AMERICAN UNIVERSITY OF BEIRUT

FROM GLOBAL CHOREOGRAPHY TO EFFICIENT DISTRIBUTED IMPLEMENTATION

by
RAYAN ALI HALLAL

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
January 2019

AMERICAN UNIVERSITY OF BEIRUT

FROM GLOBAL CHOREOGRAPHY TO EFFICIENT DISTRIBUTED IMPLEMENTATION

by

RAYAN ALI HALLAL

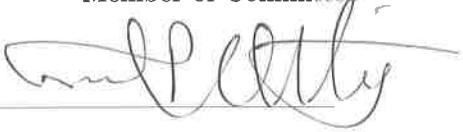
Approved by:

Dr. Mohamad Jaber, Assistant Professor — Advisor

Computer Science

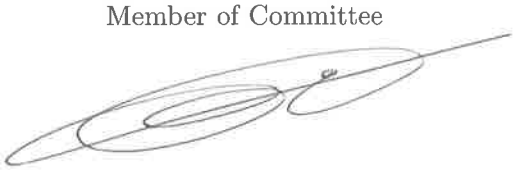
Dr. Paul Attie, Professor — Member of Committee

Computer Science



Dr. Mohamad Nassar, Assistant Professor — Member of Committee

Computer Science



Date of thesis defense: January 3, 2019

AMERICAN UNIVERSITY OF BEIRUT

THESIS, DISSERTATION, PROJECT RELEASE FORM

Student Name: Hallal (Last) Rayan (First) Ali (Middle)

● Master's Thesis ○ Master's Project ○ Doctoral Dissertation

☐ I authorize the American University of Beirut to: (a) reproduce hard or electronic copies of my thesis, dissertation, or project; (b) include such copies in the archives and digital repositories of the University; and (c) make freely available such copies to third parties for research or educational purposes.

☑ I authorize the American University of Beirut, to: (a) reproduce hard or electronic copies of it; (b) include such copies in the archives and digital repositories of the University; and (c) make freely available such copies to third parties for research or educational purposes after: **One ___ year from the date of submission of my thesis, dissertation or project.**
Two ___ years from the date of submission of my thesis , dissertation or project.
Three ✓ years from the date of submission of my thesis , dissertation or project.

Rayan (Signature) February 1, 2019 (Date)

This form is signed when submitting the thesis, dissertation, or project to the University Libraries

Acknowledgements

I will forever be indebted to my advisor Professor Mohamad Jaber for his continuous support and mentorship, not just during this semester but throughout most of my academic career at the department. I cannot possibly hope to give him the credit that he deserves in these few pages, but I can confidently say that he has taught me by his example what a good scientist (and a person) should be. Needless to say, this work would not have been possible without his assistance and guidance and I am extremely grateful for that.

I would like to extend my gratitude to every one of my classmates for the stimulating discussions we had that spurred me on to accomplish this work. It was a pleasure getting to know them and have the chance to work alongside such gifted minds.

I would also like to thank Murex S.A.L for their sponsorship of this work and for giving me the opportunity to discover new and exciting fields of research through my discussions with them. Special thanks to Rasha Abdallah, whose collaboration was instrumental for this project.

Nobody has been more important to me in the pursuit of this project than the members of my family. I would like to thank my parents, whose love and guidance are with me in whatever I pursue. Their support is the platform upon which I have been able to complete this thesis.

An Abstract of the Thesis of

Rayan Ali Hallal for Master of Science

Major: Computer Science

Title: From Global Choreography to Efficient Distributed Implementation

We introduce a methodology to automatically synthesize efficient distributed implementation starting from high-level global choreography. A global choreography describes the communication logic between the interfaces of a set of predefined processes. The operations provided by the choreography (e.g., multiparty, choice, loop, branching) are master-triggered and conflict-free by construction (no conflict parallel interleaving), which permits the generation of fully distributed implementations (i.e., no need for controllers). The synthesized implementation of the distributed system does not need controllers to synchronize and behaves as described by the choreography. This, in particular, ensures the efficiency of the implementation and reduces the communication needed at runtime. Moreover, we define a translation of the distributed implementations to equivalent Promela versions. The translation allows to verify the distributed system against behavioral properties. We apply our methodology to automatically synthesize micro-services architectures. We illustrate our method on the automatic synthesis of a verified

distributed buying system.

Contents

List of Figures

Chapter 1

Introduction

Developing correct distributed software is a notoriously difficult task. This is mainly due to their complex structure that consists of complex interactions between distributed processes. These interactions introduce a significant complexity overhead on programmers who wish to ensure the correctness of their implementations, to add on top of the already challenging task of traditional software development.

The need for distributed software arises due to many concerns. Chief among those is the need for improved performance in applications. Nowadays, most applications are required to process large amounts of data. In the cases where processing of each chunk of data is independent of the other, it would be very beneficial if each chunk could be processed in parallel. To achieve this, distributing the data is an integral step.

Another case where distributed software is needed is when fault tolerance is desired in an application. To avoid having a single point of failure, several replicas of a component are created. The replicas then have to coordinate among themselves to maintain consistency and to select a course of action upon failure

of a node.

In some cases, distribution is simply part of the problem domain (such as in communication systems). The design of such software falls into two categories: 1) A centralized server that replies to requests from multiple nodes, or forwards messages to the intended recipient, and 2) Peer-to-peer communication among multiple nodes. Many applications follow one of these 2 paradigms and hence, we can see that simplifying the process of developing such software would be of great benefit.

Traditional programming methodologies for distributed software require programmers to explicitly define message passing primitives and insert them appropriately. This approach suffers from being error prone, as programmers have to develop each component separately, making it more difficult to reason about the global interaction model.

This can lead to situations where programmers inadvertently introduce *deadlock*, a state in which the program cannot advance because of processes waiting for each other to release resources or, in case of synchronous communication, a process waiting for another process to proceed together. Moreover, programmers have to be careful to avoid *race conditions*, situations where data is accessed concurrently by multiple processes with random undefined outcome.

Other programming models such as Spark [1], allow users to specify a computation to be performed and then automatically handle distribution of data and parallelization of the computation. This approach, while being very practical for certain applications, is not applicable in general for many distributed systems.

There is a need for an approach that allows for more control over how processes interact, while maintaining the ease of specifying how such interactions occur. To answer this need, this thesis proposes a new programming methodology, one that

allows the user to easily specify interactions among distributed processes without getting burdened with implementation details and ensuring correctness of such interactions.

We mainly distinguish two possible directions to cope with the complexity of the interaction model: (1) high-level modeling frameworks [2]; (2) session types [3, 4, 5, 6, 7, 8].

The former allows easy expression of the communication models, however, it lacks the efficiency of code generation. High-level and expressive communication models require the generation of controllers to implement their communication logic. For instance, if we consider multiparty interactions with non-deterministic behavior that may introduce conflicts between processes, such conflicts would be resolved by creating new processes (controllers). Additionally, it would be easier to develop distributed systems by reasoning about the global communication model and not local processes.

For this reason, session types were introduced. They are mainly based on the following principles: (1) communication model is described as a *global protocol* between processes; (2) Automatically synthesize *local types* which are the projection of global protocol w.r.t. processes; (3) develop the code of processes; (4) statically type-check the local code of the processes w.r.t. their local protocols. Consequently, the distributed software follows the stipulated global protocol. This approach mainly suffers from some redundancy when developing the code of local processes. For instance, any modification to the global protocol requires a re-implementation of the local code of the processes. Moreover, it does not make a clear separation between computation and communication logic in the local processes.

In this thesis, we introduce a new framework that allows the automatic syn-

thesis of the local code of the processes starting from a global choreography. First, we consider a set of components/processes with their interfaces and a configuration file that defines the variables of each component as well as the mapping between ports and their computation blocks. Then, given a global choreography, which is defined on the set of ports of the components and models coordination and composition operators, we automatically synthesize the local code of the processes that embed all communication and control flow logic. The choreography allows to define: (1) multiparty interaction; (2) branching; (3) loop; (4) sequential composition; and (5) parallel composition. Without loss of generality, which is the case in most distributed system applications, we consider master based protocols, which allow for the generation of fully distributed implementations, i.e., without the need of controllers. We apply our technique to automatically synthesize micro-services architecture starting from global protocols.

The remainder of this paper is structured as follows. In Chapter 3, we introduce a distributed component-based model that is used to define semantics of our choreography model. In Chapter 4, we define the syntax and the semantics of the choreography model. The semantics is defined by transforming it into the distributed component-based model. In Chapter 5, we provide an efficient code generation of the obtained distributed component-based model. Moreover, we present a real case study in collaboration with Murex Services S.A.L. industry [9] that uses our methodology to automatically synthesize the code of micro-services architecture. In Chapter 6, we define a transformation to Promela [10], a verification modeling language and present a non-trivial example to model. In Chapter 7, we present the work flow of our methodology and present its advantages over other methodologies.

Finally we draw conclusions and present future work in Chapter 8.

Chapter 2

Literature Review

The objective of this literature survey is to give a general overview about some important concepts related to the thesis work. We give an overview about current research directions in the field of modeling distributed protocols and then follow up by describing state-of-the-art innovations.

2.1 Distributed and parallel computing

The concept of concurrent computation dates back to the early 60s and has its origins in operating system architecture design. Back then, the need for concurrent computing was to have the ability to handle several computations at the same time to increase throughput and reduce the time taken to process requests. Thereafter, with the advent of telecommunications, distributed computing became a reality and many methodologies and techniques have since sprung up to tackle the issue of how to program for distributed systems.

Programming for parallel and distributed environments introduces several problems that are not found in traditional software design. Some of the key

issues are [11]:

- **Mutual Exclusion**: Mutual exclusion refers to the property about a program execution that states that no more than one thread of execution is allowed to enter certain blocks of code, commonly known as the *critical section*. Mutual exclusion is required to control access to shared resources, in order to prevent *race conditions*. A race condition is one in which more than one process attempts to modify a shared resource at the same time. This results in undefined behavior due to modern hardware architectures.

- **Deadlock**: Deadlock is a situation in which a group of processes are waiting for one another to perform some action, most commonly release of resources or message exchange. Processes belonging to this group are referred to as deadlocked processes. Mutual exclusion is a necessary condition for deadlock to occur, as is circular wait among the deadlocked processes.

- **Fault-tolerance**: A central feature of distributed systems is their decentralization. While this implies availability of more computing resources, this can also lead to situations where a part of the system is unavailable, either due to a hardware failure in one of the components or due to loss of messages. Algorithms that work for distributed systems despite the presence of faults are essential when dealing with large computer clusters, where failures are the norm rather than the exception.

- **Consensus**: The problem of getting all nodes in a distributed system to agree on a common value in the presence of faults in an asynchronous networking environment. In a famous result [12], distributed consensus was shown to be impossible to achieve in a fully asynchronous setting if *one* process is faulty.

Several techniques and algorithms have been proposed to address these issues, such as Ricart-Agrawala [13] and the Token Ring algorithms for distributed mutual exclusion, semaphores and condition variables for mutual exclusion with access to shared memory, and Paxos and Raft for distributed consensus. Obviously, due to the impossibility result in [12], Paxos and Raft cannot guarantee termination, but they offer some guarantees w.r.t to safety properties, i.e. they do not leave the system in an incorrect state which can lead to violation of the integrity of the data, but they may not terminate at all. However these cases rarely occur in practice, which explains the widespread adoption of these algorithms.

Due to the unique nature of these issues, programming for concurrent systems in general and distributed systems in particular is a challenging task. It requires skill and expertise, and is prone to many implementation errors. This adds to the complexity of the development process, as these protocols have to be seamlessly integrated with the main computation logic of the application. Moreover, while standard debugging and error-detection techniques allow for relatively easy analysis of errors in single threaded applications, logical errors in distributed applications are much more difficult to analyze because of the sheer amount of possible execution traces.

2.2 Model Checking

An important technique for verifying correctness properties of a program is model checking [14]. Model checking works in the following manner: First, a description of the system and the specifications it must meet is formulated in temporal logic. Next, armed with the formalization of the system and the specification, an exhaustive check is applied to ensure that the system meets the specification

at the relevant states. It logically follows that for this technique to work, the systems under consideration must be finite state in nature.

Correctness properties can be classified into two types:

- **Safety Properties**: This type of property specifies that nothing bad ever happens in the system. For example, in a banking application, it would be very useful to be able to verify that money is only withdrawn once in a transaction. Such properties are usually easier to check, since they can be violated in a finite execution of the system.

- **Liveness Properties**: This type of property specifies that something good will eventually happen, i.e. the system will eventually make progress. For example, in a messaging application, the property that a message will eventually reach its intended recipient is a liveness property. Such properties are harder to verify, since they cannot be violated in a finite trace of execution because the good event might occur after the trace ends.

As mentioned earlier, model checking takes as input a formal description of the system in temporal logic. Temporal logic is any system of rules and symbolism for representing, and reasoning about, propositions qualified in terms of time. There are many temporal logic models, but two of the most important are:

1. **Linear Temporal Logic (LTL)**: LTL is typically used to model a trace of a program's execution. Properties in LTL are built up from a set of propositional variables, the logical operators $\wedge$, $\vee$ and $\neg$, and the temporal operators *until* (U) and *next* (X). Two other temporal operators *eventually* (F) and *always* (G) can be expressed in terms of the other operators.

    - p - p holds in the current state

- $X(p)$ - p holds in the next state
- $p\ U\ q$ - The property p holds now and continues to hold *until* q holds
- $F(p)$ - p eventually holds in the trace. Note that this can be expressed as $T\ U\ p$
- $G(p)$ - p holds for all states in the trace. Note that this can be expressed $\neg(F(\neg p))$

2. **Computation Tree Logic (CTL)**: CTL is a branching time logic. As its name implies, its time model has a tree structure to indicate the different choices a process may take during its execution. This form of logic is typically used to model the states of a program and the transitions between them. CTL uses similar symbols to LTL, but includes the path quantifiers *all* (A) and *exists* (E).

2.3 Current research

Many coordination models exist to simplify the modeling of interactions in concurrent and distributed systems, such as in [15, 16]. Using these models requires the definition of the local behaviors of the processes and use of the communication model to implement the interactions between them, in contrast to our case where we automatically synthesize the local code of the processes.

Moreover, in order to reason about the correctness of coordinated processes, Session types [3, 4, 5, 6, 7, 8] and choreographies [17] have been proposed to statically verify the implementations of communication protocols based on the following methodology: (1) define communication protocol between processes using a *global protocol*; (2) Automatically synthesize *local types* which are the pro-

jection of global protocol w.r.t. processes; (3) develop the code of processes; (4) statically type-check the code of the processes w.r.t. local types. Consequently, the distributed software follows the stipulated global protocol. In our case, we automatically generate a more refined version of the local types that embeds all the communication and synchronization logic as well as control-flows, and which is correct-by-construction with respect to the global choreography.

In [18, 19], the authors present a method to synthesize a global choreography from a set local types. The global view allows for the reasoning and analysis of distributed systems. In our approach, we consider the inverse of that transformation, i.e., we create template with all the necessary communication and control flows of the end-point processes starting from a global choreography.

In [20, 21], the authors introduce syntactic transformations to refine distributed system programs starting from high-level specifications. In [20], the proposed specification differs from our choreography model as it is not possible to express multiparty interactions, or guarded loop, which makes it impractical in the context of distributed systems. In [21], the paper mainly targets multiparty interactions, where the main objective is to decouple multiparty interaction while preserving its semantics. In our case, there is no need to decouple multiparty interaction as we distinguish between synchronous and asynchronous ports. Additionally, in [20, 21], it is not clear how to automatically generate code from the refined programs.

BPMN [22] (Business Process Model and Notation) is an industry standard that allows to model process choreographies. An extension of BPMN was introduced in [23] to automatically derive a local choreography from a global one. Nonetheless, the extension only considers exchange of messages and does not formally define other composition operators such as synchronous multiparty com-

munications, parallelism, choice, sequential and loop.

In [24], the authors discuss a method to synthesize local implementations from a set of specifications. However, in their approach, the specifications are given in terms of the individual processes and are then *relativized* to obtain the global specifications before generating the implementations.

Chapter 3

Distributed Component-based Framework

In this section, we introduce a component-based framework, which is inspired by BIP [16]. The framework mainly consists of atomic components that communicate through the interaction model defined on the interface ports of the atomic components. Unlike BIP, we distinguish between four types of ports: (1) synchronous send; (2) asynchronous send; (3) asynchronous receive; and (4) internal ports. In BIP, all ports have the same type that only allow to build multiparty interactions. The new additional port types allow to (1) easily model distributed system communication models; (2) provide efficient code generation, under some constraints, that does not require to build controllers to handle conflicts between multiparty interactions. For the sake of simplicity, we omit variables from atomic components at this stage. Formally a port is defined as follows.

Definition 1 (Port) *A port p is defined by a (1) port identifier p; (2) its data type $p.\texttt{dtype} \in \{\texttt{int}, \texttt{str}, \texttt{bool}, \ldots\}$; and (3) and its communication type $p.\texttt{ctype} \in \{\texttt{ss}, \texttt{as}, \texttt{r}, \texttt{in}\}$, which denotes synchronous send, asynchronous send, receive, in-*

ternal, respectively. A receive port p has field p.`buff` *denoting the number of signals/data pending on that port.*

Given a port p, we define the predicate `isSSend`(p) (resp., `isASend`, `isRecv`, `isInternal`) that is `true` iff p is a synchronous send (resp., asynchronous send, receive, internal) port.

Atomic components are the main computation blocks. An atomic component is defined as follows.

Definition 2 (Atomic Component) *An atomic component B_i is defined as a tuple (P_i, Q_i, T_i), where (1) P_i is a set of ports; (2) Q_i is a set of states; (3) $T_i \subseteq Q_i \times P_i \times Q_i$ is a set of transitions.*

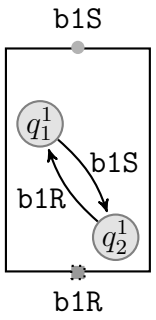


Figure 3.1: An atomic component

Example 1 *Figure 3.1 shows an atomic component with two states and two transitions. The two transitions are labeled by the ports,* `b1S` *and* `b1R`*. At state q_1^1, port* `b1S` *is enabled and at state q_2^1,* `b1R` *is enabled.*

In the sequel, we consider a system of n atomic components $\{B_i = (P_i, Q_i, T_i)\}_{i=1}^n$. We define the set $\mathcal{P} = \cup_{i=1}^n P_i$ (resp. $\mathcal{P}^{ss} = \{p \mid p \in \mathcal{P} \wedge \texttt{isSSend}(p)\}$, $\mathcal{P}^{as} = \{p \mid p \in \mathcal{P} \wedge \texttt{isASend}(p)\}$, $\mathcal{P}^{r} = \{p \mid p \in \mathcal{P} \wedge \texttt{isRecv}(p)\}$) of all

the ports (resp. synchronous send port, asynchronous send ports, receive ports) of the system. Moreover, we denote $\mathcal{P}_i^{ss}$ (resp. $\mathcal{P}_i^{as}$, $\mathcal{P}_i^{r}$) to be the set of all synchronous send (resp., asynchronous send, receive) ports of atomic component B_i. We also consider that port p_i belongs to component i. Given a state q_i, we consider that all the outgoing ports are enabled.

Synchronization between the atomic components is defined using the notion of interaction.

Definition 3 (Interaction) *An interaction is $a = (p_i, \{p_j\}_{j \in J})$, where $i \notin J$, is defined by (1) its send port p_i (synchronous or asynchronous) that belongs to the send ports of atomic component B_i, i.e., $p_i \in \mathcal{P}_i^{ss} \cup \mathcal{P}_i^{as}$; (2) its receive ports $\{p_j\}_{j \in J}$ each of which belongs to the receive ports of atomic component B_j, i.e., $p_j \in \mathcal{P}_j^{r}$.*

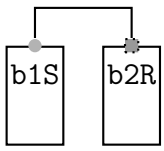


Figure 3.2: A basic interaction: $\{\texttt{b1S}, \texttt{b2R}\}$

An interaction $a = (p_i, \{p_j\}_{j \in J})$ is synchronous (resp. asynchronous) interaction iff $\texttt{isSSend}(p_i)$ (resp. $\texttt{isASend}(p_i)$).

A composite component consists of several atomic components and a set of interactions. Its semantics is defined as labeled transition system that is defined w.r.t. interaction types.

First, Equation 3.1 represents synchronous interaction, i.e., $a = (p_i, \{p_j\}_{j \in J})$ and $\texttt{isSSend}(p_i)$, which requires all the corresponding receive ports to be enabled with no pending messages (their buffers are empty) and which results in updating the states all the involved components simultaneously. Second, Equations 3.2

and 3.4 represent asynchronous interactions. Equation 3.2 represents the asynchronous execution of the send port without requiring the participation of the corresponding receive ports, however, upon its execution it places the data or synchronization notice in the buffers of the corresponding receive ports. Then, Equation 3.3 represents the autonomous execution of receive ports with no empty buffers. Finally, Equation 3.4 represents the autonomous execution of internal ports that only allow to change local state of atomic components.

Definition 4 (Composite Component) *A composite component B is defined by a composition operator parameterized by a set of interactions γ. $B = \gamma(B_1, \ldots, B_n)$, is a transition system $(Q, \gamma, \rightarrow)$, where $Q = \bigotimes_{i=1}^{n} Q_i$ and $\rightarrow$ is the least set of transitions satisfying the following rules:*

$$\text{synch-send:}\frac{\begin{array}{c} a = (p_i, \{p_j\}_{j\in J}) \in \gamma \quad \texttt{isSSend}(p_i) \\ \forall k \in J \cup \{i\} : q_k \xrightarrow{p_k} q'_k \quad \forall k \notin J \cup \{i\} : q_k = q'_k \\ \forall k \in J : p_k.\texttt{buff} = 0 \end{array}}{(q_1, \ldots, q_n) \xrightarrow{a} (q'_1, \ldots, q'_n)} \tag{3.1}$$

Informally, this means the following: To execute a synchronous interaction, all its ports must be enabled in the state of the corresponding component. Upon execution, all components involved in the interaction execute synchronously and move to the next state specified by their transition relation. Components that are not involved in the interaction are unaffected and remain the same state.

$$\text{asynch-send:}\frac{\begin{array}{c} a = (p_i, \{p_j\}_{j\in J}) \in \gamma \\ \texttt{isASend}(p_i) \quad q_i \xrightarrow{p_i} q'_i \quad \forall k \neq i : q_k = q'_k \end{array}}{\begin{array}{c} (q_1, \ldots, q_n) \xrightarrow{a} (q'_1, \ldots, q'_n) \\ \forall j \in J : p_j.\texttt{buff} := p_j.\texttt{buff} + 1 \end{array}} \tag{3.2}$$

Informally, this means the following: To execute the sending part of an asynchronous interaction, p_i (the sending port) must be enabled in the state of the

corresponding component. Upon execution, only the sending component moves to the next state defined by its transition relation. It also updates the buffer of all receiving components to indicate that a message is ready to be consumed. Other components (including components corresponding to the receive ports in the interaction) are unaffected and remain in the same state.

$$\text{recv:}\frac{q_j \xrightarrow{p_j} q'_j \quad \texttt{isRecv}(p_j) \quad p_j.\texttt{buff} > 0 \quad \forall k \neq j : q_k = q'_k}{(q_1, \ldots, q_n) \xrightarrow{a} (q'_1, \ldots, q'_n) \quad p_j.\texttt{buff} := p_j.\texttt{buff} - 1} \tag{3.3}$$

Informally, this means the following: To execute the receiving part of an asynchronous interaction, one of the receive ports of the interaction must be enabled AND a message must have already been sent (indicated by the buffer counter being greater than zero). Upon execution, the receiving component consumes the message, decrements the buffer counter and moves to the next state specified by its transition relation. Other components are unaffected and remain in the same state.

$$\text{internal:}\frac{q_i \xrightarrow{p_i} q'_i \quad \texttt{isInternal}(p_i) \quad \forall k \neq i : q_k = q'_k}{(q_1, \ldots, q_n) \xrightarrow{\epsilon} (q'_1, \ldots, q'_n)} \tag{3.4}$$

Informally, this means the following: To execute an internal transition, it is sufficient for the port to be enabled in the corresponding component. Other components remain unaffected.

Example 2 *Figure 3.3 shows a composite component that consists of two atomic ones. Both of them have one port enabled, assuming the starting state of the system is* (q_1^1, q_1^2)*. This component has one interaction only (*$\texttt{b1S}, \texttt{b2R}$*).*

Finally, a system is defined as a composite component where we specify the

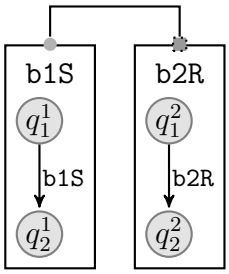


Figure 3.3: A composite component

initial state of its atomic components.

Definition 5 (System) *A system is a pair* $S = (B, \texttt{init})$*, where* $B = \gamma(B_1, \ldots, B_n)$ *is a composite component and* $\texttt{init} \in \bigotimes_{i=1}^{n} Q_i$ *is the the initial state of* B*.*

For example, the component B defined in 3.3 can have 4 possible associated systems that differ only by their initial state. The set of possible initial states corresponding to these 4 systems is given by $Q_1 \times Q_2$.

Chapter 4

Global Choreography

In this chapter, we introduce the concept of a choreography. A choreography is a model of a system that specifies how the components of the system are to interact with each other. We define how to automatically synthesize the behaviors of atomic components given a global choreography model. In particular, we consider that we have a composite component consisting of atomic components with their basic interface ports. That is, the behaviors of atomic components are empty. Atomic components can be considered as services with their interfaces but with undefined behaviors.

We first introduce the abstract syntax of the global choreography model, which allows for: (1) synchronous and asynchronous communications between interface ports; (2) sequential composition of two choreographies; (3) parallel composition of two choreographies; (4) conditional master branching, i.e., the branching decision is taken by a specific component; (5) conditional master loops, i.e., the loop condition is guided by a specific component. Listing 4.1 depicts the abstract syntax of the choreography model.

```
ch ::= snd ⟶ rcvs : <T> # send/receive
 | B_i ⊕ {snd_j : ch}_{j∈J} # Branching -- where snd_j ∈ P_i^ss
 |  while(snd) ch end # loop
 |  ch • ch # sequential
 |  ch ‖ ch # parallelism
 |  ϵ

snd ::= p_i # sender -- where p_i ∈ P_i^ss ∪ P_i^as

rcvs ::= p_i | p_i rcvs # receivers -- where p_i ∈ P_i^r

T ::= bool | int | str
```

Listing 4.1: Abstract syntax of the global choreography model

4.1 Preliminary Notations

We introduce some preliminary concepts and notations. We define $\texttt{start}(\texttt{ch})$ (where $\texttt{ch}$ is a choreography) as the set of indices of the components in $\texttt{ch}$ that should be notified to trigger the start of $\texttt{ch}$. Intuitively, a simple synchronous or asynchronous send/receive can start when notifying its corresponding send port. A choreography consisting of the sequential composition of two choreographies can be triggered by notifying the first choreography in the composition. A choreography consisting of the parallel composition of two choreographies can be triggered by notifying the two choreographies of the composition. A branching and loop choreography can be triggered by notifying their corresponding master component. Formally:

$$
\begin{aligned}
\texttt{start}(p_i \longrightarrow \{p_j\}_{j\in J}) &= \{i\} \\
\texttt{start}(\texttt{ch}_1 \bullet \texttt{ch}_2) &= \texttt{start}(\texttt{ch}_1) \\
\texttt{start}(\texttt{ch}_1 \parallel \texttt{ch}_2) &= \texttt{start}(\texttt{ch}_1) \cup \texttt{start}(\texttt{ch}_2) \\
\texttt{start}(B_i \oplus \{p_l : \texttt{ch}_l\}_{l\in L}) &= \{i\} \\
\texttt{start}\big(\texttt{while}(p_i)\ \texttt{ch}\ \texttt{end}\big) &= \{i\}
\end{aligned}
$$

Similarly, we define $\texttt{end}(\texttt{ch})$ as the index of the components in $\texttt{ch}$ that implies the termination of all the corresponding send operations in $\texttt{ch}$. We consider that a simple send/receive is done when the sending part is done. If the send part is synchronous then it implicitly requires the ending of all the corresponding receiving parts. However, if the send part is asynchronous, any subsequent choreography can start after the sending is complete, otherwise, synchronous and asynchronous would be almost identical w.r.t. the composition operators. A choreography consisting of the sequential composition of two choreographies is considered to be done when the second choreography in the composition is done. A choreography consisting of the parallel composition of two choreographies is considered to be done when the first and second choreographies in the composition are done. A branching choreography is considered to be done when *all* components in the choreographies to choose from are done executing their part of the chosen choreography. A loop choreography is considered to be done when the corresponding master component is done. Formally:

$$
\begin{aligned}
\texttt{end}(p_i \longrightarrow \{p_j\}_{j \in J}) &= \{i\} \\
\texttt{end}(\texttt{ch}_1 \bullet \texttt{ch}_2) &= \texttt{end}(\texttt{ch}_2) \\
\texttt{end}(\texttt{ch}_1 \parallel \texttt{ch}_2) &= \texttt{end}(\texttt{ch}_1) \cup \texttt{end}(\texttt{ch}_2) \\
\texttt{end}(B_i \oplus \{p_l : \texttt{ch}_l\}_{l \in L}) &= \textstyle\bigcup_{l \in L} \texttt{end}(\texttt{ch}_l) \\
\texttt{end}\big(\texttt{while}(p_i)\,\texttt{ch}\,\texttt{end}\big) &= \{i\}
\end{aligned}
$$

We define $\mathcal{C}(\texttt{ch})$ as the index of all components in choreography $\texttt{ch}$. Formally:

$$
\begin{aligned}
\mathcal{C}(p_i \longrightarrow \{p_j\}_{j\in J}) &= \{i\} \cup J \\
\mathcal{C}(\texttt{ch}_1 \bullet \texttt{ch}_2) &= \mathcal{C}(\texttt{ch}_1) \cup \mathcal{C}(\texttt{ch}_2) \\
\mathcal{C}(\texttt{ch}_1 \parallel \texttt{ch}_2) &= \mathcal{C}(\texttt{ch}_1) \cup \mathcal{C}(\texttt{ch}_2) \\
\mathcal{C}(B_i \oplus \{p_l : \texttt{ch}_l\}_{l\in L}) &= \bigcup_{l\in L} \mathcal{C}(\texttt{ch}_l) \cup \{i\} \\
\mathcal{C}\big(\texttt{while}(p_i)\,\texttt{ch}\,\texttt{end}\big) &= \{i\} \cup \mathcal{C}(\texttt{ch})
\end{aligned}
$$

Moreover, in the sequel, we represent receive ports (resp. synchronous send, asynchronous send) using dashed square (resp. circle with solid border, circle with dashed border). We also omit the border for send ports when synchrony is out of context.

4.2 Semantics (Transformations) - Projection

Given a global choreography $\texttt{ch}$ that is defined on the set of ports $\mathcal{P} = \cup_{i=1}^{n} P_i$ of a given atomic components (with empty behavior), its semantics is defined by a set of transformations that build system $S = (B, \texttt{init})$ containing atomic components with their behaviors and the corresponding interactions. Moreover, as we progressively build system S, we consider that it has a context to denote the current state where a choreography should be appended. For this, $\mathcal{S} = (S, \texttt{context})$ denotes a system with its corresponding context. $\texttt{context}$ takes an atomic component as input and returns a state, i.e., $\texttt{context}(B_i) \in Q_i$ to denote the current context of atomic component B_i. We ensure that the application of choreographies guarantees that the context of any components would consist of only one state. Initially, we consider $\mathcal{S} = (S, \texttt{context})$, where $B = \gamma(B_1, \ldots, B_n)$ with: (1) $\gamma = \emptyset$; (2) $B_i = (P_i, \{l_i\}, \emptyset)$; (3) $\texttt{init} = (q_1^{\texttt{init}}, \ldots, q_n^{\texttt{init}})$; and (4)

$\texttt{context}(B_i) = q_i^{init}$. The initial state of the obtained system that corresponds to a given choreography remains unchanged, i.e., is equal to `init`. As such, for the sake of clarity, we omit it in our construction. In the sequel, we define how to build the behaviors of atomic components given a choreography. We distinguish between the different types of choreographies: (1) send/receive, (2) sequential; (3) parallelism: (4) loop; and (5) branching.

4.2.1 Send/Receive

Send/receive choreography updates the participating components by adding a transition from the current context and labeling it by the corresponding send or receive port from the choreography. In order to avoid inconsistencies between same ports but from different choreographies, we create a copy of each port of the choreography (`copy`). However, we keep track of this mapping to avoid creating too many ports in the code generation phase and when mapping code to interface ports (the same code should be mapped to all the copies). We also add the corresponding interaction between the send and the receive ports. Finally, we update the context of the participants to be the corresponding new added states. As such, if the initial context of each component consists of one state, then the resulting system (after applying the send/receive choreography) also guarantees that each of its components also consists of one state. Note that, an interaction connected to a synchronous send port and receive ports can be considered as a multiparty interaction with a master trigger, which is the send port. As such this allows to efficiently implement multiparty interactions.

Definition 6 (Send/Receive) *Consider a system with its context* $\mathcal{S} = (S, \texttt{context})$*, we define the synchronization as an operator that transforms* $\mathcal{S}$ *into a new system*

$\mathcal{S}' = (S', \texttt{context}') = \mathcal{S}[\![p_i \longrightarrow \{p_j\}_{j \in J}]\!]$, *with* $S' = \gamma'(B'_1, \ldots, B'_n)$, *where:*

- *Atomic components are updated as follows,*

$$B'_k = \begin{cases} (P_k, Q'_k, T'_k) & \forall k \in J \cup \{i\} \\ B_k & \textit{otherwise} \end{cases}$$

  where, (1) $Q'_k = Q_k \cup \{q_k^{s'}\}$*; (2)* $T'_k = T_k \cup \{\texttt{context}(B_k) \xrightarrow{\texttt{copy}(p_k)} q_k^{s'}\}$

- *Interactions are updated as follows,* $\gamma' = \gamma \cup a$*, where* $a = (\texttt{copy}(p_i), \{\texttt{copy}(p_j)\}_{j \in J})$

- *Context is updated as follows,*

$$\texttt{context}'(B'_k) = \begin{cases} q_k^{s'} & \forall k \in J \cup \{i\} \\ \texttt{context}(B_k) & \textit{otherwise} \end{cases}$$

Example 3 (Send/Receive) *Figure 4.1 shows an abstract example on how to transform a simple send/receive choreography,* `b1S` $\longrightarrow$ `b2R`, `b3R`*, into an initial system consisting of three components with interfaces:* `b1S` *(send, synchronous or asynchronous),* `b2R` *(receive), and* `b3R` *(receive), respectively.*

4.2.2 Sequential Composition

The binary operator $\bullet$ allows to sequentially compose two choreographies, $\texttt{ch}_1 \bullet \texttt{ch}_2$. For this, its semantics is defined by (1) applying $\texttt{ch}_1$; (2) notifying the start of $\texttt{ch}_2$; and finally (3) applying $\texttt{ch}_2$. As we require that $\texttt{ch}_1$ must terminate

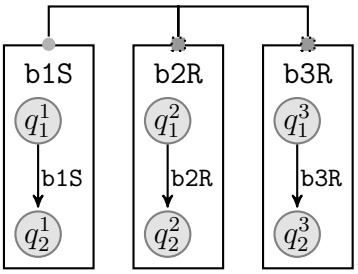


Figure 4.1: Send/Receive Transformation

before the start of $\texttt{ch}_2$, we need to synchronize all the end components of $\texttt{ch}_1$ with all the start components of $\texttt{ch}_2$. To do so, it is sufficient to pick one of the end components of $\texttt{ch}_1$ and create a synchronous send port, which is connected to new receive ports added to the remaining end components of $\texttt{ch}_1$ and start components of $\texttt{ch}_2$. Moreover, the application of the sequential composition guarantees that each component of the resulting system consists of exactly one state, provided that the context of each component of the initial system consists of one state. Formally, the semantics of the sequential composition is defined as follows.

Definition 7 (Sequential Composition) *Consider a system with its context* $\mathcal{S} = (S, \texttt{context})$*, we define the* $\bullet$ *binary operator that sequentially composes two choreographies. Its semantics is defined by transforming* $\mathcal{S}$ *into a new system* $\mathcal{S}' = (S, \texttt{context}) = \mathcal{S}[\![\texttt{ch}_1 \bullet \texttt{ch}_2]\!]$*, with* $\mathcal{S}' = \mathcal{S}[\![\texttt{ch}_1]\!][\![\texttt{ch}_{synch}]\!][\![\texttt{ch}_2]\!]$*, where:* $\texttt{ch}_{synch} = p_i^{\texttt{cs}} \longrightarrow \{p_j^{\texttt{cr}}\}_{j \in J}$ *such that: (1)* $i \in \texttt{end}(\texttt{ch}_1)$*; (2)* $J = \texttt{end}(\texttt{ch}_1) \cup \texttt{start}(\texttt{ch}_2) \setminus \{i\}$*; (3)* $p_i^{\texttt{cs}}$ *is a new synchronous send port to be added to* $\mathcal{P}_i^{ss}$*; and (4)* $\{p_j^{\texttt{cr}}\}_{j \in J}$ *are new receive ports to be added to* $\mathcal{P}_j^{r}$*.*

Example 4 (Sequential composition) *Figure 4.2 shows an abstract example on how to transform sequential composition of two choreographies,* $\texttt{ch}_1 \bullet \texttt{ch}_2$*, into an initial system consisting of five components. Here we only consider components that are involved in those choreographies, where (1) components* b_1*,* b_2*,* b_3 *and*

b_4 *are involved in choreography* $\mathtt{ch}_1$*; and (2) components* b_1*,* b_2*,* b_3 *and* b_5 *are involved in choreography* $\mathtt{ch}_2$*. Note, components that are not involved are kept unchanged. The transformation requires to: (1) apply first choreography* $\mathtt{ch}_1$ *to its participated components (i.e.,* b_1*,* b_2*,* b_3 *and* b_4*); (2) synchronize the end of choreography* $\mathtt{ch}_1$ *(e.g.,* b_1*) with the start of choreography* $\mathtt{ch}_2$ *(e.g.,* b_2 *and* b_3*). To do so, we create a synchronous send port to one of the end components of* $\mathtt{ch}_1$ *(e.g.,* b_1^{cs}*) and connect it to all the remaining end components of* $\mathtt{ch}_1$ *(e.g.,* $\emptyset$ *and the start components of* $\mathtt{ch}_2$ *(e.g.,* b_2 *and* b_3*); finally (3) we apply choreography* $\mathtt{ch}_2$*.*

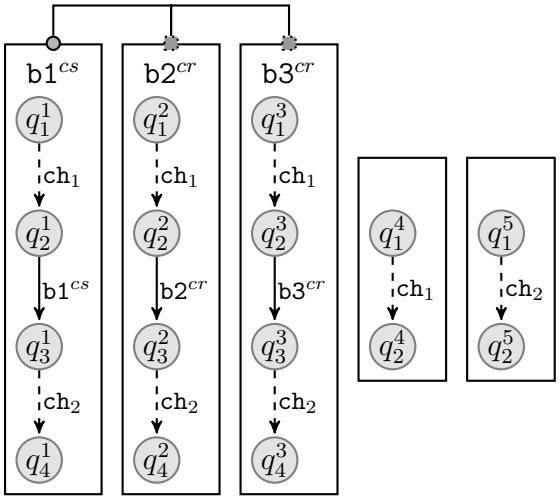


Figure 4.2: Sequential composition transformation

4.2.3 Parallel Composition

The binary operator $\|$ allows for the parallel compositions of two independent choreographies. Two choreographies are independent if their participating components are disjoint.

Definition 8 (Independent Choreographies) *Two choreographies* $\mathtt{ch}_1$ *and* $\mathtt{ch}_2$ *are said to be independent iff* $\mathcal{C}(\mathtt{ch}_1) \cap \mathcal{C}(\mathtt{ch}_2) = \emptyset$*.*

We consider independent choreographies to avoid conflicts and interleaving of executions within components. Add to that, this simplifies reasoning and writing choreographies as well as for efficient code generation. Note that parallelizing independent choreographies implies that each component has a single execution flow. This would not reduce the expressiveness of our model as parallel execution flows can be modelled in separate components. The semantics of the parallel composition $\texttt{ch}_1 \parallel \texttt{ch}_2$ is simply defined by applying $\texttt{ch}_1$ and $\texttt{ch}_2$ in any order, which leads to the same system as the two choreographies are independent, i.e., they behave on different set of components. Moreover, the application of the parallel composition guarantees that each component of the resulting system consists of exactly one state, provided that the context of each component of the initial system consists of one state.

Definition 9 (Parallel Composition) *Consider a system with its context* $\mathcal{S} = (S, \texttt{context})$*, we define the* $\parallel$ *operator that parallelizes two independent choreographies.* $\parallel$ *is defined as an operator that transforms* $\mathcal{S}$ *into a new system* $\mathcal{S}' = (S', \texttt{context}') = \mathcal{S}[\![\texttt{ch}_1 \parallel \texttt{ch}_2]\!] = \mathcal{S}[\![\texttt{ch}_1]\!][\![\texttt{ch}_2]\!] = \mathcal{S}[\![\texttt{ch}_2]\!][\![\texttt{ch}_1]\!]$.

Example 5 (Parallel Composition) *Figure 4.2 shows an abstract example on how to transform parallel composition of two choreographies,* $\texttt{ch}_1 \parallel \texttt{ch}_2$*, into an initial system consisting of five components. Here, we consider that* $\texttt{ch}_1$ *(resp.* $\texttt{ch}_2$*) involves components* B_1 *and* B_2 *(resp.* B_3 *and* B_4*).*

Remark 1 *Note also in case of parallelizing choreographies that have a component in common (i.e., not independent), we can still apply the parallel composition either by (1) enforcing any arbitrary order of execution. As such, in the case of independent choreographies, true parallelism is achieved, otherwise, we apply them in any order to avoid non-deterministic execution; (2) using of product automata*

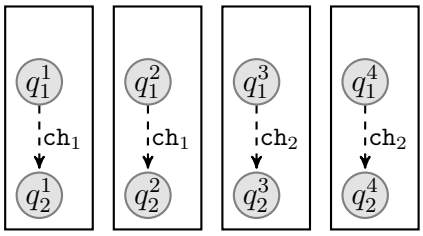


Figure 4.3: Parallel composition transformation

as defined in [17]; (3) use of multiple execution flows (i.e., multi-threading within a component).

4.2.4 Branching Composition

Branching, $B_i \oplus \{p_i^l : \mathtt{ch}_l\}_{l \in L}$, allows for the modeling of choice between several choreographies. The choice is made by a specific component (B_i), which depending on its internal state would notify, by sending a label (p_i^l), the appropriate components to follow the taken choice (i.e., the corresponding choreography, $\mathtt{ch}_l$). We apply branching by independently integrating the choreography for each choice. This can be done by letting B_i notify the participants of the choreography of that choice. Then, we define a union operator `union` that takes a set of systems with their contexts and (1) unions all of their states, transitions and ports; then (2) it updates the contexts of the obtained components by joining each of their input contexts with internal transitions. Therefore, after applying the branching we guarantee that each component will have one and only one context state. Formally, the `union` operator is defined as follows.

Definition 10 (Union) *Given a set of systems with their contexts* $\{\mathcal{S}_l = (S_l, \mathtt{context}_l)\}_{l \in L}$, *with* $S_l = (\gamma^l(B_1^l, \ldots, B_n^l), \mathtt{init})$, *we define* $\mathcal{S} = \mathtt{union}(\{\mathcal{S}_i\}_{l \in L}) = (S, \mathtt{context})$ *with* $S = (\gamma(B_1, \ldots, B_n), \mathtt{init})$, *where : (1)* $\gamma = \bigcup_{l \in L} \gamma^l$; *(2)* $B_i = (\bigcup_{l \in L} P_i^l, \bigcup_{l \in L} Q_i^l \cup \{q_i^c\}_{l \in L}, \bigcup_{l \in L} T_i^l \cup T_i^{\mathtt{merge}})$ *such that* $T_i^{\mathtt{merge}} = \{\mathtt{context}_l(B_i^l) \xrightarrow{\epsilon} q_i^c \mid l \in L\}$, *(3)*

$\texttt{context}(B_i) = q_i^c$.

Then, branching as described above can be defined by independently applying each choice, then doing the union. Formally:

Definition 11 (Branching) *Consider a system with its context* $\mathcal{S} = (S, \texttt{context})$*, we define the* $\oplus$ *operator that allows a specific component to select from a set of choreographies. It can be defined as an operator that transforms* $\mathcal{S}$ *into a new system* $\mathcal{S}' = (S', \texttt{context}') = \mathcal{S}[\![B_i \oplus \{p_i^l : \texttt{ch}_l\}_{l \in L}]\!] = \texttt{union}(\{\mathcal{S}[\![p_i^l \longrightarrow \{p_k^{\texttt{cr}_l}\}_{k \in K}]\!][\![\texttt{ch}_l]\!]\}_{l \in L})$*, where (1)* $K = \mathcal{C}(B_i \oplus \{p_i^l : \texttt{ch}_l\}_{l \in L}) \setminus \{i\}$ *are the components to be notified of choice made by component* i*; (2)* $\{p_k^{\texttt{cr}_l}\}_{k \in K}$ *are new control receive ports that are added to components with indices in* K*.*

Remark 2 *Note that, we require to notify all the participants of a choice and not only the start components. Consider the following choreography (where* α *and* β *denote some choreographies):*

$$B_1 \oplus \{p_1^l : xp_2 \longrightarrow rp_3 \bullet \alpha; p_2^l : p_2 \longrightarrow p_3 \bullet \beta\}$$

In this choreography, if we would have not sent the choice made by component 1 *to component* 3*, then component* 3 *cannot know about the decision that was taken by component* 1*. Hence, it cannot decide whether to then follow choreography* α *or* β*.*

Example 6 (Branching) *Figure 4.4 shows an abstract example on how to apply a branching operation that consists of two choices* $B_1 \oplus \{b1^{l_1} : \texttt{ch}_1, b2^{l_2} : \texttt{ch}_2\}$*. First we add choice transitions to component* B_1 *and synchronize them with the participants of* $\texttt{ch}_1$ *and* $\texttt{ch}_2$*, e.g.,* B_2 *and* B_3*. Then, we apply the choreographies accordingly. Finally, we merge the contexts with internal transitions.*

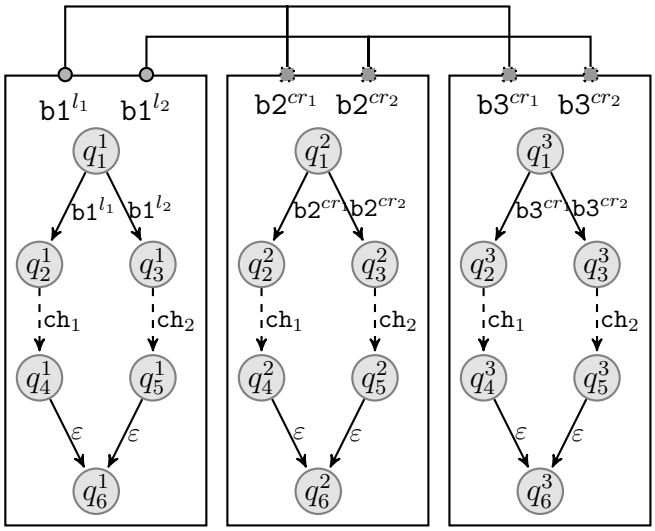


Figure 4.4: Branching transformation

4.2.5 Loop Composition

Loop, $\texttt{while}(p_i)\{\texttt{ch}\}$, allows for the modeling of a conditional repeated choreograph ch. The condition is evaluated by a specific component, which will notify, through the port p_i, the participants of the choreography to either re-execute it or break.

Definition 12 (Loop) *Consider a system* $\mathcal{S} = (S, \texttt{context})$, while *can be defined as an operator that transforms* $\mathcal{S}$ *into a new system* $\mathcal{S}' = (S', \texttt{context}') = \mathcal{S}[\![\texttt{while}(p_i)\{\texttt{ch}\}]\!]$, *where* $\mathcal{S}'$ *is defined as follows. First, we build* $\mathcal{S}^{\texttt{t}}$ *that corresponds to applying one iteration of the while loop. Formally,* $\mathcal{S}^{\texttt{t}} = (S^{\texttt{t}}, \texttt{context}^{\texttt{t}}) = \mathcal{S}[\![p_i \longrightarrow \{p_k^{\texttt{cr}}\}_{k\in K}]\!][\![\texttt{ch}]\!]$, *where* $S^{\texttt{t}} = (\gamma^{\texttt{t}}(B_1^{\texttt{t}}, \ldots, B_n^{\texttt{t}}), \texttt{init})$. *Then, we build* $\mathcal{S}'$ *by adding the reset and the end of loop. Formally,* $S' = (\gamma'(B_1', \ldots, B_n'), \texttt{init})$, *where* $B_j' = (P_j', Q_j', T_j')$, *s.t.:*

- *Adding the reset and the end loop transitions: if* $j \in K \cup \{i\}$, $T_j' = T_j^{\texttt{t}} \cup \{\texttt{context}^{\texttt{t}}(B_j) \xrightarrow{\epsilon} \texttt{context}(B_j), \texttt{context}(B_j) \xrightarrow{p_j^{\texttt{f}}} q_j^c\}$, *otherwise,* $T_j' = T_j^{\texttt{t}}$.

- P'_j *and* Q'_j *are equal to* $P^{\mathtt{t}}_j$ *and* $Q^{\mathtt{t}}_j$ *by adding the newly created ports and states, i.e.,* $p^{\mathtt{f}}_j$, q^c_j, *where* $\mathtt{isSSend}(p^{\mathtt{f}}_i)$, *i.e., is a synchronous send port.*

- $\gamma' = \gamma^{\mathtt{t}} \cup (p^{\mathtt{f}}_i, \{p^{\mathtt{f}}_j\}_{j \in K})$

- *Update the context to be the state denoting the end of the loop:*

$$\mathtt{context}'(B'_j) = \begin{cases} q^c_j & \text{if } j \in K \cup \{i\} \\ \mathtt{context}(B_j) & \text{otherwise} \end{cases}$$

Example 7 (Loop) *Figure 4.5 shows an abstract example on how to apply a loop operation guided by component* B_1 *and where its participants are components* B_1, B_2 *and* B_3.

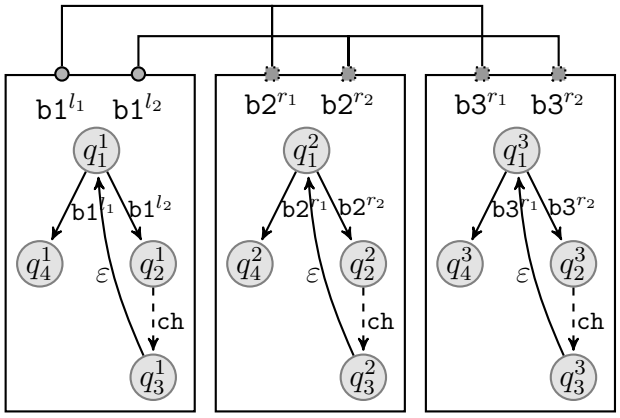


Figure 4.5: Loop composition transformation

4.2.6 Global choreography

We are now ready to define how to generate the behavior of the components of system $\mathcal{S}$ given a choreography $\mathtt{ch}$ by simply using *pattern-matching*:

$$
\begin{aligned}
\mathcal{S}[\![\mathtt{ch}]\!] \;=\; & \;\mathtt{match}\;\mathtt{ch}\;\mathtt{with} \\
\mid\; & \;\mathtt{p_i} \longrightarrow \{\mathtt{p_j}\}_{\mathtt{j}\in\mathtt{J}} \rightarrow \text{Definition 6} \\
\mid\; & \;\mathtt{ch_1} \bullet \mathtt{ch_2} \rightarrow \text{Definition 4} \\
\mid\; & \;\mathtt{ch_1} \parallel \mathtt{ch_2} \rightarrow \text{Definition 9} \\
\mid\; & \;\mathtt{B_i} \oplus \{\mathtt{p_i^l} : \mathtt{ch_l}\}_{\mathtt{l}\in\mathtt{L}} \rightarrow \text{Definition 11} \\
\mid\; & \;\mathtt{while}(\mathtt{p_i})\{\mathtt{ch}\} \rightarrow \text{Definition 12}
\end{aligned}
$$

Chapter 5

Code Generation

We describe the principle of the code generation from the generated components. The code generation takes as input a choreography and a configuration file that contains: (1) components' variables; (2) variables mapped to ports, which will be sent when executing an interaction; (3) functions mapped to send and receive ports, which are computation blocks over the variables of the corresponding component; and (3) guards mapped to send ports, which are boolean conditions over the variables of the corresponding component. Our code generation then automatically produces the corresponding implementation of each of the components. Following our choreography semantics, by definitions, the obtained components have the following characteristics: (1) they do not have a state with outgoing send and receive ports; (2) a port is connected to exactly one interaction. As such, there is no conflicting interactions that can run concurrently. Two interactions are said to be conflicting iff they share a common component. Consequently, it is possible to generate fully distributed implementations, with no need for controllers. The code structure is depicted in Algorithm 1 that requires only send/receive primitives. We distinguish between two possible cases. First, if all

outgoing transitions are labeled with send ports, we pick a random enabled port, i.e., its guard evaluated to true. Then, we notify all the receive ports that are connected to the interaction containing that port. If the port is a synchronous send port, the component waits for an acknowledgment from the corresponding receive components. Second, if all outgoing transitions are labeled with receive ports, the component waits until a message is ready/received in one of the receive ports. Upon receiving a message, we acknowledge its receipt if the port is connected to a synchronous interaction. Finally, we update the current state of the component depending on the executed port. It is worth mentioning that it is possible to provide a code generation w.r.t. a communication library (e.g., MPI, Java Message Service). In this case, the code generation can benefit from the features provided by the library, e.g., synchronous communication such as `MPI_Ssend`.

Algorithm 1: Pseudo-code - generated components.

```
1  initialization();
2  while true do
3      if all outgoing transitions are send then
4          port p = select enabled port, i.e., guard true;
5          notify all the receivers of the interaction that has port p;
6          if p is synchronous then
7              wait for ack. from the receivers;
8          end
9      end
10     else if all outgoing transitions are receive then
11         wait until a message is ready in one of the outgoing receive ports;
12         port p = select message;
13         if interaction connected is synchronous then
14             send ack. to the corresponding send port;
15         end
16     updateCurrentState();
17 end
```

5.1 Building Micro-Services Using Choreography

Traditionally, distributed applications follow a monolithic architecture, i.e., all the services are embedded within the same application. A new trend is to split complex applications up into smaller micro-services, where each micro-service can live on its own within a container. We use our choreography to model a case study that uses micro-service architecture allowing clients to deploy system images and request packages. We then automatically derive the skeleton of each micro-service. The global choreography is defined as follows. A client (`c`) sends a request to the *gateway service* (`gs`), which is the only visible micro-service to the client, containing the required version, revision, pool name, and an identifier to the testing data. `gs` forwards the request to the *deploy environment service* (`des`). `des` creates an environment id and returns it back to `gs`, which in turn forwards it back to `c`. `des` sends to the *deploy application directory service* (`dads`) and the *deploy database service* (`dds`) (i) required version, revision and pool name and (ii) testing data identifier and environment id, respectively. `c` keeps checking if the environment is ready, which is done through the gateway service with the help of the *environment info. service* (`eis`).

`dads` requests from the *machine service* (`ms`) and the *setup service* (`ss`) (i) a machine location from the pool and (ii) the package location, respectively. When `dads` receives the replies from both `ms` and `ss`, it contacts the appropriate *host machine* ($\texttt{hm}_i$) by sending the package location. Then, $\texttt{hm}_i$ sends its status to `des`. `des` upon receiving the status update, it forwards it to the `eis`.

`dds` requests from the *dumps service* (`dus`) and the *Database machines services* (`dms`) (i) testing data location, and (ii) a database server, respectively. When `dds`

receives the replies from both dus and dbs, it contacts the appropriate *database server* $\texttt{hd}_j$ by sending the testing data location. Then, $\texttt{hd}_j$ sends its status to des. des upon receiving the status update, it forwards it to the eis.

For each micro-service/component m, we denote by mSS, mAS mR a corresponding synchronous send, asynchronous send and receive port, respectively. We can then describe the global choreography CH as follows:

$$\texttt{CH} = \texttt{CH}_1 \bullet \texttt{CH}_2 \bullet \texttt{CH}_3$$

$$\texttt{CH}_1 = \texttt{cSS} \rightarrow \texttt{gsR} \bullet \texttt{gsSS} \rightarrow \texttt{desR} \bullet \texttt{desAS} \rightarrow \texttt{gsR}$$

$$\texttt{CH}_2 = \texttt{CH}_2^1 \bullet \texttt{CH}_2^2$$

$$\texttt{CH}_2^1 = \texttt{gsSS} \rightarrow \texttt{cR} \parallel (\texttt{desAS} \rightarrow \texttt{dadsR} \bullet \texttt{desAS} \rightarrow \texttt{dadsR})$$

$$\texttt{CH}_2^2 = \texttt{while}(\texttt{cSS})\ \texttt{cSS} \rightarrow \texttt{gsR} \bullet \texttt{gsSS} \rightarrow \texttt{eisR} \bullet \texttt{eisSS} \rightarrow \texttt{gsR} \bullet \texttt{gsSS} \rightarrow \texttt{cR}\ \texttt{end}$$

$$\texttt{CH}_3 = (\texttt{CH}_4 \parallel \texttt{CH}_5) \bullet \texttt{CH}_6$$

$$\texttt{CH}_4 = \texttt{CH}_4^1 \bullet \texttt{CH}_4^2 \bullet \texttt{CH}_4^3$$

$$\texttt{CH}_4^1 = \texttt{dadsAS} \rightarrow \texttt{amsR} \bullet \texttt{dadsAS} \rightarrow \texttt{SSR}$$

$$\texttt{CH}_4^2 = \texttt{amsSS} \rightarrow \texttt{dadsR} \parallel \texttt{ssSS} \rightarrow \texttt{dadsR}$$

$$\texttt{CH}_4^3 = \texttt{dads} \oplus \{l_i : \texttt{dadsSS} \rightarrow \texttt{hm}_\texttt{i}\texttt{R} \bullet \texttt{hm}_\texttt{i}\texttt{SS} \rightarrow \texttt{desR}\}$$

$$\texttt{CH}_5 = \texttt{CH}_5^1 \bullet \texttt{CH}_5^2 \bullet \texttt{CH}_5^3$$

$$\texttt{CH}_5^1 = \texttt{ddsAS} \rightarrow \texttt{dusR} \bullet \texttt{ddsAS} \rightarrow \texttt{SSR}$$

$$\texttt{CH}_5^2 = \texttt{dusSS} \rightarrow \texttt{ddsR} \parallel \texttt{dmsSS} \rightarrow \texttt{dadsR}$$

$$\texttt{CH}_5^3 = \texttt{dds} \oplus \{l_i : \texttt{ddsSS} \rightarrow \texttt{hd}_\texttt{i}\texttt{R} \bullet \texttt{hd}_\texttt{i}\texttt{SS} \rightarrow \texttt{desR}\}$$

$$\texttt{CH}_6 = \texttt{desAS} \rightarrow \texttt{eisR}$$

Given the global choreography and the code mapped to each of the ports, we automatically synthesize the code of each component. Note that, in practice, the above choreography may be updated to adjust the network speed, availability of

resources, or adding/removing new micro-services. This would require a drastic effort to re-implement the communication logic between components, which is tedious and error-prone and a very time-consuming task. Using our methodology, we only require to update the global choreography, and then automatically generate the implementation of the components.

Chapter 6

Transformation to Promela

Given a system $S = (B, \texttt{init})$, where $B = \gamma(B_1, \ldots, B_n)$, produced by applying the set of transformation corresponding to a given choreography `ch`, we define a translation of S into `Promela` [10], which can be verified with respect to properties specified in Linear Temporal Logic (LTL). We mainly define two functions (1) `createChannels`, which generates global channels (in Promela) that are used to transfer messages between processes; (2) `createProcess`, which generates the code that corresponds to each of the components. We use the `append` call to add code Promela code to the generate file. Listing 6.1 depicts the global structure of the code generation for a system S to Promela.

```
createPromela() {
  createChannels();
  foreach Bi {
    createProcess(i);
  }
}
```

Listing 6.1: Main Code Generation from System S to Promela

The functions are defined as follows:

1. `createChannels`: For every receive port, we create a channel (Promela's message carrier type). The type of the channel is the data type of the corresponding send port (i.e., p.`dtype`). For synchronous (resp. asynchronous) ports, we use a channel of length 0 (`MAX_LEN`). The main skeleton of the `createChannels` is depicted in Listing 6.2.

```
1  createChannels()
2    foreach a ∈ γ, where a = (p_s, {p_r^i}_{i∈I}) {
3      foreach p ∈ {p_r^i}_{i∈I} {
4        if (isSSend(p_s))
5          append chan channelP = [0] of {ps.dtype};
6        else
7          append chan channelP = [MAX_LEN] of
               {ps.dtype};
8        end
9      end
10   end
```

Listing 6.2: `createProcess` Skeleton

2. `createProcess`: For every component B_i, we create a process in Promela containing: (1) a variable that will hold the current state of the component, which is initialized to the initial state of the component; (2) the code generated of the LTS implementation of the component. The main skeleton of the `createProcess` is depicted in Listing 6.3.

```
createProcess(int id) {
  append proctype process(int id) {
    append int currentState = initialState;
    append currPort = _;
    append do
    append :: if
    append :: (all current outgoing trans. are
       send) ->
      append p_s = pickEnablePort(); // w.r.t. guard
      append currPort = p_s;
      foreach  p ∈ {p_r^i}_{i∈I}, where ∃a = (p_s, {p_r^i}_{i∈I}) {
        append channelP!(msg);
      }
      append if
      append :: (all outgoing are synchronous
         send) ->
        foreach  p ∈ {p_r^i}_{i∈I}, where ∃a = (p_s, {p_r^i}_{i∈I}) {
          append channelP?(_);
        }
      append fi;
      append ::  else -> // outgoing transitions
         are receive
        // listening to all current channels
        append if
         foreach p: currentState --p-->
```

```
            append ::(channelP?(val)) -> currPort
               = p;
            if(p is connected to synchronous send)
               {
               append channelP!(ack);
            }
         append fi;
      append fi;
      append doComputation(currentState);
      append currentState = updateCurrentState();
   append od;
   append }
}
```

Listing 6.3: `createProcess` Skeleton

6.1 Example

Consider a system consisting of four components: Buyer 1 (B_1), Buyer 2 (B_2), Seller (S) and Bank (Bk). Buyer 1 sends a book title to the Seller, which replies to both buyers by quoting a price for the given book. Depending on the price, Buyer 1 may try to haggle with Seller for a lower price, in which case Seller may either accept the new price or call off the transaction entirely. At this point, Buyer 2 takes Seller's response and coordinates with Buyer 1 to determine how much each should pay. In case Seller chose to abort, Buyer 2 would also abort. Otherwise, it would keep negotiating with Buyer 1 to determine how much it

should pay. Buyer 1, having a limited budget, consults with the bank before replying to Buyer 2. Once Buyer 2 deems the amount to be satisfactory, he will ask the bank to pay the seller the agreed upon amount (Buyer 1 would be doing the same thing *in parallel*). The global choreography `ch` can then be described as in Listing 6.4.

Listing 6.4: Global choreography of the Buyer/Seller example

```
CH  = B1.S→S.R • S.S → {B1.R, B2.R} • B1 ⊕ {CH1, ϵ} • CH2 •
      CH7
CH1 = B1.S→S.R • S.S → {B1.R, B2.R}
CH2 = B2 ⊕ {CH3, ϵ}
CH3 = while(B2.C) {
          B1.C→ Bk.InfR • Bk.InfS → B1.R • B1.C→ B2.R
      } • CH4
CH4 = CH5 ‖CH6
CH5 = B2.MS→Bk.MR2 • Bk.MS2 → S.R
CH6 = B1.MS→Bk.MR1 • Bk.MS1 → S.R
CH7 = B1.E → ϕ ‖ B2.E → ϕ ‖ Bk.E → ϕ ‖ S.E → ϕ
```

In the above choreography, we prefix the names of the ports by the owning components. Each port maps to a different functionality in the system so that, for example, `Bk.InfR` and `Bk.InfS` represent an enquiry handling interface. $\texttt{B}_i\texttt{.S}$ and $\texttt{B}_i\texttt{.R}$ represent simple message send/receive interfaces for Buyer i (similarly for `S.S` and `S.R`). We now define the choreography's translation to Promela. In the absence of procedures, we define the following macros for convenience and clarity. All of these macros accept a Promela channel (`ch`). We assume that `value` is a variable that will contain the value that needs to be sent.

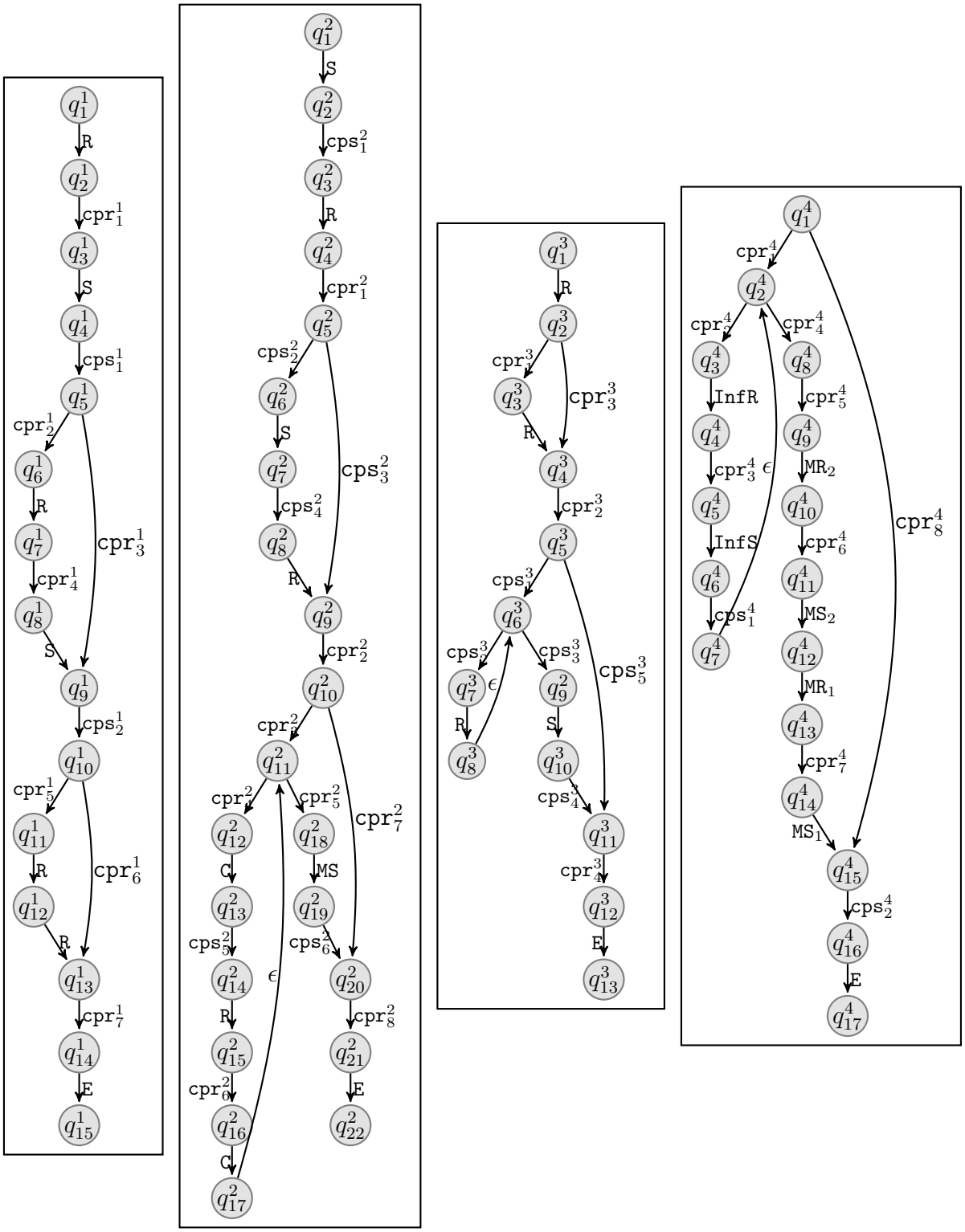


Figure 6.1: Components generated from the choreography in Listing 6.4

```
#define recv(ch) ch?value
#define recvAck(ch) ch?(_)
#define send(ch) ch!value
#define sendAck(ch) ch!ack
#define synchRecv(ch) ch?value; sendAck(ch)
```

Listing 6.5: Promela Macros

With the macros defined in Listing 6.5, the Promela code generated is depicted in Listing 6.6.

```
proctype Seller() {
  int currentState = q_1^1;
  currPort = _;
  int value;
  do
  ::if
    ::(currentState == q_1^1) -> synchRecv(S.R); currPort = S.R; currentState = q_2^1;
    ::(currentState == q_2^1) -> synchRecv(cpr_1^1); currPort = cpr_1^1; q_3^1;
    ::(currentState == q_3^1) -> send(B_1.R); send(B_2.R); recvAck(R_1.R); recvAck(B_2.R); currPort
        = S.S; currentState = q_4^1;
    ::(currentState == q_4^1) -> send(cpr_1^2); recvAck(cpr_1^2); currPort = cps_1^1 currentState = q_5^1;
    ::(currentState == q_5^1) ->
      if
      ::recv(cpr_2^1) -> sendAck(cpr_2^1); currPort = cpr_2^1; currentState = q_6^1;
      ::recv(cpr_3^1) -> sendAck(cpr_3^1); currPort = cpr_3^1; currentState = q_9^1;
      fi;
    ::(currentState == q_6^1) -> synchRecv(S.R); currPort = S.R; currentState = q_7^1;
    ::(currentState == q_7^1) -> synchRecv(cpr_4^1); currPort = cpr_4^1; currentState = q_8^1;
    ::(currentState == q_8^1) -> send(B_1.R); send(B_2.R); recvAck(B_1.R); recvAck(B_2.R); currPort
        = S.S; currentState = q_9^1;
    ::(currentState == q_9^1) -> send(cpr_2^3); recvAck(cpr_2^3); currPort = cps_2^1; currentState = q_10^1;
    ::(currentState == q_10^1) ->
      if
      ::recv(cpr_5^1) -> sendAck(cpr_5^1); currPort = cpr_5^1; currentState = q_11^1
      ::recv(cpr_6^1) -> sendAck(cpr_5^1); currPort = cpr_6^1; currentState = q_14^1
      fi;
    ::(currentState == q_11^1) -> synchRecv(S.R); currPort = S.R; currentState = q_12^1;
    ::(currentState == q_12^1) -> synchRecv(S.R); currPort = S.R; currentState = q_13^1;
    ::(currentState == q_13^1) -> synchRecv(cpr_7^1); currPort = cpr_7^1; currentState = q_14^1;
    ::(currentState == q_14^1) -> currPort = S.E; currentState = end;
    ::(currentState == end) -> break;
    fi;
    doComputation(currentState);
  od;
}
```

Listing 6.6: Seller Process in Promela

`doComputation` is a macro that takes in the `currentState` (an `int`) and performs the appropriate computation. The result of this computation would then be stored in the variable `value`.

6.2 Properties

We prefix variables local to processes with the the name of the process. Given the generated program, we verify the following properties expressed in LTL:

- *Correct termination*: All processes terminate correctly together: Let the ports suffixed by `E` represent the termination of the corresponding process. Let $\texttt{currPort}_1$ `= Buyer1.currPort,` $\texttt{currPort}_2$ `= Buyer2.currPort,` $\texttt{currPort}_3$ `= Bank.currPort,` $\texttt{currPort}_4$ `= Seller.currPort`. Then, correct termination can be expressed as the LTL formula:

$$G(\bigvee_{i=1}^{4}(\texttt{currPort}_i = E_i) \implies \bigwedge_{i=1}^{4}(\texttt{currPort}_i = E_i))$$

  where $\texttt{E}_\texttt{i}$ represents the ending interface of the appropriate process.

- *Livelock*: Progress must be made towards termination (i.e. There are no cyclic paths with no work accomplished). Intuitively, the system is in a state of live-lock if the port `Bk.InfR` is used infinitely often along an execution path. Therefore, specifying that the system is free of live-lock can be modeled as the LTL formula:

$$\neg(GF(\texttt{Bank.currPort} = \texttt{Bk.InfR}))$$

- An interface is *only called once*. In each run, money is only withdrawn

once by each process. Let the port $\texttt{Bk.MS}_1$ (resp. $\texttt{Bk.MS}_2$) represent the withdrawal of money by process 1 (resp. process 2). Then, specifying that money is withdrawn once per process can be expressed as the LTL formula:

$$O_1 \wedge O_2$$

where

$$O_i = G((\texttt{Bank.currPort} = \texttt{Bk.MS}_i) \implies XG(\neg \texttt{Bank.currPort} = \texttt{Bk.MS}_i))$$

- Money is only withdrawn either *after* Buyer1 or Buyer 2 makes a request. Let the ports $\texttt{Bk.MS}_i$ be as above and let $\texttt{B}_i\texttt{.MS}$ represent money transfer requests by Buyer i. Then specifying the order of execution is represented by the following LTL formula:

$$B_1 \wedge B_2$$

where

$$B_i = G((\neg(\texttt{Bank.currPort} = \texttt{Bk.MS}_i)) \ \mathbf{U} \ (\texttt{Bank.currPort} = \texttt{B}_\texttt{i}\texttt{.MS}))$$

Chapter 7

Design flow

After establishing the model and defining the code generation scheme, our newly proposed methodology would proceed as follows:

1. Users need to identify the interfaces/ports that will be used to communicate between different components in the system.

2. After identifying interfaces, a choreography modeling the identified behavior has to be written following the syntax provided in Listing 4.1. The choreography describes the communication logic between a set of provided processes/components using only their interfaces.

3. The user must then (1) for each component, specify its variables; and (2) for each port/interface, implement the actual function that must be executed upon referencing the corresponding port.

4. The next step is to feed code generator the choreography, along with a mapping from the ports to the actual implementations developed in Step 3.

5. All logical properties that are to be checked need to be expressed in terms of the ports used in the choreography.

6. The generated model is then checked for correctness against the specified properties (using a model-checker such as SPIN for Promela). Any violations are then analyzed and used to refine the interaction model in the first step.

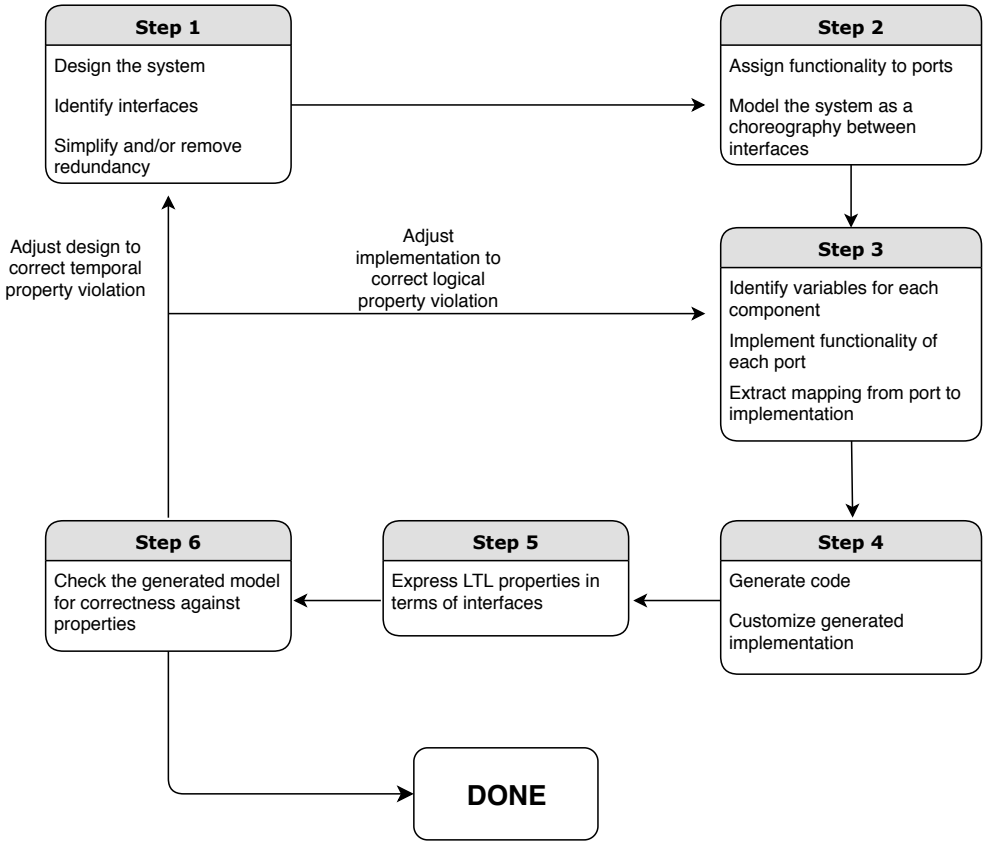


Figure 7.1: Proposed work flow

From the proposed work flow, a clear separation of concerns is apparent. In the first phase (Steps 1 and 2), users focus attention on designing the overall architecture of the system. Because of the focus on design, users can easily spot any redundancy and/or logical errors in their design and can easily rectify them. In the next phase, users focus on implementing their application in terms of interfaces (ports). Here, no communication logic is embedded into the implementation, enabling users to focus on traditional software concerns. This is in stark contrast to the traditional approach where communication and computation logic is intertwined in the implementation and significant complexity is introduced to the development process.

In the last phase, users assess the quality of the generated code in terms of correctness w.r.t to a formula expressed in Linear Temporal Logic (LTL). In case of a violation, users can immediately identify the source of error, whether in the design of the system or in the implementation, and immediately re-design/re-implement the cause of the violation.

Chapter 8

Conclusion and Future Work

Conclusion In this work, we proposed a new programming methodology for distributed systems. We first established a component based framework to model a distributed system, and formalized its description and is behavior. Next, we specified a set of transformations that can be applied to the system and defined their semantics in terms of the framework. These transformations mirror the traditional programming control flow statements and extend them to the realm of distributed systems. We can then use a DSL to specify a *choreography* that determines how a distributed system should behave. We presented a simple code generation scheme that takes as input a choreography and a mapping from ports to desired functionality and then outputs a model of the system in a language amenable to model checking and gave examples about certain properties that might be interesting to users.

This work deals with synthesizing distributed implementations of local processes. These implementations encompass everything from control flows and synchronization to verification of correctness, starting from a global choreography. This allows us to easily reason about correctness of communication protocols,

and update them without requiring changes to local processes. The proposed model for synthesizing these implementations offers many advantages, in that it requires less effort to build and promotes modular efficient design of systems by construction. Moreover, the global choreography is expressive enough to model real-world distributed applications, and we use it to synthesize micro-services architectures.

Future work Future work comprises several directions. First, we consider augmenting our choreography model by adding fault-tolerance primitives. That is, we can specify the number of replicas of each process and automatically embed a consensus protocol between them such as Paxos [25] or Raft [26]. This would further abstract away the need for users to handle fault tolerance on their side. Second, we consider integrating our framework with Spring Boot to allow for the automatic generation of RESTful web services starting from global choreography. We also consider augmenting our code generation with features provided by *Istio* [27] and *Linkerd* [28], which are used for routing, failure handling, service discovery, integration of micro-services, manage traffic flow and enforcing policies.

Bibliography

[1] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," 2010.

[2] 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.

[3] A. Bejleri and N. Yoshida, "Synchronous multiparty session types," *Electr. Notes Theor. Comput. Sci.*, vol. 241, pp. 3–33, 2009.

[4] 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*, pp. 273–284, 2008.

[5] 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*, pp. 240–256, 2007.

[6] 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.

[7] 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*, pp. 299–312, 2010.

[8] M. Charalambides, P. Dinges, and G. A. Agha, "Parameterized, concurrent session types for asynchronous multi-actor interactions," *Sci. Comput. Program.*, vol. 115-116, pp. 100–126, 2016.

[9] Murex. https://www.murex.com.

[10] G. J. Holzmann, "The model checker SPIN," *IEEE Trans. Software Eng.*, vol. 23, no. 5, pp. 279–295, 1997.

[11] G. Coulouris, J. Dollimore, and T. Kindberg, *Distributed Systems: Concepts and Design (International Computer Science)*. Addison-Wesley Longman, Amsterdam, 4th rev. ed. ed., 2005.

[12] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process," *J. ACM*, vol. 32, pp. 374–382, Apr. 1985.

[13] G. Ricart and A. K. Agrawala, "An optimal algorithm for mutual exclusion in computer networks," *Commun. ACM*, vol. 24, pp. 9–17, Jan. 1981.

[14] E. M. Clarke, E. A. Emerson, and A. P. Sistla, "Automatic verification of finite-state concurrent systems using temporal logic specifications," *ACM Trans. Program. Lang. Syst.*, vol. 8, pp. 244–263, Apr. 1986.

[15] G. A. Agha and W. Kim, "Actors: A unifying model for parallel and distributed computing," *Journal of Systems Architecture*, vol. 45, no. 15, pp. 1263–1277, 1999.

[16] A. Basu, S. Bensalem, M. Bozga, J. Combaz, M. Jaber, T. Nguyen, and J. Sifakis, "Rigorous component-based system design using the BIP framework," *IEEE Software*, vol. 28, no. 3, pp. 41–48, 2011.

[17] E. Tuosto and R. Guanciale, "Semantics of global view of choreographies," *J. Log. Algebr. Meth. Program.*, vol. 95, pp. 17–40, 2018.

[18] J. Lange and E. Tuosto, "Synthesising choreographies from local session types," in *CONCUR 2012 - Concurrency Theory - 23rd International Conference, CONCUR 2012, Newcastle upon Tyne, UK, September 4-7, 2012. Proceedings*, pp. 225–239, 2012.

[19] J. Lange, E. Tuosto, and N. Yoshida, "From communicating machines to graphical choreographies," in *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pp. 221–232, 2015.

[20] P. C. Attie and C. Das, "Automating the refinement of specifications for distributed systems via syntactic transformations," *Int. J. Systems Science*, vol. 28, no. 11, pp. 1129–1144, 1997.

[21] N. Francez and I. R. Forman, "Synchrony loosening transformations for interacting processes," in *CONCUR '91, 2nd International Conference on*

Concurrency Theory, Amsterdam, The Netherlands, August 26-29, 1991, Proceedings, pp. 203–219, 1991.

[22] OMG, Business Process Model and Notation (BPMN), Version 2.0 http://www.omg.org/spec/BPMN/2.0/, January 2011.

[23] A. Meyer, L. Pufahl, K. Batoulis, D. Fahland, and M. Weske, "Automating data exchange in process choreographies," *Inf. Syst.*, vol. 53, pp. 296–329, 2015.

[24] Z. Manna and P. Wolper, "Synthesis of communicating processes from temporal logic specifications," in *Logics of Programs* (D. Kozen, ed.), (Berlin, Heidelberg), pp. 253–281, Springer Berlin Heidelberg, 1982.

[25] L. Lamport, "Paxos made simple," pp. 51–58, December 2001.

[26] D. Ongaro and J. K. Ousterhout, "In search of an understandable consensus algorithm," in *2014 USENIX Annual Technical Conference, USENIX ATC '14, Philadelphia, PA, USA, June 19-20, 2014.*, pp. 305–319, 2014.

[27] Istio. https://github.com/istio/istio/.

[28] Linkerd. https://linkerd.io.