Implementing Choreography Extraction in Java

Bjørn Angel Kjær - bjkja17@student.sdu.dk

May 31, 2020

BADM500 - Bachelor Project in Computer Science

University of Southern Denmark
Department of Mathematics and Computer Science

Supervisor: Fabrizio Montesi - fmontesi@imada.sdu.dk

Bjørn Angel Kjær - bjkja17@student.sdu.dk University of Southern Denmark

Contents

1 Resumé

Choreographier er beskrivelse af systemer bestående af individuelle processer der kommunikerer med hindanden. Når man opbygger et sådan system har man ofte en beskrivelse af de individuelle processer, f. eks. i form a program kode. Processen at tage en beskrivelse af de individuelle processer og producerer en tilsvarende choreographi kaldes for choreography extraction. Projektet beskrevet i denne rapport beskæftiger sig med at at gen-implimenterer i Java, et program til choreography extraction originalt skrevet i Kotlin.

Projektet indebærer også at forstå algoritmen der implimenteres rent teoretisk, dels for at sikre implimentationen er korrekt, men også for at undersøge mulige optimeringer og udvidelser. Det lykkedes at implimenterer algoritment i Java, og lave en optimering af hukommelsesforbrug. Det færdige program er testet imperisk, og kører i mange tilfælde hurtigt nok til at den kan anvendes til praktiske formål

2 Introduction

Choreographies are high level representations of distributed systems that globally describe their execution. In other words, choreographies describe several intercommunicating and cooperating independent systems, as if was a single program. Usually, distributed systems are specified by the respective behaviours of individual processes, running in parallel to compose the complete system. Consider this pseudocode example from [2] inspired by the OpenID protocol [3].

u

```
procedure X:
  send cred to a
  offer to a:
    OK: receive token from w
    KO: call X
call X
```

a

```
procedure X:
  receive c from u
  if check(c):
    choose OK at u
    choose OK at w
  else:
    choose KO at u
    choose KO at w
    call X
call X
```

w

```
procedure X:
  offer to a:
  OK: send t to u
  KO: call x
call X
```

In this thesis, a composition of parallel process behaviours is called a network. From a network, it can be difficult to rule out the presence of deadlocks or livelocks. It can even be difficult to determine what the systems does. For example, the above pseudocode is much easier to understand with the additional information that a user process u tries to access a web-service w, by authenticating though an authentication service a.

Questions about the behaviour of a system are easier to answer if the system is instead described by a choreography. A choreography describing the above example could look like this:

```
def X = u.cred -> a;
    if a.check then
        a -> u[OK]; a -> w[OK]; w.t -> u
    else
        a -> u[KO]; a -> w[KO]; X

in X
```

The task of generating a choreography description of a system from its network description is called choreography extraction, or just extraction. An implementation for a choreography extraction that is usable in practice has already been developed [2]. The aim if this project is to re-implement it in Java, by using an existing implementation written in Kotlin [4] as a basis. Java is a better known and more used language, so re-implementing choreographic extraction in Java will make it more accessible, as more will be able to read the implementation, modify it to their needs, or contribute to the implementation as it is open source.

3 Theory

This section is closely based upon [2]. While much of this section is similar to the content [2], it is slightly less abstract in favour of explaining the concrete implementation of the extraction algorithm.

3.1 Describing Networks

A network represents individual processes running in parallel and communicating with one another. Networks are modelled using a set of process names $\{\mathtt{p}, \mathtt{q}, ...\}$; a set of expressions $\{e, e', ...\}$; a set of labels $\{l, l'...\}$; and a set procedure (function) names $\{X, Y, X', Y'...\}$. These sets are immutable in the model of networks for which this extraction algorithm applies.
The actions a process can perform are called behaviours. A behaviour can be thought of as an instruction a process might perform, relevant to choreography extraction. Behaviours are defined by the following grammar.

$$B ::= \mathbf{0} \mid X \mid \mathtt{p}!e; B \mid \mathtt{p}?; B \mid \mathtt{p} \oplus l; B \mid \mathtt{p}\&\{l_1 : B_1, ..., l_n : B_n\} \mid \mathtt{if}\ e\ \mathtt{then}\ B_1\ \mathtt{else}\ B_2$$

Each term do the following action within the process executing the behaviour.

- $\mathbf{0}$ is termination, and stops the process.
- X is procedure invocation. In invokes procedure (calls a function named) X.
- The send action $\mathtt{p}!e; B$ evaluates the expression e and sends the result to the process named $\mathtt{p}$. It then has the continuation B, meaning the next action the process will perform is another behaviour.

- The receive action $\mathtt{p}?;B$ receives a value from the process named $\mathtt{p}$ expecting $\mathtt{p}$ to perform the send action with this process as a target, then continuing as B.
- $\mathtt{p}\oplus l;B$ is called selection and sends the label l to process $\mathtt{p}$, then continues as B.
- The offering action $\mathtt{p}\&\{l_1 : B_1, ..., l_n : B_n\}$ offers a set of behaviours $B_1, ..., B_n$ which will be its continuation depending on which of the labels $l_1, ..., l_n$ it receives from process $\mathtt{p}$.
- $\mathtt{if}\ e\ \mathtt{then}\ B_1\ \mathtt{else}\ B_2$ is the conditional action. It evaluates a boolean expression e, and continues as B_1 if e evaluates to true. Otherwise it continues as B_2.

Using behaviours, a network can be formally defined as a map from process names to process terms $P, P', ...$ where each process term represents a running process. Process terms are of the form $\mathtt{def}\{X_i = B_i\}_{i\in I}\ \mathtt{in}\ B$ where I is a finite set of indices. Each B_i and B are behaviours, where B is the main behaviour; the initial behaviour the process executes. $\{X_i = B_i\}_{i\in I}$ assigns each procedure X_i to a corresponding behaviour B_i. The network N maps p_i to the process term P_i, such that $N(\mathrm{p}_i) = P_i$ for all $i \in [1, n]$. Such a mapping is denoted by $\mathrm{p}_1 \triangleright P_1 \mid ... \mid \mathrm{p}_n \triangleright P_n$.

As an example of a simple network, a process p that asks for another process q to respond could be formally defined as

$$\begin{aligned} p \triangleright \mathrm{def}\ X = \mathrm{q}!\mathit{greet};\ \mathrm{q}?;\ \mathbf{0}\ \mathrm{in}\ X \\ \mid q \triangleright \mathrm{def}\ X = \mathrm{p}?;\ \mathrm{p}!\mathit{resp};\ \mathbf{0}\ \mathrm{in}\ X \end{aligned}$$

3.2 Describing Choreographies

Choreographies are modelled much like networks with finite sets of process names, expressions, labels, and procedure names. The actions described by a choreography are called choreography bodies. The bodies can be though of as abstract instructions relevant to the execution of communicating processes. Choreography bodies are defined by the following grammar.

$$C ::= \mathbf{0} \mid X \mid \mathtt{p}.e \to \mathtt{q}; C \mid \mathtt{p} \to \mathtt{q}[l]; C \mid \mathtt{if}\ \mathtt{p}.e\ \mathtt{then}\ C_1\ \mathtt{else}\ C_2$$

The terms are similar to that of networks.

- $\mathbf{0}$ is termination
- X is procedure invocation
- $\mathtt{p}.e \to \mathtt{q}; C$ is communication. It evaluates e in process $\mathtt{p}$ and sends the result to process q, ten continues as C.
- $\mathtt{p} \to \mathtt{q}[l]; C$ is selection. Process $\mathtt{p}$ sends the label l to process $\mathtt{q}$, then continues as C. For each $l_i \in \{l_1, ..., l_n\}$ that $\mathtt{p}$ selects in $\mathtt{q}$ in a network, the extracted choreography will have a corresponding selection term $\mathtt{p} \to \mathtt{q}[l_i]; C_i$.
- $\mathtt{if}\ \mathtt{p}.e\ \mathtt{then}\ C_1\ \mathtt{else}\ C_2$ is the conditional term. The boolean expression e is evaluated in $\mathtt{p}$ and if it evaluates to true, it continues as C_1. Otherwise it continues as C_2.

A choreography is formally defined as $\texttt{def}\{X_i = C_i\}_{i \in I}$ $\texttt{in}$ C where I is some finite set of indices. $\{X_i = C_i\}_{i \in I}$ assigns each procedure to a choreography body, while C is the main or initial chorography body.

3.3 Extraction Algorithm

Extraction is done in two steps. First it construct a symbolic execution graph: A directed graph where each node represents a state of the network, and each edge represents the action that would make the network state from the source node into the network state from the target node. Then the graph is used to extract a choreography by traversing it recursively.

In the execution graph, procedure invocation (function calls) is not counted as a step, as it can be thought of as replacing the procedure name with the procedure definition. Such operations are called unfolding, and is done to process terms whose main behaviour is a procedure invocation. Unfolding is formally written as $\texttt{def}\ D\ \texttt{in}\ B \preceq \texttt{def}\ D\ \texttt{in}\ B'$ where D is the set of procedure definitions and B is a procedure invocation that defined as B'. For example, def $X = \text{q}!e; \mathbf{0}$ in X is effectively def $\emptyset$ in $\text{q}!e; \mathbf{0}$. Procedures can of course call other procedures, prompting repeated unfolding until a non-procedure invocation behaviour is reached. Repeated unfolding, or the transitive closure of $\preceq$, is denoted with $\preceq^*$.

A network without recursive procedures (loops) can be extracted from an unmarked Symbolic Execution Graph (uSEG).

Definition 3.1

A uSEG is defined as a directed graph with networks as nodes, edges with labels, and conform to the following conditions.

- Each node has at most two outgoing edges.
- If a node N has no outgoing edges then $N(\texttt{p}) \preceq^* \texttt{def}\ D\ \texttt{in}\ B$ for all processes p in N. Informally, all processes has terminated.
- If a node N_1 has one outgoing edge to node N_2 and the edge has the label $\texttt{p}.e \rightarrow \texttt{q}$ then for some B, B', D, and D'
  - $N_1(\texttt{p}) \preceq^* \texttt{def}\ D\ \texttt{in}\ \texttt{q}!e; B$
  - $N_1(\texttt{q}) \preceq^* \texttt{def}\ D'\ \texttt{in}\ \texttt{p}?; B'$
  - $N_2(\texttt{p}) = \texttt{def}\ D\ \texttt{in}\ B$
  - $N_2(\texttt{q}) = \texttt{def}\ D'\ \texttt{in}\ B'$
  - $N_1(\texttt{r}) = N_2(\texttt{r})$ for all other processes r
- If a node N_1 has one outgoing edge to node N_2 and the edge has the label $\texttt{p} \rightarrow \texttt{q}[l]$ then for some $B, B', D, D', B_1, ..., B_n$, and $l_1, ..., l_n$

  - $N_1(\texttt{p}) \preceq^* \texttt{def}\ D\ \texttt{in}\ \texttt{q} \oplus l; B$ where $l = l_i$ for some $i \in [1, n]$
  - $N_1(\texttt{q}) \preceq^* \texttt{def}\ D'\ \texttt{in}\ \texttt{p} \& \{l_1 : B_1, ..., l_n : B_n\}; B'$
  - $N_2(\texttt{p}) = \texttt{def}\ D\ \texttt{in}\ B$
  - $N_2(\texttt{q}) = \texttt{def}\ D'\ \texttt{in}\ B_i$ for some $i \in [1, n]$
  - $N_1(\texttt{r}) = N_2(\texttt{r})$ for all other processes r

- If a node N_1 has two outgoing edges labelled $\texttt{p}.e : \texttt{then}$ and $\texttt{p}.e : \texttt{else}$ going to node N_2 and node N_3 respectively, then for some B_2, B_3, e, and D
  - $N_1(\texttt{p}) \preceq^* \texttt{def}\ D\ \texttt{in}\ \texttt{if}\ e\ \texttt{then}\ B_2\ \texttt{else}\ B_3$
  - $N_i(\texttt{p}) = \texttt{def}\ D\ \texttt{in}\ B_i$ for $i \in [2, 3]$
  - $N_i(\texttt{r}) = N_1(\texttt{r})$ for $i \in [2, 3]$ and all processes r

- There are no other edge labels

A uSEG is constructed by creating an graph consisting of a node with the initial network, then expanding the graph by creating new nodes containing the network after "executing" the main behaviour, replacing the main behaviour with its continuation, or procedure definition depending on what kind of term it is. The edge between the nodes are labelled appropriately as to conform to the conditions above. If that is not possible, extraction is aborted. The constructed uSEG contains all possible paths the network can take during its execution. It is guaranteed to be finite because procedure calls can only be unfolded at the head of a process [1].

If the network has recursion (contain loops), whenever the construction of the graph would create a new node, it should instead create an edge to an existing identical node whenever possible. This prevents constructing infinite graphs from loops in the network. The loops however, creates the possibility that some processes are "forgotten", e.i., that they are not considered when the graph is being build. For example, the network

Network 3.1

$$
\begin{aligned}
&\texttt{p} \triangleright \texttt{def}\ X = \texttt{q}!msg;\ X\ \text{in}\ X \\
\mid\ &\texttt{q} \triangleright \texttt{def}\ X = \texttt{p}?\ X\ \texttt{in}\ X \\
\mid\ &\texttt{r} \triangleright \texttt{def}\ X = \texttt{s}!msg;\ X\ \text{in}\ X \\
\mid\ &\texttt{s} \triangleright \texttt{def}\ X = \texttt{r}?\ X\ \texttt{in}\ X
\end{aligned}
$$

might construct a uSEG where either $\texttt{p}$ never sends to $\texttt{q}$, or $\texttt{r}$ never sends to $\texttt{s}$ because the graph will form a loop before then.

To solve this problem, every process must perform some action within every loop of the graph. This is ensured by instead constructing a Symbolic Execution Graph (SEG).

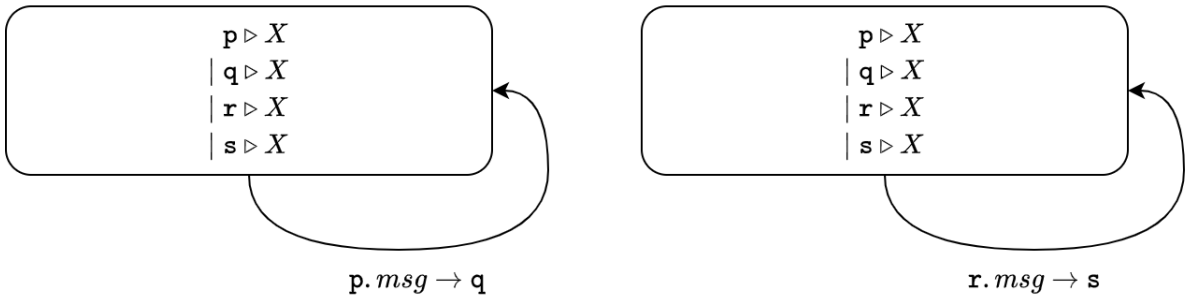


Figure 1: Two uSEGs with a loop where some processes are ignored.

Definition 3.2

A SEG is defined the same way as an uSEG, but every procedure call in the main behaviour of each process, either has the annotation • (marked) or ∘ (unmarked). The initial network has all procedure calls unmarked. If there is a edge from N_1 to N_2 and the changes from N_1 to N_2 requires the unfolding of a procedure call, then all procedure calls introduced by the unfolding are marked in N_2. If this would result in all procedure calls to be marked, the marking will be erased (all procedure calls will be annotated with ∘). This means that an SEG can contain nodes with identical networks, but with different markings. To close a loop while expanding a SEG, both the network and the marking needs to be identical.

Using the marking, it is possible to check if a SEG is valid.

Definition 3.3

A loop in a SEG is valid if:

1. The loop passes through a node where every procedure call is annotated with ∘.
2. The main behaviour of every process in each node (within the loop) contains at least one procedure call.

A SEG is valid if all its loops are valid. A network is extractable if there exists a valid SEG for that network.

Valid SEGs from network 3.1 would look like this.

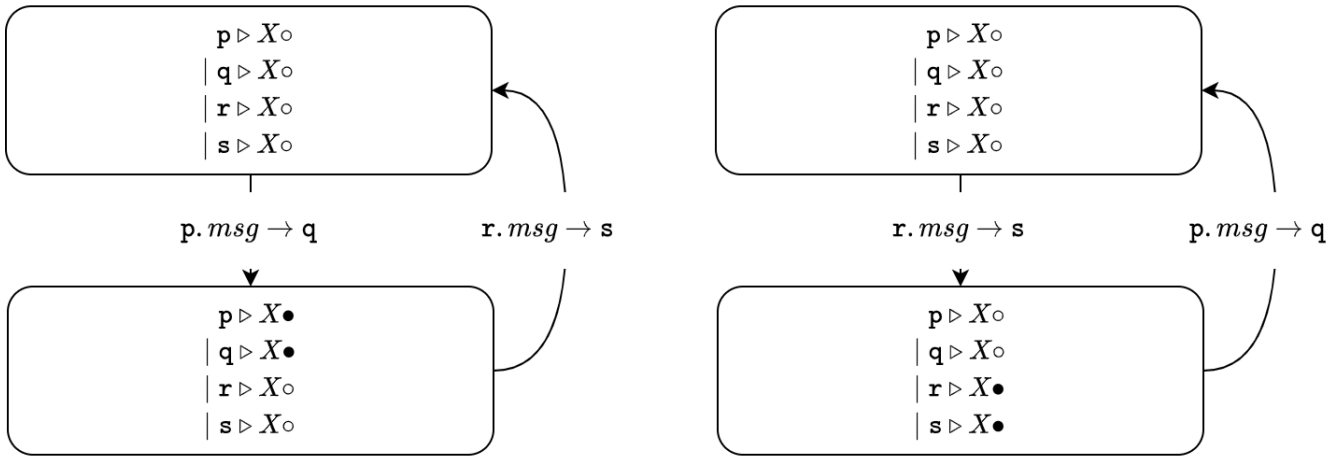


Figure 2: Two possible SEGs with a loop that includes all processes

When a valid SEG has been constructed from a network, it can be used to extract a choreography. The extraction performs the following steps

Removing loops: Every node N_i that has more than one incoming edge gets associated with a unique procedure identifier, say X_i. Then every edge pointing to N_i gets modified to point to a new leaf node with a special term X_i. The root node, that is the node with he initial network, is a special case. Instead of needing ate least two incoming edges, it only needs at least one. Since all loops in the graph have been removed, running an recursive algorithm on any node ensures it will terminate.

Defining extracted choreography: Let $\{X_i\}_{i\in I}$ be the set of procedure identifies from the previous step. The extracted choreography is then $\texttt{def}\ \{X_i = C_i\}_{i\in I}\ \texttt{in}\ C$. Each C_i is obtained by the following algorithm running on the node associated with X_i, while C is obtained by running the algorithm on the root node.

Choreography body extraction algorithm: This algorithm takes some node in an SEG as input and produces a choreography body.

- If the node is a leaf and a terminated network, return **0**
- If the node is a leaf and a special term X_i return procedure invocation X_i
- If the node has exactly one outgoing edge to N' and the edge label η is either represents communication or selection, then recursively extract choreography body C' from N' and return $\eta; C'$
- If the node has two outgoing edges to N_1 and N_2 labelled $\texttt{p}.e\texttt{:then}$ and $\texttt{p}.e\texttt{:else}$, recursively extract choreography bodies C_1 from N_1 and C_2 from N_2 and return $\texttt{if p}.e\ \texttt{then}\ C_1\ \texttt{else}\ C_2$

A choreography extracted this way is behaviourally equivalent to the initial network [1].

4 Implementing Choreography Extraction

4.1 Pre-processing Checks

To be able to extract a network, it must be well-formed. A well-formed network does not allow for self communications, nor communications with processes that do not exists in the network. Procedure calls must also only call procedures defined within the same process. If the network is not well formed it is not extractable, which can take a long time to detect. Checking for well formedness beforehand allows for quickly failing the extraction, and also allows for assuming the network is well-formed throughout the extraction process.

In the theory section, a uSEG and SEG are defined using repeated unfolding of procedure calls, until some non-procedure call action is found, noted with $\preceq^*$. The grammar for networks do not prevent recursiveness with only procedure calls however, and in that case $\preceq^*$ is not defined for that procedure call. If $\preceq^*$ cannot be defined for all procedure calls, no SEG is defined for that network. Therefore, any network containing a recursive loop which only procedure call actions fails extraction preemptively by the pre-processing check.

4.2 Overview

A graph with a single node containing the initial network is created. Then the method `buildGraph` is called on that node. `buildGraph` makes a list of possible actions to expand the graph with, either communication (sending or selection) or a condition. It automatically unfolds procedure invocations at the head of the processes' main behaviour before building the list of possible actions. Starting at the first action in the list, it tries to expand the graph, calling itself recursively on the node it creates, until all processes terminate, or a valid loop is formed. If it does so successfully it returns `OK`. If it cannot build the graph, it returns `FAIL` which means extraction is not possible. It may also return `BAD_LOOP` which means a loop cannot be closed to form a valid SEG. In that case, it removes the node it created, and tries with the next action in the list. If no actions lead to success, `buildGraph` returns `FAIL`.

When `buildGraph` tries to expand the graph, it calls two different methods depending on the action. If it is a communication, as in send, receive, selection, or offering, it calls `buildCommunication`. This function creates the network resulting from executing the action, and checks if a node with such a network already exists in the graph. If it does, it tries to add an edge to that node, returning `OK` on success. Otherwise, if the edge would not create a valid loop, it returns `BAD_LOOP`.
If on the other hand no node with an identical network exists, a node is created with the new network, and an edge pointing to it. Then `buildGraph` is called on that node. In this case, `buildCommunication` returns the result of the call to `buildGraph`.

If the action in question is instead a conditional, it calls the more complicated

`buildConditional`. This method first expands the `then` branch, and works like in the case of `buildCommunication`, except in the case of success, it do not return but do the same thing for the `else` branch. If both branches have been build successfully, then `buildConditional` can return `OK`. Otherwise, if building the `else` branch fails, the successfully builded `then` branch is removed before returning `FAIL` or `BAD_LOOP`.

If the initial call to `buildGraph` returns `OK`, then a valid SEG have been build and extraction can proceed. A function `urollGraph` replaces the loops with procedure definitions as descried in the theory section. Then, the function `buildChoreographyBody` is used to obtain choreography bodies for all the procedure definitions as well as the main body by calling it on the relevant nodes in the now tree shaped graph as described at the end of the theory section.

4.3 Closing Loops

When building the SEG, whenever an attempt is made to close a loop, the program needs to check that the loop is valid before being able to create it. Otherwise it needs to backtrack, and attempt expanding though some other action. This subsection explains how the implementation checks for loop validity.

4.3.1 Marking

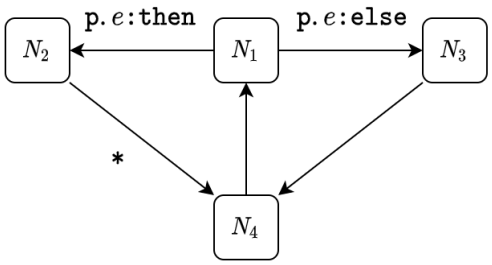
As described in the theory section, closing a loop based on networks being identical alone can lead to starvation of processes. Instead, each procedure invocation in the main behaviours should be annotated with $\circ$ (unmarked) and $\bullet$ (marked). To close a loop, the loop must contain a node where all procedure invocations are unmarked. In the actual implementation however, the processes are annotated instead. This is sufficient to ensure a valid loop as long as all processes are unmarked within some node in the loop [2, Lemma 1]. This removes the need to traverse behaviours to find procedure invocations, and forces processes with finite (non looping) behaviour to execute before a loop can be entered.
Some networks contains livelocks by design, but the extraction algorithm fails on the presence of livelocks as no valid loops can be formed. Processes which causes livelocks by design will be referred to as services. All services are always marked, which allows them to not execute any action within a loop. When the markings in a node is erased, the services stays marked. The edge with the action causing all processes to be marked, is then annotated as flipped so that loop validity can still be checked.

Each node is equipped with a list of bad nodes. The node in question is unable to form a valid loop by adding an edge to any nodes from that list. The initial node has itself in its bad node list. If a node N_2 is added from a node N_1, and all processes in N_2 are marked (thus erasing their marking) then the bad node list in N_2 is $\{N_2\}$. Otherwise, the bad node list in N_2 contains N_2 and all the nodes from the bad node list of N_1.
When attempting to add an edge between N and N', and the the edge has been flipped, the

edge creates a valid loop. If the edge is not flipped, but N' is not in the bad node list of N, the edge still forms a valid loop. In any other case the loop is invalid.

4.3.2 Choice Paths

Closing a loop using the marking technique as described above only guarantees loop validity for communications since those only has one outgoing nodes. Conditionals on the other hand could allow closing invalid loops. Consider the following SEG.



The building of the SEG starts at N_1 where a conditional is executed. The `then` branch is being build first, creating N_2. From N_2, N_4 is created, and in the process, all processes becomes marked, and therefore reset to unmarked, which is symbolized as $*$. N_4 do not have N_1 in its list of bad nodes, and can therefore create an edge to N_1 creating a valid loop. Then the `else` branch is created, first with N_3 which do not have N_4 in its list of bad nodes, and can therefore create an edge to, creating an invalid loop since not all processes in the loop $N_1 \to N_3 \to N_4 \to N_1$ are unmarked in any of the nodes.

To avoid this problem, a node can only add an edge to an existing node if that node is the current nodes predecessor in the graph. Every node gets annotated a choice path: A string of 0 and 1 that represents which conditional branches the graph expansion has been though. The initial node has the choice path "0", and all created nodes gets the same choice path as their predecessor, except in the case of conditionals. The node created from the `then` branch gets the choice path of its parent conditional with 0 appended, while the then branch gets the choice path of its parent conditional with 1 appended.

In **buildGraph** when checking if a node already exists in the graph, that node now also need to have is choice path be a prefix of the current nodes choice path. If no such node exists, a new one is created instead of creating a loop. This extra check ensures all closed loops are valid [2, Lemma 2].

5 Implementation in Java

5.1 Representing Networks and Choreographies

The grammars defining behaviours and choreography bodies have terms which contain other behaviours/bodies. To effectively structure the data, they are stored as an Abstract Syntax Tree (AST) which can be traversed using a visitor pattern. A visitor pattern is a method for working with trees, where a visitor performs some work on a node, then calls itself recursively to work on the children of the node. The implementation defines two interfaces for this purpose.

```
public interface TreeVisitor<TreeType, ReturnType> {
    ReturnType Visit(TreeType hostNode);
}

public interface TreeHost<HostType> {
    <T> T accept(TreeVisitor<T, HostType> visitor);
}
```

An abstract class `Behaviour` implements `TreeHost<Behaviour>` meaning all its subclasses can be visited by a visitor which implements `TreeVisitor<Behaviour, ReturnType>`. The type `ReturnType` depends on the type of data that the visitor want to collect. All the terms of the behaviour grammar has its own class extending `Behaviour`, which contains the relevant information for that term. `Behaviour` defines an enum `Action` with one entry for each subclass, and a method `getAction()` which returns what action the concrete class is. This allows the visitor to know the concrete type of the `Behaviour` it is looking at.
Two classes `ProcessTerm` and `Network` also extends `Behaviour` even though they are not behaviours, to simplify the code. `ProcessTerm` represents a process, and stores a hashmap from procedure names (Strings) to behaviours, as well as the process' main behaviour. `Network` stores a map from process names (Strings) to process terms.

Choreographies are structured much like networks with an abstract class `ChoreographyBody` which is a superclass for the classes representing each tern of the choreography body grammar. However `ChoreographBody` extends the abstract class `ChoreographyASTNode` which in turn implements `TreeHost<ChoreographyASTNode>`. `ChoreographyASTNode` define an enum `Type` and a function `getType()`. The classes `Choreography` and `ProcedureDefinition` extends `ChoreographyASTNode` but not `ChoreographyBody`, as unlike the case with networks, there are cases where it does not makes sense to consider them choreography bodies. `ProcedureDefinition` simply associates a procedure name (String) with a single choreography body. `Choreography` contains a list of procedure definitions, and the main choreography body.

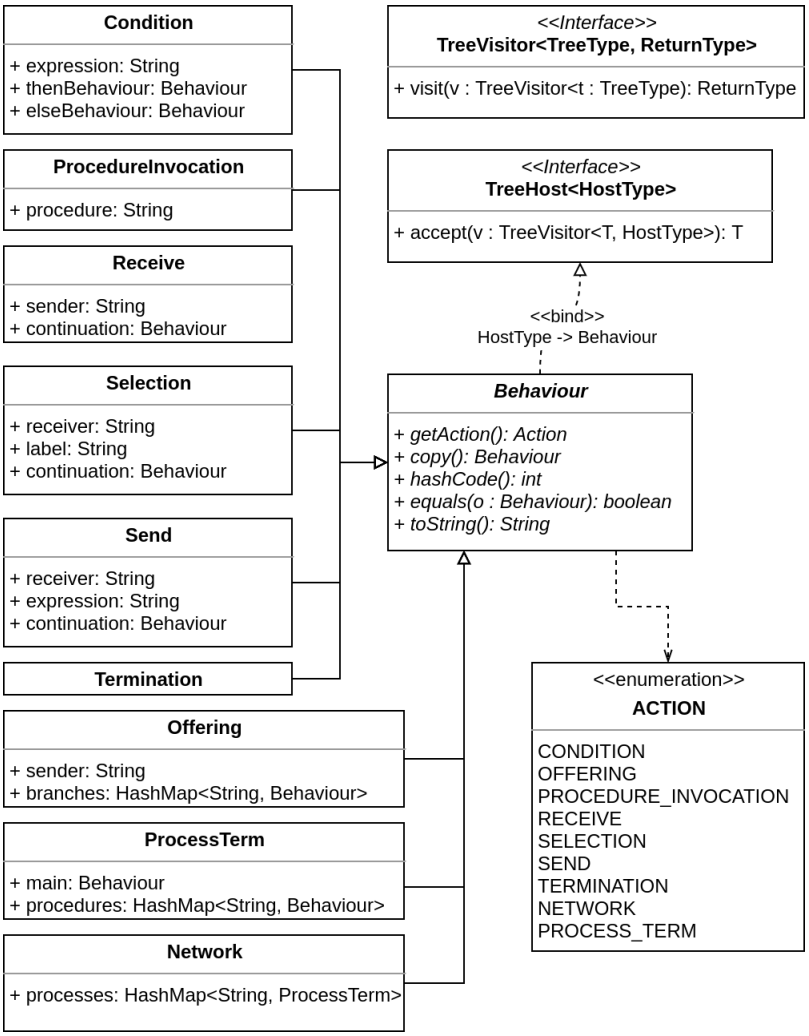


Figure 3: UML class diagram for networks

5.2 Parsing

The project uses antlr4 to generate parsers for behaviours and choreographies. Grammars for Networks and Choreographies, simmelar to the ones for behaviours and choreography bodies, are defined such that antlr4 can construct parsers that converts a String following those rules to a `Behaviour` and `ChoreographyASTNode` ASTs respectively. The `toString()` method for `Choreography` and `Network` has also been modified to produce a String following the same grammar, allowing to convert between text form for storage and readability, and AST form for practical work.

5.3 Interfacing with the Algorithm

The class `Extraction` is for interfacing with the extraction algorithm. It has two public functions: `extractChoreography` and `extractChoreographySequentially`. The latter works

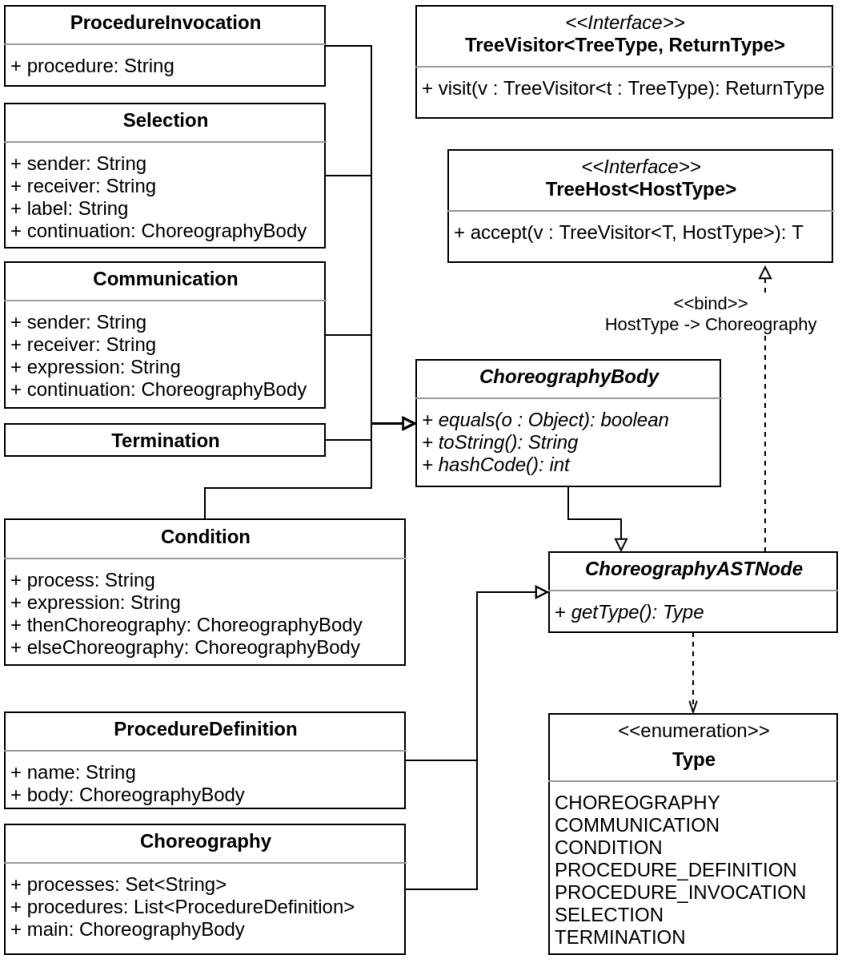


Figure 4: UML class diagram for choreographies

as follows. It takes a string representation of the network, and a set of process names that are services, as arguments. It then parses the network string to create a network AST. The network is then purges for "useless" processes. Specifically those that has termination, or procedure invocation which unfolds to termination, as their main behaviour. It then checks for well-formedness before building the Symbolic Execution Graph. From the SEG it then extracts a choreography, which it stores in a `Program`, which is a class containing choreographies, and statistics about the SEG each choreography is extracted from. The finished `Program` is the return value, and may not contain any choreography if extraction failed.

The function `extractChoreography` works the same way as `extractChoreoraphySequentially`, except it may extract multiple choreographies in parallel. The processes in the network are split into groups, such that all processes only communicate with processes within their own group. Then each group is made its own network. A choreography for every network is then extracted in parallel, and returns a `Program` containing them all. Note that if a network cannot be split, `extractChoreography` and `extractChoreographySequentially` behaves the same.

5.4 Building the Execution Graph

The SEG is build by creating a graph with a single node containing the initial network, then expanding recursively on that node, executing an action in some process' main behaviour. The class `ConcreteNode` contains the data for each node. It simply holds the following information: The node's network represented as an AST. The node's choice path represented by a string. An integer ID. A hashset containing the ID's of bad nodes. A hashmap from process names (strings) to their marking (boolean).
The initial node of course contains the initial network. Its marking is initialized to be true for services, and false for all other processes. Its choice path is "0", its ID is 0, and its set of bad nodes is empty. Subsequent node ID's will be one higher than the previously created node.
Two tightly coupled classes are used for building the Symbolic Execution Graph: `GraphProspecter` and `GraphExpander`. To put it simply, `GraphProspecter` chooses the next action to expand the graph on, while `GraphExpander` checks for loop validity and do any actual modifications to the graph. The function `buildGraph` is defined in `GraphProspector`,

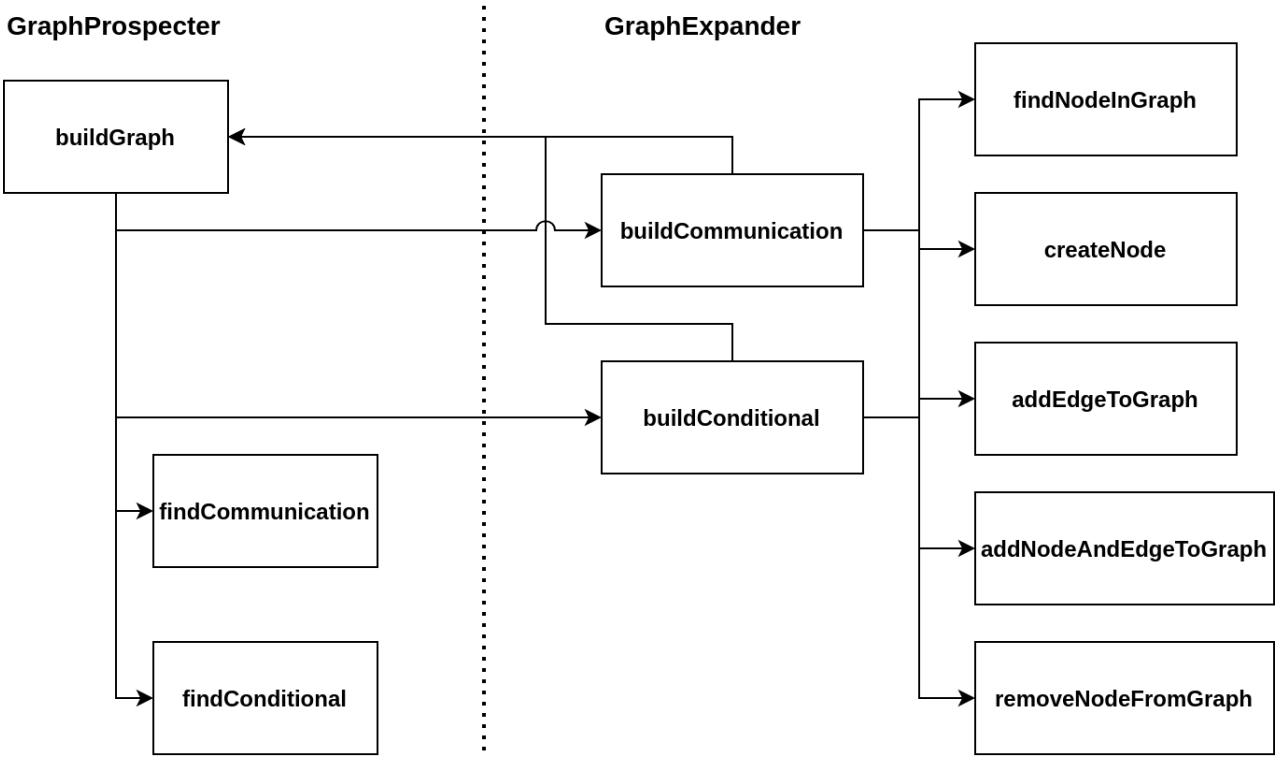


Figure 5: Simplified dependency tree

which calls `buildCommunication` and `buildConditional` which is defined in `GraphExpander`. Those two functions in turn call `buildGraph`, thus the tight coupling. In the original Kotlin implementation, all these functions, extraction, and interfacing was done in a single class. The split is intended to make the code more readable by segmenting the code based on purpose, making it easier to understand the relationships of the internal data structures and helper functions. Figure 5 shows a simplified dependency tree for building the graph.

5.4.1 GraphProspector and buildGraph

`buildGraph` is called with the initial network as argument. It starts by sorting the processes depending on some extraction strategy which are listed and explained in a later section. Every main behaviour is then unfolded, meaning if it is a procedure invocation, it is replaced by the procedures definition. Unfolding all procedure invocations, then folding back those that are unused appears to be more effective than unfolding as needed [4]. Then, for each process in their order of sorting, it tries to expand the graph by executing the main behaviour of that process. If the main behaviour is a communication, either sending, receiving, offering, or selection, it creates a target network which looks like the network after executing the action in the main behaviour. It also creates the edge label noting what action was executed. The creation of the new network and label are done using the helper function `findCommuication`. All processes except those involved in the communication is then folded back together. To continuing building the graph, it calls `buildCommunication` in `GraphExpander` with the target network, label, and current node as arguments. If `buildCommunication` returns `BAD_LOOP`, the next process is considered. If it returns `OK` or `FAIL`. `buildGraph` returns `OK` or `FAIL` respectively.
If on the other hand the main behaviour is a conditional, it works like in the case of communications, but with target networks and labels for both the then branch and else branch, which are all created by calling `findConditional`. All processes are then folded back, except for the one with the conditional. It then calls `buildConditional` in `GraphExpander` with the target networks and labels, and the current node. If `buildConditional` returns `BAD_LOOP`, the next process is considered. If it returns `OK` or `FAIL`. `buildGraph` returns `OK` or `FAIL` respectively.
If none of the processes' main behaviour can be used to expand the graph, either because it creates in an invalid loop, or because they are not communications or conditionals, it checks if the main behaviour of all processes are termination. If that is the case, `buildGraph` returns `OK`. Otherwise it returns `FAIL`.

5.4.2 GraphExpander, buildCommunication and buildConditional

As described in the above section, `buildGraph` asks `GraphExpander` to do the actual graph modifications by calling `buildCommuication` and `buildConditional`. Those functions works as follows.

`buildCommunication:`
This function attempts to expand the graph on executing a communication action. Recall the function arguments are the node the communication action is from, which will be called the source node, an edge label describing the communication, and a target network which the new node will contain. It first build a target marking, which is the marking that the new node should have. It is a copy of the original nodes markings, but where the sending and receiving process are also marked. If this results in all processes being marked, then they all become

unmarked, except for terminated processes and services, and the label is annotated as flipped. It searches the graph for an existing node which contain the target network, the target marking, and which has a choice path that is a prefix of the choice path of the source node. If it finds one, it checks if adding the edge would form a valid loop. It can do that if either the edge label is flipped, or if the found node is not in the bad node list of the source node. If adding the edge forms a valid loop, then it is added to the graph, and `buildCommunication` returns `OK`. If adding the edge forms an invalid loop, then it returns `BAD_LOOP`. If instead no matching node could be found, it will create a new one. The new node will contain the target network, and target marking. If the label is flipped, the new node's list of bad notes is empty. Otherwise its list of bad nodes will contain all entries from the source nodes bad node list, plus the source node itself. The choice path will be the same as the source node. The new node will be added to the graph, along with an edge from the source node to the new node. The edge will also store the edge label. In this latter case where a new node is created, `buildGraph` is called with the new node as argument. If `buildGraph` do not return `OK`, the new node is removed from the graph. This ensures that if `buildGraph` is unsuccessful, the graph is identical to before the call to `buildCommunication`. In any case, the return value of `buildCommunication` is the return value of its internal call to `buildGraph`.

`buildConditional:` This function is similar to `buildCommunication`. The process executing the conditional is marked, and if all processes are marked, both the then label and else label are annotated as flipped, and the target marking is erased like in `buildCommunication`. It first looks for a node matching the then network. If it finds it, and adding an edge from the source node to the then node would form a invalid loop, `buildConditional` returns `BAD_LOOP`. Otherwise it adds the edge.
If no such node is found, it creates one as described for `buildCommunication`, except the choice path for the new node is that of the source node appended with "1" for the then branch, and "0" for the else branch. It then calls `buildGraph` on the new node. If `buildGraph` do not return `OK`, it removes the new node, and returns the return value of `buildGraph`.

If creating the then branch is successful, `buildConditional` can do the same for the else branch. If building the graph on the else branch fails, it needs to remove both the else branch. If the then branch created an edge to form a loop, that edge is removed. If creating the then branch created at least one node, all those nodes are removed. The return value is therefore only `OK` if both the then branch and else branch could be created successfully.

5.4.3 Data Structures and Bookkeeping

The execution graph can become very large, so axillary data structures are implemented to remove the need to traverse the graph. A hashmap `nodeHashes` maps from integers to a list of `concreteNode`s. The integers represents a combined hash value of a network and its marking. When searching for an existing node in `buildCommunication` and `buildConditional`, a list of potential matches can be retrieved in linear time. The list still

needs to be searched for one with a prefix choicepath as hashing only works for exact matches, but it is still much fewer nodes. While searching, it is important to check if the network and marking matches too, as hashing runs the risk of collisions.
Whenever a new node is created, it is added to a list in `nodeHashes`. If a list for its hash do not already exists, one is automatically created. When a node is removed from the graph, it is also removed from its list in `nodeHashes`

When building the else branch in `buildConditional` fails, it needs to remove the then branch. For much of the SEG building, if a call to `buildGraph` do not return `OK`, any work in that iteration of the function call is undone, as to return with an unaltered graph. However, the call stack is lost when the then branch is build successfully, so simply removing the first node in the branch will leave the rest in the branches in the graph. This would not be a problem other than wasted memory, if those nodes did not also exists in `nodeHashes`. Therefore, a hashmap `choicePaths` from strings to lists of `concreteNode`s, maps every choice path to a list of nodes with that choice path. When the then branch is removed, a loop goes though all entries in the hashmap. If the key, that is the choice path for that list, has the choice path of the then branch as its prefix, all nodes in the list is removed from `nodeHashes` and the graph. The reason et goes by prefixes, and not exact matches, is because the then branch may contain conditionals which would result in nested branches with longer choice paths.

5.5 Extraction

After building the SEG, a choreography can be extracted. The function `buildChoreography` in the class `ChoreoraphyBuilder` takes the SEG and node with the initial network, which will be called the root node, as input, and produces an `Choreography`.

5.5.1 Unrolling the Graph

Extraction is performed by a recursive algorithm, which would not work on a graph with loops. Therefore the graph is turned into a tree where the loops are represented as a special procedure invocation node. The function `unrollGraph` takes root node as input and returns a list of these special procedure invocations. First a hashmap mapping from strings to `concreteNode` called `recursiveNodes` is created. For every node in the graph, it checks if there is at least two incoming edges, or at least one if it is the root node. If that is the case, the node is stored in `recursiveNodes` with a special unique procedure identifier as its key.

Then for each entry in `recursiveNodes` it creates a new `invocationNode`. This node stores the procedure identifier, as well as a reference to the `concreteNode` it is associated with. The `invocationNode` is added to the graph, and to a list of procedure invocations called `invocations`. For each incoming edge to the `concreteNode` referred by the newly created `invocationNode`, it replaces it with a new identical edge that points to the `invocationNode` instead. Finally, it returns `invocations`.

5.5.2 Building Choreography Bodies

The function `buildChoreographyBody` takes a node from the now tree-shaped SEG as input, and produces a choreography body AST. Its behaviour depends on the number of outgoing edges of the node argument.

- If there are no outgoing edges, then either the network has terminated, or the node is an `invocationNode`. In the former case, it checks if all processes has terminated. If they have, it returns a termination choreography body. Otherwise it throws an exception. If the node instead is a `invocationNode`, it returns a procedure invocation choreography body with the node's procedure identifier as its procedure.
- If there is one outgoing edge, the edge should either contain information regarding a communication or selection action. Depending on which one it is, it creates a new communication/selection choreography body using the information stored in the label. The continuation is build by calling `buildChoreographyBody` on the edge's target node.
- If there are two outgoing edges, the node represents a conditional. It returns a new conditional choreography body with the process and boolean expression obtained form the labels, and where the then and else choreography bodies are build by calling `buildChoreographyBody` on the target nodes for the then and else labels respectively.

5.5.3 Building the Choreography

`buildChoreograpy` calls `unrollGraph` and stores the returned list as the variable `invocationNodes`. This list contains every `invocationNode` in the graph. If one of the `invocationNodes` refer to the root node, then the main choreography body is defined as a procedure invocation choreography body with the procedure being that stored in that `invocationNode`. Otherwise, the main choreography is obtained by calling `buildChoreographyBody` on the root node.

A list of procedure definitions called `procedures`, is created. For every node in `invocationNodes`, it adds a new procedure definition to `procedures`. The procedure definition stores the name of the procedure obtained from the `invocationNode`, and the procedures definition by calling `buildGraph` on the `concreteNode` that the `invocationNode` refers to.

Recall that a choreography is defines as a mapping from procedure names to choreography bodies, as well as one main choreography body. A choreography can now simply be created from the main choreography body, and `procedures`. This choreography is the return value of `buildChoreography`.

6 Changes from the Kotlin Implementation

Language Differences:
Kotlin's runtime type checking capabilities, `when` statements, and automatic type casting is used extensively in the Kotlin implementation. Unfortunately, those features are not present in Java. The classes for representing choreography and network ASTs, as well as graph nodes and labels, are therefore equipped with some sort of identification function, which returns a enum value associated with their class, then manually typecasted. The many `when` statements have also been replaced by `switch` statements, which unfortunately makes the code less elegant.

Abstract Syntax Trees:
The ASTs in Kotlin are ad-hoc for the specific AST, while in the Java implementation they are defined using generic interfaces. The generic interfaces are intended to make the visitor pattern clearer. An abridged definitions of the network AST is shown here.

```
public interface TreeVisitor<TreeType, ReturnType> {
    ReturnType visit(TreeType hostNode);
}

public interface TreeHost<HostType> {
    <T> T accept(TreeVisitor<T, HostType> visitor);
}

public abstract class Behaviour implements TreeHost<Behaviour> {
    public <T> T accept(TreeVisitor<T, Behaviour> visitor) {
        return visitor.visit(this);
    }
    public enum Action {
        CONDITION,
        OFFERING,
        ...}
    public abstract Action getAction();
    public abstract Behaviour copy();
    ...
}

public class Condition extends Behaviour {...}
public class Offering extends Behaviour {...}
...
```

The interfaces for defining AST's and its visitors are unrelated to the actual network. Understanding the purpose of `TreeVisitor` and `TreeHost` is intended to make it immediately clear that `implements TreeHost<Behaviour>` means the class defines an AST. Without

reading the implementation of `Behaviour`, a developer can still understand it is intended to be read by a visitor, and that a class with `implements TreeVisitor<Behaviour, Integer>` goes though a tree of type `Behaviour` and return some integer value as result. Also, once the AST and visitor pattern is understood for networks, understanding it for choreographies is trivial as they use the same interface.

For comparison, an abridged definition of network the AST in the Kotlin implementation is shown below.

```
interface SPVisitor<T> {
    fun visit(n: ConditionSP): T
    fun visit(n: OfferingSP): T
    ...
}

interface SPNode {
    fun <T> accept(visitor: SPVisitor<T>): T
}

interface Behaviour : SPNode {
    fun copy(): Behaviour
    override fun hashCode(): Int
}

abstract class ActionSP(val process: String) : Behaviour

data class ConditionSP(val expression: String, ...) : Behaviour {...}
data class OfferingSP(val sender: String, ...) : ActionSP(sender) {...}
```

A developer seeing some class implementing `SPVisitor<T>` may not understand that the class is a visitor for networks. On the other hand, when understanding the theory, the Kotlin implementation is easy to make sense of.

Instance Re-usability:
When building the SEG, the network is modified for each node, which requires copying the existing network. Every class in the network AST therefore has a function `copy`, which returns a deep copy of itself. However, almost all fields in the network behaviour classes are `final`, which allows to use the flyweight design pattern to save memory. All networks classes, with the exception of `Network` and `ProcessTerm` have their `copy` function return `this` (a reference to their own instance) instead. This is equivalent to using deep copies as all copies will always be identical. The lower object count could also increase cache accuracy, thus increasing performance.

Command-line application improvements:
The Kotlin implementation comes with an command-line application to run tests and

benchmarks. The application needs to be run with a command-line argument specifying which test to run. Running it with no arguments prints help information. The Java implementation implements those same tests and benchmarks, also with an application that can run them through command line arguments. However, when running with no arguments, it not only prints help information, but allows the user to then type the name of the test they would like to perform. When a test is completed, it prompts for the next test, closing if the user types "exit".

```
$ java -jar -Xmx10G -Xss128M chor-extraction.jar
List of available commands (<name of command>   <description>)
        bisimcheck              Check that the choreographies extracted in the benchmark are correct,
i.e., they are bisimilar to the respective originals
        exit            Closes the application
        lty15           Run the tests from the paper [Lange, Tuosto, Yoshida @ POPL 2015]
        help            Prints this help information
        benchmark               Run the extraction benchmarking suite
        lty15-seq               Run the tests from the paper [Lange, Tuosto, Yoshida @ POPL 2015] *wit
h parallelization disabled*
        theory          Run the tests from the original theoretical paper [Cruz-Filipe, Larsen, Montes
i @ FoSSaCS 2017]
Enter command:
```

```
Enter command:
theory
Running l1
The extraction.network is well-formed and extraction can proceed
The input network has successfully been split into parallel independent networks
Graph building result: OK
input network:
       p { def X {q!<e>; q!<e>; q!<e>; X} main {X}} | q { def Y {p?; p?; Y} main {p?; Y}}
Output choreography
       def X1 { p.e->q; p.e->q; p.e->q; p.e->q; p.e->q; p.e->q; X1 } main {X1}
Elapsed time: 97ms

Running l2
The extraction.network is well-formed and extraction can proceed
The input network has successfully been split into parallel independent networks
Graph building result: OK
input network:
```

Figure 6: Interactive version of the testing and benchmarking application

Architectural changes:
Function and variable dependencies have remained mostly the same as in the Kotlin implementation, however the extraction part of the code have been majorly restructured. In the Kotlin implementation, everything was collected in a single class `Extraction`. This huge class with 800 lines of code performs every step of extraction, which goes against the single responsibility principle. In the Java implementation, extraction is split into four classes. The class `Extraction` is for interfacing with the algorithm, and preparing extraction by parsing the input string, and checking for well formedness. The SEG is build between two classes. One chooses the next action to expand the graph with, and builds the network resulting from executing the action. The other then searches the graph, looking for the possibility to close a valid loop, or building a new node if no valid loop can be closed, then asking the first class to

build the graph from that node. Finally, if a valid SEG was build, the last class unrolls the graph and builds a choreography from it.

Another architectural change is the way that the choreography and network Abstract Syntax Trees are structured. In the Kotlin implementation, networks and choreographies are in their own package, which is further divided into an interface package, and a node package. In the Java implementation, networks and choreographies also have their own package, but they share the interface related to Abstract Syntax Trees, as that interface is generic, and unspecific to both networks and choreographies. They both then have a common abstract superclass which implements the AST interface. This allows to work with their object oriented representation using generics, without relying on the AST interface. Since Java do not have the type checking features of Kotlin, and much of the code depends on what type these nodes are, an easy method for determining which class instance a node was. Networks all have a method `getType()` which returns an enum value corresponding to their class. It is defined as an abstract method in their common superclass, so it can be called even when they are treated as generic objects. Choreographies have a method `getType()` which works the same way.

7 Testing and Evaluation

This implementation have been tested on the same choreographies as those randomly generated for testing in [2]. The goal is to figure out if the implementation runs fast enough, and with small enough memory usage, to be usable in practice. In [2], the tests where performed on a computer running Arch Linux, kernel version 5.3.7, with an Intel Core i7-4790K CPU and 32 GB of RAM- The tests in this section where performed on a computer running Pop!_OS 19.10 kernel 5.3.0, with an Intel Core i7-9750H and 16 GB of RAM. Because of the difference in systems, extraction time is not comparable to those in [2]. The smaller RAM size also meant that not all tests could be completed, although the vast majority did. For testing, the JVM was allocated 10 GB heap and 128 MB stack. Increasing the stack size is necessary to run some of the tests, although less stack and heap was sufficient in most cases.

7.1 Strategies

When building the SEG, it attempts to expand it by executing the main behaviour of some process. The process it chooses, and the process it tries next if the previous one lead to a bad loop, depends on which extraction strategy is used. The implementation offers several ways to sort processes, and the graph building then tries to expand on processes in their order of sorting. The the following are the implemented strategies:

- Random: Shuffles the order of processes
- LongestFirst: Sorts the process with the longest main behaviour first

- ShortestFirst: Sorts the process with the shortest main behaviour first
- InteractionsFirst: Sorts processes with interactions first
- ConditionalsFirst: Sorts processes with conditionals first
- UnmarkedFirst: Sorts unmarked processes first
- UnmarkedThenInteractions: Sorts unmarked processes as first criteria, then interactions as a second criteria.
- UnmarkedThenSelections: Sorts unmarked processes as first criteria, interactions as second criteria, and communications as a third criteria.
- UnmarkedThenConditionals: Sorts unmarked processes as first criteria, and conditionals as a second criteria.
- UnmarkedThenRandom: Sorts unmarked processes first and shuffle their order, then anything else last, also in shuffled order.

7.2 Extraction Times

The average, median, and longest extraction times in milliseconds are listed in the table below. 910 networks where extracted once for every strategy.

Strategy	Average	Median	Longest
Random	941	25	301,251
LongestFirst	6119	1617	406,024
ShortestFirst	6090	1573	309,785
InteractionsFirst	859	25	263,144
ConditionalsFirst	1093	24	362,652
UnmarkedFirst	1154	21	331,765
UnmarkedThenInteractions	1210	23	373,128
UnmarkedThenSelections	774	30	227,194
UnmarkedThenConditionals	906	27	280,213
UnmarkedThenRandom	827	24	246,450

All strategies, except ShortestFirst and LongestFirst, have a median extraction time between 20-30 ms, which should be fast enough to use for practical applications. The average extraction times of at most 6 seconds, around 1 for most strategies, is also quite fast, although the large difference between average and median means some networks takes a very long time to extract. Indeed 300-400 seconds where some of the longest extraction times, which while very long compared to most of the networks, still might be fast enough to use for occasional static analysis. Not all of the networks used for testing in [2] was tested because of memory constraints however, and due to the exponential nature of the algorithm, extracting those may be too slow for practical purposes. On the other hand, the network which took the longest to extract (using InteractionsFirst) is 143 KB in size which is quite large considering the model for networks only contain information relevant to extraction.

The strategies LongestFirst and ShortestFirst have significantly longer average and median extraction times. This might not be because of the strategy itself, but rather because of the way they are implemented. They both sort processes by the length of their main behaviour. In the implementation, the length is obtained by converting the main behaviour to a string representation, then comparing the length of the strings. Converting to its string representation which is expensive when it is done for every comparison, so a more optimal implementation should be tested for fairer comparison with the other extraction strategies.

7.3 Comparison With the Kotlin Implementation

Running the same 910 tests with the Kotlin implementation yields the following results.

Strategy	Average	Median	Longest
Random	1236	47	371,346
LongestFirst	3633	334	374,886
ShortestFirst	3090	291	287,154
InteractionsFirst	1231	45	329,218
ConditionalsFirst	1377	46	410,041
UnmarkedFirst	1096	40	315897
UnmarkedThenInteractions	1020	41	293,717
UnmarkedThenSelections	1046	44	296827
UnmarkedThenConditionals	1126	43	341,551
UnmarkedThenRandom	1088	42	325123

The Java implementation runs on average roughly 300 ms faster for Random, InteractionsFirst, and ShortestFirst. The median and longest running time for these strategies runs faster in the Java implementation as well. The Kotlin implementation is significantly faster on ShortestFirst and InteractionsFirst on the other hand. Interestingly, the average running time for UnmarkedFirst and UnmarkedThenInteractions are fastest in Kotlin, while the median and longest running times are fastest in Java. The remaining strategies performed best in Java.

Overall it appears like the Java implementation runs slightly faster than the Kotlin implementation, possibly because of memory optimizations which reduces the amount of data being copied, thus requiring fewer operations, and possibly also reducing cache misses.

8 Conclusion and Further work

Choreography extraction was successfully (re)-implemented Java, based upon the existing implementation in Kotlin [4]. The type checking features of Kotlin unfortunately made the resulting Java code less elegant in some places, but is functionally the same.

The Abstract Syntax Trees of networks and choreographies was changed to use more generic

interfaces, and extraction had a major architectural change to split functionality into multiple classes for better overview and readability. Finally the flyweight design pattern was used to reduce memory usage.

The Java implementation seems to run slightly faster to the Kotlin implementation, and fast enough that it could be considered for practical applications. Further performance improvements might be achievable by selecting the best strategy for a given network, or part of a network. Possibly though heuristic analysis or machine learning. The implementation could also be expanded to use other linguistic features than those described by the grammars from [2].

Long term, the implementation could become a library such that developers can utilize choreography extraction without understanding, or needing to know, how the implementation works. It might also be possible to translate source code to networks, as to build an application for extracting choreography descriptions for real software. Such an application could be used to guarantee the absence of deadlocks, or give software architects a description of a system consisting of parts not designed to integrate with each other.

References

[1] Luís Cruz-Filipe, Kim S. Larsen, and Fabrizio Montesi. The paths to choreography extraction. In Javier Esparza and Andrzej S. Murawski, editors, *FoSSaCS*, volume 10203 of *LNCS*, pages 424–440. Springer, 2017.

[2] Luís Cruz-Filipe, Fabrizio Montesi, and Larisa Safina. Implementing choreography extraction. *CoRR*, abs/1910.11741, 2019.

[3] OpenID Foundation. OpenID Specification. https://openid.net/developers/specs/, 2014.

[4] Larisa Safina. *Formal Methods and Patterns for Microservices.* PhD thesis, University of Southen Denmark, 2019.