AMERICAN UNIVERSITY OF BEIRUT

A METHOD FOR VERIFYING CHOREOGRAPHIES AND THEIR IMPLEMENTATIONS

by

AL-ABBASS ADHAM KHALIL

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
April 2019

AMERICAN UNIVERSITY OF BEIRUT

A METHOD FOR VERIFYING CHOREOGRAPHIES AND THEIR IMPLEMENTATIONS

by

AL-ABBASS ADHAM KHALIL

Approved by:

Dr. Paul Attie, Professor — Advisor
Computer Science

for Mohamad Jaber

Dr. Mohamad Jaber, Assistant Professor — Member of Committee
Computer Science

Dr. Amer Mouawad, Assistant Professor — Member of Committee
Computer Science

Date of thesis defense: April 30, 2019

AMERICAN UNIVERSITY OF BEIRUT

THESIS, DISSERTATION, PROJECT RELEASE FORM

Student Name: Khalil Al-Abbass Adham

Student Name:	Khalil	Al-Abbass	Adham
	Last	First	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.

Al-Abbass May 6, 2019

Signature Date

Acknowledgements

I would like to express my deepest thanks to my advisor Professor Paul Attie for the support and mentorship of my research, for his motivation, and immense knowledge. Throughout my time at the department it was a pleasure and honour to have been taught by and alongside him. This work would not have been possible without his guidance and continuous effort.

I also would not be where I am today without the support of my family and friends. Their love and guidance are the reason I was able to complete this thesis.

An Abstract of the Thesis of

Al-Abbass Adham Khalil for Masters of Science
Major: Computer Science

Title: A Method for Verifying Choreographies and Their Implementations

A global choreography defines a communication pattern over a set of ports. The ports are partitioned into subsets, each subset being the ports that belong to a given process. From a choreography and an interaction architecture, a distributed implementation can be generated automatically. The implementation can then be analyzed for correctness using standard methods such as model checking, but this is subject to state-explosion. A more efficient approach is to verify that the choreography is correct, and to establish that the implementation automatically inherits the correctness properties of the choreography. Because the choreography is centralized, analyzing it provides a more manageable abstract view and it incurs less state explosion. We present such an approach in this thesis, along with several case studies illustrating its advantages in practice.

Contents

List of Figures

Chapter 1

Introduction

1.1 Background

Proving the correctness of a hardware or software system is an integral step in achieving its reliability. Model checking is the process where given a model of a system, check if this system satisfies a given property. Some of the properties can be safety requirements such as absence of deadlock, satisfaction of integrity constraints, and the absence (non-reachability) of bad states that can cause the system to crash or malfunction.

As the system model becomes more complex, more states and variables are involved and thus we reach "state explosion", where the size of the system state space has grown exponentially with the number of processes and state variables, and the time it takes to model check the system becomes unfeasible. One of the methods to reduce this complexity is to use *abstraction*, where first we simplify the system before proving its properties. However, the simplified system does not necessarily have the same properties as the original. The task of model checking using abstraction is difficult because:

1. if the abstraction is too coarse, it will violate the desired properties, even though the concrete system may be correct ("false negative"), and

2. if the abstraction is too fine, it will still be too large to allow efficient verification.

The challenge then is to find a faithful abstraction (which satisfies the desired properties) but is not too large. An alternative approach is to start from the abstraction, and to automatically generate a distributed implementation, by means of *correctness preserving transformations*. One such approach is presented by Hallal and Jaber [1], who give a methodology to automatically synthesize an efficient distributed implementation starting from a *high level global choreography*.

The global choreography describes a set of processes and their interactions together, and from it, we can generate a distributed system. This system can be model checked using Promela, but this suffers from the same problem of state explosion. Thus, we wish to model check the global choreography before generating the distributed implementation. Together with a theory which establishes that the generated implementation *automatically* inherits correctness properties from the choreography, we solve the verification problem.

1.2 Objective

The aim of this thesis is as follows:

1. Given a description of a global choreography, represent it as a Kripke structure and model check this structure with respect to a specification written in the temporal logic CTL [2].

2. Devise a theory which shows that satisfaction of formulae by the global choreography implies satisfaction of the same (or related) formulae by the implemented system. The formula should be drawn from a suitably interesting sublogic of CTL.

The representation and model checking of Kripke structures will be done using the tool Eshmun [3], which can be downloaded from `http://eshmuntool.blogspot.com/`.

1.3 List of Tasks

In order to achieve these objectives, we shall perform the following tasks:

1. Define a semantics for choreographies. The semantics will be given as a set of structured operational semantics rules, which take a pair consisting of a choreography and state (assignment to variables), and produce a new choreography and state as a result of executing the action described by the rule.

2. Using this semantics, we devise an algorithm for generating the Kripke structure of a choreography. We assume that choreographies are finite-state, so that the Kripke structure can be generated by simulating all possible behaviors of the choreography, until no new behaviors are produced. The resulting Kripke structure can then be model checked in Eshmun.

3. Devise *property-preservation results* which state that properties (expressed in CTL) of the choreography are also properties of the implementation.

Hence model checking of the choreography verifies properties of the implementation, i.e., the generated code. We expect these results to follow from the process of constructing the state-transitions of the components of the distributed implementation, as given in Chapter 4 of [1].

4. Apply the method to many case studies.

Chapter 2

Literature Review

2.1 Model of Concurrent Computation

We use the version of BIP that is given in [1].

In this section, we introduce a component-based framework, inspired from the Behavior Interaction Priority framework (BIP) [4]. In the BIP framework, atomic components communicate through an 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 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 consists of the following elements:*

- *a port identifier p;*
- *its data type $p.\mathtt{dtype} \in \{\mathtt{int}, \mathtt{str}, \mathtt{bool}, \ldots\}$; and*
- *its communication type denoted by $p.\mathtt{ctype}$ ranging in the set $\{\mathtt{ss}, \mathtt{as}, \mathtt{r}, \mathtt{in}\}$, where $\mathtt{ss}$ (resp. $\mathtt{as}, \mathtt{r}, \mathtt{in}$) denotes a synchronous send (resp. asynchronous send, receive, internal) communication type.*

Moreover, a receive port p has field $p.\mathtt{buff} \in \mathbb{N}$ denoting the number of signals/data pending on that port.

Given a port p, we define the predicate $\mathtt{isSSend}(p)$ (resp., $\mathtt{isASend}$, $\mathtt{isRecv}$, $\mathtt{isInternal}$) that holds $\mathtt{true}$ iff (the communication type of) 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.*

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., $p_j \in \mathcal{P}_j^r$.*

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. The semantics of a composite component is defined as a labeled transition system where the transitions depend on the interaction types (see Figure **??**). First, Equation (2.1) 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, Equation (2.2) and Equation (2.4) represent asynchronous interactions. Equation (2.2) 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, (2.3) represents the autonomous execution of receive ports with no empty buffers. Finally, Equation 2.4 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. Component $B = \gamma(B_1, \ldots, B_n)$ 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 rules in Figure* **??**.

$$\text{synch-send:}\frac{\begin{array}{c} a = (p_i, \{p_j\}_{j \in J}) \in \gamma \quad \texttt{isSSend}(p_i) \\ \forall k \in J \cup \{i\} : q_k \xrightarrow{p_k} q'_k \quad \forall k \notin J \cup \{i\} : q_k = q'_k \\ \forall k \in J : p_k.\texttt{buff} = 0 \end{array}}{(q_1, \ldots, q_n) \xrightarrow{a} (q'_1, \ldots, q'_n)} \tag{2.1}$$

$$\text{asynch-send:}\frac{\begin{array}{c} a = (p_i, \{p_j\}_{j \in J}) \in \gamma \\ \texttt{isASend}(p_i) \quad q_i \xrightarrow{p_i} q'_i \quad \forall k \neq i : q_k = q'_k \end{array}}{\begin{array}{c} (q_1, \ldots, q_n) \xrightarrow{a} (q'_1, \ldots, q'_n) \\ \forall j \in J : p_j.\texttt{buff} := p_j.\texttt{buff} + 1 \end{array}} \tag{2.2}$$

$$\text{recv:}\frac{q_j \xrightarrow{p_j} q'_j \quad \texttt{isRecv}(p_j) \quad p_j.\texttt{buff} > 0 \quad \forall k \neq j : q_k = q'_k}{(q_1, \ldots, q_n) \xrightarrow{a} (q'_1, \ldots, q'_n) \quad p_j.\texttt{buff} := p_j.\texttt{buff} - 1} \tag{2.3}$$

$$\text{internal:}\frac{q_i \xrightarrow{p_i} q'_i \quad \texttt{isInternal}(p_i) \quad \forall k \neq i : q_k = q'_k}{(q_1, \ldots, q_n) \xrightarrow{\epsilon} (q'_1, \ldots, q'_n)} \tag{2.4}$$

Figure 2.1: Semantic rules defining the behavior of composite components

Finally, a system is defined as a composite component where we specify the initial states 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*.*

2.2 CTL Syntax and Semantics

We use the temporal logic CTL to specify correctness properties.

Let AP be a set of atomic propositions, including the constants trueand false. We use true, falseas "constant" propositions whose interpretation is always the semantic truth values tt, ff, respectively. The logic CTL [?, 2] is given by the following grammar:

$$\varphi ::= \texttt{true} \mid \texttt{false} \mid p \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \mathsf{AX}\varphi \mid \mathsf{EX}\varphi \mid \mathsf{A}[\varphi \mathsf{R} \varphi] \mid \mathsf{E}[\varphi \mathsf{R} \varphi]$$

where $p \in AP$, and $\texttt{true}, \texttt{false}$ are constant propositions with interpretation $\texttt{tt}, \texttt{ff}$ respectively (i.e.,"syntactic" true, false respectively).

The semantics of CTL formulae are defined with respect to a Kripke structure.

Definition 6. *A Kripke structure is a tuple $M = (S_0, S, R, L)$ where S is a finite state of states, $S_0 \subseteq S$ is a set of initial states, $R \subseteq S \times S$ is a transition relation, and $L : S \mapsto 2^{AP}$ is a labeling function that associates each state $s \in S$ with a subset of atomic propositions, namely those that hold in the state. State t is a* sucessor *of state s in M iff $s, t \in R$.*

We assume that a Kripke structure $M = (S_0, S, R, L)$ is total, i.e.,$\forall s \in S, \exists s' \in S : (s, s') \in R$. A path in M is a (finite or infinite) sequence of states, $\pi = s_0, s_1, \ldots$ such that $\forall i \geq 0 : (s_i, s_{i+1}) \in R$. A fullpath is an infinite path. A state is reachable iff it lies on a path that starts in an initial state. Without loss of generality, we assume in the sequel that the Kripke structure M that is to be repaired does not contain any unreachable states, i.e.,every $s \in S$ is reachable.

Definition 7. *$M, s \models \varphi$ means that formula φ is true in state s of structure M and $M, s \not\models \varphi$ means that formula φ is false in state s of structure M. We define $\models$ inductively as usual:*

1. $M, s \models \texttt{true}$
2. $M, s \not\models \texttt{false}$
3. $M, s \models p$ *iff* $p \in L(s)$ *where atomic proposition* $p \in AP$
4. $M, s \models \neg\varphi$ *iff* $M, s \not\models \varphi$
5. $M, s \models \varphi \wedge \psi$ *iff* $M, s \models \varphi$ *and* $M, s \models \psi$
6. $M, s \models \varphi \vee \psi$ *iff* $M, s \models \varphi$ *or* $M, s \models \psi$
7. $M, s \models \mathsf{AX}\varphi$ *iff for all* t *such that* $(s, t) \in R : (M, t) \models \varphi$
8. $M, s \models \mathsf{EX}\varphi$ *iff there exists* t *such that* $(s, t) \in R$ *and* $(M, t) \models \varphi$
9. $M, s \models \mathsf{A}[\varphi \mathsf{R} \psi]$ *iff for all fullpaths* $\pi = s_0, s_1, \ldots$ *starting from* $s = s_0$*:* $\forall k \geq 0 : (\forall j < k : (M, s_j \not\models \varphi)$ *implies* $M, s_k \models \psi$
10. $M, s \models \mathsf{E}[\varphi \mathsf{R} \psi]$ *iff for some fullpath* $\pi = s_0, s_1, \ldots$ *starting from* $s = s_0$*:* $\forall k \geq 0 : (\forall j < k : (M, s_j \not\models \varphi)$ *implies* $M, s_k \models \psi$

We use $M \models \varphi$ to abbreviate $M, S_0 \models \varphi$. We introduce the abbreviations $\mathsf{A}[\phi \mathsf{U} \psi]$ for $\neg\mathsf{E}[\neg\varphi \mathsf{R} \neg\psi]$, $\mathsf{E}[\phi \mathsf{U} \psi]$ for $\neg\mathsf{A}[\neg\varphi \mathsf{R} \neg\psi]$, $\mathsf{AF}\varphi$ for $\mathsf{A}[\texttt{true}\mathsf{U}\varphi]$, $\mathsf{EF}\varphi$ for $\mathsf{E}[\texttt{true}\mathsf{U}\varphi]$, $\mathsf{AG}\varphi$ for $\mathsf{A}[\texttt{false}\mathsf{R}\varphi]$, $\mathsf{EG}\varphi$ for $\mathsf{E}[\texttt{false}\mathsf{R}\varphi]$.

Chapter 3

Operational Semantics of Choreographies

3.1 Choreography Grammar

We introduce the grammar of choreographies. The following description is from [1]:

> 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 3.1 depicts the abstract syntax of the choreography model.

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.

The binary operator • allows to sequentially compose two choreographies, $\mathtt{ch}_1 \bullet \mathtt{ch}_2$

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

Branching 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 the appropriate components to follow the taken choice (i.e.,

```
ch  ::=  snd  → rcvs  :  <T>  # send/receive
 | $B_i$ ⊕ {snd$_j$:ch}$_{j \in J}$ # Branching - where snd$_j$ $\in \mathcal{P}_i^{ss}$
 |   while(snd) ch end # loop
 |   ch • ch # sequential
 |   ch ∥ ch # parallelism
 |   $\epsilon$

snd  ::=  $p_i$  # sender - where $p_i \in \mathcal{P}_i^{ss} \cup \mathcal{P}_i^{as}$

rcvs  ::=  $p_i$  | $p_i$ rcvs # receivers - where $p_i \in \mathcal{P}_i^{r}$

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

Listing 3.1: Abstract syntax of the global choreography model

the corresponding choreography, $\texttt{ch}_l$).

Loop allows for the modeling of a conditional repeated choreography `ch`. The condition is evaluated by a specific component, which will notify the participants of the choreography to either re-execute it or break.

3.2 Choreography Control Predicates

For the following semantics we use these notations:

$\mathbf{F}_p(\sigma)$: is the state that results from executing a transition labeled with port p in state σ

G : guard that enables a transition if the state satisfies it

snd, p : ports

rcvs : set of ports

$\mathbf{q}_i^j$: state i of component j

ch : choreography

ch.st : set to true after first event of ch

ch.end : set to true after last event of ch

ch.in : ch.st $\wedge$ $\neg$ch.end

fires : ch fires p iff ch executes a transition labelled with port p, i.e., snd $\rightarrow$ rcvs where p $=$ snd or p $\in$ rcvs

$\widehat{p}$: port p is applied

ϵ : empty choreography

σ : state of the choreography

$\perp$: error state

$\uplus$: disjoint union operator, used between states

$\bullet$: sequential composition operator, execute one choreography and then another

$||$: parallel composition operator, execute two independent (i.e., do not share components) chorepgraphies in parallel

$\oplus$: branching composition operator, execute only one of the choreography options if its guard is satisfied

3.3 Operational Semantics

To express a notion of correct implementation of choreohraphies, we need a semantics for choreographies that is independent of the implementation. We present such a semantics, as a set of structured operational semantics rules.

3.3.1 Sequential

$$\text{sequential1:}\frac{(ch_1, \sigma) \xrightarrow{p} (ch_1', \sigma')}{(ch_1 \bullet ch_2, \sigma) \xrightarrow{p} (ch_1' \bullet ch_2, \sigma')}\sigma' = F_p(\sigma) \tag{3.1}$$

Equation (3.1) represents sequential choreographies. When ch_1 fires port p, state σ gets updated to σ' under function F and the choreography is updated to the next event, ch_1. For $ch_1 \bullet ch_2$, if ch_1 is not the empty choreograpghy, the current event of ch_1 applies port p, the state and choreography are updated accordingly to $(ch_1' \bullet ch_2, \sigma')$.

$$\text{sequential2:}\frac{}{(\epsilon \bullet ch, \sigma) \rightarrow (ch, \sigma)} \tag{3.2}$$

Equation (3.2) represents the end case for sequential choreographies. When the empty choreography is sequenced by another choreography $(\epsilon \bullet ch, \sigma)$, simply the pair is updated to the next choreography (ch, σ).

3.3.2 Parallel

$$\text{parallel1:}\frac{(ch_1, \sigma_1) \xrightarrow{p} (ch_1', \sigma_1')}{(ch_1 \parallel ch_2, \sigma_1 \uplus \sigma_2) \xrightarrow{p} (ch_1' \parallel ch_2, \sigma_1' \uplus \sigma_2)} \sigma_1' = F_p(\sigma_1) \tag{3.3}$$

Equation (3.3) represents parallel choreographies. When ch_1 fires port p, state σ_1 gets updated to σ_1' under function F and the choreography is updated to the next event, ch_1. The state of parallel choreographies is represented by the disjoint of the states of each of the involved choreographies. The choreography that is currently applying its port, updates its state. For $ch_1 \parallel ch_2$, if ch_1 is not the empty choreograpghy, the current event of ch_1 applies port p, the state and choreography are updated accordingly $(ch_1' \parallel ch_2, \sigma_1' \uplus \sigma_2)$.

$$\text{parallel2:}\frac{}{(\epsilon \parallel ch, \sigma) \rightarrow (ch, \sigma)} \tag{3.4}$$

Equation (3.4) represents the end case for parallel choreographies. When the empty choreography is paralleled with another choreography $(\epsilon \parallel ch, \sigma)$, simply the pair is updated to the next choreography (ch, σ).

3.3.3 Branching

$$\text{branching1:}\frac{}{(B_i \oplus G_j \& snd_j : ch_j, \sigma) \xrightarrow{snd_j} (ch_j, \sigma')} \sigma \models G_j, \sigma' = F_{snd_j}(\sigma) \tag{3.5}$$

Equation (3.5) represents the branching choreographies. If the state σ satisfies guard G_j, then it is possible to fire the port snd_j that G_j is guarding and to go to the respective choreography. The state is updated accordingly.

$$\text{branching2:}\frac{}{(B_i \oplus G_j \& snd_j : ch_j, \sigma) \rightarrow (\epsilon, \bot)} \sigma \not\models G_j \ \forall j \in J \tag{3.6}$$

Equation (3.6) represents an edge case. If the state σ satisfies none of the guards G_j, then move to the empty choreography and the error state.

3.3.4 While

$$\text{while:}\frac{}{(while(G \& snd)\ ch\ end, \sigma) \xrightarrow{snd} (ch \bullet while(G \& snd), \sigma')} \sigma \models G, \sigma' = F_{snd}(\sigma) \tag{3.7}$$

Equation (6.4) represents the while loop of choreographies. If the state σ satisfies guard G, then apply the snd_j that G is guarding and apply choreography ch sequenced by the loop again, the state is updated accordingly.

$$\text{whileEpsilon:}\frac{}{(while(G \& snd)\ ch\ end, \sigma) \rightarrow (\epsilon, \sigma)} \sigma \not\models G \tag{3.8}$$

Equation (3.8) represents the end case for while loop of choreographies. If the state σ does not satisfy the guard G, go to the empty choreography.

3.3.5 Send-Receive

Asynchronous Send

$$\text{sndRcvsAsynch1:}\ \frac{}{(snd \to rcvs, \sigma) \xrightarrow{snd} (rcvs, \sigma')}\ \sigma' = F_{snd}(\sigma) \tag{3.9}$$

If the sender is free, it will apply its port and update the state.

$$\text{sndRcvsAsynch2:}\ \frac{}{(rcvs, \sigma) \xrightarrow{p} (rcvs - p, \sigma')}\ p \in rcvs, \sigma' = F_p(\sigma) \tag{3.10}$$

Once a receiver is free, it applies its port and updates the state.

$$\text{sndRcvsAsynch3:}\ \frac{}{(p, \sigma) \xrightarrow{p} (\epsilon, \sigma')}\ \sigma' = F_p(\sigma) \tag{3.11}$$

Once the last receiver is free, it applies its port and updates the state. Go to the empty choreography.

Synchronous Send

$$\text{sndRcvsSynch:}\ \frac{}{(snd \to rcvs, \sigma) \xrightarrow{P_1 \ldots P_n} (\epsilon, \sigma')}\ \begin{array}{c} snd \cup rcvs = p_1 \ldots p_n, \\ \sigma' = F_{p_1}(F_{p_2}(\ldots Fp_n(\sigma))\ldots) \end{array} \tag{3.12}$$

The sender and receivers must be free, each applies its port and updates the state. Note that the resulting state σ' results from applying all of the update functions, for all of the ports, since the transition is synchronous. These update functions can be applied in any order, since they modify disjoint parts of the state.

Chapter 4

Kripke Generation

4.1 How to Write a Choreography

The following are rules on how the actual syntax of the input text file for generating the Kripke structure of a choreography should be.

Send-Receive : B.S > S.R, or S.S > B1.R, B2.R

Sequential : B.S > S.R * S.S > B1.R, B2.R

Loop : while(B2.C) B1.C > Bk.InfR ^Bk.InfS > B1.R, B2.R ^B2.S > B1.R
Note: inside loops we use ’^’ instead of ’*’ to mean sequential

Nested Loops : In order to have nested loops, the interior loop must be written in a different choreography
CH1 = while(N1.C) B1.S > S.R ^CH2 ^B2.S > S.R
CH2 = while(N2.C) H2.R > Z4.A ^B3.S > S.R ^B2 > C2.R

Branching : B1 + CH1, E E is an empty choreography where it is possible to ”skip“ the choreography

Branching inside Loops : In order to have branching inside loops, the branching must be written in a different choreography
CH4 = while(h1.te) P.R > I.PO ^CH7
CH5 = ZR1.S > B2.R
CH6 = ZR2.S > B2.R
CH7 = ZZ > RR * Br.1 + CH5, CH6, E

Parallel : Parallel choreographies are written in their own choreography and contain only sequential events
CH4 = CH5 || CH6
CH5 = B2.MS > Bk.MR2 * Bk.MS2 > S.R
CH6 = B1.MS > Bk.MR1 * Bk.MS1 > S.R

Comments : Start the line with 2 dashes ”–“ in order to ignore the line

4.2 Three-Process Example

We show an example of the generating algorithm.

4.2.1 The Choreography

Given 3 processes: P_1, P_2, and P_3. P_1 either asks P_2 or P_3 for information. They reply with the answer. Finally, all processes terminate

Listing 4.1: Three Process Example

$$
\begin{aligned}
&\mathtt{CH} = \mathtt{P}_1 \oplus \{\mathtt{CH}_1, \mathtt{CH}_2\} \bullet \mathtt{CH}_3 \\
&\mathtt{CH}_1 = \mathtt{P}_1.\mathtt{S} \rightarrow \mathtt{P}_2.\mathtt{R} \bullet \mathtt{P}_2.\mathtt{S} \rightarrow \mathtt{P}_1.\mathtt{R} \\
&\mathtt{CH}_2 = \mathtt{P}_1.\mathtt{S} \rightarrow \mathtt{P}_3.\mathtt{R} \bullet \mathtt{P}_3.\mathtt{S} \rightarrow \mathtt{P}_1.\mathtt{R} \\
&\mathtt{CH}_3 = \mathtt{P}_1 \rightarrow \phi \parallel \mathtt{P}_2 \rightarrow \phi \parallel \mathtt{P}_3 \rightarrow \phi
\end{aligned}
$$

4.2.2 Input File

```
-- This is a comment. A comment must start with 2 dashes "--"
-- and be at the start of a line
-- Each choreography is written on a single line.
-- The first choreography written will be the global choreography
-- The first event in the global choreography is the start state
-- Sequential events are delimited  by '*'
CH = P1 + {CH1, CH2} * CH3
CH1 = P1.S > P2.R * P2.S > P1.R
CH2 = P1.S > P3.R * P3.S > P1.R
-- Parallel choreographies are delimited by "||"
-- Parallel choreographies are written in their own choreography
-- Parallel choreographies contain sequential events only
CH3 = CH4 || CH5 || CH6
-- END represents the termination of a process
CH4 = P1 > END
CH5 = P2 > END
CH6 = P3 > END
```

4.2.3 Kripke Structure Generation

The following is the Kripke structure generated from the Three-Process-Example choreography. This was automatically generated by our implementation.

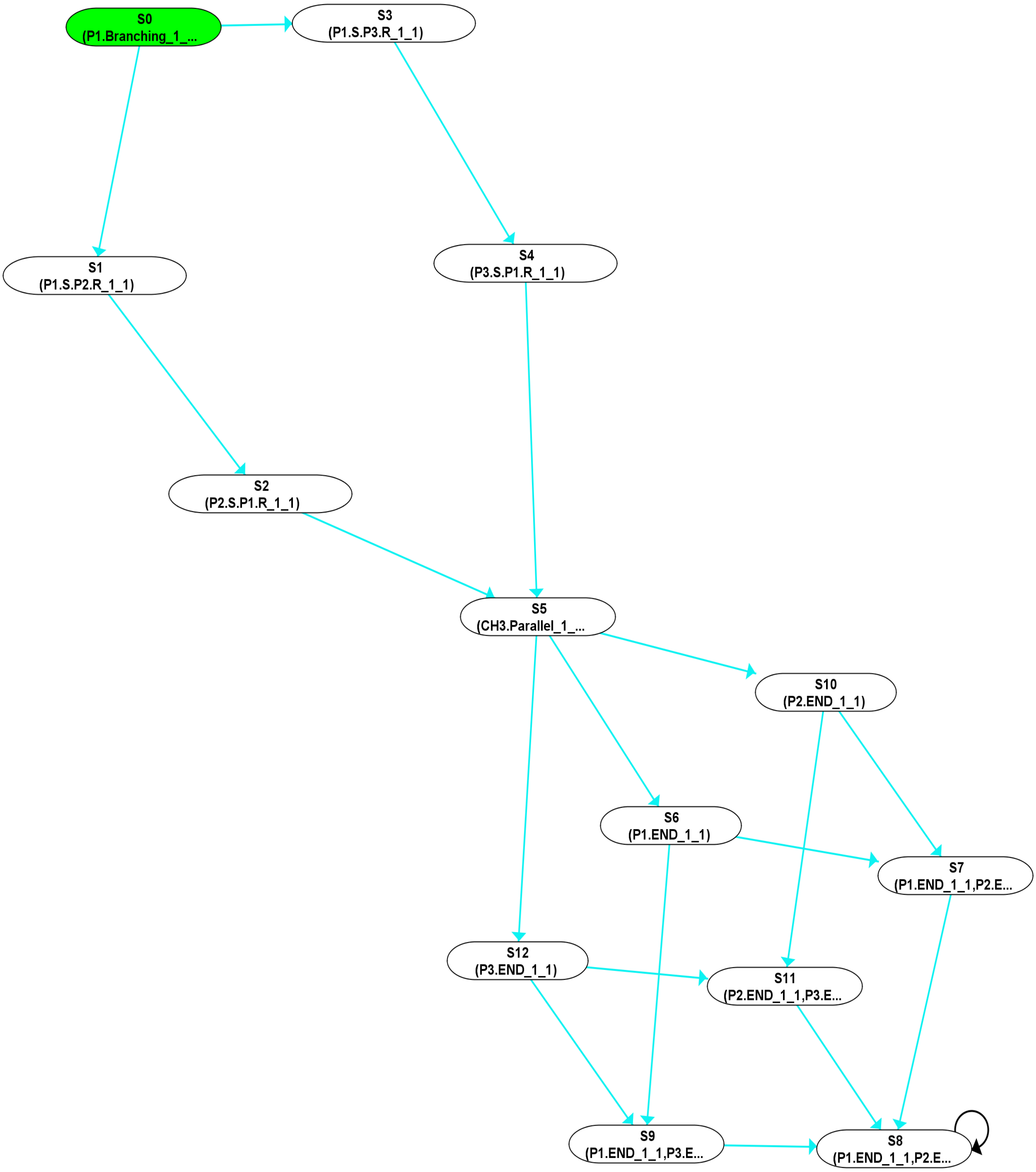


Figure 4.1: Three-Process example - Kripke Structure

4.2.4 Label Definitions

We have 3 components:

- Process 1
- Process 2
- Process 3

CTL formula are built up from atomic propositions, the usual boolean connectives, and temporal modalities. See section 2.2 and [2] for details. Each atomic proposition belongs to exactly one process. We make the convention that an atomic proposition belongs to a process if the last digit in the name of the atomic proposition is the index of the process.
The following shows the atomic propositions (comma separated) each state contains. The atomic propositions were auto generated by *name_i_*1 were name is the concatenation of the elements of an event dot separated, *i* is the counter so that the proposition is unique in case repetition of name, and 1 is to indicate that the atomic proposition belongs to process 1 (the global choreography).

```
states:
S0:P1.Branching_1_1
S1:P1.S.P2.R_1_1
S2:P2.S.P1.R_1_1
S3:P1.S.P3.R_1_1
S4:P3.S.P1.R_1_1
S5:CH3.Parallel_1_1
S6:P1.END_1_1
S7:P1.END_1_1,P2.END_1_1
S8:P1.END_1_1,P2.END_1_1,P3.END_1_1
S9:P1.END_1_1,P3.END_1_1
S10:P2.END_1_1
S11:P2.END_1_1,P3.END_1_1
S12:P3.END_1_1
```

We describe each atomic proposition:

P1.Branching_1_1 : Branching point where P1 either asks P2 or P3 for information

P1.S.P2.R_1_1 : P1 asks P2 for information

P2.S.P1.R_1_1 : P2 replies to P1

P1.S.P3.R_1_1 : P1 asks P3 for information

P3.S.P1.R_1_1 : P3 replies to P1

CH3.Parallel_1_1 : Parallel point where system terminates

P1.END_1_1 : P1 terminates

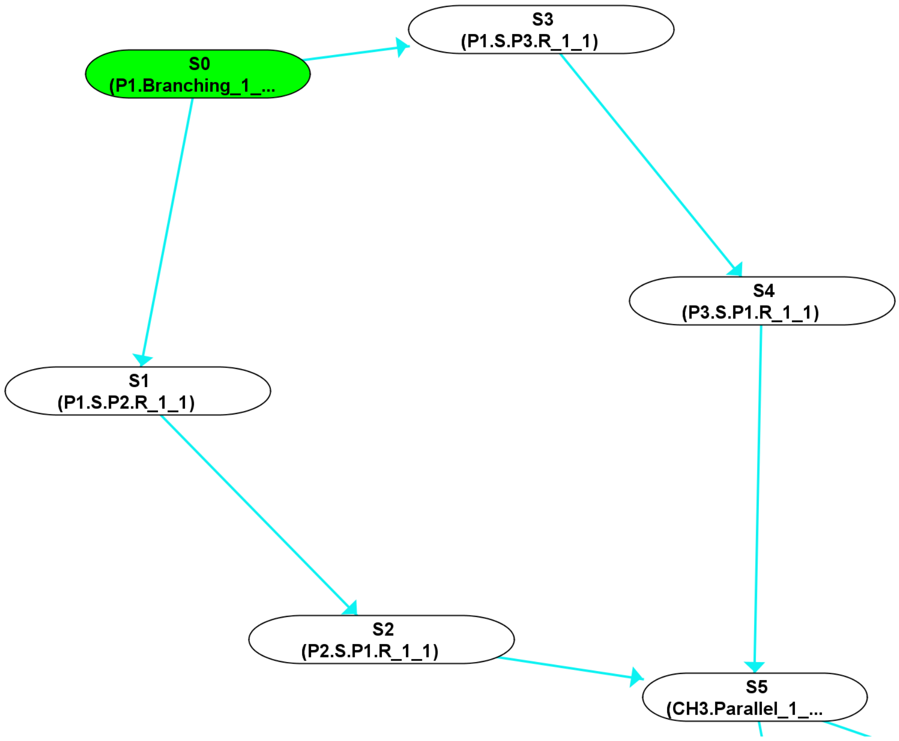
P2.END_1_1 : P2 terminates

P3.END_1_1 : P3 terminates

4.2.5 Explanation of the Kripke Structure

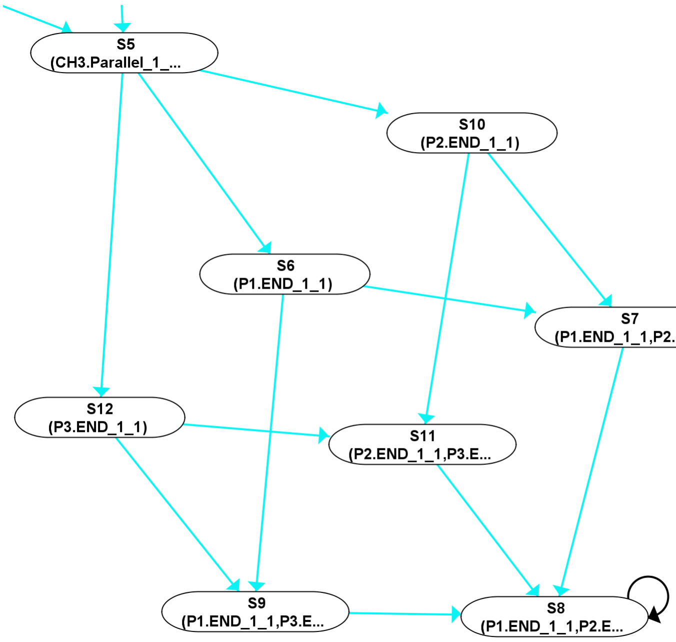
We explain below the various segments of the Kripke structure of the Three-Process example.

Request Information



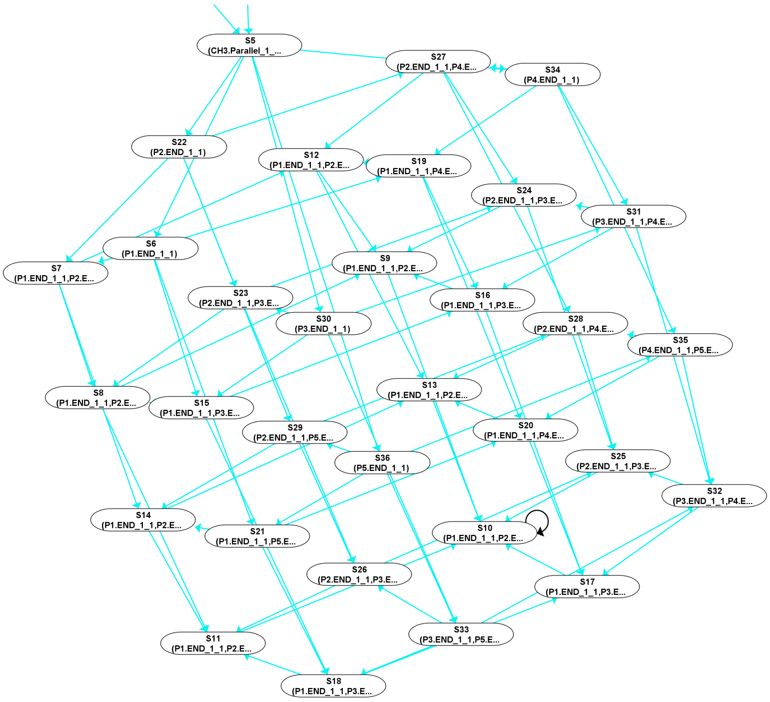
$S0$: Branching choice, P1 chooses which process to request
Case1: $S0 \rightarrow S1 \rightarrow S2$ P1 asks P2 for information and P2 replies
Case2: $S0 \rightarrow S3 \rightarrow S4$ P1 asks P3 for information and P3 replies

System Termination



All the different way for the three processes to terminate in parallel.

System Termination: 5 Processes



However, if five processes terminate in parallel, the number of unique state grows exponentially. Thus choreographies are prone to state explosion and we need to define correctness semantics for choreographies.

Chapter 5

Correctness of choreographies

Since choreograhpies include the parallel composition operator, even the verification of a choreography can be subject to state-explosion. Hence we present a set of inductive rules which enable the deduction of correctness properties of choreographies from correctness properties of the smaller choreographies from which they are built. This allows us to avoid state explosion in verifying properties of choreographies. We deal with the following

- $\mathsf{AG}(\varphi)$
- $\mathsf{AG}(\varphi \to \mathsf{AF}(\psi))$

Where φ and ψ are purely propositional

5.1 Sequential

In chapter 4 in [1], sequential semantics is defined by (i) applying ch_1; (ii) notifying the start of ch_2; and finally (iii) applying ch_2.

$$\text{SeqCorrectness1:}\frac{ch_1 \models \mathsf{AG}(\varphi)}{ch_1 \bullet ch_2 \models \mathsf{A}[\varphi\ \mathsf{W}\ ch_1.end]} \tag{5.1}$$

Given for all paths, for every state in ch_1, φ is satisfied.
Conclude $ch_1 \bullet ch_2 \models$ for all paths φ holds weak until ch_1 ends.

$$\text{SeqCorrectness2:}\frac{ch_2 \models \mathsf{AG}(\varphi)}{ch_1 \bullet ch_2 \models \mathsf{AG}(ch_2.st \implies \mathsf{AG}(\varphi))} \tag{5.2}$$

Given for all paths, for every state in ch_2, φ is satisfied.
Conclude $ch_1 \bullet ch_2 \models$ for all paths, for every state, if ch_2 starts, then for all paths, for every state in ch_2, φ holds.

$$\text{SeqCorrectness3:}\frac{ch_1 \models \mathsf{AG}(\varphi \implies \mathsf{AF}(\psi))}{ch_1 \bullet ch_2 \models \mathsf{A}[\mathsf{AG}(\varphi \implies \mathsf{AF}(\psi))\ \mathsf{W}\ ch_1.end]} \tag{5.3}$$

Given for all paths, for every state s in ch_1, if φ is satisfied, then for all paths from s, finally ψ will be satisfied.
Conclude $ch_1 \bullet ch_2 \models$ for all paths [for all paths, for every state s, if φ is satisfied, then for all paths from s, finally ψ is satisfied weak until ch_1 ends].

$$\text{SeqCorrectness4:}\frac{ch_2 \models \mathsf{AG}(\varphi \implies \mathsf{AF}(\psi))}{ch_1 \bullet ch_2 \models \mathsf{AG}[ch_2.st \implies \mathsf{AG}(\varphi \implies \mathsf{AF}(\psi))]} \tag{5.4}$$

Given for all paths, for every state s in ch_2, if φ is satisfied, then for all paths from s, finally ψ will be satisfied.
Conclude $ch_1 \bullet ch_2 \models$ for all paths, for every state s, if ch_2 starts, then for all paths from s, for every state, if φ is satisfied, then finally ψ is satisfied.

$$\text{SeqCorrectness5:}\frac{ch_1 \models \mathsf{AG}(\varphi_1 \implies \mathsf{AF}(ch_1.end)), ch_2 \models \mathsf{AG}(ch_2.st \implies \mathsf{AF}(\varphi_2))}{ch_1 \bullet ch_2 \models \mathsf{AG}(\varphi_1 \implies \mathsf{AF}(\varphi_2))} \tag{5.5}$$

Given for all paths, for every state s in ch_1, if φ_1 is satisfied, then for all paths from s, finally $ch_1.end$ will be satisfied.
Given for all paths, for every state s in ch_2, if $ch_2.st$ is satisfied, then for all paths from s, finally ψ_2 will be satisfied.
Conclude $ch_1 \bullet ch_2 \models$ for all paths, for every state s, if φ_1 is satisfied, then for all paths from s, finally φ_2 is satisfied.

5.2 Parallel

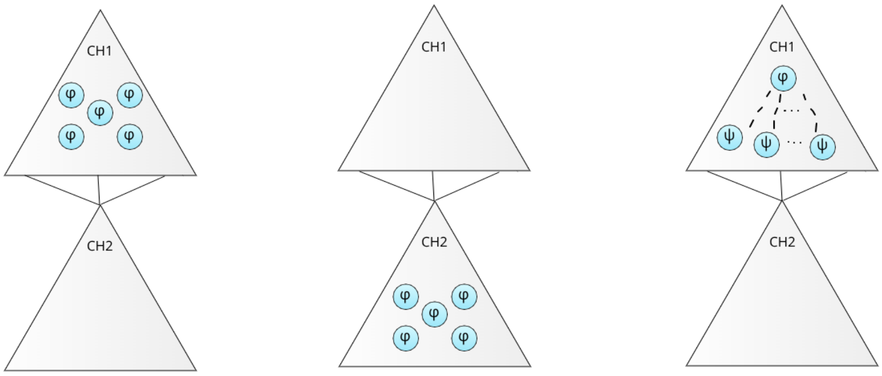
In chapter 4, the binary operator $\|$ allows for the parallel compositions of two independent choreographies. Two choreographies are independent if their participating components are disjoint. Thus, if ch_1 and ch_2 are applied in parallel, the properties of each choreography hold together.

$$\text{ParCorrectness1:}\frac{ch_1 \models \mathsf{AG}(\varphi_1), ch_2 \models \mathsf{AG}(\varphi_2)}{ch_1 \parallel ch_2 \models \mathsf{AG}(\varphi_1) \wedge \mathsf{AG}(\varphi_2)} \tag{5.6}$$

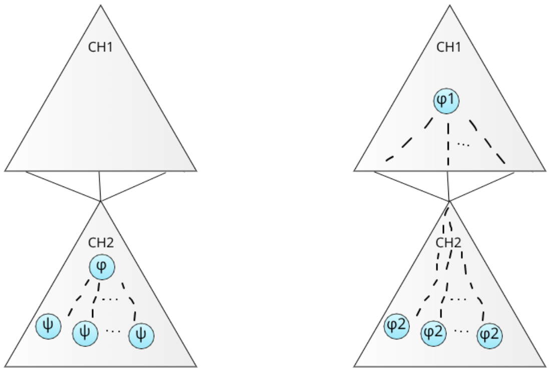
Given for all paths, for every state in ch_1, φ_1 is satisfied.
Given for all paths, for every state in ch_2, φ_2 is satisfied.
Conclude $ch_1 \parallel ch_2 \models$ for all paths, for every state, φ_1 and φ_2 hold.

$$\text{ParCorrectness2:}\frac{ch_1 \models \mathsf{AG}(\varphi_1 \implies \mathsf{AF}(\psi_1)), ch_2 \models \mathsf{AG}(\varphi_2 \implies \mathsf{AF}(\psi_2))}{ch_1 \parallel ch_2 \models \mathsf{AG}(\varphi_1 \implies \mathsf{AF}(\psi_1)) \wedge \mathsf{AG}(\varphi_2 \implies \mathsf{AF}(\psi_2))} \tag{5.7}$$

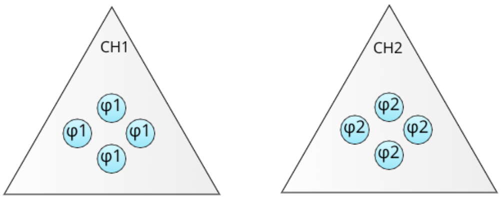
Given for all paths, for every state s in ch_1, if φ_1 is satisfied, then for all paths from s, finally ψ_1 will be satisfied.
Given for all paths, for every state s in ch_2, if φ_2 is satisfied, then for all paths from s, finally ψ_2 will be satisfied.
Conclude $ch_1 \parallel ch_2 \models$ for all paths, for every state s, if φ_1 holds then for all paths from s, finally ψ_1 holds and if φ_2 holds then for all paths, finally ψ_2 holds.



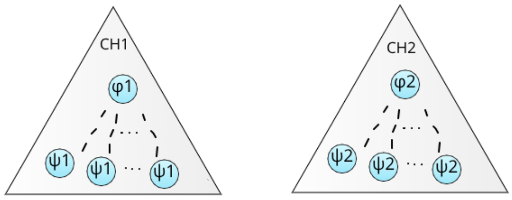
(a) Sequential 1 (b) Sequential 2 (c) Sequential 3



(d) Sequential 4 (e) Sequential 5



(f) Parallel 1



(g) Parallel 2

Figure 5.1: Sequential and Parallel Correctness

5.3 Branching

Branching allows for the modeling of choice between several choreographies. The $\oplus$ operator allows a specific component to select from a set of choreographies. Note that it is required to notify all the participants of a choice and not only the start components.

$$\text{BranCorrectness1:}\frac{ch_j \models \mathsf{AG}(\varphi_j), j \in J}{(B_i \oplus G_j \& snd_j : ch_j) \models \mathsf{AG}(G_j \wedge \widehat{snd_j} \implies \mathsf{AG}(\varphi_j))} \tag{5.8}$$

Given for all paths, for every state in ch_j, φ_j is satisfied.
Conclude $(B_i \oplus G_j \& snd_j : ch_j) \models$ for all paths, for every state s, if guard G_j is satisfied and $\widehat{snd_j}$, i.e. this branch is selected, then for all paths from s, for every state, φ_j holds.

$$\text{BranCorrectness2:}\frac{ch_j \models \mathsf{AG}(\varphi_j \implies \mathsf{AF}(\psi_j)), j \in J}{\begin{gathered}(B_i \oplus G_j \& snd_j : ch_j) \models \\ \mathsf{AG}(G_j \wedge \widehat{snd_j} \implies \mathsf{AG}(\varphi_j \implies \mathsf{AF}(\psi_j)))\end{gathered}} \tag{5.9}$$

Given for all paths, for every state s in ch_j, if φ_j is satisfied, then for all paths from s, finally ψ_j is satisfied.
Conclude $(B_i \oplus G_j \& snd_j : ch_j) \models$ for all paths, for every state, if guard G_j is satisfied and $\widehat{snd_j}$, i.e. this branch is selected, then for all paths, for every state s, if φ_j is satisfied, then for all paths from s, finally ψ_j is satisfied.

5.4 While

Loop allows for the modeling of a conditional repeated choreography 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.

$$\text{whileCorrectness1:}\frac{ch \models \mathsf{AG}(\varphi)}{while(G \& snd)\ ch\ end \models \mathsf{AG}(G \wedge \widehat{snd} \implies \mathsf{AG}(\varphi))} \tag{5.10}$$

Given for all paths, for every state in ch, φ is satisfied.
Conclude $while(G \& snd)\ ch\ end \models$ for all paths, for every state s, if guard G is satisfied and $\widehat{snd}$, i.e. we are in the loop, then for all paths from s, for every state, φ holds.

$$\text{whileCorrectness2:}\frac{ch \models \mathsf{AG}(\varphi \implies \mathsf{AF}(\psi))}{\begin{gathered}while(G \& snd)\ ch\ end \models \\ \mathsf{AG}(G \wedge \widehat{snd} \implies \mathsf{AG}(\varphi \implies \mathsf{AF}(\psi)))\end{gathered}} \tag{5.11}$$

Given for all paths, for every state s in ch, if φ is satisfied, then for all paths from s, finally ψ is satisfied.
Conclude $while(G\&snd)\ ch\ end \models$ for all paths, for every state, if guard G is satisfied and $\widehat{snd}$, i.e. we are in the loop, then for all paths, for every state s, if φ is satisfied, then for all paths from s, finally ψ is satisfied.

5.5 Send-Receive

5.5.1 Asynchronous Send

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.

$$\text{sndRecvsAsynch:}\frac{}{snd \to rcvs \models \mathsf{AG}(q_1^1 \wedge \ldots \wedge q_1^n \implies \mathsf{AF}(q_2^1) \wedge \ldots \wedge \mathsf{AF}(q_2^n))} \tag{5.12}$$

Only the starter component must be free in order to execute, and whenever a receiver component is free, it will execute. Thus we can say that then $\mathsf{AF}(q_2^1) \wedge \ldots \wedge \mathsf{AF}(q_2^n)$.

5.5.2 Synchronous Send

$$\text{sndRecvsSynch:}\frac{}{snd \to rcvs \models \mathsf{AG}(q_1^1 \wedge \ldots \wedge q_1^n \implies \mathsf{AF}(q_2^1 \wedge \ldots \wedge q_2^n))} \tag{5.13}$$

All components must be free in order to execute, thus if $q_1^1 \wedge \ldots \wedge q_1^n$ are the current states of the respective componenets, then $\mathsf{AF}(q_2^1 \wedge \ldots \wedge q_2^n)$.

Chapter 6

Correctness of Distributed Implementations of Choreographies

6.1 Implementation of Choreographies

We define implementation as a function I: Choreographies → BIP. I is defined by structural induction over the definition of a choreography. We write $I(ch) = [\![ch]\!]$ to indicate that $[\![ch]\!]$ is the BIP system that implements ch.

6.1.1 Sequential

If $I(ch_1) = [\![ch_1]\!]$, $I(ch_2) = [\![ch_2]\!]$ then $I(ch_1 \bullet ch_2)$ consists of $[\![ch_1]\!]$ then for all components common to $[\![ch_1]\!]$, $[\![ch_2]\!]$ insert $[\![ch_{synch}]\!] = snd \rightarrow rcvs$ synchronous with interaction interaction $a = snd \rightarrow rcvs$ that synchronizes the end of $[\![ch_1]\!]$ with the start of $[\![ch_2]\!]$. In this, and all subsequent rules, the semantics of the execution of an interaction is given by section 2.1.

$$\text{SequentialImp:}\frac{I(ch_1) = [\![ch_1]\!], I(ch_2) = [\![ch_2]\!]}{I(ch_1 \bullet ch_2) = [\![ch_1 \bullet ch_2]\!] = [\![ch_1]\!] \rightarrow [\![ch_{synch}]\!] \rightarrow [\![ch_2]\!]} \tag{6.1}$$

The following example is from [1]. The figure was modified.

Example 1 (Sequential composition)**.** *Figure 6.1 shows an abstract example on how to transform sequential composition of two choreographies,* $\mathtt{ch}_1 \bullet \mathtt{ch}_2$*, 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*

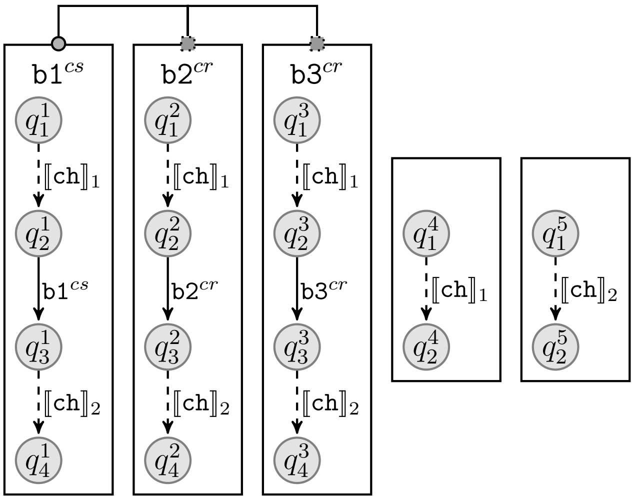


Figure 6.1: Sequential composition transformation

choreography $\texttt{ch}_1$ *(e.g.,* b_1*) with the start of choreography* $\texttt{ch}_2$ *(e.g.,* b_2 *and* b_3*). To do so, we create a synchronous send port to one of the end components of* $\texttt{ch}_1$ *(e.g.,* b_1^{cs}*) and connect it to all the remaining end components of* $\texttt{ch}_1$ *(e.g.,* $\emptyset$ *and the start components of* $\texttt{ch}_2$ *(e.g.,* b_2 *and* b_3*); finally (3) we apply choreography* $\texttt{ch}_2$*.*

6.1.2 Parallel

If $I(ch_1) = [\![ch_1]\!]$, $I(ch_2) = [\![ch_2]\!]$ then $I(ch_1 \parallel ch_2)$ consists of $[\![ch_1]\!]$ in parallel with $[\![ch_2]\!]$. The components are disjoint.

$$\text{ParallelImp:}\frac{I(ch_1) = [\![ch_1]\!], I(ch_2) = [\![ch_2]\!]}{I(ch_1 \parallel ch_2) = [\![ch_1 \parallel ch_2]\!] = \phi([\![ch_1]\!], [\![ch_2]\!])} \tag{6.2}$$

$\phi([\![ch_1]\!], [\![ch_2]\!])$ definition 4, Section 2.1 The following example is from [1]. The figure was modified.

Example 2 (Parallel Composition)**.** *Figure 6.2 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*).*

6.1.3 Branching

$$\text{branchingImp:}\frac{I(ch_j) = [\![ch_j]\!], j \in J}{I(B_i \oplus G_j \& snd_j : ch_j) = B_i \oplus G_j \& snd_j : [\![ch_j]\!]} \tag{6.3}$$

If $I(ch_j) = [\![ch_j]\!] \forall\, j \in J$, then $I(B_i \oplus G_j \& snd_j : ch_j)$ consists of $B_i \oplus G_j \& snd_j : [\![ch_j]\!]$.

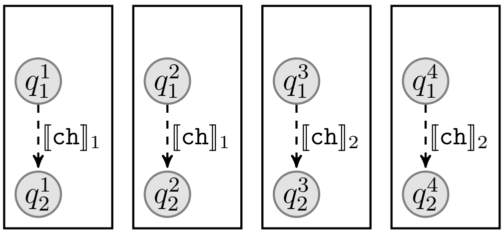


Figure 6.2: Parallel composition transformation

6.1.4 While

$$\text{while:}\frac{I(ch) = [\![ch]\!]}{I(while(G\&snd)\ ch\ end) = while(G\&snd)[\![ch]\!]end} \tag{6.4}$$

If $I(ch) = [\![ch]\!]$, then $I(while(G\&snd)\ ch\ end)$ consists of $while(G\&snd)\ [\![ch]\!]\ end$.

6.1.5 Send-Receive

$$\text{sndRcvsImp:}\frac{}{I(snd \to rcvs) = [\![snd \to rcvs]\!]} \tag{6.5}$$

$I(snd \to rcvs)$ consists of $[\![snd \to rcvs]\!]$.

6.2 Correctness

$\widehat{a}$ is an atomic proposition that is true immediately after a has executed.

6.2.1 Sequential

$$\text{SeqImpCorrectness1:}\frac{[\![ch_1]\!] \models \mathsf{AG}(\varphi)}{[\![ch_1 \bullet ch_2]\!] \models \mathsf{A}[\varphi\ \mathsf{W}\ \widehat{a}]} \tag{6.6}$$

Given for all paths, for every state in $[\![ch_1]\!]$, φ is satisfied.
Conclude $[\![ch_1 \bullet ch_2]\!] \models$ for all paths φ holds weak until $\widehat{a}$. Here $\widehat{a}$ means the end of $[\![ch_1]\!]$.

$$\text{SeqImpCorrectness2:}\frac{[\![ch_2]\!] \models \mathsf{AG}(\varphi)}{[\![ch_1 \bullet ch_2]\!] \models \mathsf{AG}(\widehat{a} \implies \mathsf{AG}(\varphi))} \tag{6.7}$$

Given for all paths, for every state in $[\![ch_2]\!]$, φ is satisfied.
Conclude $[\![ch_1 \bullet ch_2]\!] \models$ for all paths, for every state s, if $\widehat{a}$ starts, then for all paths from s, for every state in ch_2, φ holds. Here $\widehat{a}$ means the start of $[\![ch_2]\!]$.

$$\text{SeqImpCorrectness3:}\frac{[\![ch_1]\!] \models \mathsf{AG}(\varphi \implies \mathsf{AF}(\psi))}{[\![ch_1 \bullet ch_2]\!] \models \mathsf{A}[\mathsf{AG}(\varphi \implies \mathsf{AF}(\psi))\ \mathsf{W}\ \widehat{a}]} \tag{6.8}$$

Given for all paths, for every state s in $[\![ch_1]\!]$, if φ is satisfied, then for all paths from s, finally ψ will be satisfied.

Conclude $[\![ch_1 \bullet ch_2]\!] \models$ for all paths [for all paths, for every state s, if φ is satisfied, then for all paths from s, finally ψ is satisfied weak until $\widehat{a}$]. Here $\widehat{a}$ means the end of $[\![ch_1]\!]$.

$$\text{SeqImpCorrectness4:}\frac{[\![ch_2]\!] \models \mathsf{AG}(\varphi \implies \mathsf{AF}(\psi))}{[\![ch_1 \bullet ch_2]\!] \models \mathsf{AG}[\widehat{a} \implies \mathsf{AG}(\varphi \implies \mathsf{AF}(\psi))]} \tag{6.9}$$

Given for all paths, for every state s in $[\![ch_2]\!]$, if φ is satisfied, then for all paths from s, finally ψ will be satisfied.
Conclude $[\![ch_1 \bullet ch_2]\!] \models$ for all paths, for every state s, if $\widehat{a}$, then for all paths from s, for every state, if φ is satisfied, then finally ψ_2 is satisfied. Here $\widehat{a}$ signals the start of ch_2.

$$\text{SeqImpCorrectness5:}\frac{[\![ch_1]\!] \models \mathsf{AG}(\varphi \implies \mathsf{AF}(\widehat{a})), [\![ch_2]\!] \models \mathsf{AG}(\widehat{a} \implies \mathsf{AF}(\psi))}{[\![ch_1 \bullet ch_2]\!] \models \mathsf{AG}(\varphi \implies \mathsf{AF}(\psi))} \tag{6.10}$$

Given for all paths, for every state in $[\![ch_1]\!]$, if φ is satisfied, then for all paths, finally $\widehat{a}$ will be satisfied. Here $\widehat{a}$ means the end of $[\![ch_1]\!]$.
Given for all paths, for every state in $[\![ch_2]\!]$, if $\widehat{a}$ is satisfied, then for all paths, finally ψ will be satisfied.
Conclude $[\![ch_1 \bullet ch_2]\!] \models$ for all paths, for every state, if φ is satisfied, then for all paths, finally ψ is satisfied.

6.2.2 Parallel

$$\text{ParImpCorrectness1:}\frac{[\![ch_1]\!] \models \mathsf{AG}(\varphi_1), [\![ch_2]\!] \models \mathsf{AG}(\varphi_2)}{[\![ch_1 \parallel ch_2]\!] \models \mathsf{AG}(\varphi_1) \land \mathsf{AG}(\varphi_2)} \tag{6.11}$$

Given for all paths, for every state in $[\![ch_1]\!]$, φ_1 is satisfied.
Given for all paths, for every state in $[\![ch_2]\!]$, φ_2 is satisfied.
Conclude $[\![ch_1 \parallel ch_2]\!] \models$ for all paths, for every state, φ_1 and φ_2 hold.

$$\text{ParImpCorrectness2:}\frac{[\![ch_1]\!] \models \mathsf{AG}(\varphi_1 \implies \mathsf{AF}(\psi_1)), [\![ch_2]\!] \models \mathsf{AG}(\varphi_2 \implies \mathsf{AF}(\psi_2))}{[\![ch_1 \parallel ch_2]\!] \models \mathsf{AG}(\varphi_1 \implies \mathsf{AF}(\psi_1)) \land \mathsf{AG}(\varphi_2 \implies \mathsf{AF}(\psi_2))} \tag{6.12}$$

Given for all paths, for every state s in $[\![ch_1]\!]$, if φ_1 is satisfied, then for all paths from s, finally ψ_1 will be satisfied.
Given for all paths, for every state s in $[\![ch_2]\!]$, if φ_2 is satisfied, then for all paths from s, finally ψ_2 will be satisfied.
Conclude $[\![ch_1 \parallel ch_2]\!] \models$ for all paths, for every state s_1, if φ_1 holds then for all paths from s_1, finally ψ_1 holds. And for all paths, for every state s_2, if φ_2 holds then for all paths from s_2, finally ψ_2 holds.

6.2.3 Branching

$$\text{BranCorrectness1:}\frac{[\![ch_j]\!] \models \mathsf{AG}(\varphi_j), j \in J}{(B_i \oplus G_j \& snd_j : [\![ch_j]\!]) \models \mathsf{AG}(G_j \wedge \widehat{snd_j} \implies \mathsf{AG}(\varphi_j))} \tag{6.13}$$

Given for all paths, for every state in $[\![ch_j]\!]$, φ_j is satisfied.
Conclude $(B_i \oplus G_j \& snd_j : [\![ch_j]\!]) \models$ for all paths, for every state s, if guard G_j is satisfied and $\widehat{snd_j}$, i.e. this branch selected, then for all paths from s, for every state, φ_j holds.

$$\text{BranCorrectness2:}\frac{[\![ch_j]\!] \models \mathsf{AG}(\varphi_j \implies \mathsf{AF}(\psi_j)), j \in J}{\begin{array}{c}(B_i \oplus G_j \& snd_j : [\![ch_j]\!]) \models \\ \mathsf{AG}(G_j \wedge \widehat{snd_j} \implies \mathsf{AG}(\varphi_j \implies \mathsf{AF}(\psi_j)))\end{array}} \tag{6.14}$$

Given for all paths, for every state s in ch_j, if φ_j is satisfied, then for all paths from s, finally ψ_j is satisfied.
Conclude $(B_i \oplus G_j \& snd_j : [\![ch_j]\!]) \models$ for all paths, for every state, if guard G_j is satisfied and $\widehat{snd_j}$, i.e. this branch is selected, then for all paths, for every state s, if φ_j is satisfied, then for all paths from s, finally ψ_j is satisfied.

6.2.4 While

$$\text{whileImpCorrectness1:}\frac{[\![ch]\!] \models \mathsf{AG}(\varphi)}{while(G \& snd)\ [\![ch]\!]\ end \models \mathsf{AG}(G \wedge \widehat{snd} \implies \mathsf{AG}(\varphi))} \tag{6.15}$$

Given for all paths, for every state in $[\![ch]\!]$, φ is satisfied.
Conclude $while(G \& snd)\ [\![ch]\!]\ end \models$ for all paths, for every state s, if guard G is satisfied and $\widehat{snd}$, i.e. we are in the loop, then for all paths from s, for every state, φ holds.

$$\text{whileImpCorrectness2:}\frac{[\![ch]\!] \models \mathsf{AG}(\varphi \implies \mathsf{AF}(\psi))}{\begin{array}{c}while(G \& snd)\ [\![ch]\!]\ end \models \\ \mathsf{AG}(G \wedge \widehat{snd} \implies \mathsf{AG}(\varphi \implies \mathsf{AF}(\psi)))\end{array}} \tag{6.16}$$

Given for all paths, for every state s in $[\![ch]\!]$, if φ is satisfied, then for all paths from s, finally ψ is satisfied.
Conclude $while(G \& snd)\ [\![ch]\!]\ end \models$ for all paths, for every state, if guard G is satisfied and $\widehat{snd}$, i.e. we are in the loop, then for all paths, for every state s, if φ is satisfied, then for all paths from s, finally ψ is satisfied.

6.2.5 Send-Receive

Asynchronous Send

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.

$$\text{sndRecvsAsynchImp:}\,\frac{}{[\![\mathit{snd} \to \mathit{rcvs}]\!] \models \mathsf{AG}(q_1^1 \land \ldots \land q_1^n \implies \mathsf{AF}(q_2^1) \land \ldots \land \mathsf{AF}(q_2^n))} \tag{6.17}$$

Only the starter component must be free in order to execute, and whenever a receiver component is free, it will execute. Thus we can say that then $\mathsf{AF}(q_2^1) \land \ldots \land \mathsf{AF}(q_2^n)$.

Synchronous Send

$$\text{sndRecvsSynchImp:}\,\frac{}{[\![\mathit{snd} \to \mathit{rcvs}]\!] \models \mathsf{AG}(q_1^1 \land \ldots \land q_1^n \implies \mathsf{AF}(q_2^1 \land \ldots \land q_2^n))} \tag{6.18}$$

All components must be free in order to execute, thus if $q_1^1 \land \ldots \land q_1^n$ are the current states of the respective componenets, then $\mathsf{AF}(q_2^1 \land \ldots \land q_2^n)$.

Chapter 7

Correctness of Synthesis Method

7.1 Correctness Theorems

The following theorems show that, if the global choreography satisfy certain formulae, then the implemented system does as well. We deal with sequence only. The theorems are proven by induction on the length of the derivation of $[\![ch]\!]$ from ch, where $snd \to rcv$ is the base case. $snd \to rcv$ and $[\![snd \to rcv]\!]$ satisfy the same formulae.

Theorem 1 (Sequential Correctness 1)**.** *If* $ch_1 \models \mathsf{AG}(\varphi)$ *implies* $[\![ch_1]\!] \models \mathsf{AG}(\varphi)$ *then* $ch_1 \bullet ch_2 \models \mathsf{A}[\varphi \ \mathsf{W} \ ch_1.end]$ *implies* $[\![ch_1 \bullet ch_2]\!] \models \mathsf{A}[\varphi \ \mathsf{W} \ \widehat{a}]$*. Here* $\widehat{a}$ *signals end of* $[\![ch_1]\!]$

$$\frac{ch_1 \models \mathsf{AG}(\varphi) \implies [\![ch_1]\!] \models \mathsf{AG}(\varphi)}{ch_1 \bullet ch_2 \models \mathsf{A}[\varphi \ \mathsf{W} \ ch_1.end] \implies [\![ch_1 \bullet ch_2]\!] \models \mathsf{A}[\varphi \ \mathsf{W} \ \widehat{a}]} \tag{7.1}$$

Premises:

1. $ch_1 \models \mathsf{AG}(\varphi) \implies [\![ch_1]\!] \models \mathsf{AG}(\varphi)$
2. $ch_1 \bullet ch_2 \models \mathsf{A}[\varphi \ \mathsf{W} \ ch_1.end]$

Required to prove:

3. $[\![ch_1 \bullet ch_2]\!] \models \mathsf{A}[\varphi \ \mathsf{W} \ \widehat{a}]$

From 2 we get:

4. $ch_1 \models \mathsf{AG}(\varphi)$

Modus Ponens using 1 and 4 we get:

5. $[\![ch_1]\!] \models \mathsf{AG}(\varphi)$

Finally from 6.6 we conclude $[\![ch_1 \bullet ch_2]\!] \models \mathsf{A}[\varphi_1 \ \mathsf{W} \ \widehat{a}]$

Theorem 2 (Sequential Correctness 2)**.** *If* $ch_2 \models \mathsf{AG}(\varphi)$ *implies* $[\![ch_2]\!] \models \mathsf{AG}(\varphi)$ *then* $ch_1 \bullet ch_2 \models \mathsf{AG}(ch_2.st \implies \mathsf{AG}(\varphi))$ *implies* $[\![ch_1 \bullet ch_2]\!] \models \mathsf{AG}(\widehat{a} \implies \mathsf{AG}(\varphi))$*. Here* $\widehat{a}$ *signals start of* $[\![ch_2]\!]$

$$\frac{ch_2 \models \mathsf{AG}(\varphi) \implies [\![ch_2]\!] \models \mathsf{AG}(\varphi)}{\begin{array}{c} ch_1 \bullet ch_2 \models \mathsf{AG}(ch_2.st \implies \mathsf{AG}(\varphi)) \\ \implies [\![ch_1 \bullet ch_2]\!] \models \mathsf{AG}(\widehat{a} \implies \mathsf{AG}(\varphi)) \end{array}} \tag{7.2}$$

Premises:

1. $ch_2 \models \mathsf{AG}(\varphi) \implies [\![ch_2]\!] \models \mathsf{AG}(\varphi)$
2. $ch_1 \bullet ch_2 \models \mathsf{AG}(ch_2.st \implies \mathsf{AG}(\varphi))$

Required to prove:

3. $[\![ch_1 \bullet ch_2]\!] \models \mathsf{AG}(\widehat{a} \implies \mathsf{AG}(\varphi))$

From 2 we get:

4. $ch_2 \models \mathsf{AG}(\varphi)$

Modus Ponens using 1 and 4 we get:

5. $[\![ch_2]\!] \models \mathsf{AG}(\varphi)$

Finally from 6.7 we conclude $[\![ch_1 \bullet ch_2]\!] \models \mathsf{AG}(\widehat{a} \implies \mathsf{AG}(\varphi_2))$

Theorem 3 (Sequential Correctness 3)**.** *If* $ch_1 \models \mathsf{AG}(\varphi \implies \mathsf{AF}(\psi))$ *implies* $[\![ch_1]\!] \models \mathsf{AG}(\varphi \implies \mathsf{AF}(\psi))$ *then* $ch_1 \bullet ch_2 \models \mathsf{A}[\mathsf{AG}(\varphi \implies \mathsf{AF}(\psi))\ \mathsf{W}\ ch_1.end]$ *implies* $[\![ch_1 \bullet ch_2]\!] \models \mathsf{A}[\mathsf{AG}(\varphi \implies \mathsf{AF}(\psi))\ \mathsf{W}\ \widehat{a}]$*. Here* $\widehat{a}$ *signals end of* $[\![ch_1]\!]$

$$\frac{ch_1 \models \mathsf{AG}(\varphi \implies \mathsf{AF}(\psi)) \implies [\![ch_1]\!] \models \mathsf{AG}(\varphi \implies \mathsf{AF}(\psi))}{\begin{array}{c} ch_1 \bullet ch_2 \models \mathsf{A}[\mathsf{AG}(\varphi \implies \mathsf{AF}(\psi))\ \mathsf{W}\ ch_1.end] \\ \implies [\![ch_1 \bullet ch_2]\!] \models \mathsf{A}[\mathsf{AG}(\varphi \implies \mathsf{AF}(\psi))\ \mathsf{W}\ \widehat{a}] \end{array}} \tag{7.3}$$

Premises:

1. $ch_1 \models \mathsf{AG}(\varphi \implies \mathsf{AF}(\psi)) \implies [\![ch_1]\!] \models \mathsf{AG}(\varphi \implies \mathsf{AF}(\psi))$
2. $ch_1 \bullet ch_2 \models \mathsf{A}[\mathsf{AG}(\varphi \implies \mathsf{AF}(\psi))\ \mathsf{W}\ ch_1.end]$

Required to prove:

3. $\mathsf{A}[\mathsf{AG}(\varphi \implies \mathsf{AF}(\psi))\ \mathsf{W}\ \widehat{a}]$

From 2 we get:

4. $\mathit{ch}_1 \models \mathsf{AG}(\varphi \implies \mathsf{AF}(\psi))$

Modus Ponens using 1 and 4 we get:

5. $\llbracket \mathit{ch}_1 \rrbracket \models \mathsf{AG}(\varphi \implies \mathsf{AF}(\psi))$

Finally from 6.8 we conclude $\llbracket \mathit{ch}_1 \bullet \mathit{ch}_2 \rrbracket \models \mathsf{A}[\mathsf{AG}(\varphi_1 \implies \mathsf{AF}(\psi_1))\ \mathsf{W}\ \widehat{a}]$

Theorem 4 (Sequential Correctness 4)**.** *If* $\mathit{ch}_2 \models \mathsf{AG}(\varphi \implies \mathsf{AF}(\psi))$ *implies* $\llbracket \mathit{ch}_2 \rrbracket \models \mathsf{AG}(\varphi \implies \mathsf{AF}(\psi))$ *then* $\mathit{ch}_1 \bullet \mathit{ch}_2 \models \mathsf{AG}[\mathit{ch}_2.\mathit{st} \implies \mathsf{AG}(\varphi \implies \mathsf{AF}(\psi))]$ *implies* $\llbracket \mathit{ch}_1 \bullet \mathit{ch}_2 \rrbracket \models \mathsf{AG}[\widehat{a} \implies \mathsf{AG}(\varphi \implies \mathsf{AF}(\psi))]$ $\widehat{a}$ *signals start of* $\llbracket \mathit{ch}_2 \rrbracket$

$$\frac{\mathit{ch}_2 \models \mathsf{AG}(\varphi \implies \mathsf{AF}(\psi)) \implies \llbracket \mathit{ch}_2 \rrbracket \models \mathsf{AG}(\varphi \implies \mathsf{AF}(\psi))}{\begin{array}{c}\mathit{ch}_1 \bullet \mathit{ch}_2 \models \mathsf{AG}[\mathit{ch}_2.\mathit{st} \implies \mathsf{AG}(\varphi \implies \mathsf{AF}(\psi))] \\ \implies \llbracket \mathit{ch}_1 \bullet \mathit{ch}_2 \rrbracket \models \mathsf{AG}[\widehat{a} \implies \mathsf{AG}(\varphi \implies \mathsf{AF}(\psi))]\end{array}} \tag{7.4}$$

Premises:

1. $\mathit{ch}_2 \models \mathsf{AG}(\varphi \implies \mathsf{AF}(\psi))$ *implies* $\llbracket \mathit{ch}_2 \rrbracket \models \mathsf{AG}(\varphi \implies \mathsf{AF}(\psi))$
2. $\mathit{ch}_1 \bullet \mathit{ch}_2 \models \mathsf{AG}[\mathit{ch}_2.\mathit{st} \implies \mathsf{AG}(\varphi \implies \mathsf{AF}(\psi))]$

Required to prove:

3. $\llbracket \mathit{ch}_1 \bullet \mathit{ch}_2 \rrbracket \models \mathsf{AG}[\widehat{a} \implies \mathsf{AG}(\varphi \implies \mathsf{AF}(\psi))]$

From 2 we get:

4. $\mathit{ch}_2 \models \mathsf{AG}(\varphi \implies \mathsf{AF}(\psi))$

Modus Ponens using 1 and 4 we get:

5. $\llbracket \mathit{ch}_2 \rrbracket \models \mathsf{AG}(\varphi \implies \mathsf{AF}(\psi))$

Finally from 6.9 we conclude $\llbracket \mathit{ch}_1 \bullet \mathit{ch}_2 \rrbracket \models \mathsf{AG}[\widehat{a} \implies \mathsf{AG}(\varphi_2 \implies \mathsf{AF}(\psi_2))]$

7.2 Three-Process Example Correctness

We give an example of the correctness theorem by going back to the Three-Process example.

```
CH  = P1 ⊕ {CH1,CH2} • CH3
CH1  = P1.S→P2.R •P2.S  →P1.R
CH2  = P1.S→P3.R •P3.S  →P1.R
CH3  = P1  →  ϕ||  P2  →ϕ  ||  P3→  ϕ
```

- let $CH_4 = P_1 \oplus \{CH_1, CH_2\}$
- a_1 is $P_1.S \rightarrow P_2.R$
- a_2 is $P_1.S \rightarrow P_3.R$
- Required to prove: $CH \models \mathsf{A}[\mathsf{AG}(P_1.S \implies \mathsf{AF}(P_1.R)) \mathbin{\mathsf{W}} CH_4.end]$

By Model Checking:

- $CH_1 \models \mathsf{AG}(P_1.S \implies \mathsf{AF}(P_1.R))$
- $CH_2 \models \mathsf{AG}(P_1.S \implies \mathsf{AF}(P_1.R))$

Conclude by branching correctness rule:

- $CH_4 \models \mathsf{AG}(\widehat{a_1} \implies \mathsf{AG}(P_1.S \implies \mathsf{AF}(P_1.R))) \wedge$
  $\mathsf{AG}(\widehat{a_2} \implies \mathsf{AG}(P_1.S \implies \mathsf{AF}(P_1.R)))$

By CTL deduction:

- $CH_4 \models \mathsf{AG}(P_1.S \implies \mathsf{AF}(P_1.R))$

Finally:

- $CH \models \mathsf{A}[\mathsf{AG}(P_1.S \implies \mathsf{AF}(P_1.R)) \mathbin{\mathsf{W}} CH_4.end]$

Similarly:

- $[\![CH]\!] \models \mathsf{A}[\mathsf{AG}(P_1.S \implies \mathsf{AF}(P_1.R)) \mathbin{\mathsf{W}} \widehat{a_4}]$ where $\widehat{a_4}$ signals end of CH_4

Chapter 8

Buyer-Seller Example

We illustrate our approach with the Buyer-Seller example from [1]. The following description is from [1]:

> 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, who 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*).

8.1 The Choreography

Listing 8.1 depicts global choreography from the Buyer-Seller example in [1].

8.2 Input File

```
-- This is a comment. A comment must start with 2 dashes "--"
-- and be at the start of a line
-- Each choreography is written on a single line.
-- The first choreography written will be the global choreography
-- The first event in the global choreography is the start state
-- Sequential events are delimited  by '*'
```

Listing 8.1: Global choreography of the Buyer/Seller example

```
CH  = B_1.S→S.R • S.S →{B_1.R, B_2.R} • B_1 ⊕{CH_1, ϵ} • CH_2 •
   CH_7
CH_1  = B_1.S→S.R • S.S →{B_1.R, B_2.R}
CH_2  = B_2 ⊕{CH_3, ϵ}
CH_3  = while(B_2.C) {
        B_1.C→ Bk.InfR • Bk.InfS → B_1.R • B_1.C→ B_2.R
      } • CH_4
CH_4  = CH_5 ‖CH_6
CH_5  = B_2.MS→Bk.MR_2 • Bk.MS_2 → S.R
CH_6  = B_1.MS→Bk.MR_1 • Bk.MS_1 → S.R
CH_7  = B_1.E →ϕ ‖ B_2.E →ϕ ‖ Bk.E →ϕ ‖ S.E →ϕ
```

```
CH = B1.S > S.R * S.S > {B1.R, B2.R} * B1 + {CH1, E} * CH2 * CH7
-- Send/Recv messages can have one or multiple receivers
CH1 = B1.S > S.R * S.S > {B1.R, B2.R}
-- 'E' stands for epsilon, where we "skip" the branching step
CH2 = B2 + {CH3, E}
-- Inside while loops, '^' is used instead of '*' for sequential events
-- Nested loops are written in their own choreography
-- To use branching inside loops, the branching must be written
-- in its own choreography
CH3 = while(B2.C) {B1.C > Bk.InfR ^ Bk.InfS > B1.R ^ B1.C > B2.R} * CH4
-- Parallel choreographies are delimited by "||"
-- Parallel choreographies are written in their own choreography
-- Parallel choreographies contain sequential events only
CH4 = CH5 || CH6
CH5 = B2.MS > Bk.MR2 * Bk.MS2 > S.R
CH6 = B1.MS > Bk.MR1 * Bk.MS1 > S.R
CH7 = CH8 || CH9 || CH10 || CH11
-- END represents the termination of a process
CH8 = B1.E > END
CH9 = B2.E > END
CH10 = Bk.E > END
CH11 = S.E > END
```

8.3 Kripke Structure Generation

The following is the Kripke structure generated from the Buyer-Seller choreography. This was automatically generated by our implementation.

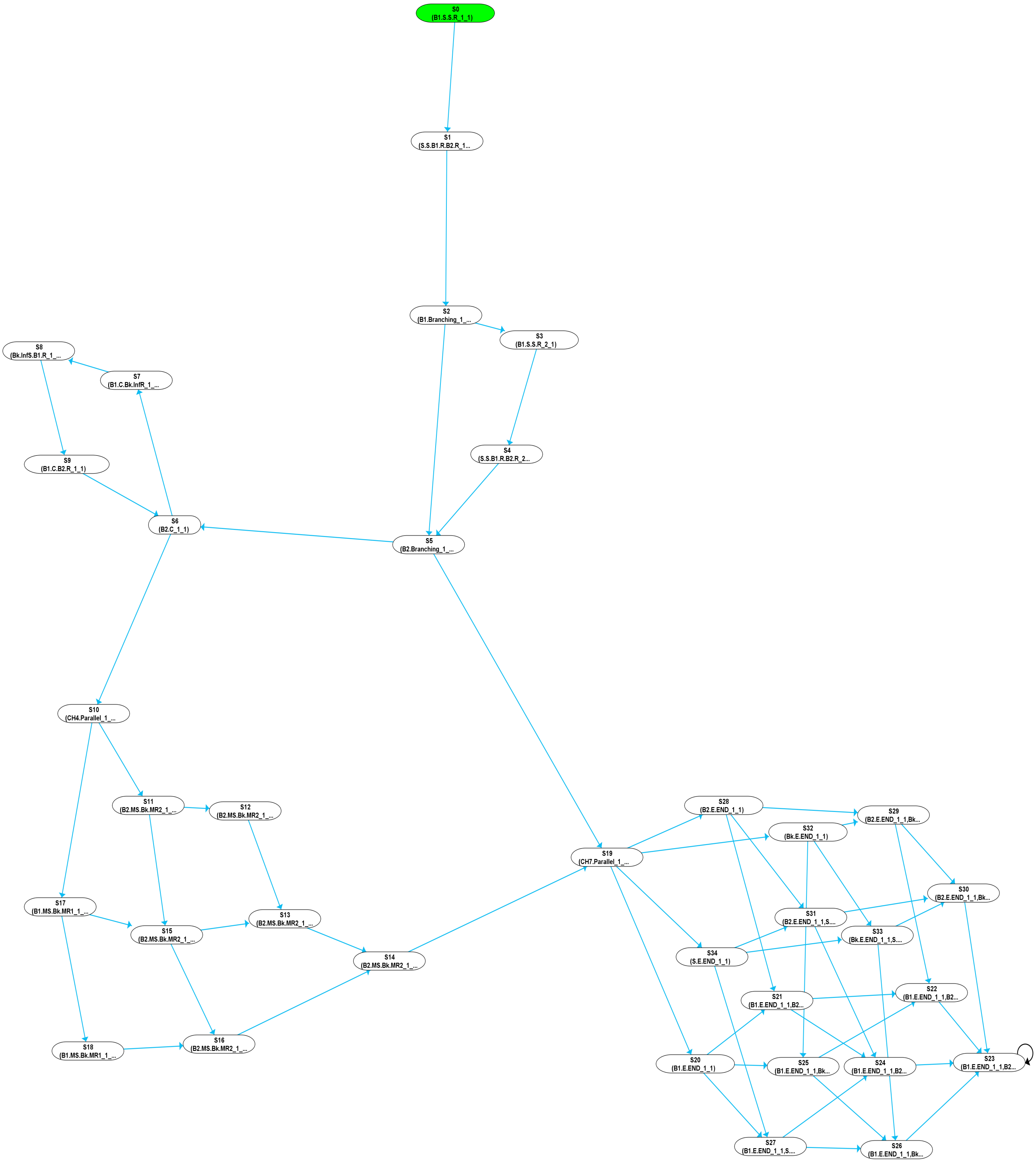


Figure 8.1: Automatically Generated Buyer-Seller Example

8.4 Label Definitions

We have 4 components:

- Buyer 1
- Buyer 2
- Seller
- Banker

The following shows the atomic propositions (comma separated) each state contains. The atomic propositions were auto generated by *name_i_1* were name is the concatenation of the elements of an event dot separated, *i* is the counter so that the proposition is unique in case repetition of name, and 1 is to indicate that the atomic proposition belongs to process 1 (the global choreography).

```
states:
S0:B1.S.S.R_1_1
S1:S.S.B1.R.B2.R_1_1
S2:B1.Branching_1_1
S3:B1.S.S.R_2_1
S4:S.S.B1.R.B2.R_2_1
S5:B2.Branching_1_1
S6:B2.C_1_1
S7:B1.C.Bk.InfR_1_1
S8:Bk.InfS.B1.R_1_1
S9:B1.C.B2.R_1_1
S10:CH4.Parallel_1_1
S11:B2.MS.Bk.MR2_1_1
S12:B2.MS.Bk.MR2_1_1,Bk.MS2.S.R_1_1
S13:B2.MS.Bk.MR2_1_1,Bk.MS2.S.R_1_1,B1.MS.Bk.MR1_1_1
S14:B2.MS.Bk.MR2_1_1,Bk.MS2.S.R_1_1,B1.MS.Bk.MR1_1_1,Bk.MS1.S.R_1_1
S15:B2.MS.Bk.MR2_1_1,B1.MS.Bk.MR1_1_1
S16:B2.MS.Bk.MR2_1_1,B1.MS.Bk.MR1_1_1,Bk.MS1.S.R_1_1
S17:B1.MS.Bk.MR1_1_1
S18:B1.MS.Bk.MR1_1_1,Bk.MS1.S.R_1_1
S19:CH7.Parallel_1_1
S20:B1.E.END_1_1
S21:B1.E.END_1_1,B2.E.END_1_1
S22:B1.E.END_1_1,B2.E.END_1_1,Bk.E.END_1_1
S23:B1.E.END_1_1,B2.E.END_1_1,Bk.E.END_1_1,S.E.END_1_1
S24:B1.E.END_1_1,B2.E.END_1_1,S.E.END_1_1
S25:B1.E.END_1_1,Bk.E.END_1_1
```

```
S26:B1.E.END_1_1,Bk.E.END_1_1,S.E.END_1_1
S27:B1.E.END_1_1,S.E.END_1_1
S28:B2.E.END_1_1
S29:B2.E.END_1_1,Bk.E.END_1_1
S30:B2.E.END_1_1,Bk.E.END_1_1,S.E.END_1_1
S31:B2.E.END_1_1,S.E.END_1_1
S32:Bk.E.END_1_1
S33:Bk.E.END_1_1,S.E.END_1_1
S34:S.E.END_1_1
```

We describe each atomic proposition:

B1.S.S.R_1_1 : Buyer 1 asks seller for price of an item

S.S.B1.R.B2.R_1_1 : Seller replies to both buyers quoting the price

B1.Branching_1_1 : Branching point to for buyer 1 to either haggle or continue

B1.S.S.R_2_1 : Buyer 1 haggles with seller for price

S.S.B1.R.B2.R_2_1 : Seller replies to both buyers either accepting buyer 1's haggle or aborting transaction

B2.Branching_1_1 : Branching point to either continue transaction or abort

B2.C_1_1 : While loop condition for Buyer 2's satisfaction of price

B1.C.Bk.InfR_1_1 : Buyer 1 consults Banker

Bk.InfS.B1.R_1_1 : Banker replies to Buyer 1

B1.C.B2.R_1_1 : Buyer 1 negotiates with Buyer 2 on the price

CH4.Parallel_1_1 : Parallel point for Buyers' payment

B2.MS.Bk.MR2_1_1 vBuyer 2 asks banker to wire the seller the agreed amount

Bk.MS2.S.R_1_1 : Banker wires Buyer 2's money to the seller

B1.MS.Bk.MR1_1_1 : Buyer 1 asks banker to wire the seller the agreed amount

Bk.MS1.S.R_1_1 : Banker wires Buyer 1's money to the seller

CH7.Parallel_1_1 : Parallel point for components' termination

B1.E.END_1_1 : Buyer 1 reaches termination

B2.E.END_1_1 : Buyer 2 reaches termination

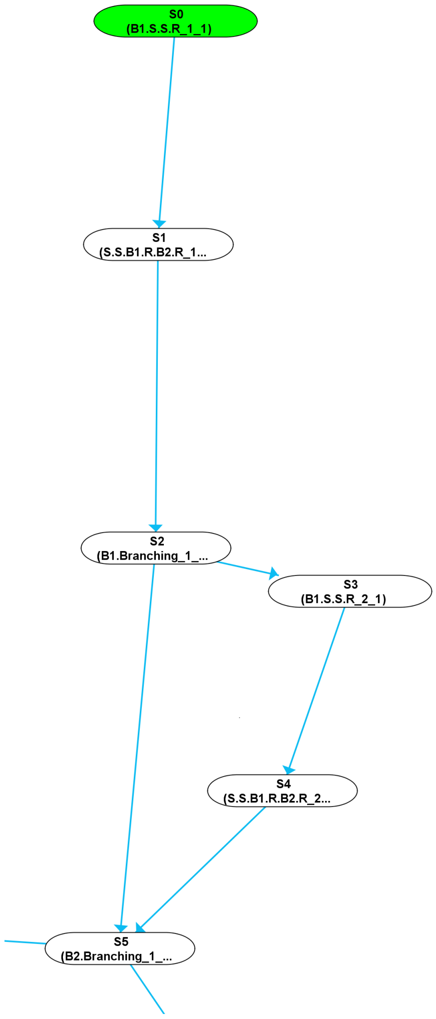
Bk.E.END_1_1 : Banker reaches termination

S.E.END_1_1 : Seller reaches termination

8.5 Explanation of the Kripke Structure

We explain below the various segments of the Kripke structure of the Buyer-Seller example.

8.5.1 Buyer 1 Haggle



S0: Buyer 1 asks Seller for price
S1: Seller replies to both Buyer 1 and Buyer 2

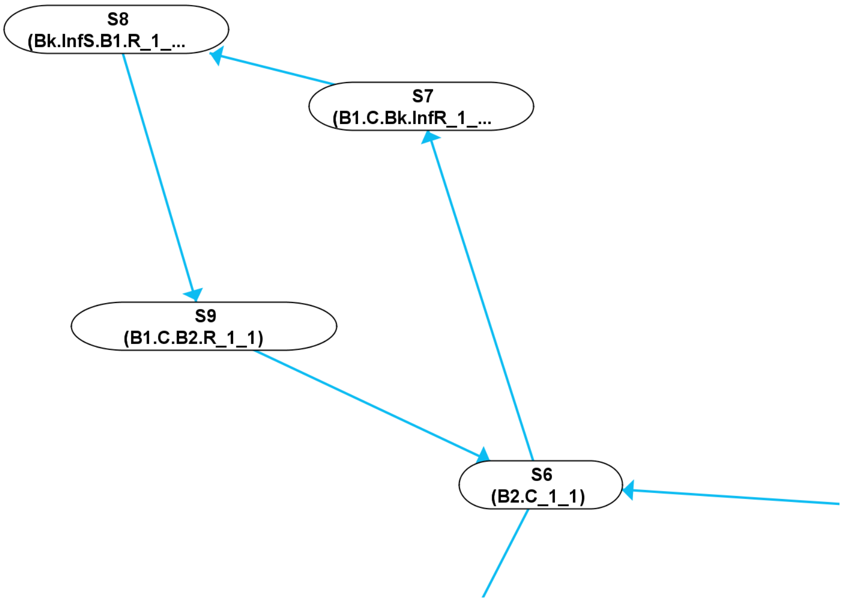
S2: Branching choice, Buyer will either decide to haggle or not

S3: Buyer 1 is haggling

S4: Seller replies to both buyers with either agreement to the haggle or calling off the transaction

S5: Branching choice, either we continue with the transaction or abort if Buyer 1 had haggled unsuccessfully

8.5.2 Banker Consulting

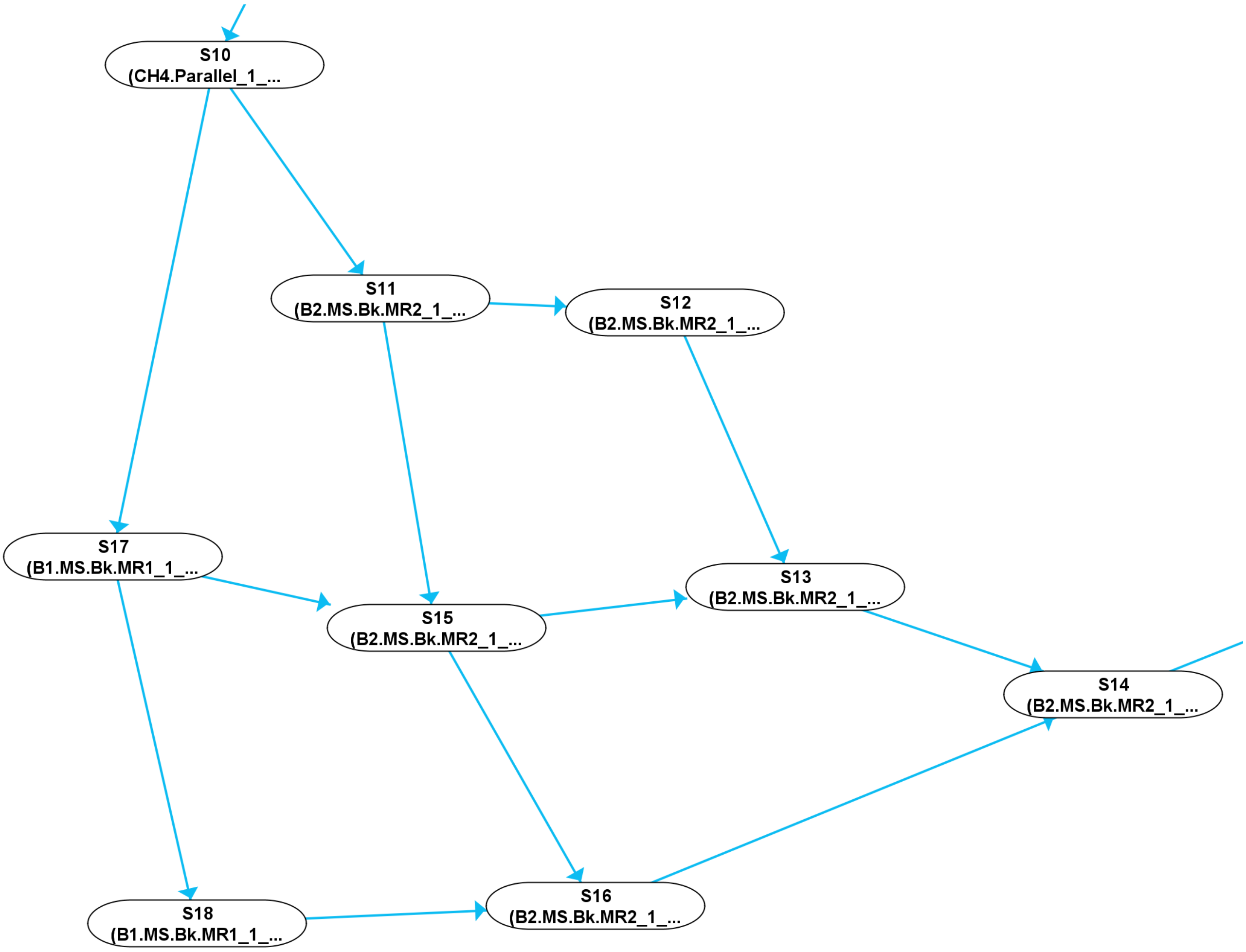


S6: Either Buyer 1 decided not to haggle ($S2 \rightarrow S5$) or haggled successfully ($S2 \rightarrow S3 \rightarrow S4 \rightarrow S5$)

S7, S8: Banker is consulting with Buyer 1

S9: Buyer 1 and Buyer 2 negotiate on price. Once Buyer 2 is satisfied with price, we exit the loop going from $S6 \rightarrow S10$

8.5.3 Buyers Authorize Payment in Parallel



All the routes from S10 to S14 are the different ways for $CH4 : CH5 \parallel CH6$ to happen

Case 1: $S10 \rightarrow S11 \rightarrow S12 \rightarrow S13 \rightarrow S14$, this simply $CH5 \bullet CH6$.

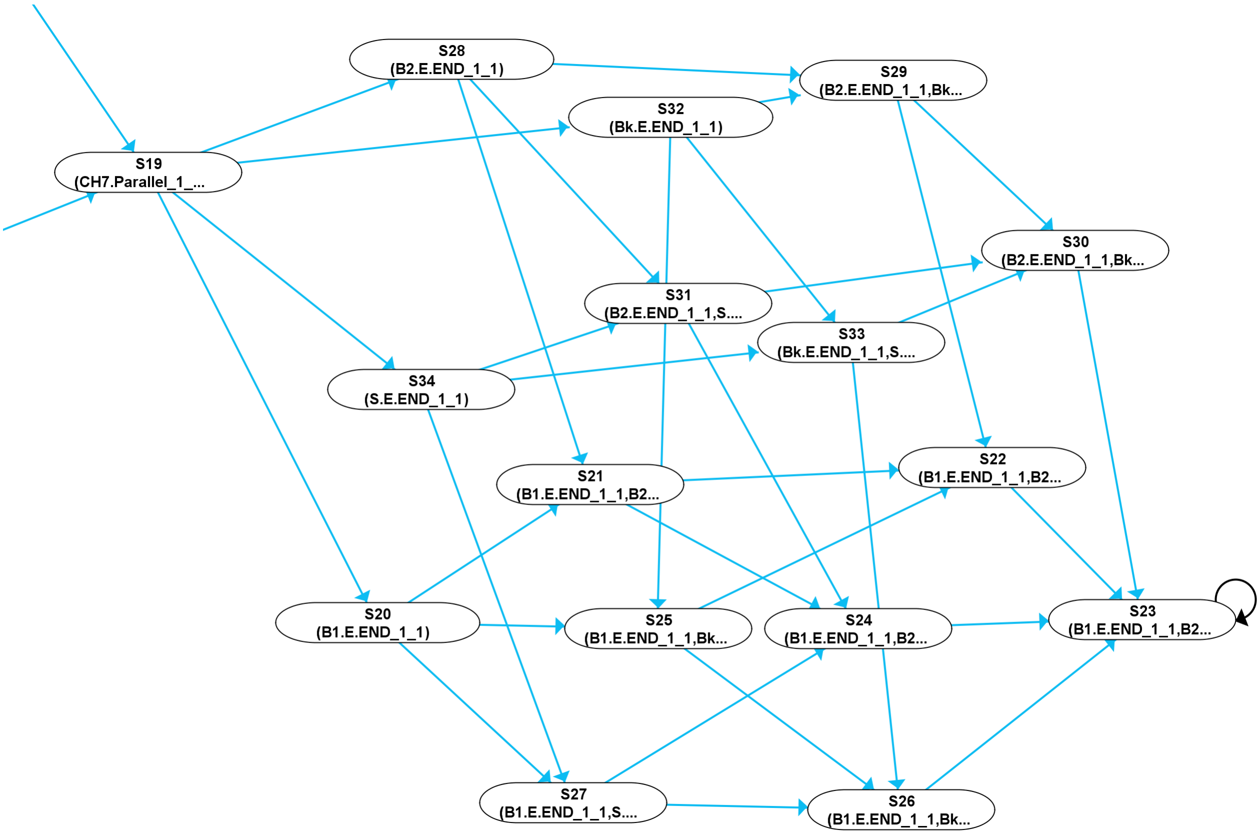
Case 2: $S10 \rightarrow S17 \rightarrow S18 \rightarrow S16 \rightarrow S14$, this simply $CH6 \bullet CH5$.

Case 3: Reaching $S15$ ($S10 \rightarrow S17 \rightarrow S15$ or $S10 \rightarrow S11 \rightarrow S15$) Both Buyers have asked for the money to wired (regardless of who asked first)

Case 3.1: $S15 \rightarrow S13 \rightarrow S14$ Banker decides to wire Buyer 2's money then Buyer 1's

Case 3.2: $S15 \rightarrow S16 \rightarrow S14$ Banker decides to wire Buyer 1's money then Buyer 2's

8.5.4 System Termination



$S19 \rightarrow S23$ are all the different ways for $CH7$ to happen

8.6 Properties Model Checked

Here are some of the properties model we checked in Eshmun for the above Kripke structure. They include the properties tested on the implemented system in [1] and some of our own.

1. **AlwaysTerminate:** for all paths, we will terminate.

$$\mathsf{AF}(B1.E.END\_1\_1 \,\&\, B2.E.END\_1\_1 \,\&\, Bk.E.END\_1\_1 \,\&\, S.E.END\_1\_1)$$

2. **AuthorizePayment:** no payment unless buyer requests.

$$(\mathsf{AG}(\mathsf{A}(!Bk.MS2.S.R\_1\_1 \;\mathsf{W}\; B2.MS.Bk.MR2\_1\_1))) \;\&\; (\mathsf{AG}(\mathsf{A}(!Bk.MS1.S.R\_1\_1 \;\mathsf{W}\; B1.MS.Bk.MR1\_1\_1)))$$

3. **EndPorts:** if a process reaches the end port, other processes will also reach their end ports.

$$\mathsf{AG}((B1.E.END\_1\_1 \mid B2.E.END\_1\_1 \mid Bk.E.END\_1\_1 \mid S.E.END\_1\_1) => \\ (\mathsf{AF}(B1.E.END\_1\_1 \,\&\, B2.E.END\_1\_1 \,\&\, Bk.E.END\_1\_1 \,\&\, S.E.END\_1\_1)))$$

4. **NoLivelock:** system doesn't suffer from livelock.

$$\mathsf{AF}(\mathsf{AG}(!B2.C\_1\_1))$$

5. **LoopOrTerminate:** either we follow a path that leads to a loop, or we terminate.

$$\mathsf{AG}(!B2.C\_1\_1 => ((\mathsf{EF}(B2.C\_1\_1)) \mid \\ (\mathsf{AF}(B1.E.END\_1\_1 \,\&\, B2.E.END\_1\_1 \,\&\, Bk.E.END\_1\_1 \,\&\, S.E.END\_1\_1))))$$

6. **NoLivelockAfterLoop:** after exiting a loop, the system doesn't enter livelock.

$$\mathsf{AG}(B2.C\_1\_1 => (\mathsf{AX}(!B1.C.Bk.InfR\_1\_1 => (\mathsf{AG}(!B2.C\_1\_1)))))$$

7. **PayOnce:** buyer only pays once.

$$(\mathsf{AG}(Bk.MS1.S.R\_1\_1 => (\mathsf{AF}(!Bk.MS1.S.R\_1\_1 => \\ (\mathsf{AG}(!Bk.MS1.S.R\_1\_1)))))) \,\&\, (\mathsf{AG}(Bk.MS2.S.R\_1\_1 => \\ (\mathsf{AF}(!Bk.MS2.S.R\_1\_1 => (\mathsf{AG}(!Bk.MS2.S.R\_1\_1))))))$$

8. **TermAfterExit:** after exiting the loop we will terminate.

$$\mathsf{AG}(B2.C\_1\_1 => (\mathsf{AX}(!B1.C.Bk.InfR\_1\_1 => \\ (\mathsf{AF}(B1.E.END\_1\_1 \,\&\, B2.E.END\_1\_1 \,\&\, Bk.E.END\_1\_1 \,\&\, S.E.END\_1\_1)))))$$

All of the above properties were model checked true, except for **AlwaysTerminate** and **NoLivelock**, which were modified to **NoLiveLockAfterLoop** and **LoopOrTerminate** respectively to better fit the scope of the example.

Chapter 9

Conclusions and Future Work

We were able to devise an operational semantics for choreographies, and used this to automatically generate the Kripke structure that gives the behavior of a choreography. We devised a method for verifying the correctness of choreographies which avoids state explosion. We also came up with the semantics and correctness for implementation of choreographies. Finally, we devised theorems which state that if the global choreography satisfy certain formulae, then the implemented system does as well (for sequential correctness). This was done for a sub-logic of CTL. Future work includes:

1. Write correctness theory for all operations of the choreography
2. Undertake more case studies
3. Increase the set of CTL formulae that can be verified
4. Consider infinite-state choreographies, i.e., the state variables have infinite domains

Bibliography

[1] R. A. Rayan Hallal, Mohamad Jaber, “From global choreography to efficient distributed implementation,” pp. 756–763, 2018.

[2] E. A. Emerson and E. M. Clarke, “Using branching time temporal logic to synthesize synchronization skeletons,” *Science of Computer Programming*, vol. 2, pp. 241 – 266, 1982.

[3] P. C. Attie, K. Dak-Al-Bab, and M. Sakr, “Model and program repair via sat solving,” *ACM Transactions on Embedded Systems*, pp. –, 2017.

[4] 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.