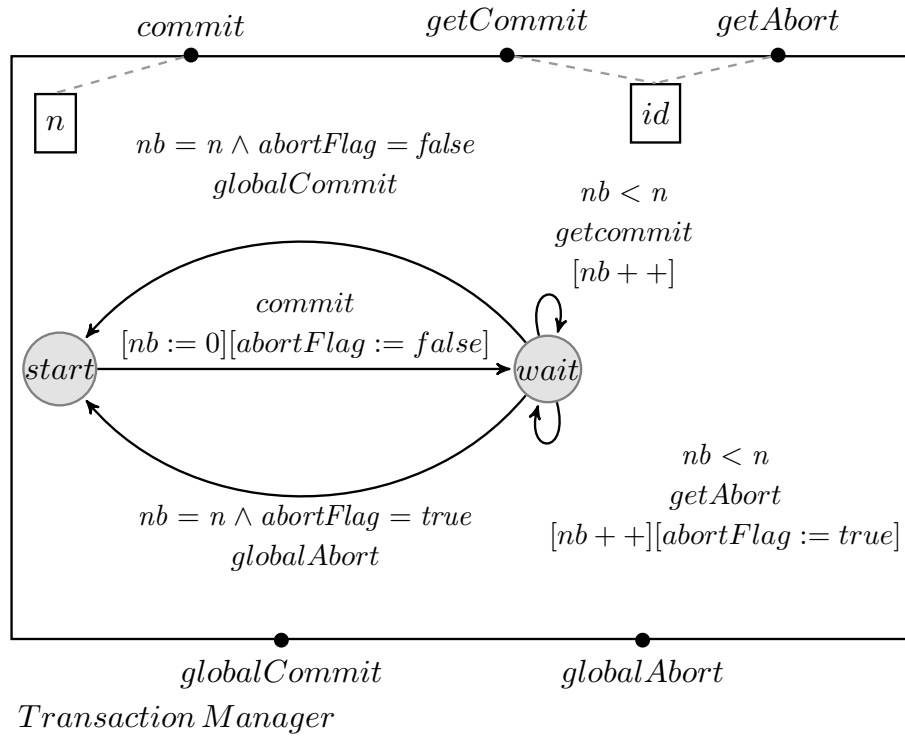


Figure 5.4: Transaction Manager.

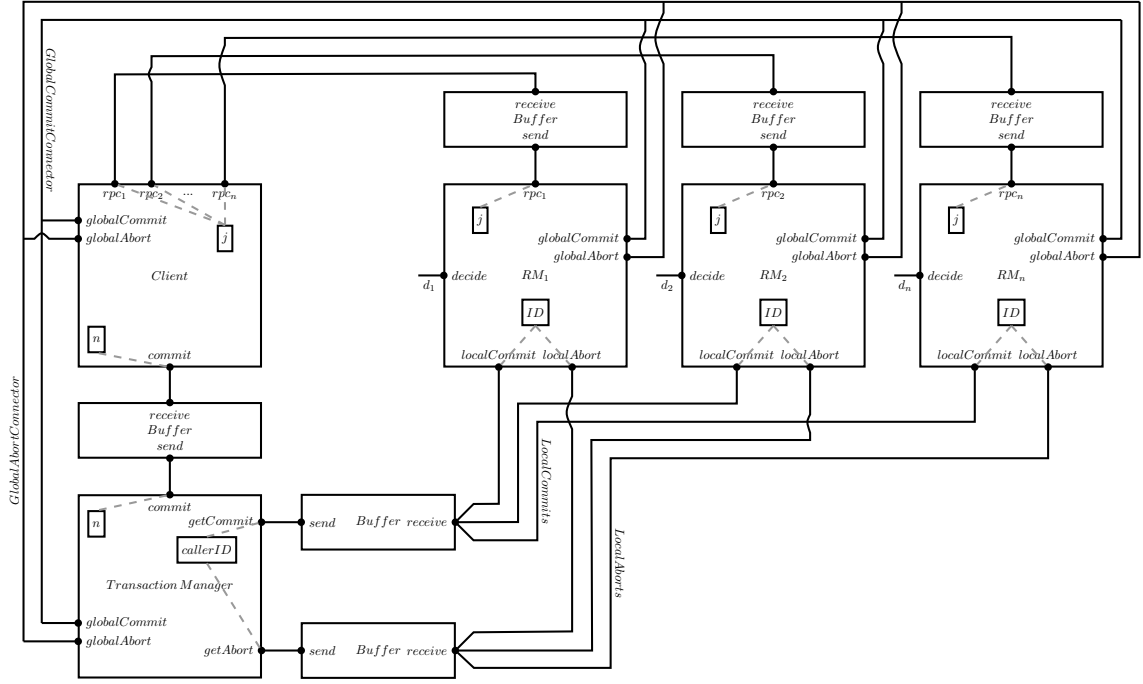


Figure 5.5: Composite Component.

of resource manager component. There are also $n + 3$ buffers, n of which are $\{Bu_1, Bu_2, \dots, Bu_n\}$ where Bu_i receives data from $c.rpc_i$, and sends it to $rm_i.rpc_i$. These buffers hold data coming from different clients (in cases where there are more than one client) and prepare it for the resource managers to fetch when they are ready to do so. Three other buffers are connected to the transaction manager, one is Bu_{n+1} which is connected to $tm.commit$, it keeps the commit requests from multiple clients, and two other buffers Bu_{n+2}, Bu_{n+3} which are connected to $tm.getCommit, tm.getAbort$ respectively, to queue the local decisions made by the rm 's and hold the data from being lost until tm is ready to process it.

5.1.2 BIP-Plus Implementation of 2PC Protocol

The PA send/receive primitives remove the need for the extra buffering components since each receive port comes equipped with a built-in system buffer. The BIP model presented above gets transformed to its equivalent BIP⁺ one by undergoing the transformations steps stated at the beginning of section 6. The resulting BIP⁺ model has the following description:

Figure 5.6 shows c^* , the client PA atomic component as it is modelled in BIP⁺. Ports $\{rpc_1, rpc_2, rpc_3, \dots, rpc_n\}$ are changed into *send* ports exporting the trans-

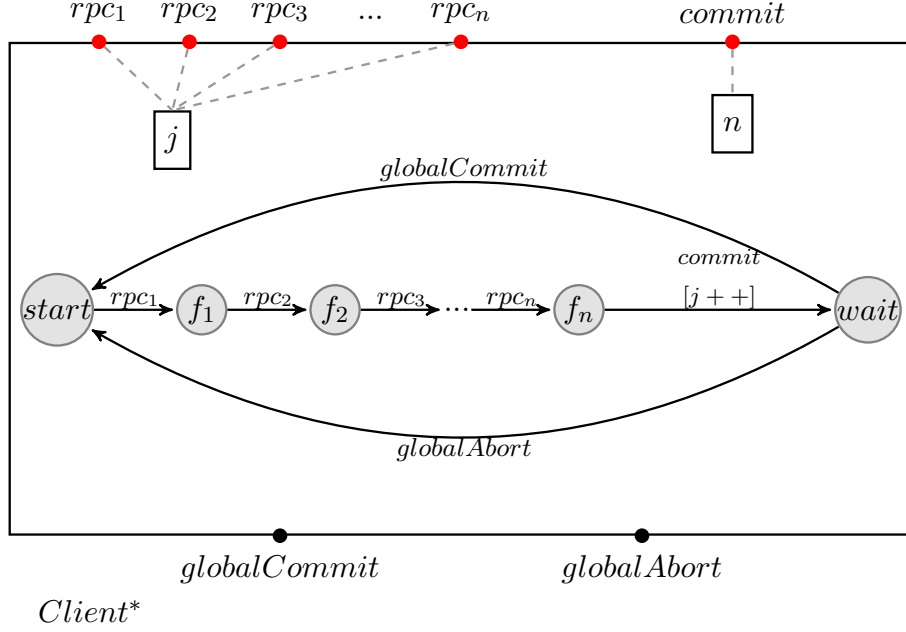


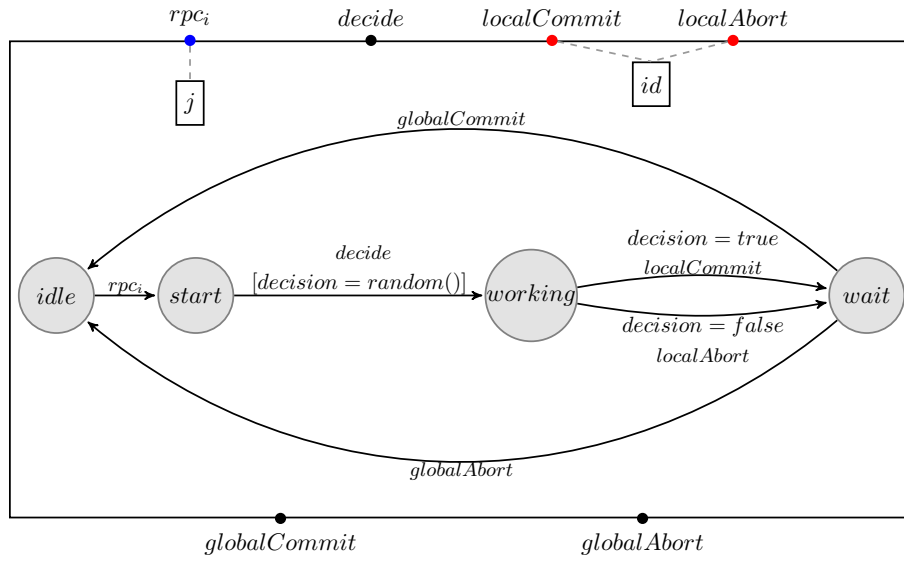
Figure 5.6: Client*.

action number j to deliver it to each corresponding rm . The *commit* port is also transformed into a *send* port with the variable n exported. All other ports are kept *ordinary*.

The Resource Manager from Figure 5.7 is transformed to BIP^+ in the same way as before. Ports *localCommit*, *localAbort* are made *send* ports, rpc_i is made a *receive* port, with *decide*, *globalCommit*, and *globalAbort* kept *ordinary*.

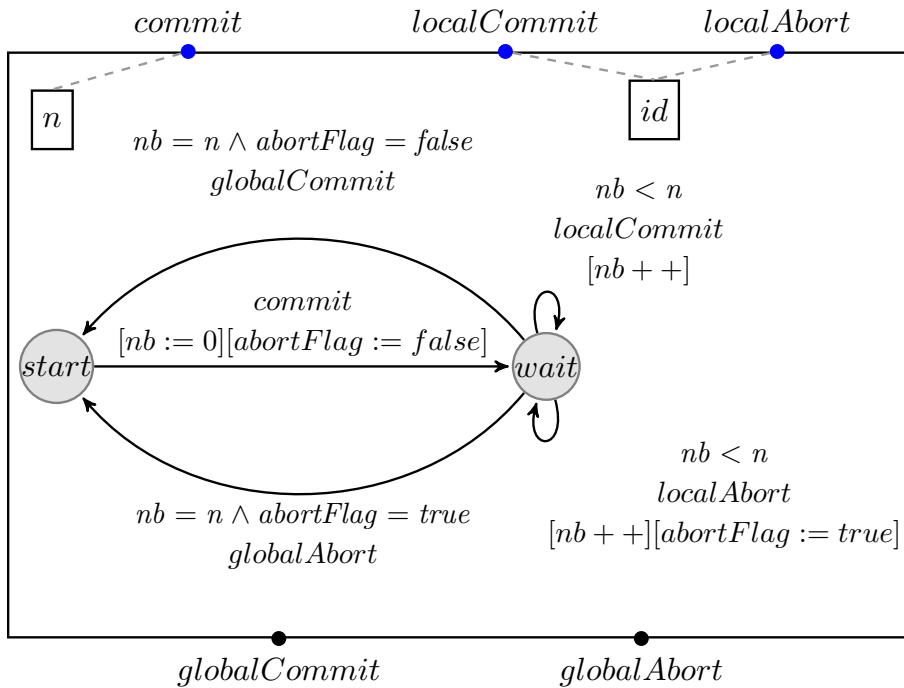
The transaction manager goes through the same transformation with $P_s = \phi$, $P_r = \{commit, localCommit, localAbort\}$, and $P_o = \{globalCommit, globalAbort\}$ to give tm^* . It is important to notice that *wait* is the source of 4 different transitions which ports are of two different types. $t(localCommit) = t(localAbort) = send$, and $t(globalCommit) = t(globalAbort) = ordinary$. This is not a violation of BIP^+ operational semantics since each set of transition type is guarded, and the two guards, governing the two sets, are mutually exclusive, which means they can never evaluate to *true* at the same time as the expression, $(nb < n) \wedge (nb = n) = false$, always holds.

The composite component, Figure 5.9, can now do without the extra buffering atomic components, and the send ports can be directly connected to the corresponding receive ports using send/receive interactions. Beside the obvious optimization gained by removing the buffers (and their propagating overhead to the top of the 3-layer send/receive model), there is an extra major performance spike gained from



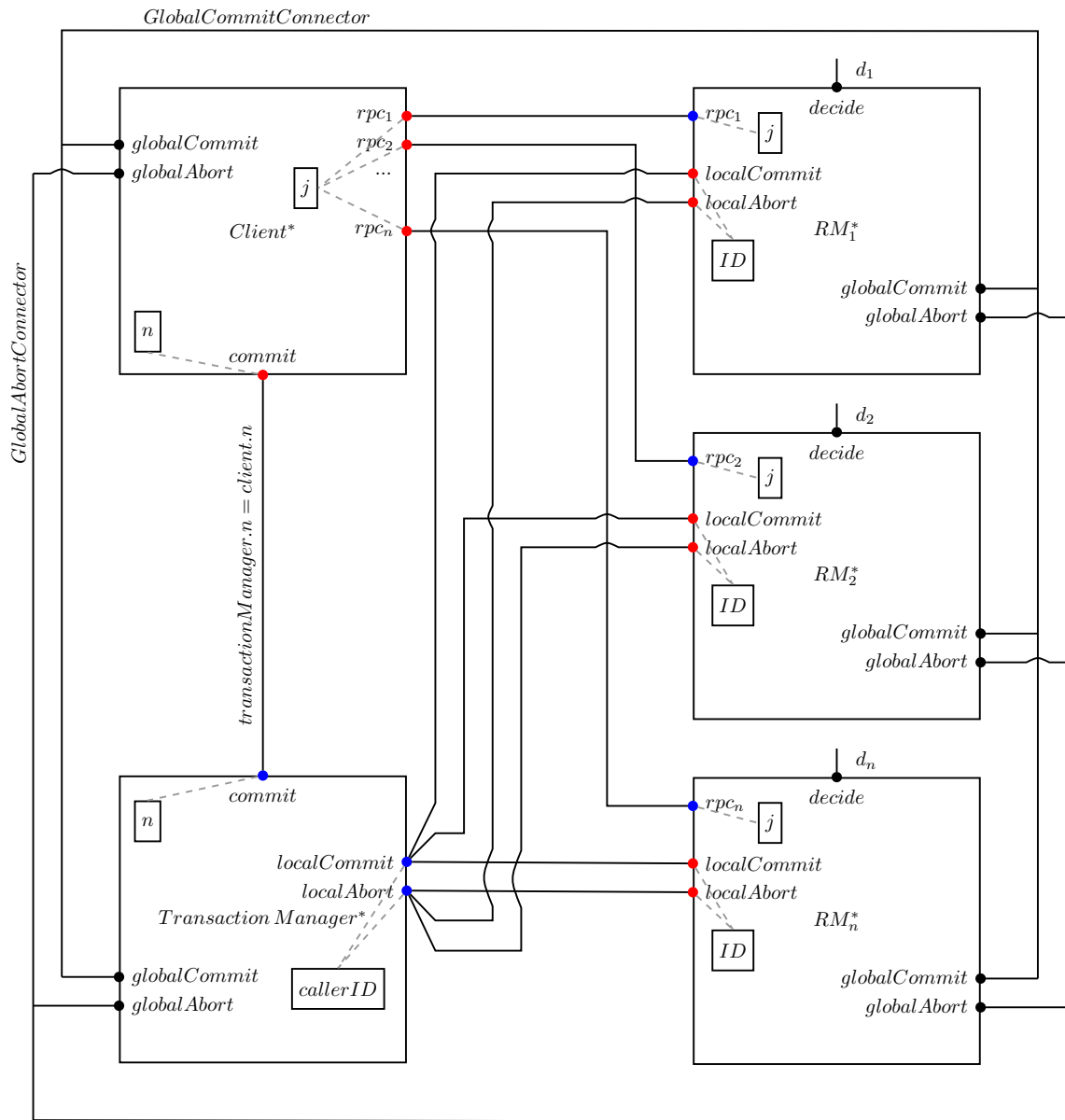
$Resource\ Manager_i^*$

Figure 5.7: $Resource\ Manager_i^*$.



$Transaction\ Manager^*$

Figure 5.8: $Transaction\ Manager^*$.



Two Phase Commit*

Figure 5.9: Composite Component*.

stating different kind of interactions. Since different types of ports have transitions that are, by construction, known to never be conflicting (either by starting from different locations or by having mutually exclusive guards), interaction protocol and reservation protocol can now focus on resolving conflicts between ordinary interactions, and the resulting BIP^{*SR} model can now benefit from direct source-to-destination conflict-free send/receive interactions.

5.1.2.1 Experimental Results

Here, we present the performance results of BIP with buffers and BIP⁺ models for the previously defined problem, Two phase commit. For all the scenarios of this benchmark, all the interactions of the model are assigned to only one component in the interaction protocol layer. Also, all the runs are done by making use of a cluster of 4 machines, each of which is an 8-core machine. For this benchmark, we have tested 4 different scenarios to show, in different cases, how the performance of both the BIP (with buffers) and BIP⁺ model is affected.

The first two scenarios are problem dependant. They are done by changing or fixing some problem parameters on only one machine to get the corresponding run time.

The first scenario is to get the processing time variation of both BIP and BIP⁺ implementations as the number of transactions change. For this case, the number of resource managers is fixed to be 10 while the number of transactions varies from 20,000 to 200,000 by a step of 20,000. The results of this scenario are presented in the graph in figure 5.10.

It is well shown in the graph that the processing time of both implementations (BIP with buffers and BIP⁺) increases as the number of transactions increases (almost linearly). However, in a BIP model with buffers, more time is needed to decide on a certain number of transactions than that in a BIP⁺ model proving that the communication in the BIP model with buffers is more expensive. Moreover, the time difference between the two implementations, as in the graph, increases as the number of transactions increases. This is because, in addition to the overhead induced by communication in BIP, the more there are transactions, the more there are client requests and decisions to be made by the resource managers and sent to the transaction managers which increases the overall use of communication. Precisely, the global decision making process will be done as many times as the number

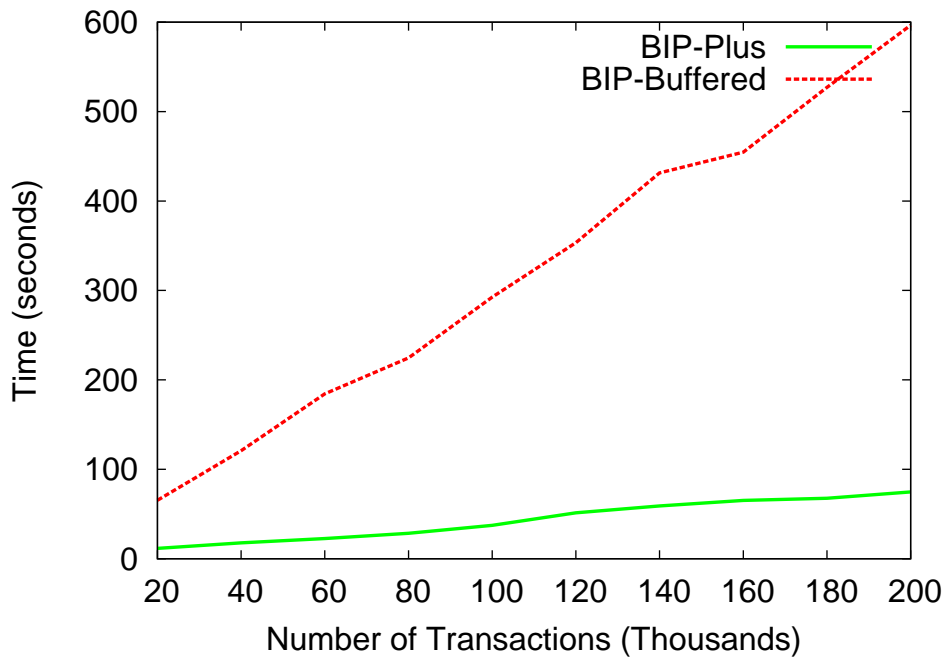


Figure 5.10: 2PC Time Performance with Respect to Number of Transactions .

of transactions which, in turns, increases overhead and time consumption. On the other hand, the increase of time, in both models, is linear because the cost of communication increased is always constant. For the same number of resource managers and for the same number of transactions added, the same tasks are done and same interactions are executed.

The second scenario is to get the processing time variation of both BIP and BIP⁺ implementations as the number of resource managers change. For this case, the number of transactions is fixed to be 10,000 while the number of resource managers change from 2 to 20 by a step of 2. The results of this scenario are presented in the graph in figure 5.11.

The results show that both run times of BIP and BIP⁺ models increase as number of RMs increases. This is because the more there are RMs, the more voters there are which adds more interactions and in turn more communication between components.

Also, these results show, for one more time, that the performance of BIP⁺ implementations is much better than that of BIP implementation and that

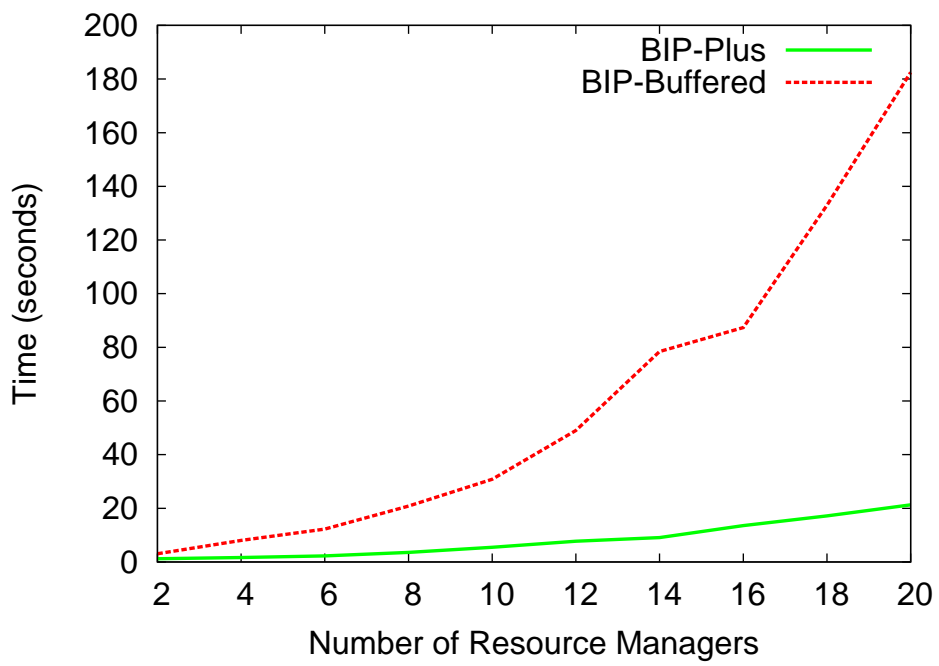


Figure 5.11: 2PC Time Performance with Respect to Number of Resource Managers

a direct send-receive communication helps getting rid of overhead and expensiveness of the buffers in BIP (more components, more interactions in the interaction protocol layer). The reason behind having a such noticeable time gap between the 2 implementations is that when new resource managers are added to the model, the corresponding buffer components to their receive ports are also added along with the interactions. This rises the load of the interaction protocol layer. However, this is not the case in the BIP⁺ model as only the RMs are added. This also cause the difference between the two curves to become much greater as the number of RMs increase.

The last two scenarios test the two implementations on different number of machines.

In the third Scenario, the run time of both models (BIP and BIP⁺) is tested on 1,2,3 and 4 machines of the cluster. For this case all the parameters of the problem are fixed such that the number of resource managers is 10 and the number of transactions is 20,000. The components of the two models are distributed in a round-robin fashion. The results of this scenario are presented in the graph of figure 5.12.

The 2PC problem is not a suitable problem to be parallelized over multiple machines because there are much more communication than computation (communication time much more expensive than computation) affecting negatively the whole run time of the problem if is executed on multiple machines. This can be noticed by the results we have got as the run time increases when components are distributed over multiple machines.

Although distribution of components over multiple machines is not convenient for this problem particularly, this scenario is still a great proof that BIP⁺ can, also, perform better than BIP with buffers on multiple machines since as seen in the graph the processing time of BIP⁺ implementations stay less than that of BIP (with buffers).

Finally, in the last scenario, we try to have a selective distribution of components among the different machines on which the implementations to be tested. By keeping the problem parameters fixed and changing the number of machines (1 to 4) on which the problem is parallelized, we do not distribute the components by only using a round-robin algorithm. In the BIP model with buffers, the client, resource managers and transaction managers are distributed using round-robin algorithm

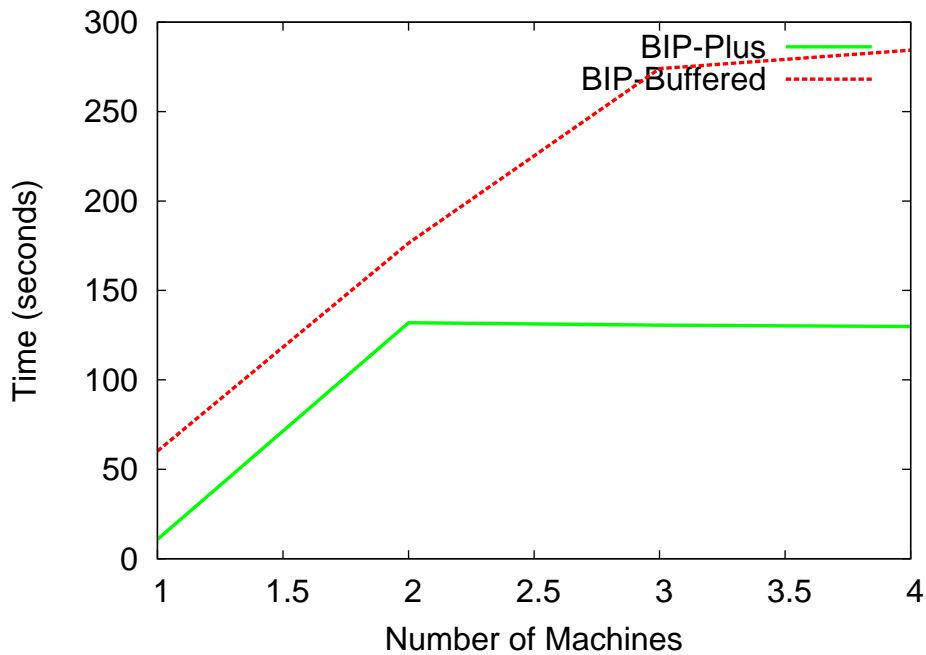


Figure 5.12: 2PC Time Performance with Respect to Number of Machines .

over machines. However, the buffer components corresponding to a particular (receive port of) component are put on the same machine on which the corresponding component is. Also, the interaction protocol is chosen to be set with the transaction manager. In addition, we tried to have equal loads on all machines, so in exceptional cases, we changed the initial round-robin distribution. As for BIP⁺ model, the components were distributed among machines in a round-robin fashion without the interaction protocol. The latter was put on the machine on which the transaction manager is.

The results of this scenario are presented in graph figure 5.13.

The main notice of these results is that there is a main advancement of performance of both (Buffered-BIP and BIP⁺) models in comparison to the results of the previous scenario. This imposes new questions about the distribution process of components among machines and how it should be. In addition to the great enhancement obtained in comparison with the third scenario, we still have BIP⁺ with a better performance than buffered-BIP. The graph in 5.14 shows clearly the difference in results of both of the third and fourth scenarios.

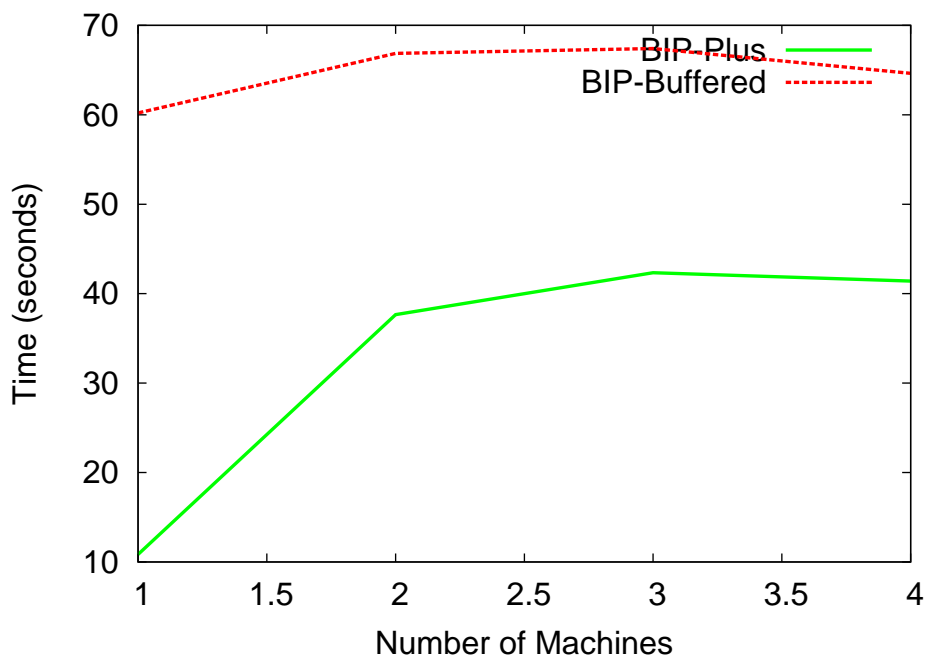


Figure 5.13: 2PC Time Performance with Respect to Number of Machines with Selective Distribution .

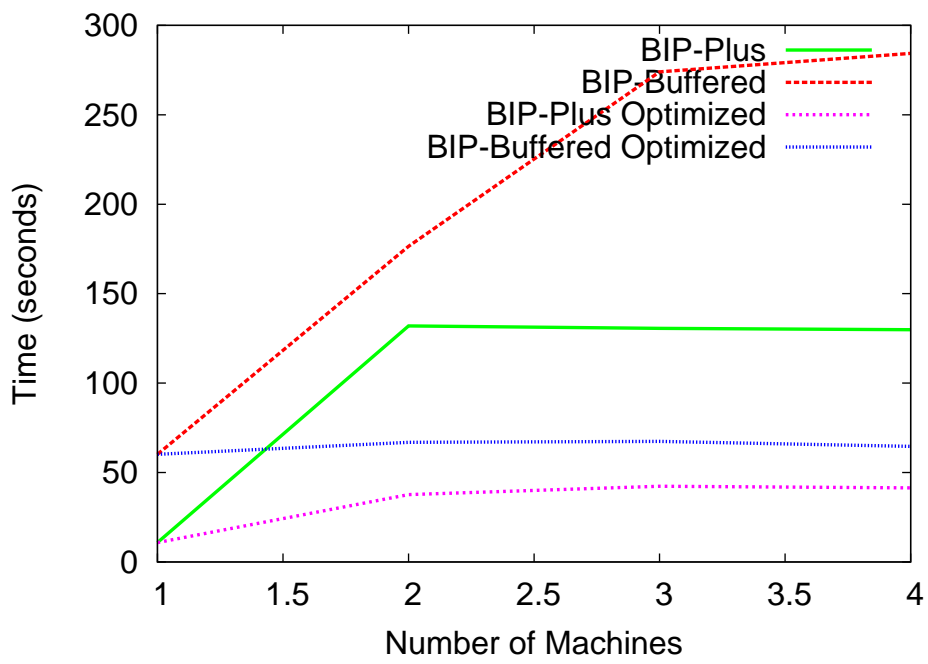


Figure 5.14: 2PC Time Performance with Respect to Number of Machines Comparison.

5.2 Support Vector Machines (SVM)

Handwritten Digit Recognition has become a classic application of machine learning with very good results in terms of accuracy. For the purpose of this paper, we implement a distributed supervised learning model that predicts the classes of handwritten digits using support vector machines (SVM) classifier.

Using different machine learning techniques, handwritten digit recognition using SVM classifier has achieved very high accuracy levels, above %99 [14], but still requires expensive computations and suffers from inefficient runtime [15], so it was proposed by [16] to distribute the model to multiple SVMs, each has a smaller local dataset to deal with, hence can perform faster, and to train these SVMs in parallel. This approach showed significant enhancement of runtime [7,8], at the cost of possible insignificant decrease in accuracy [15].

With n being the level of distribution, and starting with a training and a testing sets of images, the distributed SVM model applied here undergoes three major phases:

1. Preprocessing: The training set gets randomly divided into n disjoint and equally-sized subsets, called local training sets, ready to be distributed over n components. n preprocessing units, $\{pre_1, pre_2, \dots, pre_n\}$, work in parallel, each extracting feature vectors out of images in its local training set. The features extracted in our case are the intensity of each individual pixel.
2. Training: n units, called weak learners $\{wl_1, wl_2, \dots, wl_n\}$, collect the preprocessed feature vectors acquired from phase 1, with each wl_i receiving data only from pre_i . wl_i then compiles the vectors to perform a 1-step SVM training resulting in a local hypothesis h_i , which is a trained model ready to make predictions.
3. Testing: Finally the results of all weak learners get aggregated in one machine called strong learner sl , which constructs a final hypothesis h_f , which, in turn, handles predicting the testing set or the unlabelled data to calculate the in-sample accuracy of the final hypothesis. The problem of combining several hypothesis into one is solved by Bagging (**bootstrap aggregating**), a technique proposed by [17]. This paper implements a very well-known bagging method called majority voting. The strong learner, in possession of the testing

set, extracts feature vectors from this set, sending each vector (resembling one testing image) to each weak learner to predict its class. The strong learner collects n votes, and makes the final prediction by choosing the class with the higher votes resolving ties by random selection. This final prediction, then, gets compared against the correct labels (previously known) to calculate the accuracy of the model in the current testing set.

The advantage gained by the distribution is that as n increases, the size of local training subsets decrease, along with which decreases the computation time taken by each preprocessor and weak learner, and better speed-up results are obtained.

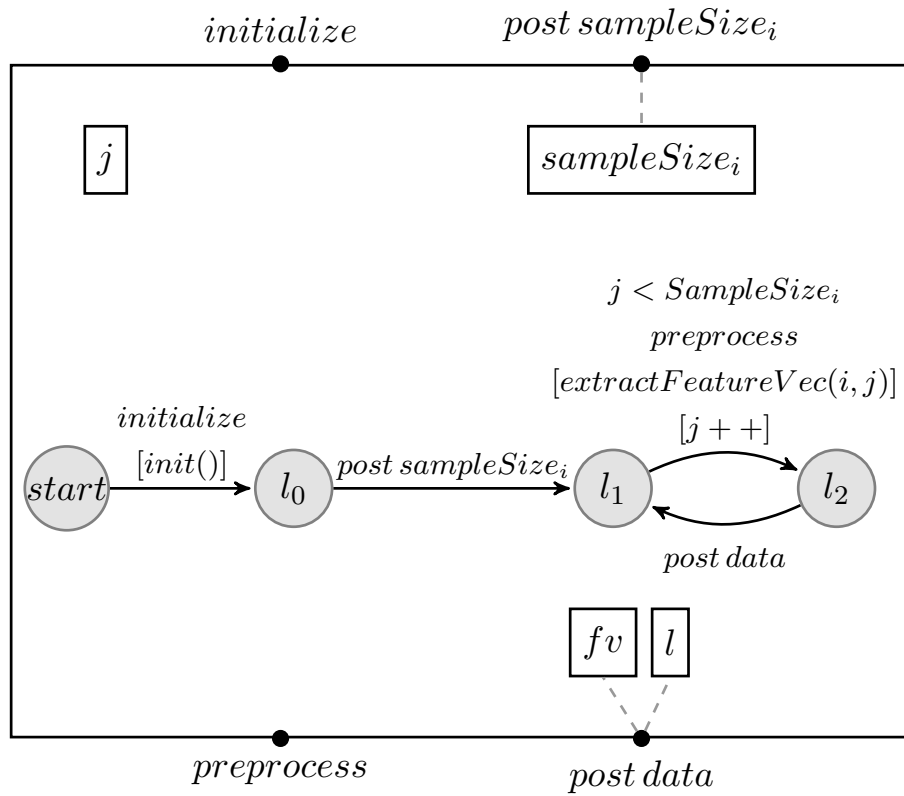
This example uses the MNIST dataset [18] to train and test the model with 60,000 training, and 10,000 testing gray-scale images. Each image has a resolution of $26 * 26$ pixels. The dataset is provided in a special format, so we transform it to jpg format using a script to highlight the role of the preprocessor, and to simulate a more convenient scenario in which other features might be extracted to achieve higher accuracy.

For the purpose of this case study, we use the C++ implementation of opencv 3.0.0 [19] for both image manipulation, and SVM training and prediction routines. A polynomial kernel is used with a polynomial function of the third degree and a termination criteria set to 800,000 iterations.

5.2.1 BIP Implementation of (SVM)

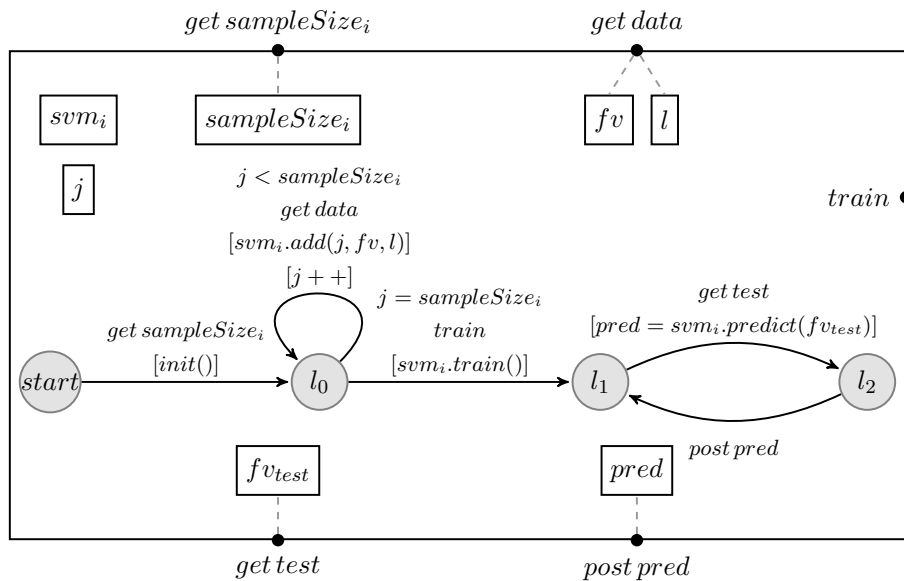
The preprocessor model, shown in Figure 5.15, starts by performing initialization steps that include determining the size of its training sample. It then synchronizes with the weak learners as it moves from location l_0 to location l_1 to post the size of the data sample. the preprocessor, then, enters a loop with as many iterations as its local sample size. As transitioning from l_1 to l_2 it processes one labelled image by loading it and extracting its feature vector. Then, as moving from l_2 back to l_1 , it sends the feature vector accompanied with the pre-known label to the weak learner via a buffer connecting the two components.

Figure 5.16 shows the *BIP* implementation of the weak learner. It starts by getting the size of the local training sample from the buffer connected to the port *get sampleSize_i*, and it performs some further initialization steps. The weak learner then enters a loop getting the labelled feature vectors from the buffer coming from



Preprocessor_i

Figure 5.15: Preprocessor.



Weak Learner_i

Figure 5.16: Weak Learner.

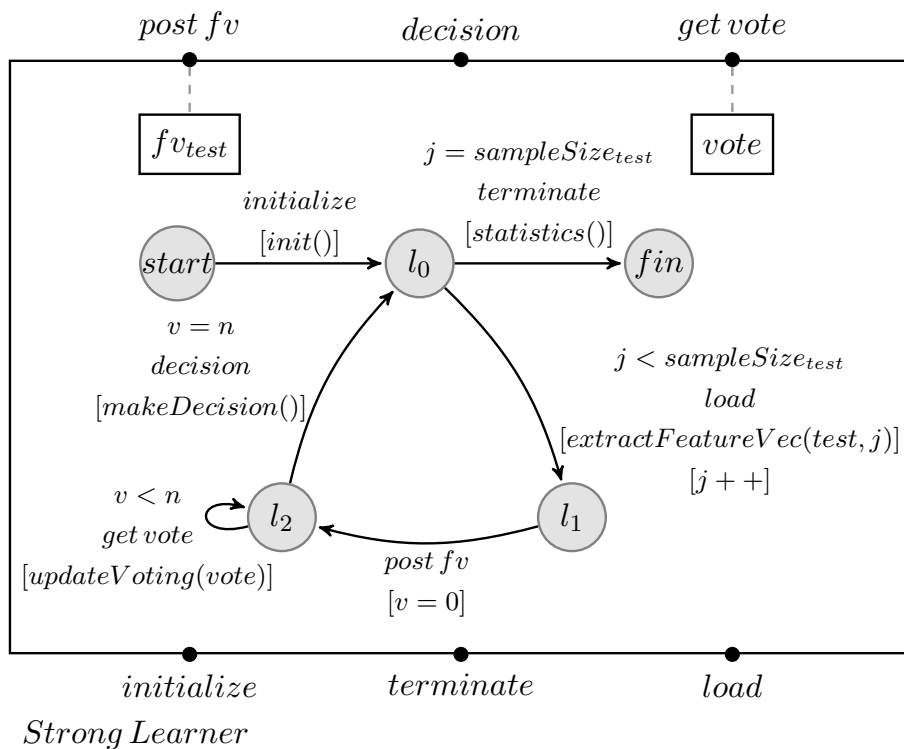


Figure 5.17: Strong Learner.

the preprocessor attached to it and adding these to the support vector machine matrix. It then trains the model to generate its local hypothesis h_i and reaches state l_1 where it synchronizes with the strong learner to start receiving the testing sample. In each iteration, as it loop between l_1 and l_2 , it receives a feature vector fv_{test} , predicts its class and sends this prediction to a buffer connected to the strong learner.

In Figure 5.17, we show the *BIP* implementation of the strong learner. After initialization, which includes determining the size of the testing set, it enters a loop where in each iteration it loads a testing image, extracts a feature vector out of it, and waits until it synchronizes with each of the weak learners components to post the feature vector as it moves from l_1 to l_2 . It stays in l_2 until it gets n votes, each of which is one weak learner's prediction of the feature vector in hand. After receiving all the votes, and as the component moves from l_2 back to l_0 it calculates the votes and makes a final decision labelling that feature vector, and resolving ties by random selection. When all the testing vectors get processed, the component exits the loop and terminates.

The composite component, shown in Figure 5.18, has n preprocessors that interacts with n weak learners in a direct strong synchronization connector to send

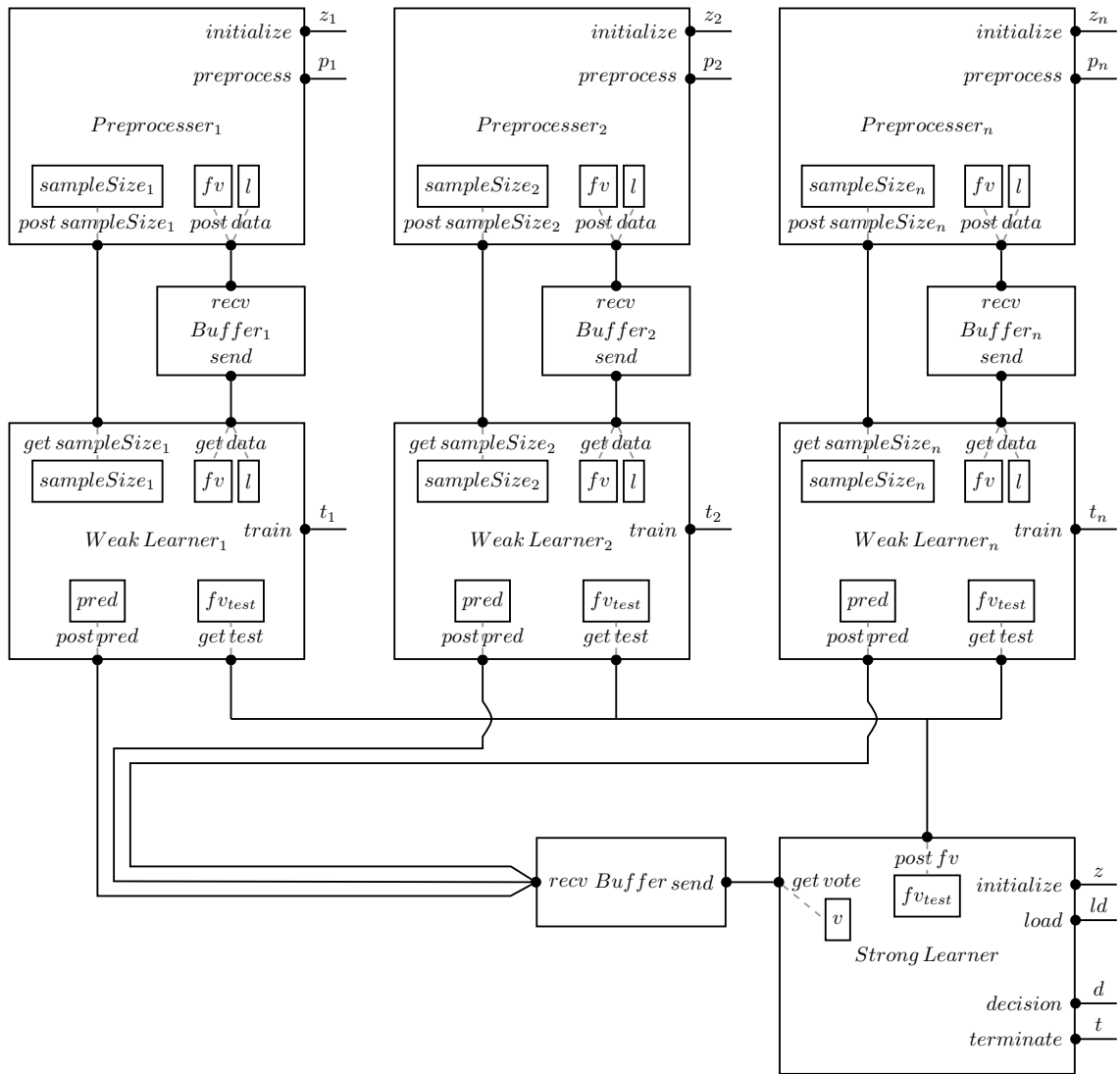


Figure 5.18: Composite Component.

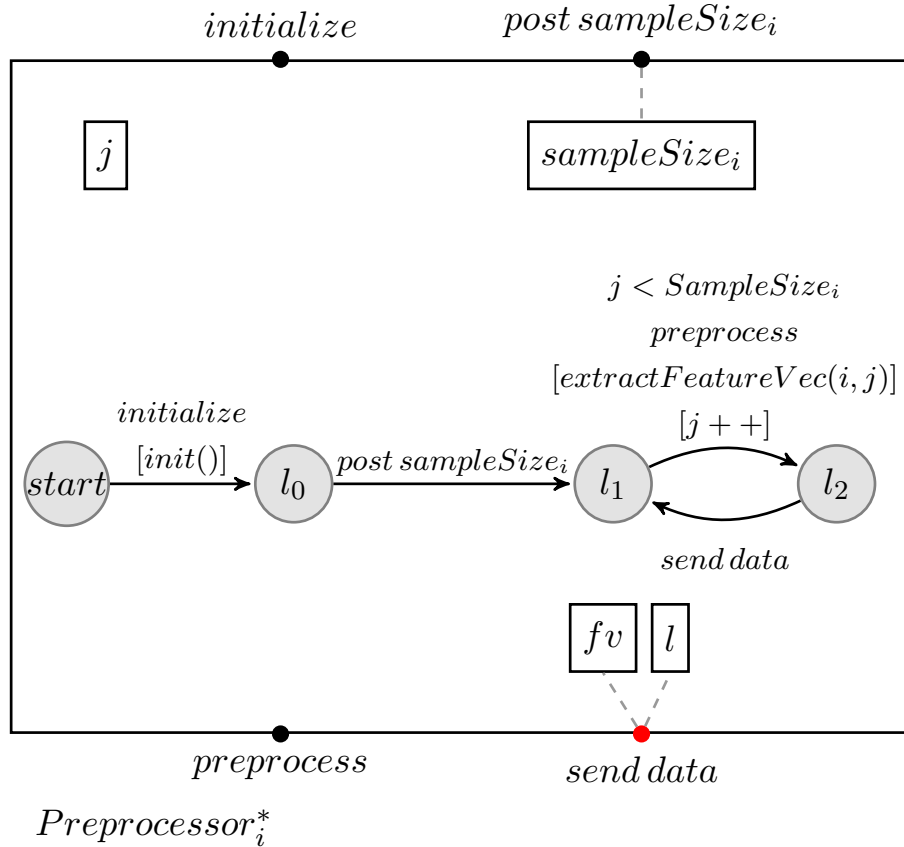


Figure 5.19: Preprocessor*.

the size of the training set, and via a buffer to send labelled feature vectors. Weak learners synchronize with the strong learner to get the feature vector to predict. And each of them, then, sends its prediction to get queued in a buffer connected to strong learner's *get vote* port, for in some cases, a plurality might be reached before receiving all the votes, in which case the strong learner would be able to make a final prediction.

5.2.2 BIP-Plus Implementation of (SVM)

As in two phase commit, BIP⁺ removes the need for explicit buffering while preserving asynchronous data communication. The following describes how to construct BIP⁺ atomic components starting with the previously described BIP components.

The preprocessor*, Figure 5.19, has $P_s = \{send\ data\}$ every other port is made ordinary. In Figure 5.20, the weak learner has $P_S = \{send\ pred\}$ and $P_r = \{recv\ data\}$, the rest are all ordinary. The strong learner, shown in Figure 5.21, has $P_r = \{recv\ vote\}$, every other port is ordinary. The ports *recv vote* and

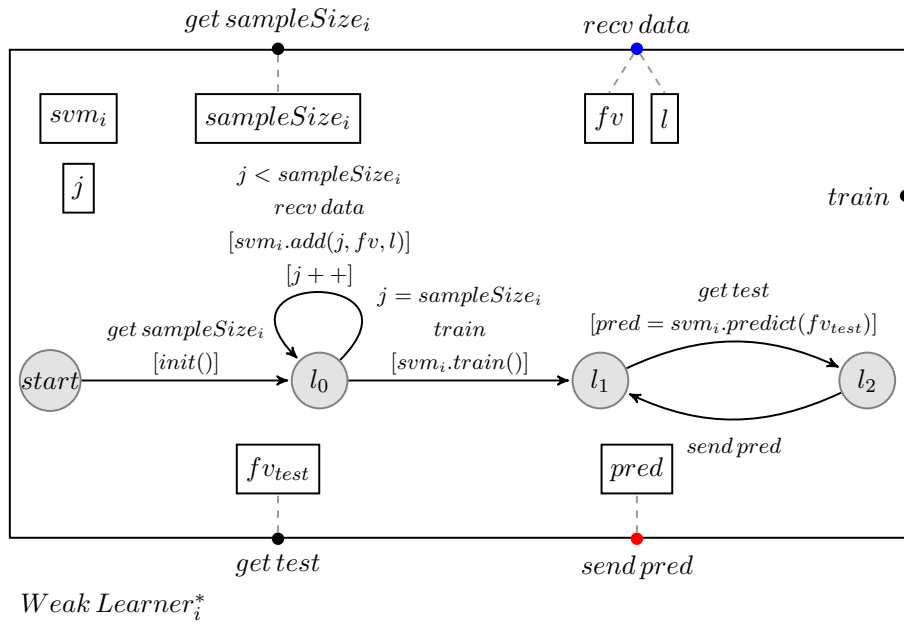


Figure 5.20: Weak Learner* .

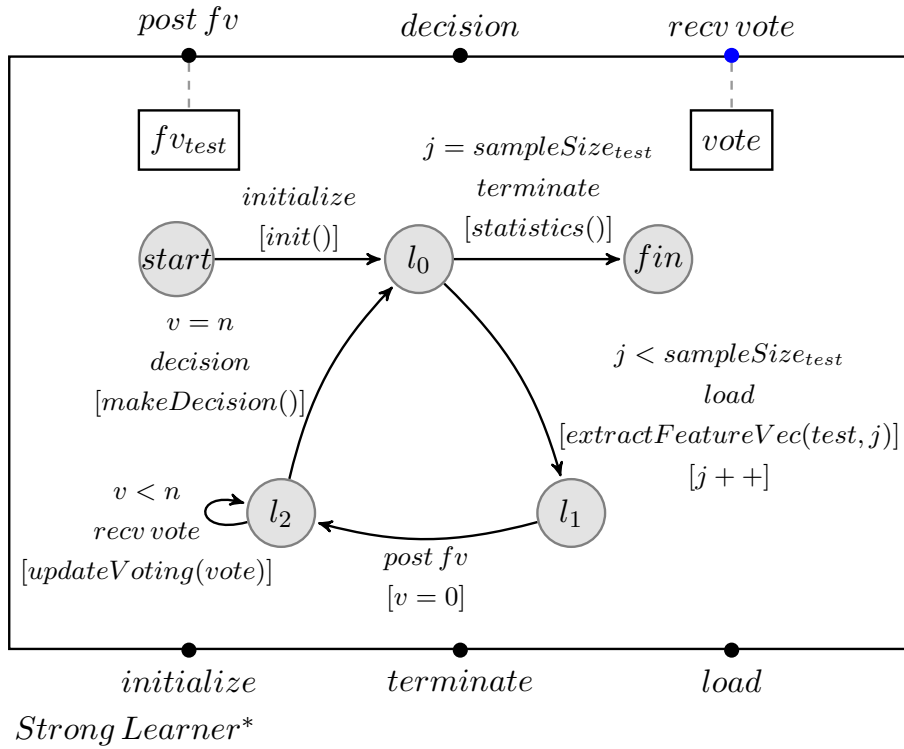


Figure 5.21: Strong Learner* .

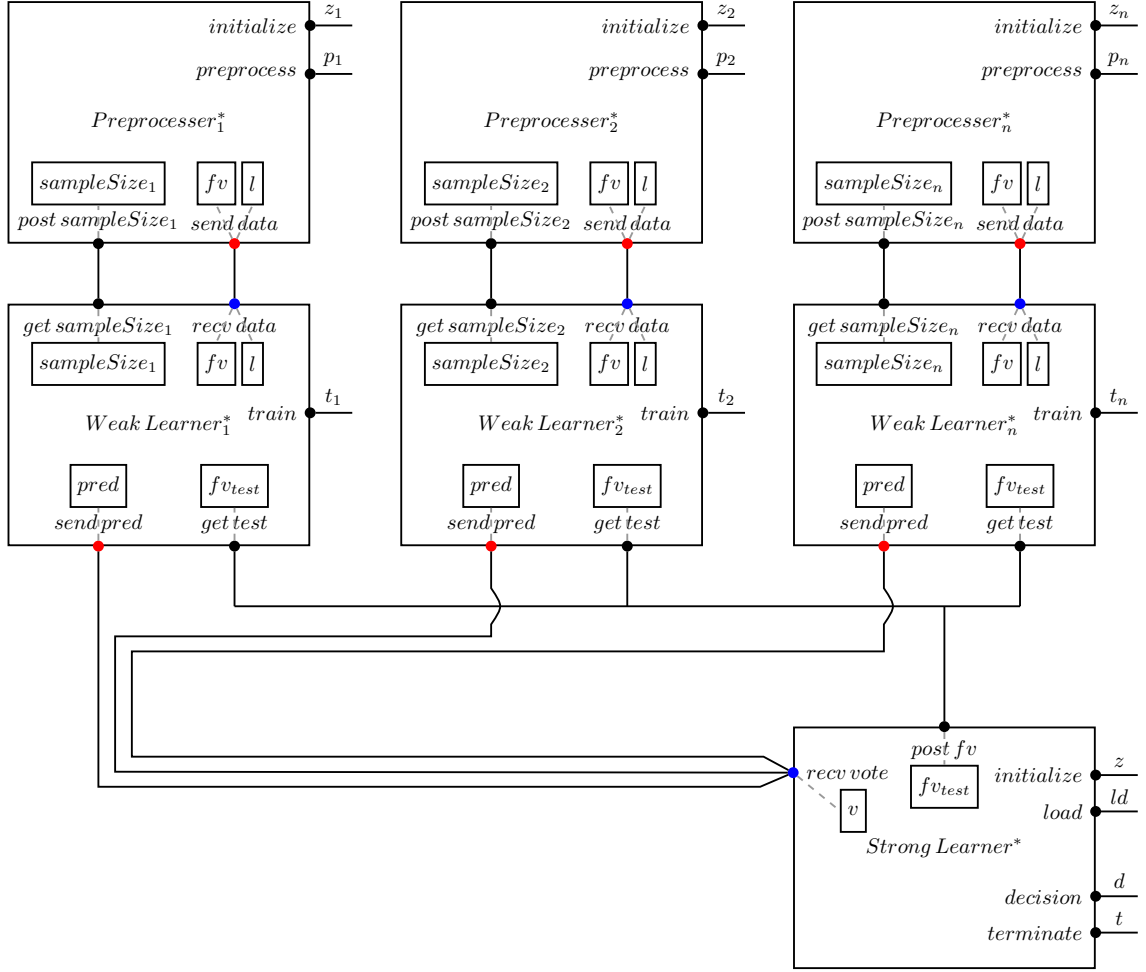


Figure 5.22: Composite Component*.

decision are of types *receive* and *ordinary* respectively, and their associating transitions are outgoing the same location (l_2). Again, this is not a violation to the operational semantics as the two guards ($v < n$) and ($v = n$) guarding the two transitions, are mutually exclusive and are never true at the same time.

As done before, to construct the composite component, and since every receive port has a built-in buffer, all buffering components get removed and send ports get directly connected to receive ports as it is illustrated in Figure 5.22.

5.2.2.1 Experimental Results

Here, we present the performance results of BIP with buffers and BIP⁺ models for the handwritten digit recognition(using SVM) problem. For all the scenarios of this benchmark, all the interactions of the model are assigned to only one component in the interaction protocol layer. Also, all the runs are done by making use of a

cluster of 4 machines, each of which is an 8-core machine.

For this benchmark, we only tested 2 scenarios to show how different the performances of both models is.

The first scenario is to get the variation of processing time of the 2 models with respect to the increase of number of preprocessors and weak learners (i.e variation of n) on a single machine. The expectation is that the processing time must decrease as n increases. One problem is that there is a bottleneck for increasing n given a particular training set/ testing set. In other words, for the particular size of training and testing sets in this problem, we can only increase n to a limit after which adding more preprocessors/weak learner will no longer improve the performance of the problem. Here, for training set = 60,000 and testing set = 10,000, the best performance we can get is for $n=6$ any value greater or less than $n=6$ is expected to give a worse performance. In order to extend this limit and be able to increase n more, we can either increase the size of data set, decrease the size of the testing set or change both. We aim to do 10 different runs to get the performance of models for 10 different n 's, so we decrease both the size of sets (training set size=6000, testing set size=50). The results of this scenario are presented in figure 5.23.

As expected, the performance of both implementations decreases as n increases. Moreover, the test results show that BIP^+ performs better than BIP model with buffers.

Although the cost of communication increases with the increase of number of pre-processor and weak learners, this does not widen the gap between the two curves. This is due to the fact that there is much more computation in the problem than communication, and, as we increase the number of components and weak learners, computation is distributed over all of them. This helps enhancing the overall performance of the problem as we add more weak learners/preprocessors because the improvement got by distributing data over components is much more the overhead got by increasing more components.

The second scenario is to get the performance of both implementations on 1,2,3 and 4 machines. For this case, the parameters for the problem (size of training set, testing set and number of preprocessors and weak learners (n)) are fixed. The

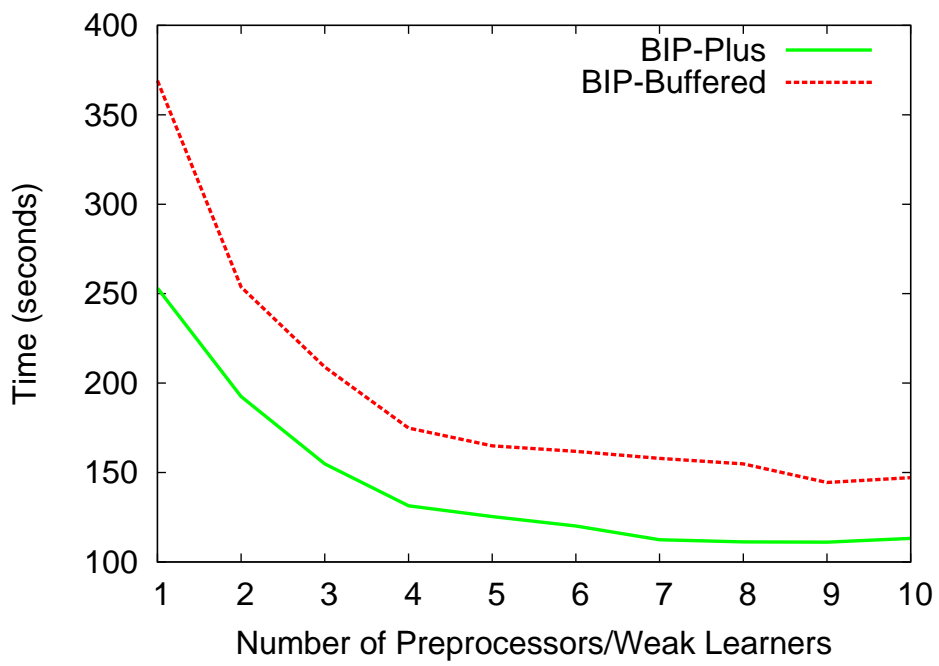


Figure 5.23: SVM Time Performance with Respect to Number of Preprocessors and Weak Learners.

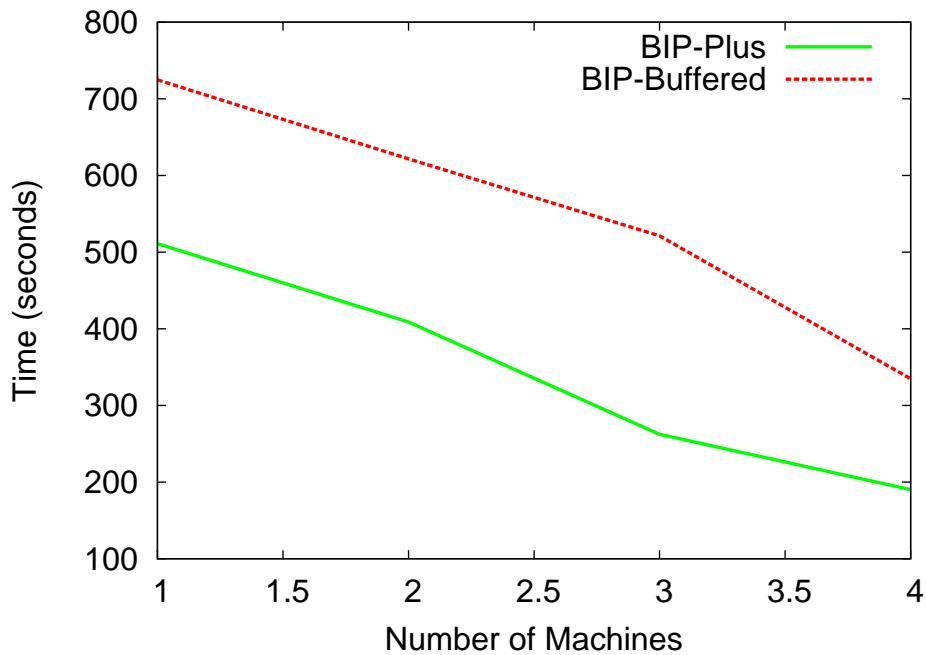


Figure 5.24: SVM Time Performance with Respect to Number of Preprocessors and Weak Learners.

training set size is set to be 60,000 and the testing set size is set to be 50, so we can set n to 30. By setting n to 30, we guarantee the need of 4 machines knowing that this problem is suitable for being parallelized over multiple machines on the cluster for it includes much more computation than communication. The results of this scenario are presented in graph in figure 5.24.

As noticed, we have achieved more performance of both implementations as their components are distributed over more machines. Nevertheless, BIP⁺ implementation still performs better than BIP.

Chapter 6

RELATED WORKS

Contents

6.1	Session Types	80
6.2	Grid Component Model (GCM)	81
6.2.1	Fractal	81
6.2.2	GCM	81
6.2.3	ProActive/GCM	82
6.3	Correct-by-construction model for asynchronously communicating systems	82
6.4	LASP	83
6.5	AzureBOT	83
6.6	FlowPools	84

Several platforms, tools, design flows and programming languages have been proposed to facilitate building efficient, scalable and reliable distributed implementations. In this chapter, we present some of these works in the scope of distributed computing.

6.1 Session Types

Session types [20, 21, 22, 23, 24] have been proposed to model interactions between distributed processes. Session type is based on the following methodology:

1. Interactions are described as a *global protocol* between processes;
2. *Local protocols* which are the projection of global protocol are automatically synthesized according to processes;
3. A team of programmers develop the code of processes;
4. Processes are statically type-checked with respect to local protocols.

The design methodology of session type has major drawbacks. First, there is a gap between design and implementation. Second, the design flow includes redundancy (global protocol, local protocol, process implementation). Third, there is no clear separation between communication and computation in local processes.

6.2 Grid Component Model (GCM)

6.2.1 *Fractal*

Fractal [25, 26, 27] is a component-based model, developed at France Telecom R&D and INRIA France. Fractal is a general component-based model for implementing, deploying and managing complex software systems. It is composed of:

- software components, such that a component consists of a membrane which can be composed of many components (sub-components)
- interfaces (similar to ports in other component models)
- explicit connections (bindings) between them

.

6.2.2 *GCM*

In a moderately different application domains from BIP, based on the Fractal component model, GCM [28] has been proposed in the CoreGrid Network of Excellence. It tackles large-scale distributed grid computing and inherits from fractal:

- its hierarchical structure
- the enforcement of separation between functional and non-functional concerns
- its extensibility, and the separation between interfaces and implementation

Additionally, it includes some further extensions which are:

- supporting collective communications
- supporting autonomic aspects and a stricter separation between functional and non-functional concerns

6.2.3 *ProActive/GCM*

The GCM component model has been implemented as the ProActive/GCM during the GridComp European project. ProActive/GCM is an extension of proActive. ProActive (Proactive Parallel Suite) [29] is a Java library (Source code under AGPL license) for parallel, distributed, and concurrent computing, also featuring mobility and security in a uniform framework. It provides an API to facilitate the development of applications distributed over a LAN, Clusters, grids and Cloud stations. It is based on the notion of active objects.

6.3 **Correct-by-construction model for asynchronously communicating systems**

This approach [30] uses the labelled transition systems as a model for choreography or conversation protocol (CP) and its corresponding distributed systems. It proposes, utilizing proof-based methods such as the B method, a correct-by-construction model in order to develop asynchronous distributed applications in which choreographies are realised. The asynchronous communications among the peers or the distributed systems is achieved by making use of FIFO buffers which is similar to our proposal for the BIP⁺ framework. The distributed systems are obtained by a refinement-based approach, consisting of two refinement steps:

1. synchronous model projection, i.e., projection from the CP model to a synchronous distributed system in which CP is realised
2. asynchronous model projection i.e., projection the synchronous composition to its asynchronous version where FIFO queues are augmented at every peer in the system.

It is an approach to be applied to construct many real-world scalable, efficient and reliable distributed applications, where communication is asynchronous without buffer size restrictions, e.g: service choreographies, singularity, channels contracts and Erlang contracts. Yet, it still does not support collective primitives such as a multicasting which is mentioned to be part of their future work.

6.4 LASP

LASP [31] is a programming model that is designed to facilitate and simplify correct, large-scale distributed computing utilizing synchronization as little as possible. It combines ideas from deterministic dataflow programming and conflict free replicated data types (CRDT) in order to provide powerful primitives for composing CRDTs into larger computations that observe the Strong Eventual Consistency (SEC). This makes LASP support distributed applications of non-monotonic behavior in a monotonic framework by defining data structures and some operations to be performed on them. LASP's only motivation is to be used for large-scale computation over replicated data, and it is still not a general purpose language. Furthermore, LASP is implemented as an Erlang Library, on the Top of Riak Core distributed systems framework. Erlang [32](the library on which LASP is built) is considered as a general purpose programming language that is mainly utilized in large telecommunications systems and internet applications and not in problems in which performance is a main requirement. In contrast, MPI (as generated from our BIP⁺ and BIP models) is well suited for hard problems that require high-performance [33].

6.5 AzureBOT

AzureBot [34] is a simple and user-friendly framework that is designed to accelerate and facilitate building scalable bag-of-tasks distributed applications only. Additionally, this framework only supports cloud platforms, mainly the Azure cloud platform. It is based on an object-oriented principle and consists of four classes:(1) the Task class that holds input data and the work to be done, (2) the Result class that preserves the output obtained by the system, (3) the BagOfTasks class that holds all the tasks of the system, and finally (4) the TaskExec class that communicates with the elements executing in the system to distribute available work and obtain the resulting output. However, it is mentioned that synchronization cannot be expressed explicitly when required in this framework which is considered as a limitation to be tackled. Moreover, it supports only applications written in C# and hosted in the Azure cloud platform.

6.6 FlowPools

FlowPools is a fundamental dataflow collections abstraction which is backed up by deterministic, lock free and composable data structures, implemented for Scala language. They are utilized to build large and complex deterministic parallel dataflow programs. However, FlowPools does not support building distributed systems. On the contrary, it is designed for multi-threaded programs only.

Chapter 7

CONCLUSION AND FUTURE WORKS

Contents

7.1 Conclusion	85
7.2 Future Works	86

7.1 Conclusion

Developing reliable, efficient and scalable distributed applications is considered as a hard task for it is hardly predictive, error-prone and time consuming although many high-level and low-level frameworks exist for this development process. Moreover, the need of such systems is rapidly rising nowadays. For these reasons, in this thesis, we try to propose a model that is capable of addressing the issues to construct such applications i.e., to simplify and facilitate developing correct, efficient and scalable systems.

High-level models are more desirable to use for building distributed applications for they abstract away the implementation details in comparison to the low-level models. As a result, we are driven to choose the BIP framework, which produces correct-by construction, scalable and efficient distributed implementations from a high-level model, as the focus of this thesis. Unfortunately, abstraction of communication primitives, in a BIP model, reduces its expressiveness, and, in addition, induces overhead in some implementations (e.g., asynchronous data transfer between processes) weakening their performance.

Accordingly, we have proposed an extension of the BIP framework, BIP^+ , that combines both of the high-level primitives and low-level send-receive primitives which intensifies the expressiveness of BIP. In a BIP^+ model, the asynchronous send and receive primitives are expressed by direct send and direct receive ports and a complete send-receive communication is possible through a DSR interaction which is not an ordinary multiparty interaction.

Inclusion of these low-level primitives in the high-level model has made it possible to guide the code generation to use the underlying platform (e.g., using system buffers). This helps getting rid of the overhead that can be caused, in an equivalent

BIP model, due to the implementation details (e.g., buffer components) that need to be added to mimic the behavior of an asynchronous send-receive communication.

Moreover, we have proposed our 3-layer architecture to derive distributed implementations directly from a particular BIP⁺ model and proved that our proposed 3-layer model is correct. In other words, it is guaranteed that the code generation, from a BIP⁺ model, produces a correct implementation, which is semantically equivalent to the original model.

Hopefully, we have succeeded to satisfy our concerns, i.e., achieve more expressiveness and efficiency in the BIP⁺ model than that of its equivalent BIP model. We have shown that a BIP⁺ model outperforms its equivalent BIP model.

To sum up, the proposed extension makes the BIP communication model more expressive while preserving the possibility to utilize abstract and high-level primitives. In addition, the obtained model still supports the development of correct-by-construction and scalable distributed systems. Furthermore, since the code generation, from the proposed model, uses the underlying platform, more efficient implementations for distributed applications, that make use of asynchronous communications, have been obtained.

7.2 Future Works

Our future work comprises the following:

- Transforming session types to BIP⁺. This is targeted by defining a novel design methodology for developing correct distributed softwares, given a global session type we automatically:
 1. generate its equivalent BIP⁺ model that mimics the same interaction structure;
 2. inject computations to local components;
 3. automatically generate efficient distributed implementations.
- Combining collective low-level primitives with the BIP⁺ framework. For instance, we tend to extend the BIP⁺ framework with the direct broadcast/multi-cast primitives. Although, in BIP⁺, we can express a broadcast or multi-cast by having a DSR interaction between one direct send and multiple receives, a direct broadcast/multi-cast can be more efficient and is expected to perform

better. This is due to the fact that a DSR interaction of one send and multiple receives is, finally, generated as a sender sending messages asynchronously and sequentially, one after the other, to all of the receivers participating in this interaction. On the other hand, utilizing a broadcast primitive, instead, optimizes the communication patterns as the implementations of these primitives, for example, can make use of a tree-based algorithm, such as splitted-binary tree algorithm, to minimize the messages traffic among processes and attain a better performance [35].

- Acquiring C++ code generations with TCP sockets for distributed implementations of BIP⁺models. BIP distributed implementations are generated to C++ code using either TCP sockets or MPI implementing the send-receive primitives between components. Yet, for now, our BIP⁺tool only supports automatic generation of C++ code employing MPI. Similar to MPI, in which the asynchronous communication primitives make use of the system buffer to preserve data sent until it is received by the receiving party, TCP buffers the received messages until the receiving application is ready to consume this data. This means that our BIP⁺model, that is a combination of both multi-party and asynchronous send-receive interactions (using buffering), after being transformed to its 3-layer SR BIP⁺model, can also generate a C++ code employing TCP sockets with the least overhead possible.

References

- [1] A. Basu, S. Bensalem, M. Bozga, J. Combaz, M. Jaber, T.-H. Nguyen, and J. Sifakis, “Rigorous component-based system design using the bip framework,” *IEEE Software*, vol. 28, no. 3, pp. 41–48, 2011.
- [2] M. Nouredine, M. Jaber, S. Bliudze, and F. A. Zaraket, “Reduction and Abstraction Techniques for BIP,” in *Proceedings of the 11th International Symposium on Formal Aspects of Component Software*, ser. Lecture Notes in Computer Science, I. Lanese and E. Madelaine, Eds., vol. 8997. Springer, 2014, pp. 288–305.
- [3] Y. Falcone, M. Jaber, T. Nguyen, M. Bozga, and S. Bensalem, “Runtime verification of component-based systems in the BIP framework with formally-proved sound and complete instrumentation,” *Software and System Modeling*, vol. 14, no. 1, pp. 173–199, 2015. <http://dx.doi.org/10.1007/s10270-013-0323-y>
- [4] B. Bonakdarpour, M. Bozga, M. Jaber, J. Quilbeuf, and J. Sifakis, “A framework for automated distributed implementation of component-based models,” *Distributed Computing*, vol. 25, no. 5, pp. 383–409, 2012.
- [5] —, “Automated conflict-free distributed implementation of component-based models,” in *IEEE Fifth International Symposium on Industrial Embedded Systems - SIES 2010, University of Trento, Italy, July 7-9, 2010*. IEEE, 2010, pp. 108–117.
- [6] —, “A framework for automated distributed implementation of component-based models,” *Distributed Computing*, vol. 25, no. 5, pp. 383–409, 2012. <http://dx.doi.org/10.1007/s00446-012-0168-6>
- [7] Verimag. (2015) Bip tools. <http://www-verimag.imag.fr/BIP-Tools,93.html>
- [8] T. E. Foundation. (2015) Eclipse modeling project. Accessed on August 13, 2015. <https://eclipse.org/modeling/emf/>
- [9] S. Bensalem, A. Griesmayer, A. Legay, T.-H. Nguyen, J. Sifakis, and R. Yan, “D-finder 2: Towards efficient correctness of incremental design,” in *NASA*

- Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings*, ser. Lecture Notes in Computer Science, vol. 6617. Springer, 2011, pp. 453–458.
- [10] W. Gropp, E. Lusk, and R. Thakur, *Using MPI-2: Advanced Features of the Message-Passing Interface*. Cambridge, MA: MIT Press, 1999.
- [11] J. Gray and L. Lamport, “Consensus on transaction commit,” *ACM Trans. Database Syst.*, vol. 31, no. 1, pp. 133–160, Mar. 2006. <http://doi.acm.org/10.1145/1132863.1132867>
- [12] A. S. Tanenbaum and M. V. Steen, *Fault Tolerance*, 2007, pp. 355–372.
- [13] J. N. Gray, *Notes on data base operating systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1978, pp. 393–481. http://dx.doi.org/10.1007/3-540-08755-9_9
- [14] C.-L. Liu, K. Nakashima, H. Sako, and H. Fujisawa, “Handwritten digit recognition: benchmarking of state-of-the-art techniques,” *Pattern Recognition*, vol. 36, no. 10, p. 22712285, 2003.
- [15] O. Meyer, B. Bischl, and C. Weihs, *Support Vector Machines on Large Data Sets: Simple Parallel Approaches*. Cham: Springer International Publishing, 2014, pp. 87–95. http://dx.doi.org/10.1007/978-3-319-01595-8_10
- [16] R. Collobert, S. Bengio, and Y. Bengio, “A parallel mixture of svms for very large scale problems.” *Neural Computation*, vol. 14, no. 5, pp. 1105–1114, 2002. <http://dblp.uni-trier.de/db/journals/neco/neco14.html#CollobertBB02>
- [17] L. Breiman, “Bagging predictors,” *Mach. Learn.*, vol. 24, no. 2, pp. 123–140, Aug. 1996. <http://dx.doi.org/10.1023/A:1018054314350>
- [18] “The mnist database.” <http://yann.lecun.com/exdb/mnist/>
- [19] “Opencv — opencv.” <http://opencv.org/>
- [20] A. Bejleri and N. Yoshida, “Synchronous multiparty session types,” *Electr. Notes Theor. Comput. Sci.*, vol. 241, pp. 3–33, 2009. <http://dx.doi.org/10.1016/j.entcs.2009.06.002>

- [21] K. Honda, N. Yoshida, and M. Carbone, “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*, 2008, pp. 273–284. <http://doi.acm.org/10.1145/1328438.1328472>
- [22] E. Bonelli and A. B. Compagnoni, “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*, 2007, pp. 240–256. http://dx.doi.org/10.1007/978-3-540-78663-4_17
- [23] A. Vallecillo, V. T. Vasconcelos, and A. Ravara, “Typing the behavior of software components using session types,” *Fundam. Inform.*, vol. 73, no. 4, pp. 583–598, 2006. <http://iospress.metapress.com/content/82bf1qafeel5g8n4/>
- [24] S. J. Gay, V. T. Vasconcelos, A. Ravara, N. Gesbert, and A. Z. Caldeira, “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*, 2010, pp. 299–312. <http://doi.acm.org/10.1145/1706299.1706335>
- [25] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani, “The fractal component model and its support in java: Experiences with auto-adaptive and reconfigurable systems,” *Softw. Pract. Exper.*, vol. 36, no. 11-12, pp. 1257–1284, Sep. 2006. <http://dx.doi.org/10.1002/spe.v36:11/12>
- [26] G. S. Blair, T. Coupaye, and J. Stefani, “Component-based architecture: the fractal initiative,” *Annales des Télécommunications*, vol. 64, no. 1-2, pp. 1–4, 2009. <http://dx.doi.org/10.1007/s12243-009-0086-1>
- [27] A. Cansado, D. Caromel, L. Henrio, E. Madelaine, M. Rivera, and E. Salageanu, *The Common Component Modeling Example: Comparing Software Component Models*, ser. Lecture Notes in Computer Science. Springer, 2008, vol. 5153, ch. A Specification Language for Distributed Components implemented in GCM/ProActive, <http://agrausch.informatik.uni-kl.de/CoCoME>.

- [28] L. Henrio and E. Madelaine, “Behavioural Verification of Distributed Components,” in *ICE 2013*, Florence, Italy, Jun. 2013. <https://hal.inria.fr/hal-00850025>
- [29] Inria. Proactive. <https://team.inria.fr/scale/software/proactive/>
- [30] Z. Farah, Y. Ait-Ameur, M. Ouederni, and K. Tari, “A correct-by-construction model for asynchronously communicating systems,” *International Journal on Software Tools for Technology Transfer Int J Softw Tools Technol Transfer*, 2016.
- [31] C. Meiklejohn and P. Van Roy, “Lasp: A language for distributed, coordination-free programming,” in *Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming*, ser. PPDP ’15. New York, NY, USA: ACM, 2015, pp. 184–195. <http://doi.acm.org/10.1145/2790449.2790525>
- [32] Erlang libraries. <http://erlang.org/faq/libraries.html>
- [33] Open mpi: Open source high performance computing. <https://www.open-mpi.org/>
- [34] D. Agarwal and S. K. Prasad, “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.
- [35] T. Angskun, G. Bosilca, G. E. Fagg, E. Gabriel, and J. J. Dongarra, “Performance analysis of mpi collective operations,” in *In: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS05) - Workshop 15*, 2005.

