Imperial College London

Departement of Computing

Graphical Representation For Global Protocols

by

Charlotte Pichot

Submitted in partial fulfilment of the requirements for the MSc Degree in Advanced Computing of Imperial College London

September 2012

Abstract

A session encapsulates a safe communication scheme between two or more participants in a context of distributed systems. We associate to this structure a typing system in order to statically check programs. Multiparty session types distinguish global types, which ensure coherence of the scenario, and local types, which specify what the end-point programs should conform. The language Scribble has been developed to specify communication protocols supported by this framework. We focus for this project on global types, as well as on the Scribble specification for global protocol: we design graphical notations to match these protocols. Our goal is to develop a tool to obtain a correspondence between global protocols in Scribble and their graphical representation.

Therefore, we establish notations for graphs that correspond to global protocols in Scribble and that are supported by generalised multiparty session types. We provide also the implementation of a software tool to perform the transformation from global protocols in Scribble to graphs, and vice versa. The correspondence is based on a library of graphical objects in Python. To go further with this work on graph, we introduce new notations to represent a notion of time. Therefore, we extend the syntax for Scribble protocols, as well as for generalised global and local types with clocks. We clearly define structures to express timeouts for message passing, delays and other time constraints. We define a temporal satisfiability criterion for clock conditions and conjecture progress and error-freedom properties for processes.

As far as we know, graphical representations for global protocols has not yet been developed, neither extensions of multiparty session types with time. This report presents all the work that has been conducted to support these contributions.

Acknowledgements

I thank my supervisor Dr Nobuko Yoshida for her proposal and for introducing me in her research team. Her support has helped me conducting my project until the end. I also thank Rumyana Neykova for her time and explanations to help me understand and define the scope of my project. I enjoyed the discussion with both of them about important issues concerning the project.

I thank Dr. Iain Phillips to accept being the second marker for my individual project.

Furthermore I have really appreciated the meetings of the "Mobility Reading Group". I have learned a lot during the presentations and discussions about various topics linked to my project. Therefore I thank all the members of the group.

Finally I thank Guillaume Lozachmeur for all the discussions and the good moments we shared during this period, having the same supervisor.

Contents

Introduction

Session types, a new structure with an associated typing system, were first introduced by [21] to safely model communication in distributed networks. There was indeed a need in distributed programming for a clear structure to express conversations. For this purpose, we have seen a great development in the area of distributed systems over the past few years.

Networks head more and more to distributed architectures. They are continuously growing, connecting various devices together. For instance, we use in our daily life, not only computers and smart phones, but also devices, like smart meters connected to the electric grid. At a lower level, even the internal architecture of a cell is now designed as a distributed system. Therefore lots of engineers are interested in development of distributed systems, which seem to allow greater performance, as well as providing a more suitable model for physical systems.

In this context, to be connected altogether, we need to develop communication features for these systems: this is the past few years new challenge. Furthermore, as soon as communication is concerned, we must ensure some fundamental properties like safety, that is there should be no deadlocks, no mismatches and processes should be able to progress. Session types aim at solving these issues.

Some of the courses offered as part of the master degree Advanced Computing bring us the basics on the topic. We start studying structures for modelling conversations among several processes. In the course Models of Concurrent Computation, we discovered some ways of modelling interactions between processes with the study of CCS and Pi-calculus. CCS is a specification language for describing concurrent communicating processes. It gives rise to model-checking tools for proving properties about concurrent systems. On the other hand, Pi-calculus evolved from CCS. It models the changing connectivity of interacting systems and forms the basis of languages for supporting distributed concurrent programming. Along with the specification of processes using these languages, we are interested in their behaviour and in particular in bisimulations between processes. More generally, as far as finite state machines are concerned, bisismulations support verification aspects of a program, that is checking that the behaviour of an implementation matches a specification. As well as being a semantical issue, it refers to a complexity issue. We want it to be tractable and we learnt in the Complexity course that Bisimulation is a P-complete problem. Finally, in the same way as researchers developed typing systems for programming languages, such as Java and CC+, as we learnt in the Advanced Issues in Object Oriented Programming Languages course, there is a need for sessions, modelling conversations, to have an associated typing system to ensure safety properties.

The on-going research on session types does already handle a significant amount of communicating features: theoretical work has been done on the syntax and the operational semantics of the calculus, as well as on proofs for key properties and theorems. However, for the moment, there is no agreed representation of these sessions. Lots of work has been done on graphical representation of workflows and also of local processes, but not on whole conversations. That is why we have been focusing on this aspect for the project.

Motivations

First we explain with some details the motivation of our work. Along with the theoretical development of session types, a software tool chain is being implemented. It aims at generating all the necessary code from a global specification. From an engineer point of view, it is also more accurate to specify a scenario with a graphical tool rather than using some specific language. Therefore our goal is to develop such a graphic tool.

Example: Data Acquisition Use Case

We consider a real world use case. Starting from this example we present the steps of the development for a given scenario. We choose for a use case one currently used in session types research team as part of a partnership with the Ocean Observatories Initiative (OOI). The use case is as follows: UC.R2.13 “Acquire Data From Instrument” from OOI Use Case library (Release 2). We give here the scenario as described in [5].

"This use case describes a scenario where a user program (U) is connected to the Integrated Observatory Network (ION), which provides the infrastructure between users and remote sensing instruments.

The user requests, via an ION agent service (A), the acquisition of data from an instrument (I), e.g. a temperature sensor, registered with the ION. One of two data acquisition methods can be requested by U. One is push mode, where the data is streamed by I (driven by I’s clock). The other is poll mode, where the data is periodically polled by A (driven by A’s clock) on behalf of U. However, I may support only one of these modes. If not supported, A is responsible for emulating the requested mode over the available one. In both modes, A formats and relays the acquired data to U as a stream; U can interrupt the stream at any point to end the conversation."

Engineers need then to formalise these narrative specifications in order to develop some software. This is where our project brings a solution. We offer a tool to graphically represent the scenario, an easier way to formalise it. In this case, we could obtain the graph in Figure 1.

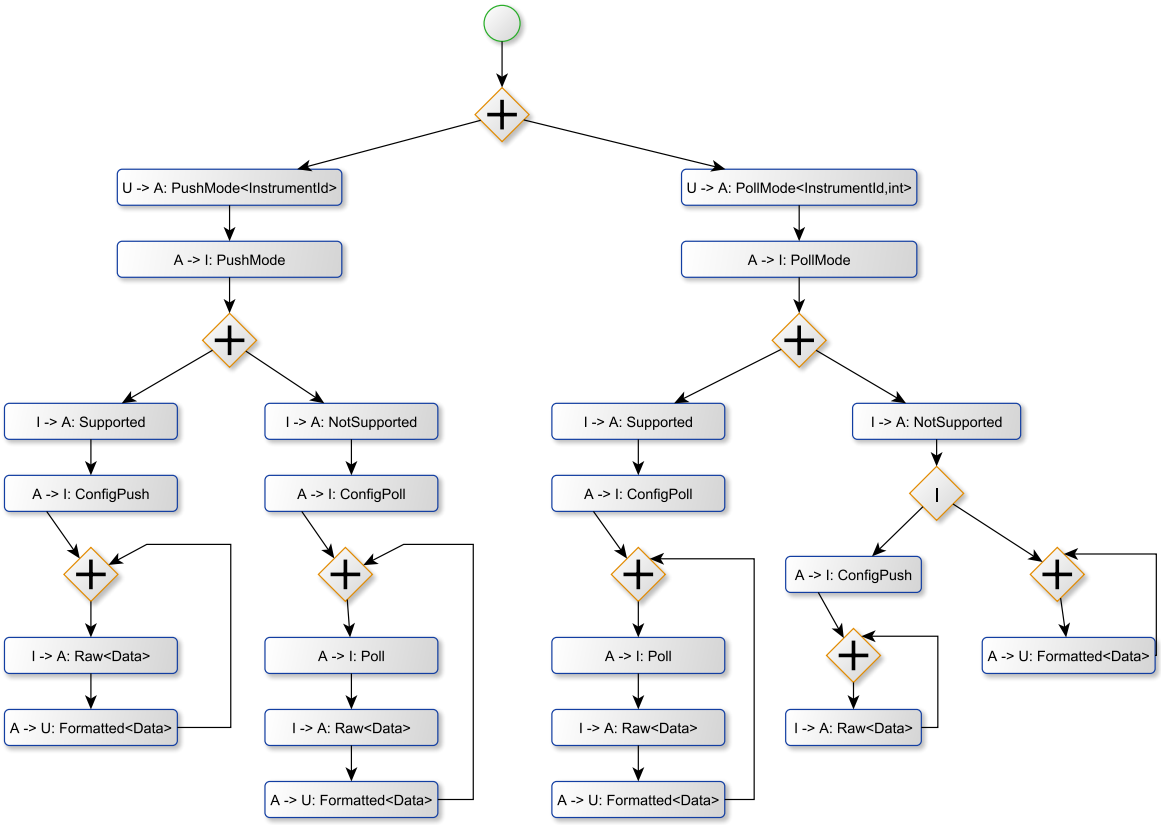


Figure 1: The choreography for the OOI “Acquire Data from Instrument” use case

To garantee the safety of this tool, our representation is supported by generalised multiparty session types. Therefore we have established global types for this scenario (Figure 2). All x_i are state variables. x_0 represents the initial state. $x_0 = x_{push} + x_{poll}$ is a choice: U will make the choice between continuing with x_{push} or x_{poll}. $x_{pushsup2} + x_{pushsup3} = x_{pushsup4}$ is a merge and in this case expresses recursion. $x_{pollnsup1} = x_{pollnsup2} \mid x_{pollnsup3}$ is a fork to model interleaving of actions at $x_{pollnsup2}$ and $x_{pollnsup3}$.

We can see the exact match between the graph representation and the global protocol: indeed, edges are state variables and nodes are the constructions such as interaction, choice, join, etc. For instance, if we

$$
\begin{array}{lrcl}
\mathrm{G} = \ \mathrm{def} & x_0 & = & x_{push} + x_{poll} \\
& x_{push} & = & \mathrm{U} \to \mathrm{A}\text{: PushMode}\langle\ \text{InstrumentId}\ \rangle; x_{push1} \\
& x_{push1} & = & \mathrm{A} \to \mathrm{I}\text{: PushMode}\ ; x_{push2} \\
& x_{push2} & = & x_{pushsup} + x_{pushnsup} \\
& x_{pushsup} & = & \mathrm{I} \to \mathrm{A}\text{: Supported}; x_{pushsup1} \\
& x_{pushsup1} & = & \mathrm{A} \to \mathrm{I}\text{: ConfigPush}; x_{pushsup2} \\
& x_{pushsup2} + x_{pushsup3} & = & x_{pushsup4} \\
& x_{pushsup4} & = & \mathrm{I} \to \mathrm{A}\text{: Raw}\langle\ \text{Data}\ \rangle; x_{pushsup5} \\
& x_{pushsup5} & = & \mathrm{A} \to \mathrm{U}\text{: Formatted}\langle\ \text{Data}\ \rangle; x_{pushsup3} \\
& x_{pushnsup} & = & \mathrm{I} \to \mathrm{A}\text{: NotSupported}; x_{pushnsup1} \\
& x_{pushnsup1} & = & \mathrm{A} \to \mathrm{I}\text{: ConfigPoll}; x_{pushnsup2} \\
& x_{pushnsup2} + x_{pushnsup3} & = & x_{pushnsup4} \\
& x_{pushnsup4} & = & \mathrm{A} \to \mathrm{I}\text{: Poll}; x_{pushnsup5} \\
& x_{pushnsup5} & = & \mathrm{I} \to \mathrm{A}\text{: Raw}\langle\ \text{Data}\ \rangle; x_{pushnsup6} \\
& x_{pushnsup6} & = & \mathrm{A} \to \mathrm{U}\text{: Formatted}\langle\ \text{Data}\ \rangle; x_{pushnsup3} \\
& x_{poll} & = & \mathrm{U} \to \mathrm{A}\text{: PollMode}\langle\ \text{InstrumentId, int}\ \rangle; x_{poll1} \\
& x_{poll1} & = & \mathrm{A} \to \mathrm{I}\text{: PollMode}\ ; x_{poll2} \\
& x_{poll2} & = & x_{pollsup} + x_{pollnsup} \\
& x_{pollsup} & = & \mathrm{I} \to \mathrm{A}\text{: Supported}; x_{pollsup1} \\
& x_{pollsup1} & = & \mathrm{A} \to \mathrm{I}\text{: ConfigPoll}; x_{pollsup2} \\
& x_{pollsup2} + x_{pollsup3} & = & x_{pollsup4} \\
& x_{pollsup4} & = & \mathrm{A} \to \mathrm{I}\text{: Poll}; x_{pollsup5} \\
& x_{pollsup5} & = & \mathrm{I} \to \mathrm{A}\text{: Raw}\langle\ \text{Data}\ \rangle; x_{pollsup6} \\
& x_{pollsup6} & = & \mathrm{A} \to \mathrm{U}\text{: Formatted}\langle\ \text{Data}\ \rangle; x_{pollsup3} \\
& x_{pollnsup} & = & \mathrm{I} \to \mathrm{A}\text{: NotSupported}; x_{pollnsup1} \\
& x_{pollnsup1} & = & x_{pollnsup2} \mid x_{pollnsup3} \\
& x_{pollnsup2} & = & \mathrm{A} \to \mathrm{I}\text{: ConfigPush}; x_{pollnsup4} \\
& x_{pollnsup4} + x_{pollnsup5} & = & x_{pollnsup6} \\
& x_{pollnsup6} & = & \mathrm{I} \to \mathrm{A}\text{: Raw}\langle\ \text{Data}\ \rangle; x_{pollnsup5} \\
& x_{pollnsup3} + x_{pollnsup7} & = & x_{pollnsup8} \\
& x_{pollnsup8} & = & \mathrm{A} \to \mathrm{U}\text{: Formatted}\langle\ \text{Data}\ \rangle; x_{pollnsup7} \\
\text{in } x_0 &&&
\end{array}
$$

Figure 2: The global protocol for the OOI "Acquire Data from Instrument" use case

focus on the beginning of the protocol declaration:

$$
\begin{array}{lrcl}
\mathrm{G} = \ \mathrm{def} & x_0 & = & x_{push} + x_{poll} \\
& x_{push} & = & \mathrm{U} \to \mathrm{A}\text{: PushMode}\langle\ \text{InstrumentId}\ \rangle; x_{push1} \\
& x_{poll} & = & \mathrm{U} \to \mathrm{A}\text{: PollMode}\langle\ \text{InstrumentId, int}\ \rangle; x_{poll1}
\end{array}
$$

the mapping between global protocol and graph could be represented as in Figure 4.

Furthermore, there exists a programming language, called Scribble, that aims at providing a simple language, as its name states it, to express global protocols as well as local protocols. Besides, a complete tool chain from global protocols in Scribble to the generation of the classes for the end-point programs is being developed. It uses the projection from global to local protocols. Our goal is to add the first step of this tool chain with a graph representation. For this purpose we want to establish a clear correspondence between global protocols in Scribble and our graphical notations. As an example, we give in Figure 3 the corresponding global protocol in Scribble of our scenario. The part of the protocol we presented in more details in Figure 4 corresponds to this extract from the program in Scribble:

```
choice at U {
  PushMode(InstrumentId) from U to A;
} or {
  PollMode(InstrumentId,int) from U to A;
}
```

Code in Scribble corresponding to Figure 4

```
// U is User, A is ION Agent, I is Instrument
global protocol DataAcquisition(role U, role A, role I) {
  interruptible {  choice at U {
            PushMode(InstrumentId) from U to A;
            PushMode from A to I;
            choice at I {
              Supported from I to A;
              ConfigPush from A to I;
              rec PUSH {
                Raw(Data) from I to A;
                Formatted(Data) from A to U;
                continue PUSH;
              }
            } or {
              NotSupported from I to A;
              ConfigPoll from A to I;
              rec POLL {
                Poll from A to I;
                Raw(Data) from I to A;
                Formatted(Data) from A to U;
                continue POLL;
            } }
          } or {
            PollMode(InstrumentId,int) from U to A;
            PollMode from A to I;
            choice at I {
              Supported from I to A;
              ConfigPoll from A to I;
              rec POLL {
                Poll from A to I;
                Raw(Data) from I to A;
                Formatted(Data) from A to U;
                continue POLL;
              }
            } or {
              NotSupported from I to A;
              parallel {
                ConfigPush from A to I;
                rec PUSH {
                  Raw(Data) from I to A;
                  continue PUSH;
                }
              } and {
                rec POLL {
                  Formatted(Data) from A to U;
                  continue POLL; }
            } }
          }
  } by U with Stop
}
```

Figure 3: The Scribble global protocol for the OOI “Acquire Data from Instrument” use case

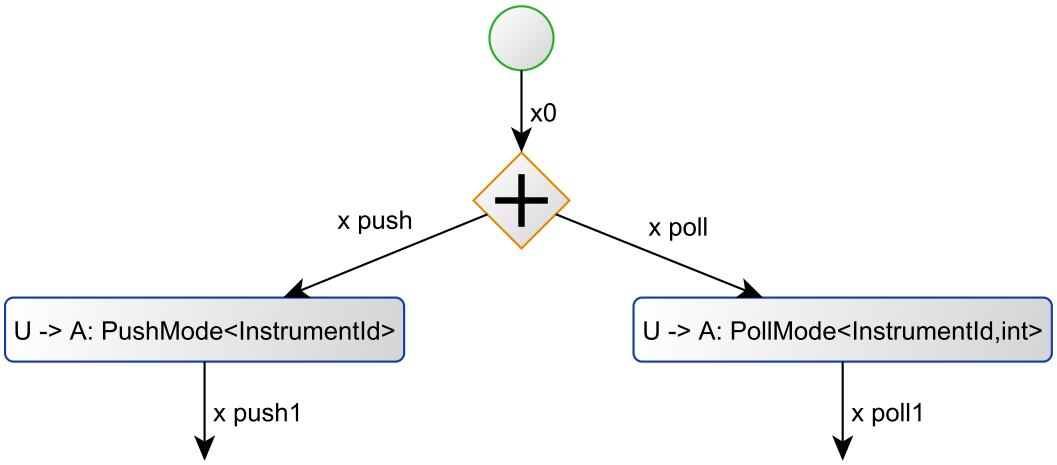


Figure 4: Mapping between global protocol and graph

Contributions

We begin with exposing in this report all the background work we have done to obtain, as well as an overview of the on-going research, a deep knowledge in session types. This was a prerequisite to become familiar with the key elements: session types, Scribble language.

The main topic is the session types. Even if some features of sessions are related to Pi-calculus, the theory behind is based on different elements. To understand step by step where generalised multiparty session types (section 1.1.3) come from, we first study binary session types (section 1.1.1) and work on simple examples. Then we turn to multiparty session types (section 1.1.2) for which the major part of understanding work has to be done. Common features to all these topics are verification issues. We will look at some specific work in section 1.1.4. The second key element of our project is the Scribble language (section 1.2.1). In order to build our graph notations in correspondence with it, we studied the syntax (section 1.2.2) and its features (section 1.2.3). To conclude the background work, we looked at work related to implementation issues based on session types (section 1.3). The implementation is indeed a great part of the work once theory has been developed.

We proceed with our work and in particular the design we have chosen for graphs (chapter 2): we start from general principles we obtained from the background work (section 2.1) to continue with a discussion about existing representations (section 2.2) before presenting our notations (section 2.3).

Once our notations have been set, the major work concerns the development (chapter 3). We expose our general architecture as an overview (section 3.1) and then get into more details to explain how from a global protocol in Scribble we obtain a graph (section 3.2) and vice versa (section 3.3).

Then we present our implementation (chapter 4): we explain first the general structure of the code (section 4.1), then the library in Python we have developed to create the needed graphical objects (section 4.2) and finally some details about both ways of the correspondence (sections 4.3 and 4.4).

To summarize our contributions concerning the graph representation:

- We establish notations for graphs that correspond to Scribble global protocol and which are supported by generalised multiparty session types.
- We provide the implementation of a software tool to perform the transformation from Scribble global protocol into graph, and vice versa. The correspondence is based on a library of graphical objects in Python.

In a second time, to go further, we extended our work to provide a notion of time to such protocols and

graphs. In chapter 5, we introduce this new feature with general principles (section 5.1), the corresponding design (section 5.2) with the extension of the Scribble language (section 5.3), as well as the changes on the implementation (section 5.4). To concretise our work, we developed timed global protocol (chapter 6). Starting from related work (section 6.1), we established our own syntax (section 6.2). We present the expressiveness of our syntax (section 6.3), before presenting some key results we obtained or we conjecture (section 6.4).

To summarize our contributions regarding time:

- We extend the syntax for Scribble protocols, as well as for generalised local and global types.
- We clearly define structures to express timeouts for message passing, delays and other time constraints.
- We define a temporal satisfiability criterion for clock conditions and conjecture progress and error-freedom properties for processes.

Finally, we have evaluated the expressiveness and usability of our graph representation (chapter 7). We define a method for the evaluation (section 7.1), and we perform the implementation of several test cases (section 7.2). As a result, we achieve to demonstrate the conformance of our graphical representation to well-formed global protocols (section 7.3). We conclude with a summary of what we have achieved, some feedbacks on the project with the improvements that can be made and possible future work.

Chapter 1

Background

1.1 Session types

As we have seen in the introduction there is a need in distributed programming for a clear structure to express conversation.

A session is a structure to encapsulate a safe communication scheme between two or more peers in a context of distributed processes. A typing system is associated to this structure in order to statically check the programs from the processes, in particular safety and consistency. A session is introduced to describe a sequence of interactions between the processes.

As the case of two processes was first studied, we begin with exposing the binary session types to get the idea of session types. Then it has been extended to more than two processes with multiparty session types in order to express more complex interactions. Therefore we will continue with a detailed presentation of multiparty session.

1.1.1 Binary session types

Binary session types contain the expected communication features, as studied in the Models of Concurrent Computation course: value sending and receiving, recursion, choice of interactions. What is new is the session initiation and session delegation.

We proceed with an example that we have established to explain in more details the syntax (Table 1.1), operational semantics and typing system involved in binary sessions. We consider the case of a client using the National Rail website to book a train ticket. The scenario is as follows:

- The client first chooses the destination.
- Then the service (National Rail) contacts the right company to delegate the deal. The client will continue the transaction, unaware that she now communicates directly with the company.
- The client and the company exchange the date, the price and the money.

Here is the syntax for the system made up of one client, *Client*, the National Rail Service, *NRS*, and one of the companies, *Company*.

```
def      Client  =  request a(k) in k ◁ bristol;k!(date);k?(price);k!(money);inact
and      NRS     =  accept a(v) in k ▷ {bristol:request c(k') in throw k'[v];NRS ||...|| newcastle: ... }
and      Company =  accept c(k') in catch k'[k] in k?(date);k!(price);k?(money);Company
in                  Client | NRS | Company
```

The three first lines introduce the recursion that is used for the process defined in the last line. The first line specifies that a client first requests a session with the national rail service (*request a(k) in*), then selects the destination labelled Bristol (*k ◁ bristol;*), and finally sends the date (*k!(date);*), receives the corresponding price (*k?(price);*) and sends the money (*k!(money);*) before halting (*inact*). The second line describes the interactions from the side of the national rail service: it accepts the request of session (*accept a(v) in k*) and then has a branching choice (*k ▷ { ... }*). We only specify the case for the label called Bristol: NRS

P	::=	request a(k) in P	Session Request
	|	accept a(k) in P	Session Acceptance
	|	$k!\langle e\rangle;P$	Data sending
	|	$k?(x);P$	Data reception
	|	throw k[k']; P	Channel Sending
	|	catch k(k') in P	Channel Reception
	|	$k\triangleleft$;P	Label Selection
	|	$k\triangleright\{l_1 : P_1 \| ... \| l_n : P_n\}$	Label Branching
	|	if e then P else Q	Conditional Branch
	|	$P \mid Q$	Parallel
	|	inact	Inaction
	|	$(\nu u)P$	Name/Channel Hiding
	|	def D in P	Recursion
	|	$X[ek]$	Process Variables
e	::=	c	Constant
	|	e+e' | e-e' | e+e' | not e ...x	Expression
D	::=	$X(x,y) = P$	Declaration

Table 1.1: Syntax for user-defined processes

requests a new session with the corresponding company (*request c(k') in*) and delegates the deal (*throw k'[v];*), before being available again for other interactions. The third line specifies that a company first accepts the request for a new session (*accept c(k') in*) then receives the delegation of the deal (*catch k'[k] in*) and finally operates the receiving of the date (*k?(date);*), the sending of the price (*k!(price);*) and the receiving of the money (*k?(money);*), before being available again.

Rather than exposing all the tables for operational semantics and typing system, that can be found in [21, 32], we apply the rules to reduce our example. Let call P the system above defined and D the declaration for recursion inside P, and apply to it the reduction rules.

P	$\rightarrow$	def D in (ν k) (k $\triangleleft$ bristol;k!(date);k?(price);k!(money);inact | k $\triangleright$ {bristol:request c(k') in throw k'[k];NRS $\|\|$...$\|\|$ newcastle: ... }) | accept c(k') in catch k'[k] in k?(date);k!(price);k?(money);Company	[Link]
	$\rightarrow$	def D in (ν k) (k!(date);k?(price);k!(money);inact | request c(k') in throw k'[k];NRS) | accept c(k') in catch k'[k] in k?(date);k!(price);k?(money);Company	[Label]
	$\rightarrow$	def D in (ν k) (k!(date);k?(price);k!(money);inact | (ν k')(throw k'[k];NRS | catch k'[k] in k?(date);k!(price);k?(money);Company))	[Link]
	$\rightarrow$	def D in (ν k) (k!(date);k?(price);k!(money);inact | (ν k')(NRS | k?(date);k!(price);k?(money);Company))	[Pass]
	$\rightarrow$	def D in (ν k) (k?(price);k!(money);inact | (ν k')(NRS | k!(price);k?(money);Company))	[Com]
	$\rightarrow$	def D in (ν k)(k!(money);inact | (ν k')(NRS | k?(money);Company))	[Com]
	$\rightarrow$	def D in (ν k) (inact | (ν k')(NRS | Company))	[Com]
	$\equiv$	def D in (NRS | Company)	

Besides the above specified reduction rules, we used [Def] at each step and [Str] at the second [Link] step in order to put the third part inside (ν k).

The process P is also typable with respect to the typing system. We give here two results of this typing:

$$\Gamma \vdash k \triangleleft bristol; k!(date); k?(price); k!(money); inact \triangleright \Delta, k : \oplus\{bristol :![nat];?[nat];![nat];end\}$$

$$\Gamma \vdash catch\ k'[k] in\ k?(date); k!(price); k?(money); inact \triangleright \Delta, k' :?[\ ![nat];?[nat];![nat];end\ ];end$$

Therefore binary session types allow expressing complex communication structure between two peers and to ensure its safety. Nevertheless, in a real world context, conversations usually involve more than two entities. To solve this issue, one can first think of modelling interactions between more than two processes

using binary session types for each pair of them in the same way as the previous example. It appears however that it removes the desired clarity of the structure. Furthermore it cannot express the situation as a whole but only separate interactions. That is why researchers introduce multiparty session types.

1.1.2 Multiparty session types

Overview

A multiparty session describes the interactions between several processes. Besides specifying the user-defined processes involved in a session, we have to define, at a higher level, a global protocol that ensures the coherence of the session. The typing system allows specifying these two levels: first there is a global type and then projections of this global type to each endpoint processes, the local types.

In multiparty session programming, a developer will have to define the global type and the user-defined processes. The key step in the methodology consists then of checking the correspondence between the projections from the global type and the type of the endpoint programs. We summarize the structure of multiparty session types in Figure 1.1.

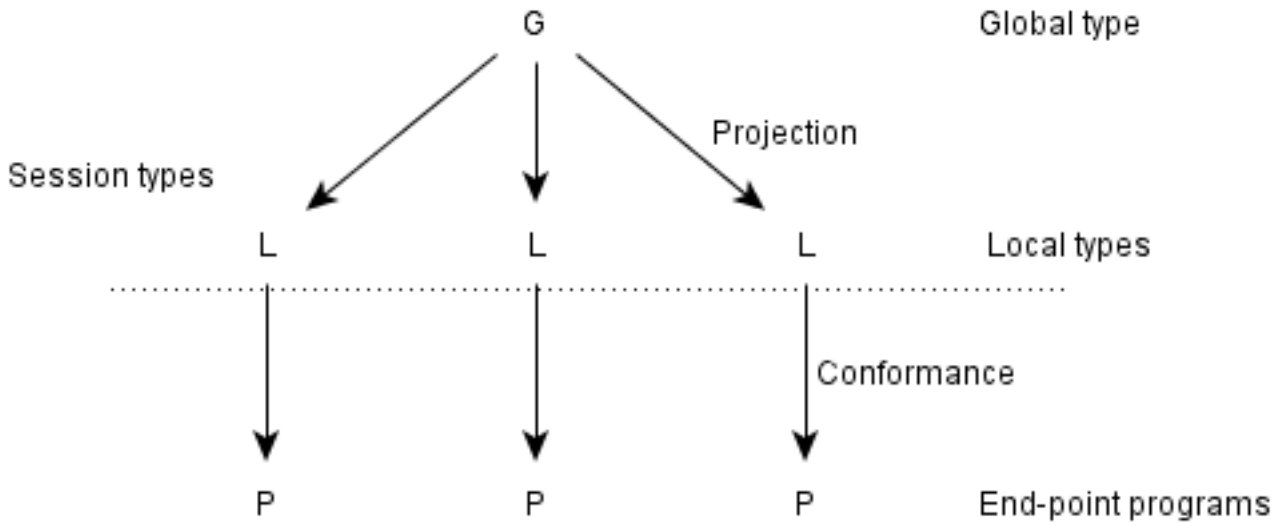


Figure 1.1: General structure for multiparty session types

With respect to the procedure, we get several properties: type safety, session fidelity, progress, linearity. Also the structure of sessions makes it possible to deal with interferences and interleaving of interactions among participants.

To describe with more details and more formally what is explained above, we look at the example of the three buyers protocol and then introduce syntax, operational semantics and the typing system, as defined in [14].

The three buyers protocol example

We first introduce the calculus through the example of the three-buyer protocol, which includes the expected communication features, as well as session-multicasting and dynamically merging two conversations. This example is extracted from [14]. "The overall scenario proceeds as follows.

1. Alice sends a book title to Seller, then Seller sends back a quote to Alice and Bob. Then Alice tells Bob how much she can contribute.

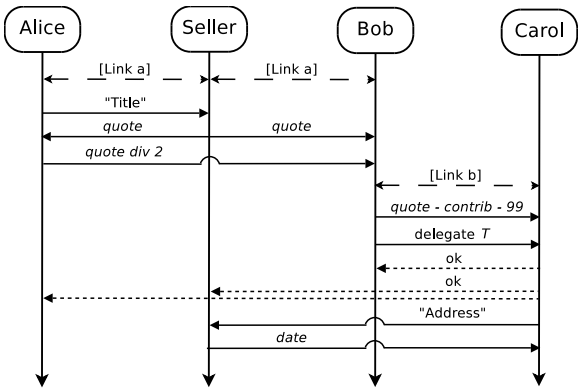


Figure 1.2: The three buyer protocol interactions

2. If the price is within Bob's budget, Bob notifies both Seller and Alice he accepts, then sends his address, and Seller sends back the delivery date.

3. If the price exceeds the budget, Bob asks Carol to collaborate together by establishing a new session. Then Bob sends how much Carol must pay, then *delegates* the remaining interactions with Alice and Seller to Carol.

4. If the rest of the price is within Carol's budget, Carol accepts the quote and notifies Alice, Bob and Seller, and continues the rest of the protocol with Seller and Alice transparently, *as if she were Bob*. Otherwise she notifies Alice, Bob and Seller to quit the protocol.

Figure 1.2 depicts an execution of the above protocol where Bob asks Carol to collaborate (by delegating the remaining interactions with Alice and Seller) and the transaction terminates successfully.

Then multiparty session programming consists of two steps: (1) specifying the intended communication protocols using global types, and (2) implementing these protocols using processes. The specifications of the three-buyer protocol are given as two separated global types: one is G_a among Alice, Bob and Seller and the other is G_b between Bob and Carol. We write principals with legible symbols though they will actually be coded by numbers: in G_a we have $\mathtt{A}=1$, $\mathtt{B}=2$, and $\mathtt{S}=3$, while in G_b we have $\mathtt{B}=2$, $\mathtt{C}=1$.

$G_a =$

1. $\mathtt{A} \longrightarrow \mathtt{S} : \langle \mathsf{string} \rangle.$
2. $\mathtt{S} \longrightarrow \{\mathtt{A},\mathtt{B}\} : \langle \mathsf{int} \rangle.$
3. $\mathtt{A} \longrightarrow \mathtt{B} : \langle \mathsf{int} \rangle.$
4. $\mathtt{B} \longrightarrow \{\mathtt{S},\mathtt{A}\} : \{\mathsf{ok} : \mathtt{B} \longrightarrow \mathtt{S} : \langle \mathsf{string} \rangle.$
5. $\qquad\qquad\qquad\qquad\qquad\qquad \mathtt{S} \longrightarrow \mathtt{B} : \langle \mathsf{date} \rangle; \mathsf{end}$
6. $\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{quit} : \mathsf{end}\}$

$G_b =$

1. $\mathtt{B} \longrightarrow \mathtt{C} : \langle \mathsf{int} \rangle.$
2. $\mathtt{B} \longrightarrow \mathtt{C} : \langle T \rangle.$
3. $\mathtt{C} \longrightarrow \mathtt{B} : \{\mathsf{ok} : \mathsf{end}, \quad \mathsf{quit} : \mathsf{end}\}.$

$T =$

$\oplus\langle \{\mathtt{S},\mathtt{A}\},$
$\{\mathsf{ok} : !\langle \mathtt{S}, \mathsf{string} \rangle; ?\langle \mathtt{S}, \mathsf{date} \rangle; \mathsf{end},$
$\mathsf{quit} : \mathsf{end}\}\rangle$

The types give a global view of the two conversations, directly abstracting the scenario given by the diagram. In G_a, line 1 denotes A sends a string value to S. Line 2 says S multicasts the same integer value to A and B and line 3 says that A sends an integer to B. In lines 4-6 B sends either ok or quit to S and A. In the first case B sends a string to S and receives a date from S, in the second case there are no further communications.

Line 2 in G_b represents the delegation of the capability specified by the session type T of channels (formally defined later) from B to C (note that S and A in T concern the session on a).

Figure 1.3 gives the code, associated to G_a and G_b, for S, A, B and C in a "user" syntax formally defined later[1]: Session name a establishes the session corresponding to G_a. S initiates a session involving three bodies as third participant by $\overline{a}[3](y_3)$: A and B participate as first and second participants by $a[1](y_1)$ and $a[2](y_2)$, respectively. Then S, A and B communicate using the channels y_3, y_1 and y_2, respectively. Each

[1] In the examples we will use the following font conventions: variables (bound by an input action) are in *italics* and constants are in sans serif; string literals are in monospace font and double quoted.

$$
\begin{array}{lcl}
\mathtt{S} & = & \overline{a}[3](y_3).y_3?(\mathit{title});y_3!\langle \mathsf{quote}\rangle;y_3\&\{\mathsf{ok} : y_3?(\mathit{address});y_3!\langle \mathsf{date}\rangle;\mathbf{0},\ \mathsf{quit} : \mathbf{0}\} \\
\mathtt{A} & = & a[1](y_1).y_1!\langle \texttt{"Title"}\rangle;y_1?(\mathit{quote});y_1!\langle \mathit{quote}\ \mathsf{div}\ 2\rangle;y_1\&\{\mathsf{ok} : \mathbf{0},\ \mathsf{quit} : \mathbf{0}\} \\
\mathtt{B} & = & a[2](y_2).y_2?(\mathit{quote});y_2?(\mathit{contrib}); \\
 & & \mathsf{if}\ (\mathit{quote}\text{ - }\mathit{contrib} < 100)\ \mathsf{then}\ \ y_2 \oplus \mathsf{ok};y_2!\langle \texttt{"Address"}\rangle;y_2?(\mathit{date});\mathbf{0} \\
 & & \mathsf{else}\ \overline{b}[2](z_2).z_2!\langle \mathit{quote}\text{ - }\mathit{contrib}\text{ - }99\rangle;z_2!\langle\langle y_2\rangle\rangle;z_2\&\{\mathsf{ok} : \mathbf{0},\ \mathsf{quit} : \mathbf{0}\} \\
\mathtt{C} & = & b[1](z_1).z_1?(x);z_1?((t)); \\
 & & \mathsf{if}\ (x < 100)\ \mathsf{then}\ z_1 \oplus \mathsf{ok};t \oplus \mathsf{ok};t!\langle \texttt{"Address"}\rangle;t?(\mathit{date});\mathbf{0} \\
 & & \mathsf{else}\ z_1 \oplus \mathsf{quit};t \oplus \mathsf{quit};\mathbf{0}
\end{array}
$$

Figure 1.3: The three buyer example

P	::=	$\overline{u}[\mathtt{p}](y).P$	Multicast Request		$\mid$	$\mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ Q$	Conditional
	$\mid$	$u[\mathtt{p}](y).P$	Accept		$\mid$	$P \mid Q$	Parallel
	$\mid$	$y!\langle e\rangle;P$	Value sending		$\mid$	$\mathbf{0}$	Inaction
	$\mid$	$y?(x);P$	Value reception		$\mid$	$(\nu a)P$	Hiding
	$\mid$	$y!\langle\langle z\rangle\rangle;P$	Session delegation		$\mid$	$\mathsf{def}\ D\ \mathsf{in}\ P$	Recursion
	$\mid$	$y?((z));P$	Session reception		$\mid$	$X\langle e,y\rangle$	Process call
	$\mid$	$y \oplus l;P$	Selection				
	$\mid$	$y\&\{l_i : P_i\}_{i\in I}$	Branching	e	::=	$v \mid x$	
u	::=	$x \mid a$	Identifier		$\mid$	$e\ \mathsf{and}\ e' \mid \mathsf{not}\ e \ldots$	Expression
v	::=	$a \mid \mathsf{true} \mid \mathsf{false}$	Value	D	::=	$X(x,y) = P$	Declaration

Table 1.2: Syntax for user-defined processes

channel $y_\mathtt{p}$ can be seen as a port connecting participant p with all other ones; the receivers of the data sent on $y_\mathtt{p}$ are specified by the global type (this information will be included in the runtime code). The first line of G_a is implemented by the input and output actions $y_3?(\mathit{title})$ and $y_1!\langle\texttt{"Title"}\rangle$. The last line of G_b is implemented by the branching and selection actions $z_2\&\{\mathsf{ok} : \mathbf{0},\ \mathsf{quit} : \mathbf{0}\}$ and $z_1 \oplus \mathsf{ok}$, $z_1 \oplus \mathsf{quit}$.

In B, if the quote minus A's contribution exceeds 100€ (i.e., $\mathit{quote}\text{ - }\mathit{contrib} \geq 100$), another session between B and C is established dynamically through shared name b. The delegation is performed by passing the channel y_2 from B to C (actions $z_2!\langle\langle y_2\rangle\rangle$ and $z_1?((t))$), and so the rest of the session is carried out by C with S and A."

We chose to study this example as it is widely-used in the literature. This example illustrates indeed how multiparty session types are used for such a protocol and guarantees safe communication. To become more familiar with multiparty session types, we worked on this example and applied to it the semantics rules, as we did for binary session types.

Syntax and semantics

The real improvement in multiparty session types, compared to binary session types, is the notion of global protocol, which will be used as a choreography for the whole system. The syntax for global protocols can be found in [14]. The grammar for processes, ranged over by $P, Q \ldots$, and for expressions, ranged over by $e, e', \ldots$, similar to the binary session case, is given by the Table 1.2.

The rules for operational semantics are also from [14] and given in Tables 1.3 (r ranges over a, s and z ranges over v, $s[\mathtt{p}]$ and l.), 1.4.

Types and projection

As we have introduced the notion of global protocol, we now define global types. As explained in the three buyers protocol, from the global type, one can deduce the local types thank to projections.

A *global type*, ranged over by $G, G', ..$ describes the whole conversation scenario of a multiparty session as a type signature. Its grammar is given below:

$$P \mid \mathbf{0} \equiv P \quad P \mid Q \equiv Q \mid P \quad (P \mid Q) \mid R \equiv P \mid (Q \mid R)$$

$$(\nu r)P \mid Q \equiv (\nu r)(P \mid Q) \quad \text{if } r \notin \mathrm{fn}(Q)$$

$$(\nu rr')P \equiv (\nu r'r)P \quad (\nu r)\mathbf{0} \equiv \mathbf{0} \quad \mathsf{def}\ D\ \mathsf{in}\ \mathbf{0} \equiv \mathbf{0}$$

$$\mathsf{def}\ D\ \mathsf{in}\ (\nu r)P \equiv (\nu r)\mathsf{def}\ D\ \mathsf{in}\ P \quad \text{if } r \notin \mathrm{fn}(D)$$

$$(\mathsf{def}\ D\ \mathsf{in}\ P) \mid Q \equiv \mathsf{def}\ D\ \mathsf{in}\ (P \mid Q) \quad \text{if } \mathrm{dpv}(D) \cap \mathrm{fpv}(Q) = \emptyset$$

$$\mathsf{def}\ D\ \mathsf{in}\ (\mathsf{def}\ D'\ \mathsf{in}\ P) \equiv \mathsf{def}\ D\ \mathsf{and}\ D'\ \mathsf{in}\ P \quad \text{if } \mathrm{dpv}(D) \cap \mathrm{dpv}(D') = \emptyset$$

$$s : (\mathtt{q}, \Pi, z) \cdot (\mathtt{q}', \Pi', z') \cdot h \equiv s : (\mathtt{q}', \Pi', z') \cdot (\mathtt{q}, \Pi, z) \cdot h \quad \text{if } \Pi \cap \Pi' = \emptyset \text{ or } \mathtt{q} \neq \mathtt{q}'$$

$$s : (\mathtt{q}, \Pi, z) \cdot h \equiv s : (\mathtt{q}, \Pi', z) \cdot (\mathtt{q}, \Pi'', z) \cdot h \quad \text{where } \Pi = \Pi' \cup \Pi'' \text{ and } \Pi' \cap \Pi'' = \emptyset$$

Table 1.3: Structural equivalence

$$a[1](y_1).P_1 \mid \ldots \mid \overline{a}[n](y_n).P_n \longrightarrow (\nu s)(P_1\{s[1]/y_1\} \mid \ldots \mid P_n\{s[n]/y_n\} \mid s : \varnothing) \quad \text{[Link]}$$

$$s[\mathtt{p}]!\langle \Pi, e\rangle; P \mid s : h \longrightarrow P \mid s : h \cdot (\mathtt{p}, \Pi, v) \quad (e \downarrow v) \quad \text{[Send]}$$

$$s[\mathtt{p}]!\langle\langle \mathtt{q}, s'[\mathtt{p}']\rangle\rangle; P \mid s : h \longrightarrow P \mid s : h \cdot (\mathtt{p}, \mathtt{q}, s'[\mathtt{p}']) \quad \text{[Deleg]}$$

$$s[\mathtt{p}] \oplus \langle \Pi, l\rangle; P \mid s : h \longrightarrow P \mid s : h \cdot (\mathtt{p}, \Pi, l) \quad \text{[Label]}$$

$$s[\mathtt{p}]?(\mathtt{q}, x); P \mid s : (\mathtt{q}, \{\mathtt{p}\}, v) \cdot h \longrightarrow P\{v/x\} \mid s : h \quad \text{[Recv]}$$

$$s[\mathtt{p}]?((\mathtt{q}, y)); P \mid s : (\mathtt{q}, \mathtt{p}, s'[\mathtt{p}']) \cdot h \longrightarrow P\{s'[\mathtt{p}']/y\} \mid s : h \quad \text{[Srec]}$$

$$s[\mathtt{p}]\&(\mathtt{q}, \{l_i : P_i\}_{i \in I}) \mid s : (\mathtt{q}, \{\mathtt{p}\}, l_{i_0}) \cdot h \longrightarrow P_{i_0} \mid s : h \quad (i_0 \in I) \quad \text{[Branch]}$$

$$\mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ Q \longrightarrow P \quad (e \downarrow \mathsf{true}) \qquad \mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ Q \longrightarrow Q \quad (e \downarrow \mathsf{false}) \quad \text{[If-T, If-F]}$$

$$\mathsf{def}\ X(x,y) = P\ \mathsf{in}\ (X\langle e, s[\mathtt{p}]\rangle \mid Q) \longrightarrow \mathsf{def}\ X(x,y) = P\ \mathsf{in}\ (P\{v/x\}\{s[\mathtt{p}]/y\} \mid Q) \quad (e \downarrow v) \quad \text{[ProcCall]}$$

$$P \longrightarrow P' \quad \Rightarrow \quad (\nu r)P \longrightarrow (\nu r)P' \qquad P \longrightarrow P' \quad \Rightarrow \quad P \mid Q \longrightarrow P' \mid Q \quad \text{[Scop,Par]}$$

$$P \longrightarrow P' \quad \Rightarrow \quad \mathsf{def}\ D\ \mathsf{in}\ P \longrightarrow \mathsf{def}\ D\ \mathsf{in}\ P' \quad \text{[Defin]}$$

$$P \equiv P' \text{ and } P' \longrightarrow Q' \text{ and } Q \equiv Q' \quad \Rightarrow \quad P \longrightarrow Q \quad \text{[Str]}$$

Table 1.4: Reduction rules

$$\begin{array}{llll} \text{Global} & G & ::= & \mathtt{p} \to \Pi : \langle U \rangle.G' \\ & & \mid & \mathtt{p} \to \Pi : \{l_i : G_i\}_{i \in I} \\ & & \mid & \mu \mathbf{t}.G \mid \mathbf{t} \mid \mathsf{end} \end{array} \qquad \begin{array}{llll} \text{Exchange} & U & ::= & S \mid T \\ \text{Sorts} & S & ::= & \mathsf{bool} \mid \ldots \mid G \end{array}$$

We now define local types of pure processes, called *session types*. While global types represent the whole protocol, local types correspond to the communication actions, representing sessions from the viewpoints of single participants.

$$\begin{array}{llllll} \text{Action} & T & ::= & !\langle \Pi, U\rangle; T & \textit{send} & \mid \ \mu \mathbf{t}.T \quad \textit{recursive} \\ & & \mid & ?(\mathtt{p}, U); T & \textit{receive} & \mid \ \mathbf{t} \quad \textit{variable} \\ & & \mid & \oplus\langle \Pi, \{l_i : T_i\}_{i \in I}\rangle & \textit{selection} & \mid \ \mathsf{end} \quad \textit{end} \\ & & \mid & \&(\mathtt{p}, \{l_i : T_i\}_{i \in I}) & \textit{branching} & \end{array}$$

The relationship between sessions and global types is formalised by the notion of projection. The *projection of G onto* $\mathtt{q}$ ($G \upharpoonright \mathtt{q}$) is defined by induction on G:

$$(\mathtt{p} \to \Pi : \langle U \rangle.G') \upharpoonright \mathtt{q} = \begin{cases} !\langle \Pi, U\rangle; (G' \upharpoonright \mathtt{q}) & \text{if } \mathtt{q} = \mathtt{p}, \\ ?(\mathtt{p}, U); (G' \upharpoonright \mathtt{q}) & \text{if } \mathtt{q} \in \Pi, \\ G' \upharpoonright \mathtt{q} & \text{otherwise.} \end{cases}$$

$$\Gamma, u : S \vdash u : S \ \lfloor \textsc{Name} \rfloor \quad \Gamma \vdash \mathsf{true}, \mathsf{false} : \mathsf{bool} \quad \frac{\Gamma \vdash e_i : \mathsf{bool}}{\Gamma \vdash e_1 \ \mathsf{and} \ e_2 : \mathsf{bool}} \ \lfloor \textsc{Bool} \rfloor, \lfloor \textsc{And} \rfloor$$

$$\frac{\Gamma \vdash u : \langle G \rangle \quad \Gamma \vdash P \triangleright \Delta, y : G \upharpoonright \mathtt{p} \quad \mathrm{pn}(G) \leq \mathtt{p}}{\Gamma \vdash \overline{u}[\mathtt{p}](y).P \triangleright \Delta} \ \lfloor \textsc{MCast} \rfloor \quad \frac{\Gamma \vdash u : \langle G \rangle \quad \Gamma \vdash P \triangleright \Delta, y : G \upharpoonright \mathtt{p}}{\Gamma \vdash u[\mathtt{p}](y).P \triangleright \Delta} \ \lfloor \textsc{MAcc} \rfloor$$

$$\frac{\Gamma \vdash e : S \quad \Gamma \vdash P \triangleright \Delta, c : T}{\Gamma \vdash c!\langle \Pi, e \rangle; P \triangleright \Delta, c : !\langle \Pi, S \rangle; T} \ \lfloor \textsc{Send} \rfloor \quad \frac{\Gamma, x : S \vdash P \triangleright \Delta, c : T}{\Gamma \vdash c?(\mathtt{q}, x); P \triangleright \Delta, c : ?(\mathtt{q}, S); T} \ \lfloor \textsc{Rcv} \rfloor$$

$$\frac{\Gamma \vdash P \triangleright \Delta, c : T}{\Gamma \vdash c!\langle\langle \mathtt{p}, c' \rangle\rangle; P \triangleright \Delta, c : !\langle \{\mathtt{p}\}, T' \rangle; T, c' : T'} \ \lfloor \textsc{Deleg} \rfloor \quad \frac{\Gamma \vdash P \triangleright \Delta, c : T, y : T'}{\Gamma \vdash c?((\mathtt{q}, y)); P \triangleright \Delta, c : ?(\mathtt{q}, T'); T} \ \lfloor \textsc{Srec} \rfloor$$

$$\frac{\Gamma \vdash P \triangleright \Delta, c : T_j \quad j \in I}{\Gamma \vdash c \oplus \langle \Pi, l_j \rangle; P \triangleright \Delta, c : \oplus\langle \Pi, \{l_i : T_i\}_{i \in I} \rangle} \ \lfloor \textsc{Sel} \rfloor \quad \frac{\Gamma \vdash P_i \triangleright \Delta, c : T_i \quad \forall i \in I}{\Gamma \vdash c \& (\mathtt{p}, \{l_i : P_i\}_{i \in I}) \triangleright \Delta, c : \& (\mathtt{p}, \{l_i : T_i\}_{i \in I})} \ \lfloor \textsc{Branch} \rfloor$$

$$\frac{\Gamma \vdash P \triangleright \Delta \quad \Gamma \vdash Q \triangleright \Delta'}{\Gamma \vdash P \mid Q \triangleright \Delta, \Delta'} \ \lfloor \textsc{Conc} \rfloor$$

$$\frac{\Gamma \vdash e : \mathsf{bool} \quad \Gamma \vdash P \triangleright \Delta \quad \Gamma \vdash Q \triangleright \Delta}{\Gamma \vdash \mathsf{if} \ e \ \mathsf{then} \ P \ \mathsf{else} \ Q \triangleright \Delta} \ \lfloor \textsc{If} \rfloor \quad \frac{\Delta \ \mathsf{end \ only}}{\Gamma \vdash \mathbf{0} \triangleright \Delta} \ \lfloor \textsc{Inact} \rfloor \quad \frac{\Gamma, a : \langle G \rangle \vdash P \triangleright \Delta}{\Gamma \vdash (\nu a) P \triangleright \Delta} \ \lfloor \textsc{NRes} \rfloor$$

$$\frac{\Gamma \vdash e : S \quad \Delta \ \mathsf{end \ only}}{\Gamma, X : S \, T, x : S \vdash X\langle e, c \rangle \triangleright \Delta, c : T} \ \lfloor \textsc{Var} \rfloor \quad \frac{\Gamma, X : S \, T, x : S \vdash P \triangleright y : T \qquad \Gamma, X : S \, T \vdash Q \triangleright \Delta}{\Gamma \vdash \mathsf{def} \ X(x, y) = P \ \mathsf{in} \ Q \triangleright \Delta} \ \lfloor \textsc{Def} \rfloor$$

Table 1.5: Typing rules for pure processes

$$(\mathtt{p} \to \Pi : \{l_i : G_i\}_{i \in I}) \upharpoonright \mathtt{q} = \begin{cases} \oplus(\Pi, \{l_i : G_i \upharpoonright \mathtt{q}\}_{i \in I}) & \text{if } \mathtt{q} = \mathtt{p} \\ \&(\mathtt{p}, \{l_i : G_i \upharpoonright \mathtt{q}\}_{i \in I}) & \text{if } \mathtt{q} \in \Pi \\ G_1 \upharpoonright \mathtt{q} & \text{if } \mathtt{q} \neq \mathtt{p}, \mathtt{q} \notin \Pi \text{ and} \\ & G_i \upharpoonright \mathtt{q} = G_j \upharpoonright \mathtt{q} \text{ for all } i, j \in I. \end{cases}$$

$$(\mu \mathbf{t}.G) \upharpoonright \mathtt{q} = \mu \mathbf{t}.(G \upharpoonright \mathtt{q}) \quad \mathbf{t} \upharpoonright \mathtt{q} = \mathbf{t} \quad \mathsf{end} \upharpoonright \mathtt{q} = \mathsf{end}.$$

As an example, we list two of the projections of the global types G_a and G_b of the first three-buyer protocol, that we have established:

$$\begin{array}{lcl} G_a \upharpoonright 1 & = & !\langle \{3\}, \mathsf{string} \rangle; ?(3, \mathsf{int}); !\langle \{2\}, \mathsf{int} \rangle; \&(2, \{\mathsf{ok} : \mathsf{end}, \mathsf{quit} : \mathsf{end}\}) \\ G_b \upharpoonright 2 & = & !\langle \{1\}, \mathsf{int} \rangle; !\langle \{1\}, T \rangle; \&(1, \{\mathsf{ok} : \mathsf{end}, \mathsf{quit} : \mathsf{end}\}) \end{array}$$

where $T = \oplus\langle \{1,3\}, \{\mathsf{ok} : !\langle \{3\}, \mathsf{string} \rangle; ?\langle 3, \mathsf{date} \rangle; \mathsf{end}, \ \mathsf{quit} : \mathsf{end}\} \rangle$.

Type checking and properties

The typing judgements for expressions and pure processes are of the shape:

$$\Gamma \vdash e : S \quad \text{and} \quad \Gamma \vdash P \triangleright \Delta$$

where Γ is the *standard environment* which associates variables to sort types, service names to global types and process variables to pairs of sort types and session types; Δ is the *session environment* which associates channels to session types. Formally we define:

$$\Gamma ::= \emptyset \mid \Gamma, u : S \mid \Gamma, X : S \, T \quad \text{and} \quad \Delta ::= \emptyset \mid \Delta, c : T$$

assuming that we can write $\Gamma, u : S$ only if u does not occur in Γ, briefly $u \notin dom(\Gamma)$ ($dom(\Gamma)$ denotes the domain of Γ, i.e., the set of identifiers which occur in Γ). We use the same convention for $X : S\ T$ and Δ (thus we can write Δ, Δ' only if $dom(\Delta) \cap dom(\Delta') = \emptyset$).
Table 1.5 presents the typing rules for pure processes. They allow proving type safety.

1.1.3 Generalised Multiparty session types

In recent work ([17]), new global type syntax has been introduced, called generalised multiparty session types. It does indeed extend multiparty session types, as it allows expressing more complex scenarii.

Overview

The new syntax is based on the declaration of state variables. They represent the successive distributed states of the interaction. This extension introduces also join and merge constructs, which therefore explicitly distinguish the branching points from the forking ones. Besides this new definition of global protocol, they present a new syntax for local types, processes and networks that corresponds to the notation of global types. Then, the goal of developing this new syntax is to build a subclass of safe Communicating Finite State Machines (CFSM), called multiparty session automata, which implement the local types, issued from the projection of the global type. The framework can therefore be summarised in two main steps: (1) projection of the generalised global type into local types, the latter being equivalent to a multiparty session automata, (2) type checking from the multiparty session automata to get the generalised multiparty processes.

Syntax, semantics and projection

What is interesting for our purpose is that this global type syntax was chosen to support general graphs. Then it is easier to match this generalised graph syntax to a representation. Therefore we will focus on this syntax for the project. We have already discussed an example at the beginning of this report: the scenario of Figure 2, which combines for instance recursion, fork and choice.

We start with the grammar for global protocols, which is given below:

G	::=	def G in x	Global type
G	::=	x = p $\rightarrow q : l\langle U\rangle; x'$	Labelled messages
	|	x = x' | x"	Fork
	|	x = x' + x"	Choice
	|	x | x' = x"	Join
	|	x + x' = x"	Merge
	|	x = end	End
U	::=	$\langle G\rangle$ | bool | nat | ...	Sorts

A global type G ::= def G in x_0 describes an interaction between a fixed number of participants. x_0 is the initial states, from where interactions start, then the state variables in G specify the other interactions that can be done within this global type. It is worth noting that recursion is not explicitly defined in this syntax: it uses the merge construct as in the above mentioned example.

A global type is well-formed if it satisfies the sanity, local choice and linearity conditions as defined in [17]. These conditions are related to similar CFSM properties.

We now proceed with local types. The grammar is very similar to the one for global types:

T	::=	def T in x	Local type
T	::=	x = $!\langle p, l\langle U\rangle\rangle.x'$	Send
	|	x = $?\langle p, l\langle U\rangle\rangle.x'$	Receive
	|	x = x'	Indirection
	|	x = x' | x"	Fork
	|	x = x' $\oplus$ x"	Internal choice
	|	x = x' & x"	External choice
	|	x | x' = x"	Join
	|	x + x' = x"	Merge
	|	x = end	End
U	::=	$\langle G\rangle$ | bool | nat | ...	Sorts

Local types represent the actions of session end-points that each process must follow within its implementation.

The projection of a well-formed global protocol to local types is straightforward: labelled messages are an output from p's point of view and an input from q's point of view; otherwise it creates an indirection link from x to x'. Internal choice represents selection and external choice corresponds to branching.

Finally, they develop syntax for processes and networks to match local types. We will not give here all the details of the syntax and the semantics rules; we only state that the syntax abstracts the control flow of any standard programming language equipped with fork and join constructs.

Multiparty Session Automata

Multiparty session automata are defined in [17] as a new communicating automata subclass that automatically satisfy strong distributed safety properties. They introduce communicating finite state machines to represent the local behaviour: (Q, C, q_0, A, δ), where Q is a finite set of states, C a finite set of channels, q_0 an initial state, A is a finite alphabet of messages and δ a finite set of transitions. There is a transition for each interaction, i.e. a message receiving or a message sending. δ is included in Q x (C x {!,?} x A) x Q. Further details about notations and definitions can be found in [17].

They also introduce communicating systems S as tuples of CFSMs. Finally for each local type there is a corresponding automaton. Therefore they call "multiparty session automata" communicating systems S such that the CFSMs are the automata corresponding to the projections of a well-formed global type.

Properties and type checking

Several properties are proven on multiparty session automata: local choice, deadlock-freedom, strong boundedness, progress and liveness. Interpreting some well-formedness conditions and standard properties is one of the interesting features of communicating finite state machines.

They also develop a new type system to verify the processes they introduced. With the correspondence they established with multiparty session automata, typed processes enjoy the same properties.

1.1.4 Verification issues

In the three preceding approaches, the typing system statically ensures safety, but it is also possible to check safety properties locally with the introduction of monitors ([5]), which is an important feature in distributed networks. Indeed, while static verification has many advantages, it is usually not suitable for real-world large-scale distributed systems: we need dynamic verification.

In [5], they introduce a light weight runtime layer, underlying the dynamic verification framework: it is called conversation runtime. It is based on the attribution of an unambiguous conversation identifier and a message label to message flows of a given conversation. As well as the static verification technique, the dynamic one is decentralised. Each principal in a network is wrapped by a dedicated monitor ensuring that the on-going communication conform a given specification. Then it is possible to mix the two techniques and ensure deadlock-freedom and protocol conformance for networks formed of verified unmonitored principals and monitored ones.

1.2 Scribble

With the on-going endeavour to build a core descriptive framework and the associated development environment for large-scale distributed systems based on session types, there is a real need of a simple language for defining session types protocols. Indeed a protocol offers an agreement on the ways interactions proceed among two or more participants. Scribble was introduced to describe these interactions ([20]). We have seen an example of a program in Scribble in Figure 3. We will again refer to it in this section.

1.2.1 Overview

Scribble is a protocol description language. It is designed to enable accurate, clear and concise specification of application-level message-passing communication protocols.

The main elements of a Scribble protocol are: the conversation (here DataAcquisition) and the principals (here U, A and I). A conversation, or a session, is an instantiation of a protocol that follows the protocol's rules of engagement. Principals represent entities, such as corporations and individuals, who are responsible for performing communication actions in distributed applications. When a principal participates in a conversation (i.e. becoming its participant), it does so by taking up specific role(s) stipulated in the underlying protocol. There are two main constructs: the preamble and the protocol definition with the role declarations and interactions description.

Scribble ensures for transport asynchrony, message order preservation and reliability. The safety assurance is given by session types.

1.2.2 Syntax

Scribble syntax is specified using a BNF-like notation. Terminal symbols are in purple with typewriter font. Non-terminal symbols are in italic. Each grammar rule starts with a non-terminal (the name defined by the rule) and ::=. Parentheses ( ) are used for grouping grammar elements, square brackets [ ] for an optional element, vertical bar | for separating alternatives, and star * for zero or more repetitions of the preceding element.

A Scribble protocol comprises four parts: a package declaration, an optional list of declarations for including external packages, an optional list of declarations for importing message types, and a list of global protocol declarations. A scribble-protocol should have an extension spr.

We present here only the main rules of Scribble syntax; the complete syntax can be find in [7].

scribble-protocol ::= (*include-decl*)* (*type-decl*)* (*global-protocol-decl*)+

global-protocol-decl ::= `global protocol` *global-protocol-signature*

global-protocol-signature ::= *identifier* [< *parameter-list* >] ([*role-list*]) *global-protocol-body*

| *identifier* ([*role-list*]) `instantiates` *identifier* [< *argument-list* >] ;

global-protocol-body ::= *global-interaction-block*

global-interaction-block ::= { *global-interaction-sequence* }

global-interaction-sequence ::= (*global-interaction*)*

global-interaction ::= *message*

| *choice*

| *parallel*

| *recursion*

| *continue*

| *interruptible*

| *do*

| *spawn*

message ::= [*message-signature* | *identifier*] `from` *role-name* `to` *role-name* ;

choice ::= `choice at` *role-name global-interaction-block* (`or` *global-interaction-block*)+

parallel ::= `par` *global-interaction-block* (`and` *global-interaction-block*)*

recursion ::= `rec` *identifier global-interaction-block*

continue ::= `continue` *identifier* ;

In the following, we will define several global protocols in Scribble. They will each begin with the key word `global protocol`, followed by its name and the involved roles. Then the protocol body declare the interactions actions. For instance, the program in Figure 3, from the motivation example, follows these rules.

To explain how to apply these rules, we study in details the choice example we had selected earlier:

```
choice at U {
  PushMode(InstrumentId) from U to A;
} or {
  PollMode(InstrumentId,int) from U to A;
}
```

The choice rule is as follows:
choice ::= `choice at` *role-name global-interaction-block* (`or` *global-interaction-block*)+
It begins with the key word `choice` as in the code. Then we have to specify which role is responsible for the choice that is formalized by `at` *role-name*. In our example it is "at U". Then we have a *global-interaction-block*, which is `PushMode(InstrumentId) from U to A;`. Finally, we define the alternative blocks with the key word `or` as follows: (`or` *global-interaction-block*)+. The + symbol means that there is at least two choice blocks.

1.2.3 Communicating features and associated conditions

We state here the main features we are interested in for this report, and their associated conditions. We extract the information from the Scribble tutorial.

Interaction

Message passing in Scribble protocols has the following characteristics: (1) Message transfers are reliable, (2) Message transfers are ordered between each pair of roles. The second one means that, for any pair of roles A and B in a given protocol, messages sent by A to B will be received by B in the same order that they were originally dispatched by A.

Causality in Scribble protocols is determined solely by the explicitly specified message transfers, and in particular, by the configuration of roles performing the send and receive actions in the message transfer sequence.

Choice

A choice must contain at least two choice blocks.

The result of the choice decision by the deciding role should be observable by all other roles in the protocol via explicit and unambiguous message transfers. This refers to the local choice condition in generalised multiparty session types.

Recursion

A recursion label is a non-empty sequence of ASCII characters and digits that does not start with a digit. Nested rec constructs must declare unique labels: this is really important in order not to have mismatch between nested recursion constructs.

The recursion label of a continue must be declared by an outer rec within which the continue occurs.

Parallel

When using the parallel construct, the following well-formedness conditions should be obeyed: (1) Two parallel blocks should not have confusing message signature – i.e. if the sender and the receiver are common, then the message signature should be distinct, (2) We distinguish two message signatures with the same operator but different argument numbers. The first condition refers to the linearity condition in generalised multiparty session types.

Interruption

To avoid semantic complexity, we assume interruptible to be used only as a top-level interaction — there should be no preceding or subsequent interactions, nor can an interruptible be inside another block.

Multiple interrupts can be thrown in one conversation: after an interrupt, it will be broadcasted to all participants, who will go out of the block, i.e. finishes its part in the conversation. The specified message signatures for interrupt should be distinct from those for ordinary messages used inside the interruptible block. When programming with interrupts, a role can issue an interrupt if and only if the role is specified and the interrupt message has the same signature as specified in the protocol.

1.3 Implementation issues

We now concentrate on the implementation side. For these new theoretic concepts to be useful for developers, it must indeed be easily implementable. We present some developments that can be found in the literature.

1.3.1 Previous work

Extension of Java

We first expose briefly an extension of Java, called SJ, shorthand for session-based extension of Java, from [22]. It offers a full compilation-runtime framework that supports session abstraction over concrete transports, for instance TCP.

Session programming consists of two steps:

- specifying the intended interaction protocols using session types,
- implementing these protocols using session operations.

In SJ, session types are called protocols. At the application-level, the operators handle, transport-independently, with all the expected features of a language for session-based distributed programming, ie session sockets, session server socket, session server-address, session communication (send and receive, iteration, branching), session failure and session delegation.

From this basis, the compilation-runtime framework of SJ works across three layers:

- Layer 1: SJ source code,
- Layer 2: Java translation and session runtime APIs,
- Layer 3: Runtime execution: JVM and SJ libraries.

It is the role of the SJ compiler to map layer 1 to layer 2. Besides translating the session operations into the communication primitives of the session APIs, it type-checks the source according to the constraints of both standard Java typing and session typing.

Session C

We now introduce another extension: Multiparty Session C. It defines a session runtime library and a session type checker to support a full guarantee of deadlock-freedom, type-safety, communication safety and global progress for any well-typed programs. As the framework is based on theory of multiparty session types, a session C program, i.e. a C program that calls the session runtime library, is developed in a top-down approach: from the global protocol, using Scribble, to the individual program. The session type-checker validates the source code against its endpoint protocol. Details of the code can be found in [28].

1.3.2 A tool chain in Python

So far we have presented some work based on session types: theoretical work as well as implementation work. There is now a need to link these different parts altogether. What we want to achieve is to provide to developers a complete tool chain: from the global scenario to the endpoint programs. Our goal is to develop software able to generate, from the global protocol, the local ones by projection, and then generate the code for the classes to obtain the end-point programs. In this way, the Scribble language, as defined earlier, is supported by the Scribble toolchain ([6]).

In many use cases, it will indeed be global protocols that designers will specify which are automatically projected to local protocols. Local protocols are used to guide the implementation of individual system components, and may be used to statically verify, e.g. by type-checking programs at compile time, or dynamically verify, by monitoring runtime messages, that each component conforms to the original global protocol.

SJ and Session C concern implementation issues for the end of the tool chain. In a similar way, our project contributes to the development of this tool chain. We build indeed the first step of the tool chain, which is the graph representation of global protocol.

Chapter 2

The graph design

As we have seen in the previous chapter, some work can be done on a correspondence between a graph representation of a choreography and the corresponding global protocol. Therefore we will now focus on this aspect. The goal is: starting from the existing theory of multiparty session types and the language Scribble, we want to develop a visual representation that is more easily usable for design than the programming language.

We first explain the design of the graph representation we have adopted. In order to demonstrate how we arrive to this notation, we begin with the expecting features of the representation from the studied theory, and then we present the existing tools to finally expose our choice.

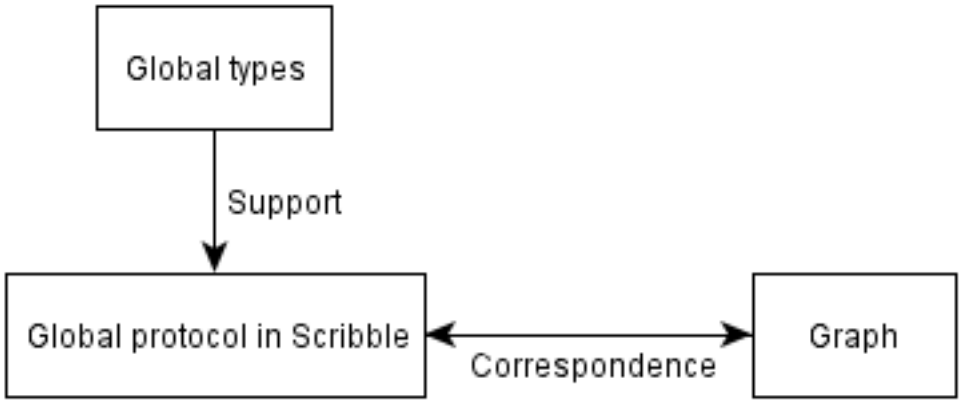


Figure 2.1: Architecture of the design

2.1 Choices for the syntax

2.1.1 The method

As mentioned earlier, we drew our inspiration for our notation from Generalised Multiparty Session Types. One of its features is indeed to support general graph with an adapted syntax. Therefore one can more easily deduce some properties usually linked to graphs as connexity, liveness and deadlock-freedom.

Furthermore global protocols, as defined in the Generalised Multiparty Session Types, give us some guidelines to develop a graph representation.

- There should be exactly one start point and at most one end point.
- Join should only happen for parallel threads.
- The graph has to support the basics: interaction, choice, parallel, recursion.

It is also worth noting that Generalised Multiparty Session Types allow for a greater expressiveness than the actual Scribble language. For instance, in a choice construct with three branches, in Generalised Multiparty Session Types the two first branches can merge at some point and eventually merge with the last one before proceeding, whereas in Scribble it is not possible as there is no specified merge construct. However we need to work on both as Generalised Multiparty Session Types offer types for global protocol, whereas Scribble is a programming language.

We have summarized the design architecture in Figure 2.1.

2.1.2 The chosen global type

We remind here the syntax we have adopted for global types:

G	::=	def $\tilde{G}$ in x	Global type
G	::=	x = p $\rightarrow p' : l\langle U \rangle ; x'$	Labelled messages
	|	x = x' | x"	Fork
	|	x = x' + x"	Choice
	|	x | x' = x"	Join
	|	x + x' = x"	Merge
	|	x = end	End
U	::=	$\langle G \rangle$ | bool | nat | ...	Sorts

2.2 Discussion about existing representation

In the literature, there are already some graphic tools. We present some of them here, the commonly used ones, and discuss their accuracy with respect to the representation of global protocols.

2.2.1 Free Choice Petri Nets

As we have studied a formal representation of global protocol, the first representation we can think of is a Petri net. It is indeed quite popular in concurrent systems modelling.

Petri nets are defined by a set of places and a set of transitions. Two transitions can be in causal relation, in conflict, or concurrent. There are commonly used to prove dynamic properties, such as liveness, deadlock-freedom, bounding, and cycle.

There is a class of Petri nets, called free-choice Petri nets, that respects this condition: If there is an arc from a place s to a transition t, then there must be an arc from any input place of t to any output transition of s. Free-choice Petri nets can be used to model the flow of control in networks of processors. On the other hand, the data manipulated by the processors are not modeled but like black dots.

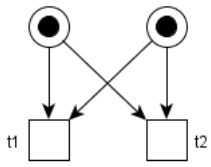


Figure 2.2: A Free Choice Petri Net

To conclude, it is possible to deal with a class of Petri nets that matches, by means of constraints, our structure and behaviour. However, this representation is more useful for demonstrations of dynamic properties for a specific model rather than for a visual tool. It is for instance used to verify well-formedness in [17].

2.2.2 Finite State Machines

In the same way, finite state machines are commonly used to model processes.

From the local protocol, one can easily generate a finite state machine to perform dynamic verifications. The states represent the conversation states relevant to the target role; the transitions correspond to the messages expected to be sent or received by the endpoint in the current protocol state.

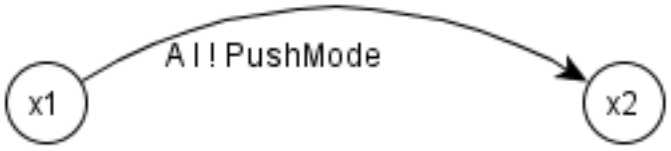


Figure 2.3: A Finite State Machine

For instance, Figure 2.3 represents a scenario where A sends *PushMode* to I, from the point of view of A. The label on the arrow states that the message *PushMode* will be sent (!) on channel A I.

This kind of notation is also not really accurate for representing global protocol, as it is meant to represent processes.

2.2.3 Business Process Modelling Notation

There is a widely-used among process designers notation, called BPMN Choreography (BPMN stands for Business Process Modelling Notation).

BPMN 2.0 from the Object Management Group (OMG) [1, 2] is a standardized graphical notation for modelling business processes. It is intended to provide a notation that is readily understandable by all business users (including business analysts, technical developers, and those who will manage and monitor the processes after implementation) and to create a standardized bridge between business process design and XML-based business execution languages, such as BPEL4WS and Sybase Integration Orchestrator. In [8], they have also worked on similar issues.

BPMN notation is based on three basic elements:

- Activities: An activity is a work that is performed within a business process. It can be atomic, or non-atomic. A task is a type of atomic activity.
- Events: An event is something that *happens* during the course of a business process. Events affect the flow of the process and usually have a trigger or a result. In particular, intermediate events attached to boundary of an activity indicate that the activity should be interrupted when the event is triggered. It is used mainly for error handling, exception handling or compensation.
- Gateways: They are modelling elements that are used to control how Sequence Flows interact as they converge and diverge within a process.

BPMN 2.0 provides the following diagrams:

- Conversation diagrams - which provide an overview of the communications between participants,
- Choreography diagrams - which focus on the detail of the conversation between two or more participants, and which are often linked to specific conversation nodes,
- Collaboration diagrams - which focus on the messages that pass between participants; participants can be shown as black boxes or with processes inside them,
- Process diagrams - which focus on the sequence flow in a single process in a participant.

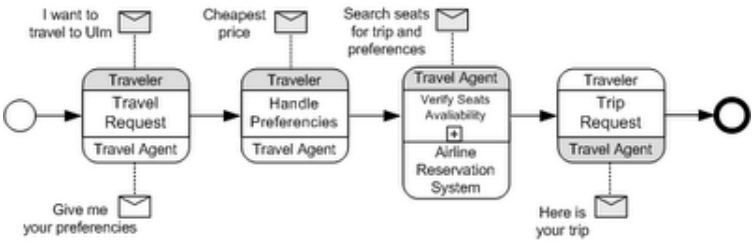


Figure 2.4: An example of BPMN choreography from bpm-research.blogspot.co.uk

Conversation diagrams allow linking the participants together if they have a communication at some point in the scenario. There is no detail on the steps of the conversion: sequence, parallel, choice, etc.

Choreography diagrams are used to analyse how participants exchange information to coordinate their interactions. A diagram consists in a sequence of choreographic tasks, which represent an interaction between two participants, i.e. for each task, messages can be attached: one from the initiator of the task, and possibly a reply message from the receiver. The diagram therefore shows the interactions between the participants. There is also the possibility to call another choreography or sub-choreography.

Collaboration diagrams, which focus on the messages, are designed in a way that shows the interaction in a user/process point of view. It separates visually each participant in a pool. Therefore it corresponds better to a local view, rather than a global view.

Process diagrams concern a single participant. Therefore it suits more for a local view purpose.

The most appropriate for our purpose seems to be the choreography diagram, which represents the interactions between participants. One of its drawback is however to allow that in one task a message is sent and also its reply, whereas we are focusing on single interactions between processes.

This analysis proves that BPMN is a tool that offers a lot of possibility in modelling business processes. Furthermore it is already adopted by a wide range of process designers. On the other hand, it seems not to be completely adapted to our purpose of global protocol representation. That is why we suggest using a different notation that we have developed.

2.3 The chosen notations

2.3.1 General principles

As we have discussed in the previous section some notations that already exist in the literature, we now present the graphical representation that we have chosen for our work.

The main principles that have guided our choice are:

- Simplicity
- Readability
- Clear correspondence with Scribble global protocol

Furthermore, our purpose is to have notations that not only match global protocol but also look like BPMN in order to be easily adopted by engineers. To respect these principles, we proceeded with discussing the key information that are needed and have to appear on the graph. This helps us define our notations. As we were looking for a representation that matches the Scribble language, we assume that graphs only represent constructs that are supported by Scribble. This restriction against the Generalised Multiparty Session Types will disappear with future extensions of Scribble.

We now present the notations for the communicating features we are using in our protocols. All the following examples are extracted from the one at the beginning of this report. The generic entity is a box and corresponds to a message interaction. Then there are several diamond gates: choice/merge, fork/join, recursion; and also a start point and an end point. All these ones are similar to BPMN notations. We add an interruptible node.

2.3.2 Message

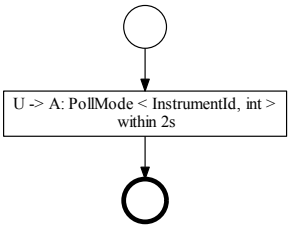


Figure 2.5: Representation of a message passing

We study here a simple protocol containing one message. Its name is *FirstMessage* and it involves two roles: A and U. The protocol specifies that U sends a message to A. This message has for an operator *PollMode* and carried two data, which have for payload types *InstrumentId* and *int*.
The graph representing this protocol, Figure 2.5, is composed of three nodes:

- The start node: a simple circle,
- The message node: a box with the message signature,
- The end node: a circle in bold style.

The program in Scribble for this protocol is as follows:

```
global protocol FirstMessage (role A, role U) {
  PollMode(InstrumentId,int) from U to A within 2s;
}
```

It starts with the key word `global protocol`, and then specifies the name, the roles and the body of the protocol. We will give more details about the *within* statement later.

As we have set the basic structure, we now explore other constructs with inside interactions.

2.3.3 Choice

We study here a protocol containing a choice. Its name is *FirstChoice* and it involves three roles: A, U and I. The protocol specifies that U sends a message to A, followed by A sends a message to I, and then I has a choice between two branches: sending to A the message *Supported*, or the message *NotSupported*.
The graph representing this protocol, Figure 2.6, is composed of two new nodes:

- The choice node: a diamond with a + inside,
- The merge node: a diamond in bold style with a + inside.

The choice node states that there are several possible branches. When the interactions inside the branches finish, branches merge to the merge node. If there is a recursion inside a branch, this branch will not merge.

The program in Scribble for this protocol is as follows:

```
global protocol FirstChoice (role A, role U, role I) {
  PushMode(InstrumentId) from U to A within 2s;
  PushMode from A to I within 1s;
  choice at I {
    Supported from I to A within 1s;
    } or {
    NotSupported from I to A within 1s;
    }
}
```

The Scribble protocol clearly specifies the role responsible for the choice. On the graph, we can check the local choice property, that is we can check if there is a unique sender if the receivers are the same also, as defined in [17].

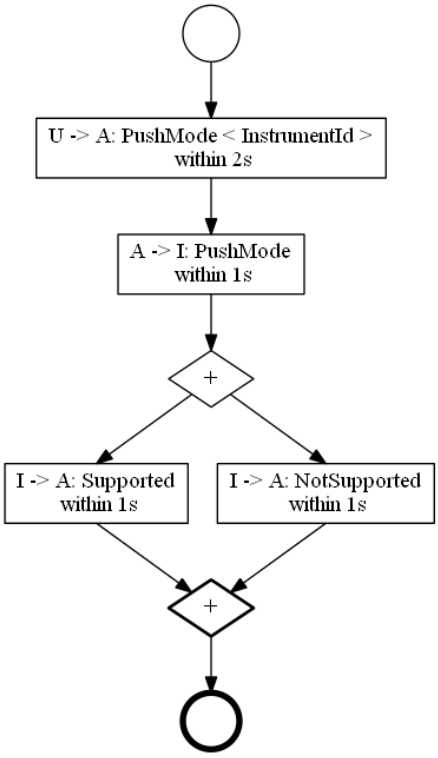


Figure 2.6: Representation of a choice construct

2.3.4 Parallel

We study here a protocol containing two threads in parallel. Its name is *FirstParallel* and it involves three roles: A, I and U. The protocol specifies that I sends a message to A, and then, in parallel, we have A sending to I the message *ConfigPush*, followed by I sendin to A the message *Raw(Data)*, and A sending to U the message *Formatted(Data)*.
The graph representing this protocol, Figure 2.7, is composed of two new nodes:

- The fork node: a diamond with a | inside,
- The join node: a diamond in bold style with a | inside.

The fork node states that, from this point, several threads will be executed in parallel. When the interactions inside the threads finish for all of them, threads join to the join node. If there is a recursion inside one thread, there will be not join node. Only parallel threads can be joined, in order to avoid deadlocks.

The program in Scribble for this protocol is as follows:

```
global protocol FirstParallel ( role A, role I, role U) {
  NotSupported from I to A within 1s;
  parallel {
    ConfigPush from A to I within 1s;
    Raw(Data) from I to A within 1s;
    } and {
    Formatted(Data) from A to U within 2s;
    }
}
```

On the graph we can check the linearity property, that is we can check if there is in parallel threads there is not concurrent messages that have the same label, as well as the same sender and receiver, as defined in [17].

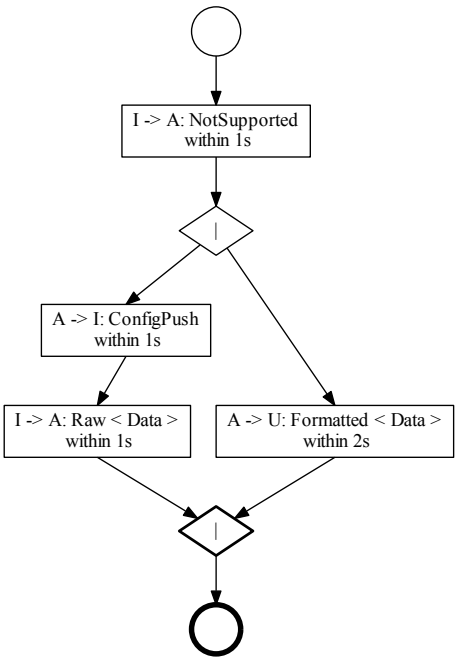


Figure 2.7: Representation of a parallel construct

2.3.5 Recursion

We study here a protocol containing a recursion. Its name is *FirstRecursion* and it involves three roles: A, I and U. The protocol consists only in a recursion: I sends a message to A and A send a message to U, and then it comes back at the beginning of the protocol again.
The graph representing this protocol, Figure 2.8, is composed of one new node:

- The recursion node: a empty diamond.

The recursion node signals the beginning of the recursion loop.

The program in Scribble for this protocol is as follows:

```
global protocol FirstRecursion ( role A, role I, role U) {
  rec PUSH {
    Raw(Data) from I to A within 1s;
    Formatted(Data) from A to U within 2s;
           continue PUSH;
    }
}
```

Global protocol in Scribble

We need to specify exactly the same label for the recursion and the corresponding *continue* statement: here it is *PUSH*. This has indeed an influence in the case of nested recursion constructs as we will see in the chapter 7.

2.3.6 Interruption

We study here a protocol containing a interruption. Its name is *FirstInterruption* and it involves two roles: A and U. The protocol consists in one message that can be interrupted: U continuously sends the same message to A unless A interrupts it with the message *stop*.
The graph representing this protocol, Figure 2.9, is composed of one new node:

- The interruption node: an oval.

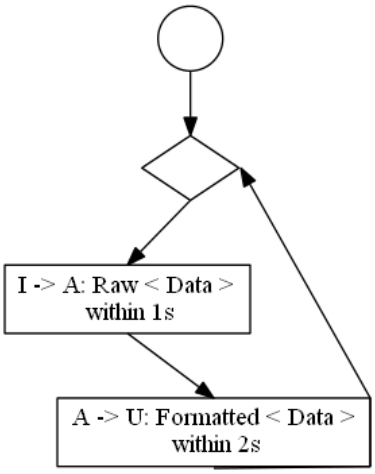


Figure 2.8: Representation of a recursion

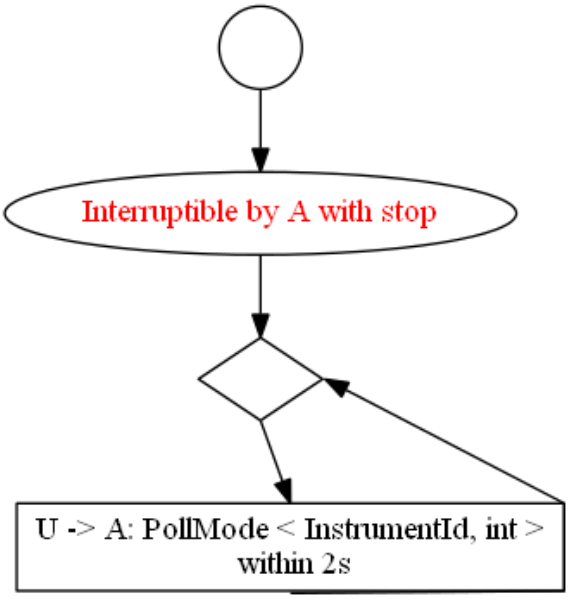


Figure 2.9: Representation of a interruption

The interruption node signals the beginning of the block of interactions that can be interrupted.

The program in Scribble for this protocol is as follows:

```
global protocol TestInterrupt ( role A, role U) {
  interruptible {
    rec Loop {
      PollMode(InstrumentId,int) from U to A within 2s;
      continue Loop;
    }
  } by A with stop
}
```

Global protocol in Scribble

2.3.7 Well-formedness conditions

As we assume that graphs only represent constructs that are supported by Scribble, a choice activity (respectively a fork activity) is supposed to end with a unique merge action (a join action), except if there is a recursion inside. In this last case, there will be an end-point in the graph if one of the branch of the choice stops, otherwise there will be no end-point.

In all cases we assume that graphs and protocols are well-formed, since in [17] they have demonstrated that it can be checked in polynomial time.

2.4 Further remarks

The graph representation we have presented here may be extended or reused in several ways: it is supposed to evolve with the research progress. Furthermore, as the graphical objects could deal with more features, they can be inserted in other software tools that need a graph representation.

As Scribble is supposed to evolve, so will the graph notation. The first possible extension could be to deal with delegation. This feature is already in Scribble but not yet in our graph representation. Another one is the timeout that we will present in the following chapters. We can also think of the treatment of exception and extend the notation for interruption.

Chapter 3

The development structure

3.1 General structure

As we now have set the notations, we give the steps of the development. One of the first questions we asked ourselves was: in which way do we need the correspondence? i.e. from graph to protocol or from protocol to graph? It appears that both were needed and interesting. Therefore we obtain the above mentioned correspondence in both ways.

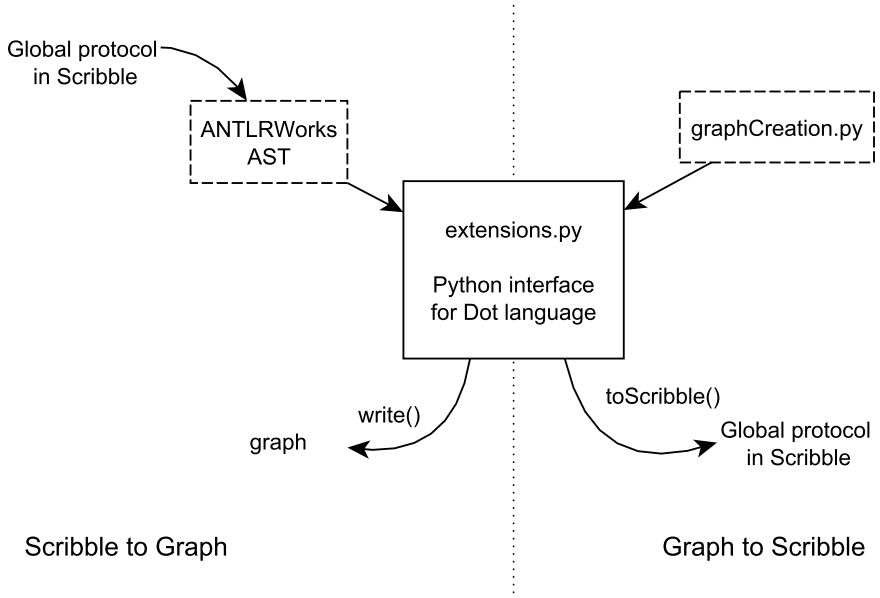


Figure 3.1: General structure for the development

Figure 3.1 represents the general structure that supports our development. The central data structure of our correspondence between graph representations, following the design we have defined in chapter 2, and global protocols in Scribble, is a script describing the graph in the Dot language. We will introduce the Dot language in chapter 4. To write this script, we use a Python file that we have developed and called *extensions*. This file indeed extends a Python library, which implements an interface for the Dot language, as we will explain later.

On one hand, we start with the Scribble protocol and parse it to obtain the script, and then from the script we print the graph. On the other hand, we start with the code of the script and print the protocol in Scribble. We explain in more details the structure in the following sections.

3.2 From Scribble protocol to graph

We begin with exposing how we manage from a global protocol in the Scribble language to obtain a graph. We chose to start with this way of the correspondence because the global protocol is already fixed by the research community, whereas the graphic objects were to define.

3.2.1 Method

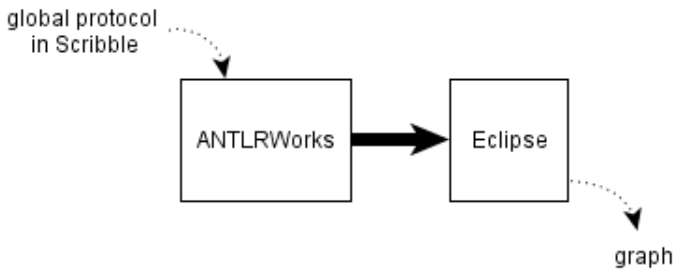


Figure 3.2: Structure of the development

We can divide the work into two parts: (1) read the protocol and store the data, (2) while going through the collected data, create the graph. The first part was done within ANTLRWorks and the second in an Eclipse environment, as represented in Figure 3.2.

Given a global protocol in Scribble, the first issue for our program is to read it. The corresponding step in our development is to define a grammar for the global protocol in Scribble. This grammar allows decomposing a protocol into several predefined structures by defining lexer rules and parser rules. For each of these structures, we build a part of a tree with the essential information extracted from the protocol.

While parsing a Scribble global protocol, we build an abstract syntax tree. This will later be used to create the graphical representation of the protocol. An abstract syntax tree is, as its name states it, a data structure created from the protocol, in order to reuse the data of the protocol which were read.

Once an abstract syntax tree is built from the parsing of the protocol, it is possible to walk through it. To do this, ANTLRWorks allows defining a grammar for trees, which is: for each sub-structure of the tree we can state some code in Python. As well as providing an easy tool to build grammars, ANTLRWorks can generate Python files that can be reused and modified in a larger project.

Figure 3.3 represents the structure of the development with some details that we explain next.

3.2.2 ANTLRWorks

Following this method, we have developed a lexer and a parser using the ANTLRWorks tool. ANTLRWorks is a grammar development environment for ANTLR v3 grammars. It combines a grammar-aware editor with an interpreter for rapid prototyping and a language-agnostic debugger for isolating grammar errors. ANTLRWorks helps eliminate grammar nondeterminisms by highlighting nondeterministic paths in the syntax diagram associated with a grammar.

3.2.3 Lexer, Parser

The lexer and parser are defined with some rules. Lexer rules define the tokens used to identify parts of the protocol during the reading, whereas a parser rule states how a structure of the protocol will be stored in the abstract syntax tree, as explained next.

The parsing rules we developed follow the grammar underlying the Scribble language, as specified in [7]. We define tokens to be subtrees' roots. The grammar states the way it is built. One significant feature of ANTLR is the rewrite rule: it allows creating our own tokens to structure the tree rather than using the ones read by the lexer.

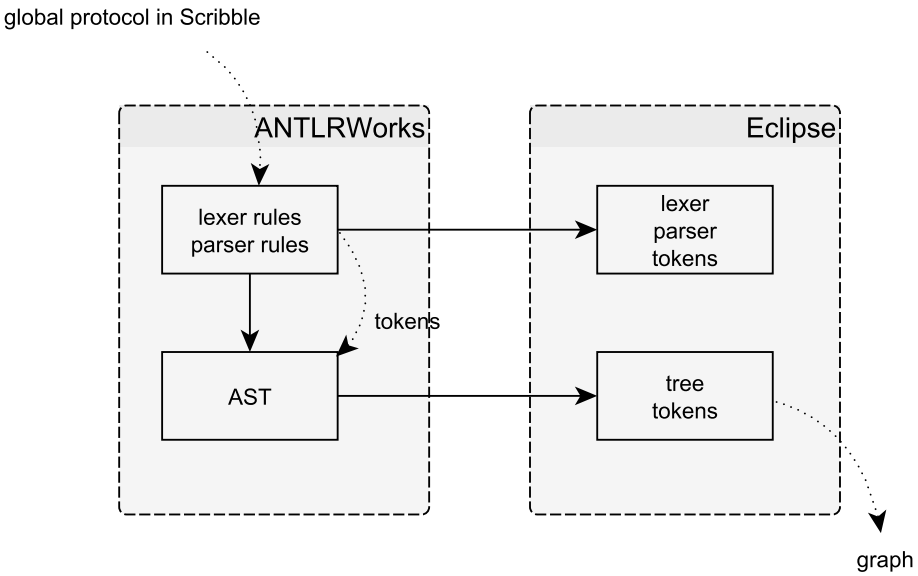


Figure 3.3: Details of the structure for the development

```
choice
        :       Choice 'at' roleName block ( 'or'  block )* -> ^(Choice roleName block+ );
```

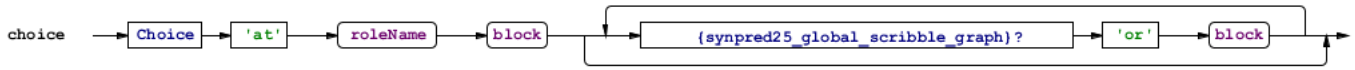


Figure 3.4: An example of a parser rule and its diagram: the choice case

For each activities (message, recursion, choice, etc.), there is a node in the corresponding abstract syntax tree. Its children are the added information. For instance a choice node will have a role as its first son, corresponding to the role that makes the choice, and then as many other sons as there are alternatives, as represented in Figure 3.4. An alternative will be represented as a block. The protocol body is also a block. A block is a node whose son is an activity list. The latter is also a node whose sons are consecutives activities, in the same order as defined in the protocol. These different levels of syntax allow avoiding grammar nondeterminism.

Therefore given a Scribble global protocol in input of the program we obtain, after parsing and walking through the abstract syntax tree, the graph representation of the protocol. More details will follow in the next chapter.

3.3 From graph to Scribble protocol

We have described in the previous section how we have carried out the generation of a graph from the global protocol, we will now explain the other way round. It is indeed often easier for an engineer to draw a graph rather than to write the code for a protocol. Moreover discussing about a graph is more interactive for designing rather than commenting a code. That is why we want to be able to build first the graph and obtain from it the global protocol.

The first step is to create the graph we want to start from. For this purpose, we do not use any particular

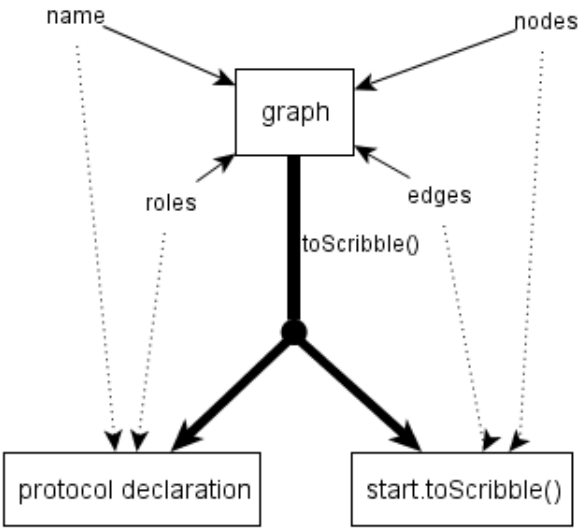


Figure 3.5: Representation of the development

software but the same graphic objects we have developed in Python to generate the graph. While creating the graph, we store added pieces of information, which are its name and the concerned roles. It will be useful to write the protocol declaration as represented in Figure 3.5.

The next step to obtain the global protocol is to walk through the created graph and write the protocol with the information stored in the nodes. From one node to the next one, we walk through the whole graph to retrieve all necessary pieces of information. We begin with the first node of the graph, which is the start node. Therefore we are able to print the global protocol in Scribble.

Chapter 4

Details of the implementation

Along with the development we kept on improving our graph notations in order to obtain the ones presented in chapter 2. Implementing was indeed useful to refine or improve our first thoughts. We present here some details about the implementation: the issues faced, the structure of the code, and the choices made for the design. Most of the work exposed hereunder was done in an Eclipse environment as it allows for more flexibility in the code.

4.1 Structure of the code

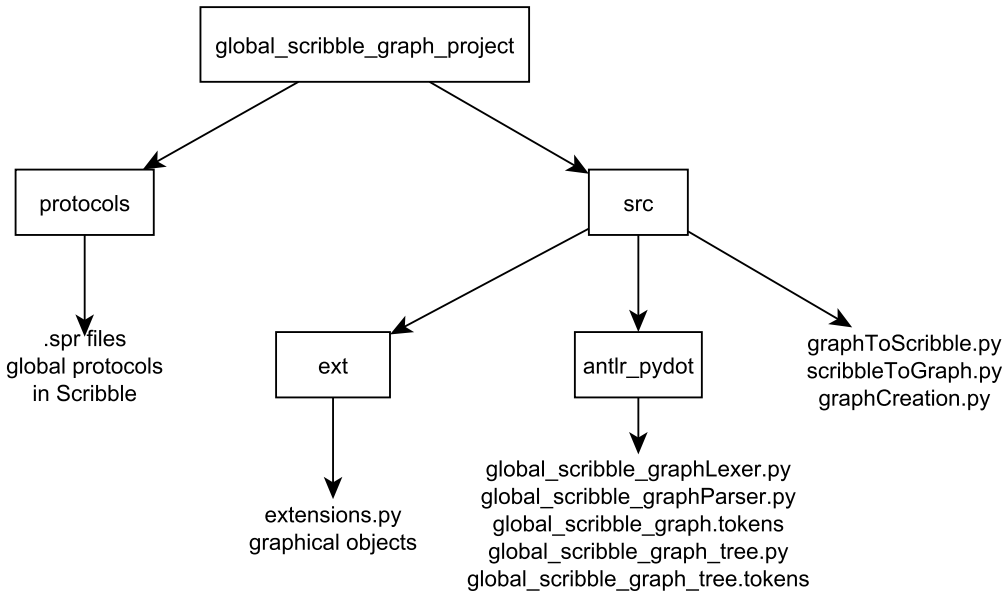


Figure 4.1: Overview of the project

We present in Figure4.1 the project that we developed in Python in an Eclipse environment. We called our project *global_scribble_graph_project*. We have defined two separate packages:

- Protocols: it contains all the example of protocols we used during the development.
- Source: it contains a sub-package with all the files generated with ANTLRWorks, another sub-package ext with the extensions of Pydot (This file defines the graphical objects we use in our project), and also other Python files: the main method for both ways and a file to specify graphs in Python.

We present in the following sections the details of the implementation of this project.

4.2 The Python library: an extension of pydot

The choice of developing our own graphical notations implies to create some graphical objects. We have chosen to develop the software in Python as it is the language already used in the research group.

4.2.1 Pydot

In the literature there are several libraries for graph development. We have decided to use pydot, a Python interface to Graphviz's Dot language [4].

Graphviz is an open source graph visualization software [3]. The Graphviz layout programs take descriptions of graphs in a simple text language, called Dot, and make diagrams in useful formats, such as images and SVG for web pages, PDF or Postscript for inclusion in other documents; or display in an interactive graph browser. (Graphviz also supports GXL, an XML dialect.) Graphviz has many useful features for concrete diagrams, such as options for colors, fonts, tabular node layouts, line styles, hyperlinks, roll and custom shapes. The algorithm underlying the construction of a graph in Dot language is based on assigning nodes to discrete ranks and ordering nodes within ranks.

Pydot allows easily creating both directed and non-directed graphs from Python. Currently all attributes implemented in the Dot language are supported (up to Graphviz 2.26.3). As well as downloading the pydot package, it is needed to download also graphviz and pyparsing in order to use it, as explains in Appendix with more details.

4.2.2 Extensions

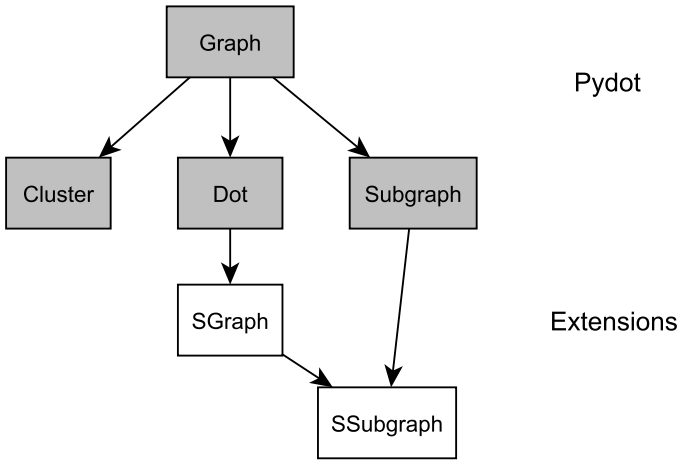


Figure 4.2: Class diagram for graphs

From this existing library, we extended the code in order to create our own graphic objects: SNode, SEdge, SGraph, SSubgraph - S stands for Scribble. These objects inherit from the pydot objects Node, Edge, Dot, Subgraph, as presented in Figures 4.2 and 4.3 and are designed to have the desired features. The SNode class has several derived class, corresponding to each type of nodes we expect, i.e. Start, Message, Choice, etc. As an example, we give here the code to define a Join node, as deriving from the SNode class:

```
class Join(SNode):

    def __init__(self, name = '', label = "|", style = 'bold', shape = 'diamond', obj_dict = None, **attrs):
        pydot.Node.__init__(self, name = name, label = label, style = style, shape = shape,
        obj_dict = obj_dict, **attrs)

        if obj_dict is None:
            self.obj_dict['activity']='join'
```

We also develop some new methods to deal with the operation we need, as explained in the following sections, or change some existing method.
For instance we add a method to obtain an edge from its source node:

```
def get_edge_src(self, src):
    """Get the list of Edge instances which source is src.
    """
    edge_objs = list()

    for edge, obj_dict_list in self.obj_dict['edges'].iteritems():
        if edge[0] == src:
            edge_objs.extend( [ SEdge( obj_dict = obj_d ) for obj_d in obj_dict_list ] )

    return edge_objs
```

We change for example the method to add a node in a graph:

```
def add_Snode(self,snode):
    """Adds a SNode object to the graph and store the roles.

    It takes a node object as its only argument and returns
    None.
    """
    if isinstance(snode, Message):

        self.add_role(snode.get_role1())
        self.add_role(snode.get_role2())

    if isinstance(snode, Choice):
        self.add_role(snode.get_role())

    self.add_node(snode)
```

We used this adapted library to deal with graphical objects in the development of the correspondence between the global protocol and the representation.

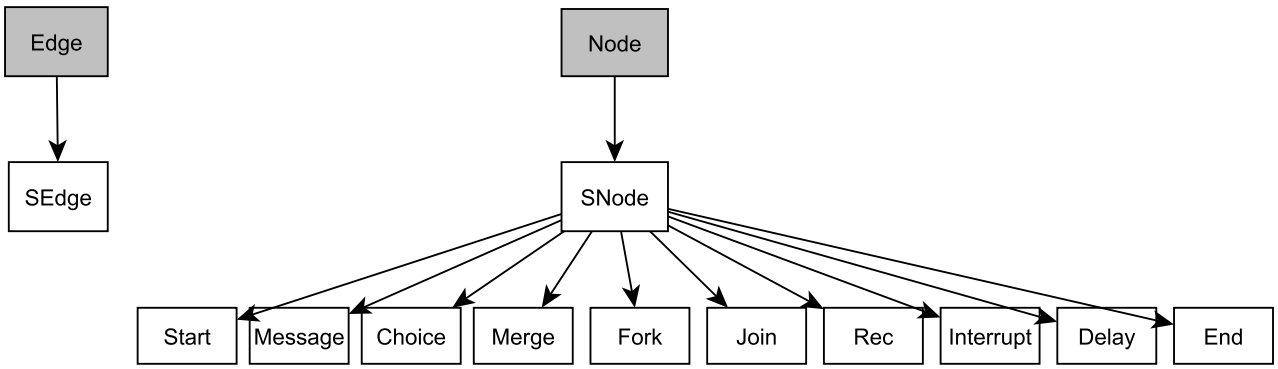


Figure 4.3: Class diagram for edges and nodes

4.3 From Scribble to graph

4.3.1 Main method

Our code is based on the Python files generated with ANTLRWorks: the lexer, the parser and the tree. We use methods, predefined with the ANTLR tool, to write the main method: first we prepare the tree from the input with the lexer and the parser, and then we obtain a stream of all the nodes in a depth-first search way and execute the code associated with them. The processus is described in Figure 4.4.

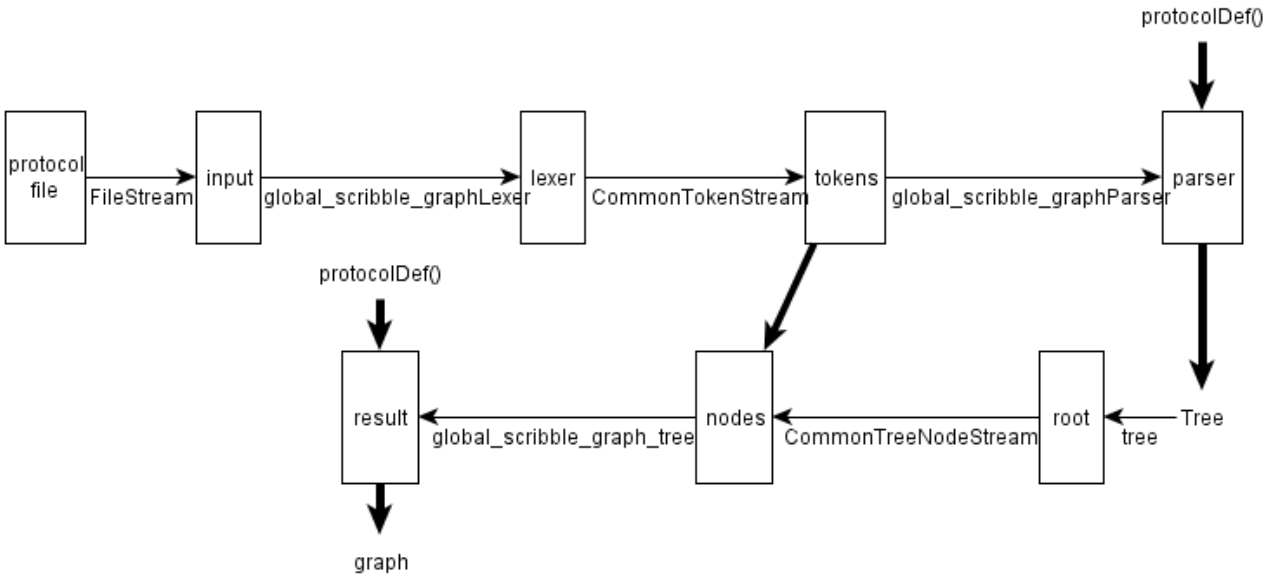


Figure 4.4: Implementation's structure from Scribble to graph

As a result, walking through all the nodes of the abstract syntax tree allows creating all the nodes and edges needed for the graph. The last step consists in printing the graph. This method is defined in the pydot library: from the Python code it generates a script in the Dot language which in turn can be executed to produce the graph.

4.3.2 Choices and assumptions

We first give as an example the case of a simple interaction:

```
msg = extensions.Message( nodeName, Message5.getChild(1).getText(),
Message5.getChild(2).getText(), msgLabel, timeLabel)
graph.add_Snode(msg)
```

When we meet a node Message, we program to create a graphic object Message (derived from SNode) with attributes Sender, Receiver, MessageLabel and TimeLabel, that are the ones stated in the leaves of the node Message. For this purpose we use the predefined methods `getChild()` and `getText()`.

In the same way we build nodes for the other activities. For choice (respectively fork) nodes, after creating them we proceed with the alternative blocks which number is evaluated with the methods `getChildCount()`. In the case where there is no recursion inside the branches (respectively threads), we also create a merge (respectively join) node to close the choice (respectively parallel) section. This part of code may evolve when Scribble will support merge and join constructs, as well as explicit end. For the recursion, we assume that each recursion label is unique, as it will be the name of the node.

It is worth noting that each node should have a different name. To solve this issue we associate to the graph a counter that generates integers to name the nodes.

The execution of each activity returns two nodes the first one and last one: for instance, for a message both are the created node Message and for a choice it is the choice node and the merge node. Also an activity list, as well as a block, take in argument the previous node and return its last one in order to be able to build transitions between blocks. Then for the edges of the graph, we build them in the activity list section from one son to the next one, retrieving the nodes from the activities.

```
begin, end = self.activity(graph)
# action start
# at this point, we make the connection between the previous node (start) and the first
activity and within the activities in the list
edge = extensions.SEdge(currentNode, begin)
```

```
import sys
import antlr3
from ext import extensions
from antlr_pydot5.global_scribble_graphLexer import global_scribble_graphLexer
from antlr_pydot5.global_scribble_graphParser import global_scribble_graphParser
from antlr_pydot5.global_scribble_graph_tree import global_scribble_graph_tree

def main (argv):
    # first step: parse the global protocol into a tree
    filename = "C:/workspaceProject/global_scribble_graph/protocols/ooi_example_final.spr"
    input = antlr3.FileStream (filename)
    lexer = global_scribble_graphLexer (input)
    tokens = antlr3.CommonTokenStream (lexer)
    parser = global_scribble_graphParser (tokens)
    res = parser.protocolDef ()
    root = res.tree

    # second step: walk through the tree
    nodes = antlr3.tree.CommonTreeNodeStream(root)
    nodes.setTokenStream(tokens)
    result = global_scribble_graph_tree(nodes)

    # third step: create the graph
    graph = extensions.SGraph("G")
    result.protocolDef(graph)


if __name__ == "__main__":
    sys.exit (main (sys.argv))
```

Figure 4.5: Code of the main method

```
graph.add_Sedge(edge)
currentNode = end
# action end
```

4.4 From graph to Scribble

4.4.1 Creation of the graph

For the creation of the graph, we use the graphic library described in a previous section: the extension of pydot. It is indeed the central part of our correspondence between graphs and Scribble protocols, as presented in Figure 3.1. It allows creating graph, nodes, edges and subgraphs. We can also create edges to link these objects together and specify, if needed, the headport and tailport of the edge to obtain the expected visualization. The call to the method `graph.print()` generates the given graph. The development of this way of the correspondence brings us to continue extending the library.

4.4.2 Generation of the protocol

For this purpose, we have created a method called `toScribble()`, defined in the classes SGraph and SNode, and implemented in SGraph and the derived classes of SNode. The call of this method for a SNode prints the protocol concerning this node and also obtains the next node in the graph to go ahead. Therefore we develop also a method `next_nodes()` as follows:

```
def next_nodes(self, graph):
        edges =graph.get_edge_src(self.get_name())
        names =[]
```

```
next_nodes = []
for e in edges:
    names = names+[e.get_destination()]
for n in names:
    next_nodes = next_nodes + graph.get_Snode(n)
return next_nodes
```

For the parallel case, we first deal with the inside blocks from the fork node until the join node, then we continue with the node after the join one. It works in the same way for the choice case. However, we have to pay attention whether there is a join node (respectively a merge node) or not, to know when to stops. That is why the `toScribble()` method returns both the code and the end node of the constructs if there is one, for instance it could be a join node, and also why we use a flag to mark if one thread does not finish because of a recursion. The detail of the code is in Appendix.

We assume that after a message, recursion or interrupt node there is only one node. For the recursion - continue construct, we set a flag in the recursion node the first time we reach it and change it the second time in order to distinguish between the two syntax.

The method also manages to indent the protocol and print it in a pretty way in order to get a readable global protocol.

To conclude, we now have a correspondence between global protocols in Scribble and a graph representation. It means for the following that all work done to extend the graph representation should also extend Scribble protocols and the underlying theory, and vice-versa. We will apply this on an extension handling with a notion of time.

Chapter 5

Towards graphs with time

5.1 General presentation

As we aim at modelling real situations where several processes are interacting, such as the use case from the Ocean Observatories Initiative, we have to handle a notion of time, which is an important feature of physical systems. That is why we explore now how to extend our graph with time. This chapter introduces the notations, whereas the next one presents the theory underlying our model.

5.1.1 Useful definitions

We first give here some definitions from the Oxford dictionary in order to clarify terms we will use in this chapter.

Delay A period of time by which something is late or postponed.

Timer An automatic mechanism for activating a device at a preset time.

Timeout A cancellation or cessation that automatically occurs when a predefined interval of time has passed without a certain event occurring.

5.1.2 Motivations

Our current model allows representing complex interactions among several processes such as concurrency, recursion and branching. Theoretically there should be neither errors nor deadlocks. However, for critical software for instance, it sounds more realistic to have timeouts for each interaction. It allows indeed to avoid situation where a process is stuck waiting for a message. This is major issue for distributed systems.

Furthermore, to be able to handle time constraints in a scenario extends the expressiveness of our model: that is not only timeouts, but also several types of time constraints like delays. Time as a physical parameter has indeed an influence in most of real use cases.

Therefore we aim at introducing continuous time in our protocols by adding a set of clocks. To achieve this, we first focus on the graphical representation to figure out how to express time with our graph notations, and also on Scribble, as it corresponds to the graph. A second step is to define the theory underlying this notion of time. For this purpose we want to extend our previous syntax for generalised global protocols to continue supporting the extended Scribble language. Finally we will adapt our previous implementation work to encompass all these changes and guarantee the equivalence we have established between graphs and Scribble global protocols.

5.2 Design

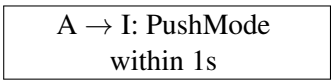
We introduce in this section the design we have chosen to represent timeouts, delays and other time constraints.

5.2.1 General example

To illustrate our motivations, we look at a variation of the example from the Ocean Observatories Initiative mentioned at the beginning of this report. Our goal does indeed remain to develop a graphical representation that expresses as precisely as possible a global scenario. Therefore we begin with defining which features we want to express on the graph and how to represent them.

The graph in Figure 5.1 presents three main differences against the first one: (1) the *within* statement inside the message boxes, (2) the clock node, and (3) the constraints on arrows.

The *within* statement provides a notion of timeout on interactions. For instance,



means that I should receive the message *PushMode* at most one unit of time after A has sent it.

The clock node represents a notion of delay. It can be expressed either as relative time or as absolute time. A clock with the label t + 3min means that the next interaction will happen in three minutes with respect to the clock t. While a clock with the label t = 5s means that the next interaction will happen when the clock t will indicate 5s.

The labels on the arrows refer to the different time constraints we require. It also specifies clocks to be reset with a label like t1 = 0s. Therefore clocks encompass the notion of timer. It is possible to define several constraints on one arrow.

It is possible to define as many clocks as needed. In our example, we use six clocks. We assume that at the beginning all the clocks are set to 0.

5.2.2 A new node: Delay

We study here a protocol containing a delay specification. Its name is *FirstDelay* and it involves two roles: A and I. The protocol specifies that I sends a message to A, and then there is a delay before A sends to I the message *ConfigPush*. A has indeed to wait for three seconds before continuing with the next interaction. The graph representing this protocol, Figure 5.2, is composed of one new node:

- The delay node: a clock with the time to wait on the top right.

The node states the delay to be waited for. t represents the current time on clock t. Therefore t + 3s means that if the time on clock t states 5s, we have to wait until t = 8s. In constrast to other time constraints, which are represented on the arrows, we have chosen to represent delay with a node. In a sense it is indeed an activity with an action, i.e. waiting.

The program in Scribble for this protocol is as follows:

```
global protocol FirstDelay ( role A, role I) {
  NotSupported from I to A within 1s;
  wait for t + 3s
  ConfigPush from A to I within 1s;
}
```

5.3 Extension of the Scribble language

To conform the notations we have defined for graphs, we extend the Scribble language. We use the same syntax notations as the one introduced in a previous section to declare the grammar of the language.

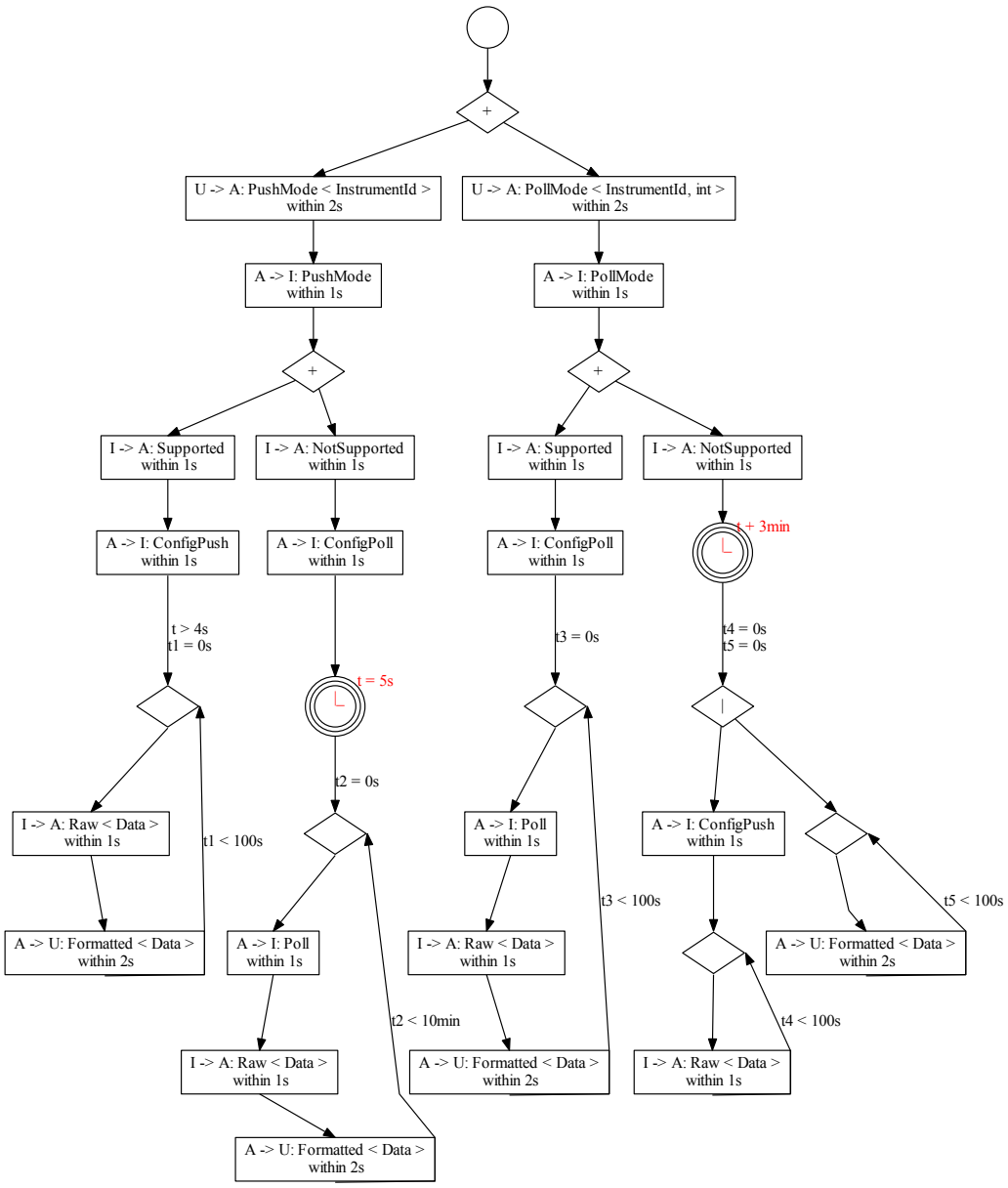


Figure 5.1: A variation of Use Case "Acquire Data from Instrument"

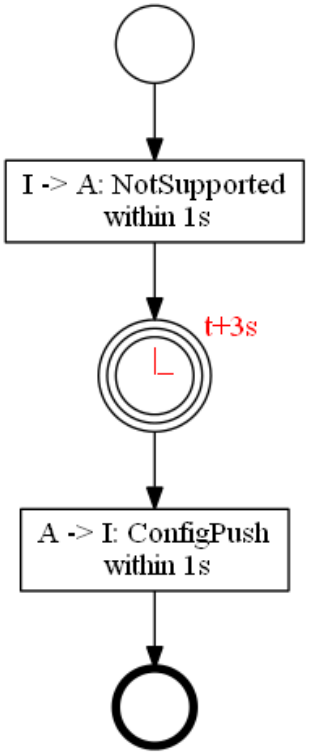


Figure 5.2: Representation of the delay

global-protocol-body	::=	global-interaction-block
global-interaction-block	::=	{ global-interaction-sequence }
global-interaction-sequence	::=	(global-interaction)*
global-interaction	::=	[time-constraints] message
	|	[time-constraints] choice
	|	[time-constraints] parallel
	|	[time-constraints] recursion
	|	[time-constraints] continue
	|	[time-constraints] delay
message	::=	(message-signature | identifier) `from` role-name `to` role-name `within` time;
delay	::=	`wait for` time-identifier symbol time ;
	|	`wait for` time-identifier `is` time ;
time-constraints	::=	constraint (`and` constraint)*
constraint	::=	time-identifier `after` time
	|	time-identifier `before` time
	|	time-identifier `is` time
time-identifier	::=	identifier
time	::=	(digit)* identifier
symbol	::=	('+' | '*')

There are three major differences as for the graph representation: (1) the time constraints, (2) the delay, and (3) the *within* statement.

Time constraints can be defined before each activity as an optional element. It represents constraints that a user writes on the arrows of the graph representing the global protocol. We assume that there is no constraint on the last arrow pointing at the end node. Time constraints consist of one or more constraints. A date represents an absolute time, for instance 10h.

Delay is a new activity that consists in waiting for a specific time: it could be a relative time, for instance t+3h, or an absolute time, for instance t= 10h. It corresponds to the clock notation on the graph.

The *within* statement has the same meaning as the one define for the graph, for instance within 2s.

Therefore this language clearly match the notations we have introduced for graphs: it is now possible to obtain the equivalence between Scribble global protocols and graphs again.

5.4 Implementation

In order for our implementation to remain accurate with this extension, we extend the work previously done. It consists in three main changes:

- We extend the grammar in ANTLR to support the new Scribble grammar.
- We create a new object node, called Delay, to correspond to the *wait for* statement. It derives from the class SNode and has the symbol of a clock.
- We add a new way to define an edge by specifying a time constraint besides the existing features with the method `graph.add_Sedge_with_label(edge,label)`.

Based on these changes, our implementation handles now the cases of timed global protocols and timed graphs.

Chapter 6

Timed global protocol

As we have now set our graphical notations, we focus on the way to express them theoretically. We will first present some results one can find in the literature about time.

6.1 Related work

We discuss here some related issues, that is in which way a notion of time was introduced in several contexts: in contract formalism, in Pi-calculus, in colored Petri-nets and in automata.

6.1.1 Contract formalism

We first look at contract formalism. We will not give much details as we can find a survey about the topic in [23]. In this thesis, the author defines the contract formalisation as "mathematical models and domain-specific languages for specifying and reasoning about contract". Furthermore, "a contract is a legally binding agreement between two or more parties, and a formalisation is an unambiguous, machine-interpretable representation". In practice, a contract will describe the actions that participants agree to perform.

Most of the contracts are based on a notion of time and in particular of deadline. To handle with this issue, several approaches were developed: (1) logic-based, (2) event-condition-action based, (3) trace-based. Two kind of temporal constraints can be distinguished: the absolute ones and the relative ones.

In [24], they propose a trace-based approach. The main idea is that actions of one trace are time stamped and each obligation must have a deadline. Then "all deadlines that occur in clauses are relative to unspecified reference points which are given by the starting time of the specification and by the time of event occurrences. Thus, these relative deadlines are lifted to absolute deadlines when the CSL (Contract Specification Language) specification is executed." An obligation clause looks like:

<p> k(x) where e due after n_1 within n_2 remaining z then c

where p are the parties, k a type, x binds to e an expression,n_i are times that binds z and c is a clause.

To specify an alternative continuation branch, which becomes active if the deadline passes, they define external choices.

This formalisation can inspire us a way to include time in global protocols. We could define deadlines (after t_1 before t_2) and associate them to blocks of activities inside a protocol. It would mean: this block of activity has to be done within the period defined by the deadline. We have tried to include this feature in our work. However it is not easily representable in a graph in an intuitive way, so we will not consider this possibility in our later work.

6.1.2 Timer

We present here an approach to introduce timer in Pi-calculus. In [12, 13] they introduce a timer to offer channel-based distributed liveness, ie a timer is a converter from (potential) non-terminating rule to terminating channels. The definition for processes is extended with:

$timer^t \langle$ a($\tilde{x}$).P,Q$\rangle$, with t an integer such that t>0

It means that after t steps it turns into process Q, unless it has been stopped, i.e. a message has been received by a timer at a.

The flow of time is communicated at each step in the computation by a time stepper function Φ, which acts on processes. It models the implicit broadcast of time passing.

$$\begin{array}{lrcl} & \Phi(timer^{t+1}\langle P, Q\rangle) & = & timer^{t}\langle P, Q\rangle \\ & \Phi(timer^{1}\langle P, Q\rangle) & = & \mathrm{Q} \\ & \Phi(P \| Q) & = & \Phi(P) \| \Phi(Q) \\ & \Phi((\nu a)P) & = & (\nu a)\Phi(P) \\ \text{otherwise} & \Phi(P) & = & P \end{array}$$

This timer aims at an efficient implementation of periodic behaviour and error recovery. It is indeed possible to extend the calculus to define a recovery mechanism.

In the same way, in [16] they define timed distributed Pi-calculus and introduce time constraints by considering timeout timers for channels. The possible communications are performed at each tick of a universal clock. Active channels are those which could be involved in these communications. The time-stepping function affects the active channels which do not communicate at that tick; the timers of the affected channels are decreased by one unit of time. Furthermore they guarantee some properties like time determinism, maximal progress and time continuity.

Finally, in [25] they develop an extended algebra allowing both time values as well as names to be transmitted between processes, called the πRT-calculus. The new feature of the model is based on timeouts and on the distinction between when an action is possible (enabled) and when an action actually occurs (fires) since they might be delay between these two states. Therefore they propose a timed labelled transition system.

6.1.3 Timed Colored Petri-Nets

We can find lots of work on Petri-nets in the literature as they are appropriate for the modelling of distributed systems, since they allow for the representation of parallelism and synchronization ([18, 31]).

In [31] they introduce timed colored Petri-nets. A colored Petri-net is one where tokens have a value, often referred to as 'colour'. Then each place has a type (a set of colours) and tokens in a place have a value (a colour) belonging to the corresponding type. As they aim at expressing quantitative temporal properties like deadlines, activity duration, response times and delays, they also add a notion of time. They attach a time-stamp to every token. This time-stamp indicates the time a token becomes available. The enabling time of a transition is the maximum time-stamp of the tokens to be consumed. The difference between the firing time and the time-stamp of such a produced token is called the firing delay. To describe time delays, they use intervals. It allows for the representation of time constraints.

In [30] Petri-nets are applied to workflow management with the introduction of a trigger for each transition. There are four types of triggering: time, message, user and automatic (the default one). The principle underpinning the trigger is that an external condition leads to the execution of an enabled task.

6.1.4 Timed Automata

We now expose the main principles of timed automata as introduced in [9–11]. They aim at modelling the behaviour of real-time systems over time. The timed (finite) automata are defined to be a way to annotate state-transition graphs with timing constraints using finitely many real-valued clocks.

These automata are based on timed words which are obtained by coupling a real-valued time with each symbol in a word. Then, when an automaton makes a transition, the choice of the next state depends not only upon the input symbol, but also on the time of this input symbol relative to the times of the previously read symbols. For this purpose, they associate a finite set of (real-valued) clocks with each transition table. A clock can be set to zero simultaneously with any transition. At any instant, the reading of a clock equals the time elapsed since the last time was reset. With each transition, they associate a clock constraint, and require that the transition may be taken only if the current values of the clocks satisfy this constraint. Having multiple clocks allows multiple concurrent delays. The clocks of the automaton do not correspond to the

local clocks of different components in a distributed system. They also assume that all the clocks increase at the uniform rate counting time with respect to a fixed global time frame.

They demonstrate in the paper how timed automata can capture several aspects of real-time systems: qualitative features such as liveness, fairness, and non-determinism; and quantitative features such as periodicity, bounded response, and timing delays. We will present some of these aspects in more details in the following sections.

6.1.5 Timed Pi-calculus

In [29] they introduce clocks (or stopwatches) and two types of clock operations: clock resets and clock constraints in order to extend Pi-calculus with real-time. Then the future behaviour of a real-time system depends not only on the content of a receiving message, but also on the time-stamp of the message. In other words, the time-stamp of the arriving/sent message should satisfy certain time constraints specified by the system, for future transitions to take place. Therefore, there are two new variables: (1) the time-stamp of the message t_m, (2) the clock of the process which sends the message c. A message is then represented by $\langle m, t_m, c \rangle$. They assume that processes have access to countable set of clocks when they are needed and that all clocks advanced at the same rate.

The calculus is defined as follows:

$$
\begin{array}{rcll}
\text{C} & ::= & \mathit{Reset} \parallel \mathit{Constraint} \parallel \varepsilon & \text{(Clock operations)} \\
\text{Reset} & ::= & (\mathit{resetClock}) \parallel \mathit{Constraint}(\mathit{resetClock}) & \\
\text{Constraint} & ::= & \text{ce} \sim 0 & \\
\text{ce} & ::= & x + ce \parallel x - ce \parallel x & \\
\text{x} & ::= & \mathit{Numbe} \parallel \mathit{Clock} & \\
\text{tild} & ::= & < \parallel > \parallel \leq \parallel \geq \parallel = & \\
\Pi & ::= & C\overline{x}\langle y, t_y, c \rangle \parallel Cx(\langle z, t_z, c \rangle) \parallel C\tau \parallel [x = y]\pi \parallel \mathit{Reset} \parallel \mathit{Constraint} & \text{(Processes)}
\end{array}
$$

6.1.6 Other work

We can find several other pieces of work on the same topic in the literature. For instance, in [27] they work on runtime verification framework based on linear temporal logic and colored automata. The framework continuously verifies compliance with respect to a predefined constraint model and is able to recover after the first constraint violation. We also mention here a timed session request defined in [26]:

$$
\begin{array}{rll}
\text{P} ::= & \text{request a(k) during m in P} & \text{Timed session request} \\
& \parallel \text{ accept a(k) given c in P} & \\
& \quad ... & \\
& \parallel \text{kill } c_k & \text{Session abortion}
\end{array}
$$

Furthermore, we can find other extensions that use similar techniques, such as a theory of multiparty session types with assertions based on symmetric sum types ([19]) or the introduction of global assertion which specifies a session type with logical predicates ([5, 15]).

Based on this analysis of previous work, we now present the theoretical part of this work. What is new in our extension is to introduce time for global protocols: both in the Scribble syntax and in global/local types.

6.2 The syntax

As we have fixed the notations concerning time for graphs and Scribble language, we now present the chosen syntax to support these notations. We propose new types for the theory to support the extended Scribble language. For this purpose, we adapt some work from [10, 29] to our case.

6.2.1 Primarily notations

Before stating the protocols types, we introduce a few notations. The definition are similar to the ones for timed automata in [10].

We first declare a set of clocks to express several time notions and constraints. For a set X of clock variables, the set Φ(X) of clock constraints δ is defined inductively by:

$$\delta := x \leq c \mid x \geq c \mid \neg\delta \mid \delta_1 \wedge \delta_2 \mid \varepsilon$$

where x is a clock in X and c is a constant in $\mathbb{Q}$. We also introduce useful abbreviations:

x = c	means	x $\leq$ c and x $\geq$ c
x < c	means	$\neg$ x $\geq$ c
x > c	means	$\neg$ x $\leq$ c

Clock constraints that follow this syntax are well-formed constrainst. For instance $t_1 = 3 \wedge t_2 < 5$ is a well-formed clock constraint.

A clock interpretation v for a set X of clocks assigns a real value to each clock, i.e. it is a mapping from X to $\mathbb{R}$. At the beginning, all clocks are assigned to 0. Then the values change as time goes on, or clocks are reset to 0.

6.2.2 As a generalised global type

From the global point of view, the syntax is defined as follows:

G	::=	def $\tilde{G}$ in x	Global type
G	::=	x = p $\rightarrow$ p' : $l\langle U\rangle, \lambda_O, \delta_O, \lambda_I, \delta_I; x'$	Labelled messages
	|	x = x' | x"	Fork
	|	x = x' + x"	Choice
	|	x | x' = x"	Join
	|	x + x' = x"	Merge
	|	x = end	End
U	::=	$\langle G\rangle$ | bool | *nat* | ...	Sorts

The major difference against generalised global types defined in [17] is the labelled message:

$$\text{x = p} \rightarrow p' : l\langle U\rangle, \lambda_O, \delta_O, \lambda_I, \delta_I \text{ ;x'}$$

p and p' denote respectively the sending and receiving participants. U is the payload type of the message and l its label.

Let C be a finite set of clocks. λ_O, λ_I are included in C and denote the set of clocks to reset. O stands for output and I for input. This means that λ_O denote the clocks to be reset during the output action, and λ_I the clocks to be reset during the input action. δ_O, δ_I are clock constraints over C. δ_O denote the constraint to satisfy to send the message and δ_I the one to satisfy to receive the message.

For instance we could have,

$$\text{x = A} \rightarrow \text{B : TheMessage <bool> , } \{t_x\}, t > 3, \{t_1\}, t_x < 2 \text{ ; x'}$$

This means that to go from state x to state x',

- A has to reset the clock t_x while sending the message labelled *TheMessage* and carrying a boolean to B if the condition t > 3 is satisfied.
- B has to reset the clocks t_1 while receiving the message labelled *TheMessage* and carrying a boolean from A if the condition $t_x < 2$ is satisfied.

t, t_x, t_1 are clocks associated to the corresponding global protocol.

This interaction could be a part of this protocol:

```
global protocol ExampleCFSM (role A, role B){
  t before 3s
  TheMessage from A to B within 2s;
  wait for t1 + 1s
}
```

Figure 6.1 represents the protocol.

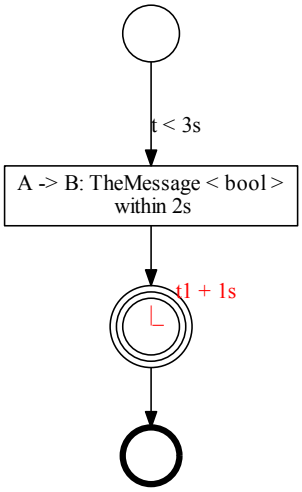


Figure 6.1: Example to illustrate the new syntax

6.2.3 As a local type

In order for the local types syntax to correspond to the communicating finite states machines previously defined, we extend the local types syntax.

T	::=	def $\tilde{T}$ in x	Local type
T	::=	x = $!\langle p, l\langle U\rangle, \lambda, \delta\rangle.x'$	Message sending
	|	x = $?\langle p, l\langle U\rangle, \lambda, \delta\rangle.x'$	Message receiving
	|	x = x' | x"	Fork
	|	x = x' $\oplus$ x"	Internal choice
	|	x = x' & x"	External choice
	|	x | x' = x"	Join
	|	x + x' = x"	Merge
	|	x = x'	Inaction
	|	x = end	End
U	::=	$\langle G\rangle$ | bool | *nat* | ...	Sorts

As for global types, the major differences against generalised global types defined in [17] are the message sending and the message receiving:

$$x = !\langle p, l\langle U\rangle, \lambda, \delta\rangle.x'$$
$$x = ?\langle p, l\langle U\rangle, \lambda, \delta\rangle.x'$$

p denotes respectively the receiving and sending participants. U is the payload type of the message and l its label.

Let C be a finite set of clocks. λ is included in C and denotes the set of clocks to reset during the action. δ is a clock constraint over C to satisfy to respectively send the message and receive the message.

For instance our previous example will be written from the point of view of A like:

$$x = !\langle \text{ B, TheMessage <bool> , } \{t_x\}, t > 3 \rangle .\ x'$$

and from the point of view of B like:

$$x = ?\langle \text{ A, TheMessage <bool> , } \{t_1\}, t_x < 2 \rangle .x'$$

This has the same meaning as the one explained before.

6.2.4 As communicating finite state machines

Let define now a communicating finite state machine to represent the local behaviour: (Q, K, q_0, A, C, T), where Q is a finite set of states, K a finite set of channels, q_0 a initial state, C a finite state of clocks and T a finite set of transitions. There is a transition for each interaction, ie a message receiving or a message sending. These transitions carry the time information as constraints and clocks resetting, ie T is included in Q x (K x {!,?} x A x 2^C x Φ(C)) x Q. It extends the communicating finite state machine exposed in a previous section.

For instance, the two ommunicating finite state machines corresponding to our previous example are in Figure 6.2 and 6.3.

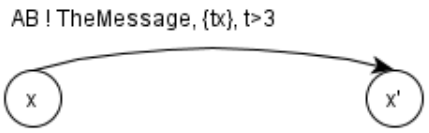


Figure 6.2: From the point of view of A

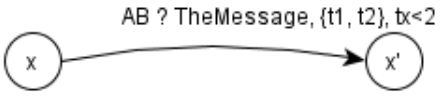


Figure 6.3: From the point of view of B

As for timed automata,we are interested in timed words (σ, τ_O, τ_I), where $\tau = \tau_1\tau_2...$ is a time sequence. We need to guarantee $\tau_{O_0} < \tau_{I_0} < \tau_{O_1} < \tau_{I_1} < \tau_{O_2}$ ... In our example, corresponding time sequences have only one character, as there is only one transition, for instance 3.2 and 4.5 work. In fact any real number strictly greater than three is possible for the first one and the second one should add to the first one no more than 2. The associated timed word is *(TheMessage, 3.2, 4.5)*.

At time τ_{O_i}, the transition $\langle x_{i-1}, k_i, !, l_i, \lambda_{O_i}, \delta_{O_i}, x_i \rangle$ is such that:

- $(v_{I_{i-1}} + \tau_{O_i} - \tau_{I_{i-1}})$ satisfies δ_{O_i}
- v_i equals $[\lambda_{O_i} \rightarrow 0](v_{I_{i-1}} + \tau_{O_i} - \tau_{I_{i-1}})$, which means that clocks in λ_{O_i} are reset to 0.

At time τ_{I_i}, the transition $\langle x_{i-1}, k_i, ?, l_i, \lambda_{I_i}, \delta_{I_i}, x_i \rangle$ is such that:

- $(v_{O_i} + \tau_{I_i} - \tau_{O_i})$ satisfies δ_{I_i}
- v_i equals $[\lambda_{I_i} \rightarrow 0](v_{O_i} + \tau_{I_i} - \tau_{O_i})$, which means that clocks in λ_{I_i} are reset to 0.

We used here some notations in [10]. In our example, v_0 maps t, t_1 to 0, τ_{O_0} and τ_{I_0} are 0 and τ_{O_1} is 3.2 and τ_{I_1} is 4.5. Then $v_{I_0} + \tau_{O_1} - \tau_{I_0}$ is 3.2 for every clocks. Therefore t > 3 is satisfied. In the same way, $v_{O_1} + \tau_{I_1} - \tau_{O_1}$ is 1.3 for t_x (it has been previously reset) and 4.5 for the other clocks. So $t_x < 2$ is also

satisfied.

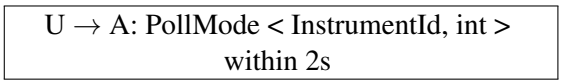
Therefore, we have defined all the syntax we need to match the graphs we want to be able to declare. We will now expose in more details how to express some features relative to time with these new notations.

6.3 Expressiveness of the syntax

We present here some examples to demonstrate the expressiveness of the syntax we have developed. At the starting point, all clocks are set to 0.

6.3.1 Timeout for message passing

We begin with a simple graph with three nodes: a start node, an end node and a message node as follows.



The complete graph is in Figure 2.5. For this graph, the corresponding Scribble global protocol is:

```
global protocol TestMessage (role A, role U) {
  PollMode(InstrumentId,int) from U to A within 2s;
}
```

To support this protocol we define the global type as follows:

G = def x = U → A : PollMode <InstrumentId, int> , $\{t_x\}, \varepsilon, \varnothing, t_x < 2$; x'
x' = end
in x

This means that we associate a unique, fresh clock, called t_x, to this message interaction and this clock will ensure the timeout for the message passing. Indeed, when U sends the message *PollMode*, U sets the clock t_x to 0. As A has also access to this clock, A can check that she receives the message *PollMode* with respect to the constraint $t_x < 2$. It means that A will not continue with x' if the timeout t_x is over. Therefore it garantees that the interaction is performed within two units of time and the whole structure models therefore the *within* statement from the global protocol in Scribble.

We also have the local types in the same way:

T_U = def x = $!\langle$ A, PollMode <InstrumentId, int> ,$\{t_x\}, \varepsilon\rangle$. x'
x' = end
in x

T_A = def x = $?\langle$ U,PollMode <InstrumentId, int> , $\varnothing, t_x < 2\rangle$. x'
x' = end
in x

6.3.2 Delay

We now add a delay before sending the message. First we look at the case of absolute time. The corresponding Scribble global protocol is:

```
global protocol TestAbsoluteDelay (role A, role U) {
  wait for t is 17 h
  PollMode(InstrumentId,int) from U to A within 2 s;
}
```

Figure 6.4 represents this protocol.

To support this protocol we define the global type as follows:

G = def x = U → A : PollMode <InstrumentId, int> , $\{t_x\}, t = 17, \varnothing, t_x < 2$; x'
x' = end
in x

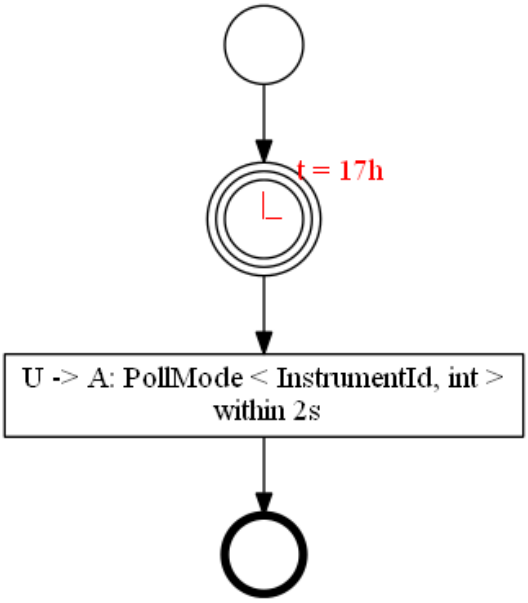


Figure 6.4: Representation of the case of an absolute delay

What is new from the previous example is the output constraint t = 17. This means that U has to wait for t to be 17h to be able to send the message *PollMode*. Therefore it models a delay.

We also have the local types in the same way:

T_U = def x = !⟨ A, PollMode <InstrumentId, int> ,{t_x}, t=17 ⟩ . x'
x' = end
in x

T_A = def x = ?⟨ U,PollMode <InstrumentId, int> , $\varnothing$,$t_x < 2$ ⟩ . x'
x' = end
in x

We now look at the case of relative time. The corresponding Scribble global protocol is:

```
global protocol TestRelativeDelay (role A, role U) {
  PollMode(InstrumentId,int) from U to A within 2s;
  wait for t + 3 h
  PollMode from A to I within 1s;
}
```

Figure 6.5 represents this protocol.

To support this protocol we define the global type as follows:

G = def x = U → A : PollMode <InstrumentId, int> , { t_x},ε,{t},$t_x < 2$; x'
x' = A → I : PollMode, { $t_{x'}$},$t = 3$,$\varnothing$,$t_{x'} < 1 \wedge t = 3$; x"
x" = end
in x

There are two differences against the previous example: (1) {t}, and (2) t=3. These mean that t has to be reset to 0 by A when receiving the first message. In this case t will play the role of a timer. Then A will have to check when t equals three hours. Only when this constraint will be also satisfied, A will be able to send the second message.

We also have the local types in the same way:

T_U = def x = !⟨ A, PollMode <InstrumentId, int> ,{ t_x}, ε⟩ . x'
x' = end
in x

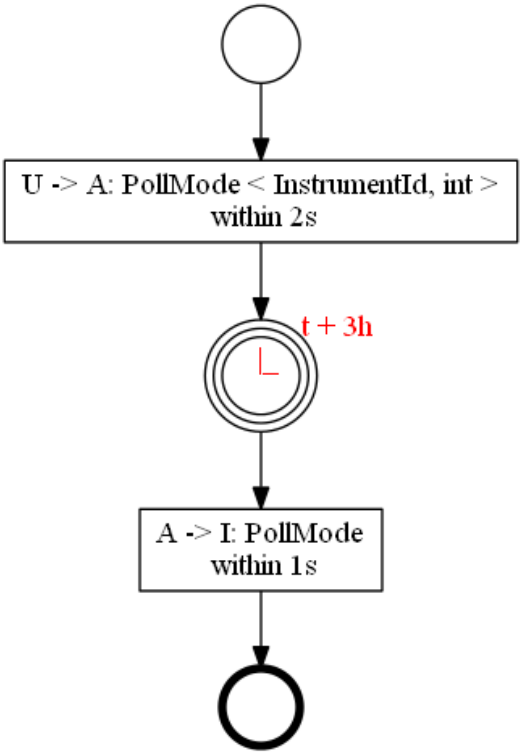


Figure 6.5: Representation of the case of a relative delay

$T_A =$ def x = ?⟨ U,PollMode <InstrumentId, int> , $\{t\}, t_x < 2$⟩ . x'
x' = !⟨ I, PollMode,{ $t_{x'}$ }, $t = 3$⟩ . x"
x" = end
in x

$T_I =$ def x' = ?⟨ A,PollMode, $\varnothing, t_{x'} < 1$⟩ . x"
x" = end
in x'

6.3.3 Time constraints

Finally we consider the case of a constraint on the arrow going to the Message box. The corresponding Scribble global protocol is:

```
global protocol TestConstraint (role A, role U) {
  t before 4 min
  PollMode(InstrumentId,int) from U to A within 2 s;
}
```

Figure 6.6 represents this protocol.

To support this protocol we define the global type as follows:

G = def x = U → A : PollMode <InstrumentId, int> , $\{t_x\}$, t < 4, $\varnothing, t_x < 2$; x'
x' = end
in x

What is new is t < 4. This means that the constraint t < 4 has to be satisfied in order for U to send the message *PollMode*.

We also have the local types in the same way:

$T_U =$ def x = !⟨ A, PollMode <InstrumentId, int> ,$\{t_x\}$, t < 4 ⟩ . x'
x' = end
in x

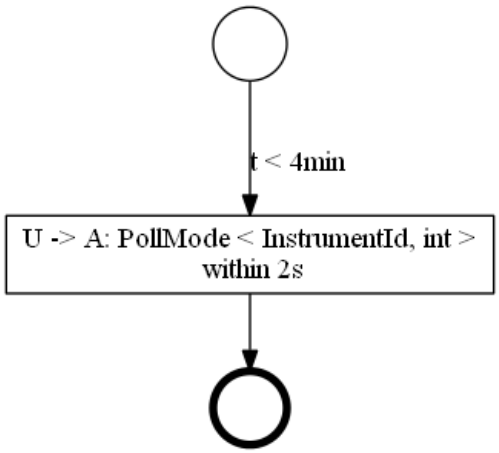


Figure 6.6: Representation of the case of a constraint

$T_A =$ def x = ?⟨ U,PollMode <InstrumentId, int> , $\varnothing, t_x < 2$ ⟩ . x'
x' = end
in x

6.4 Results

6.4.1 Well-formedness

Clocks constraints should be formed as stated in section 6.2.1. We suggest an additional condition that should be checked.

Temporal satisfiability *If* δ*, the clock condition of a given transition, is satisfiable at some point, for each constraint* δ*', appearing in a later transition, it is eventually possible to satisfy* δ*'.*

To demonstrate the need of this additional condition, we consider the following example.

```
global protocol TestTemporalSatisfiability (role A, role B, role C) {
  wait for t = 5s
  Msg1(no1) from A to B within 2 s;
  choice at B {
    t after 6s
    Msg2(no2) from B to C within 1s;
  } or {
    Msg3(no3) from B to C within 1s;
  }
}
```

Figure 6.7 represents this protocol.

To support this protocol we define the global type as follows:

G = def x_0 = A → B : Msg1 <nat> , $\{t_{x_0}\}$, t = 5, $\varnothing, t_{x_0} < 2$; x_1
$x_1 = x_2 + x_3$
x_2 = B → C : Msg2 <nat> , $\{t_{x_2}\}$, t < 6, $\varnothing, t_{x_2} < 1$; x_4
x_3 = B → C : Msg3 <nat> , $\{t_{x_3}\}$, ε, $\varnothing, t_{x_3} < 1$; x_5
$x_4 + x_5 = x_6$
x_6 = end
in x_0

This means that when A send the message *Msg1* to B, t equals five seconds. Then the message should be received within two seconds. Therefore if the condition $\delta = t_{x_0} < 2$ is satisfied, this means also that t < 7 is satisfied. In this case the condition δ' = t > 6 can be satisfied at some point: either it is already satisfied because we could have 6 < t < 7, or it is not yet satisfied, but will eventually be satisfied when time goes on. The latter case could be assimilate to a delay, but it is not exactly the same. In the case of a delay, we know

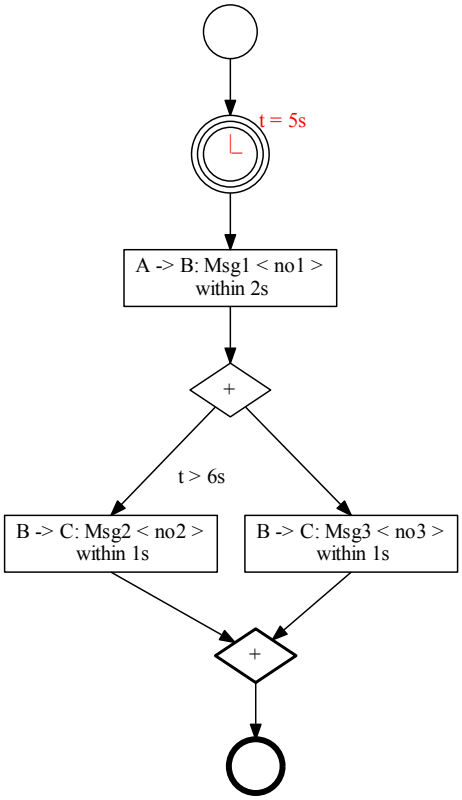


Figure 6.7: Example of temporal satisfiability

exactly the time when the delay has been achieved, whereas in this case, the message *Msg2* can be sent whenever t is greater than six seconds. Therefore this example satisfies the temporal satisfiability condition.

On the other hand, if we consider the following global protocol in Scribble:

```
global protocol TestTemporalSatisfiability (role A, role B, role C) {
  wait for t = 5s
  Msg1(no1) from A to B within 2 s;
  choice at B {
    t before 6s
    Msg2(no2) from B to C within 1s;
  } or {
    t after 6s
    Msg3(no3) from B to C within 1s;
  } or {
    Msg4(no4) from B to C within 1s;
  }
}
```

we do not verify the temporal satisfiability condition, as there is the case when $\delta = t_{x_0} < 2$ is satisfied, and also $t < 7$, but $\delta' = t < 6$ is not satisfiable: for instance when $t = 6.5$ after the receiving of the message *Msg1*. Such a situation means that the choice could be done only by temporal conditions, as it is not the case in our model. We state that the choice is done by a participant and all timed branches should be enabled at some point.

We conjecture that a global type G = def $\tilde{G}$ in x_0 is *well-formed*, if it satisfies the sanity, local choice,

linearity conditions[1] and clock constraints the temporal satisfiability conditions. It is worth noting, that we may revise the local choice condition for global protocols in order to include the case when the choice only depends on time conditions.

6.4.2 Projection

The projection from a well-formed global protocol to local protocols, written G$\upharpoonright$ p, is straightforward from the syntax. We give here the algorithm:

$$
\begin{array}{rcl}
\text{def } \tilde{G} \text{ in x} \upharpoonright \text{p} & = & \text{def } \tilde{G} \upharpoonright_{\tilde{G}} \text{p in x} \\
\text{x = p} \to p' : l\langle U\rangle, \lambda_O, \delta_O, \lambda_I, \delta_I; x' \upharpoonright_{\tilde{G}} \text{p} & = & \text{x} = !\langle p', l\langle U\rangle, \lambda_O, \delta_O\rangle.x' \\
\text{x = p} \to p' : l\langle U\rangle, \lambda_O, \delta_O, \lambda_I, \delta_I; x' \upharpoonright_{\tilde{G}} \text{p'} & = & \text{x} = ?\langle p, l\langle U\rangle, \lambda_I, \delta_I\rangle.x' \\
\text{x = p} \to p' : l\langle U\rangle, \lambda_O, \delta_O, \lambda_I, \delta_I; x' \upharpoonright_{\tilde{G}} \text{p''} & = & \text{x = x' (p''} \notin \{\text{p,p'}\}) \\
\text{x = x' | x''} \upharpoonright_{\tilde{G}} \text{p} & = & \text{x = x' | x''} \\
\text{x = x' + x''} \upharpoonright_{\tilde{G}} \text{p} & = & \text{x = x'} \oplus \text{x'' (if p = ASend}(\tilde{G})(\text{x})) \\
\text{x = x' + x''} \upharpoonright_{\tilde{G}} \text{p} & = & \text{x = x' \& x'' (otherwise)} \\
\text{x | x' = x''} \upharpoonright_{\tilde{G}} \text{p} & = & \text{x | x' = x''} \\
\text{x + x' = x''} \upharpoonright_{\tilde{G}} \text{p} & = & \text{x + x' = x''} \\
\text{x = end} \upharpoonright_{\tilde{G}} \text{p} & = & \text{x = end}
\end{array}
$$

Thanks to the simplicity of the projection, we have:
Proposition (Projection). *Given a well-formed G, the computation of G$\upharpoonright$ p is linear in the size of G.*

6.4.3 Properties and conjectures

We conjecture some more properties about projection: coherence, progress and deadlock-freedom. Starting from untimed typing judgements, we would like to establish a timed simulation to prove these properties. For this purpose, we will need to use the Pi-calculus and in particular processes syntax that we have established in Table 6.1.

As for global types, a process starts with the definition def $\tilde{P}$ in x($\tilde{v}$) and the actions follow the same syntax. On one hand, for the definition, the parameters in x will instantiate the ones in P. On the other hand, for process actions, variables $\tilde{x}$ occurring on the left-hand side are binding variables on the right-hand side.

A session is initialised by a transition of the form x($\tilde{x}$) = x$\langle G, C\rangle$.x'($\tilde{e}$) where G is a global type and C the set of clocks used in G. It attributes a global interaction pattern G with its set of clocks to the shared channel a that x gets substituted to. The variables in $\tilde{e}$ are all bound by $\tilde{x}$. After a session initialisation, participants can accept the session with x($\tilde{x}$) = x[p](y): x'($\tilde{e}$), starting the interactions: the variables in $\tilde{e}$ are bound by $\tilde{x}$ and by y, which, at run-time, receives the session channel.

The sending action allows resetting clocks in λ and sending to p in session x a value e labelled by a constant l if the constraint δ is satisfied. The reception expects from p a message with a label l, if the condition δ is satisfied, as well as resets the clocks in λ. Other actions are similar to the ones in [17].

If we want to allow cases where the choice depends only on time, we would have to change the temporal satisfiability condition, as well as the definition for expressions to:

$$e ::= \text{v} \mid \text{x} \mid \delta \mid \text{e} \wedge \text{e} \mid \ldots$$

From this syntax, we can develop the operational semantics for processes, using the labels α and β, and then define judgements. This may enable to prove theorems about error-freedom and completeness.

[1] These conditions are defined in [17]

P	::=	def $\tilde{P}$ in X	definition
P	::=		definition
	|	x($\tilde{x}$) = x$\langle G, C\rangle$.x'($\tilde{e}$)	init
	|	x($\tilde{x}$) = x[p](y): x'($\tilde{e}$)	accept
	|	x($\tilde{x}$) = x!$\langle p; l < e >, \lambda_O, \delta_O\rangle$: x'($\tilde{e}$)	send
	|	x($\tilde{x}$) = x?$\langle p; l(y), \lambda_I, \delta_I\rangle$: x'($\tilde{e}$)	receive
	|	x($\tilde{x}$) = x'($\tilde{y}$) | x''($\tilde{z}$)	parallel
	|	x($\tilde{x}$) | x'($\tilde{y}$) = x''($\tilde{z}$)	join
	|	x($\tilde{x}$)+ x'($\tilde{x}$) = x''($\tilde{x}$)	merge
	|	x($\tilde{x}$) = x'($\tilde{x}$) & x''($\tilde{x}$)	external choice
	|	if e then $x'(\tilde{e}')$ else $x''(\tilde{e}'')$	conditional
	|	x($\tilde{x}$) = 0	null
	|	x($\tilde{x}$) = $(\nu a)x'(a\tilde{x})$	new name
X	::=		state
	|	x($\tilde{v}$)	thread
	|	X | X	parallel
	|	x$(\nu a)X$	restriction
	|	0	null
e	::=	v | x | e $\wedge$ δ | e $\wedge$ e | ...	expressions
v	::=	a | s[p] | true | false | ...	values
α, β	::=	s[p,q]!$\langle p; l < e >, \lambda, \delta\rangle$	labels
	|	s[p,q]?$\langle p; l(y), \lambda, \delta\rangle$	
	|	a$\langle G, C\rangle$	
	|	a$\langle p\rangle$ [s]	
	|	$\langle \tau, \lambda, \delta\rangle$	

Table 6.1: Syntax for timed processes

Chapter 7

Evaluation

As we have described so far our work, we now present an evaluation of it. The major part of our project concerns the implementation of a graphic tool as introduced in chapter 3, that is why we will focus our evaluation on this aspect.

7.1 Method

In order to evaluate our implementation about the correspondence between global protocols in Scribble and the graph representation, we have tested several scenarii and checked the conformance of the result. We are interested in well-formed programs in Scribble and well-formed graphs: we assume indeed this well-formedness criterion.

Besides the basic cases exposed in previous chapters (FirstMessage, FirstChoice, FirstParallel, FirstRecursion and FirstDelay), we have considered several protocols with complex constructs. We have selected scenarii involving merge, join and recursion constructs. They are indeed the ones we have to pay particular attention to. On one hand, for merge and join constructs, the issue is that they are explicit represented on the graph, but not in the Scribble language. On the other hand, for recursions, the issue is the cycle it introduces: it adds a difficulty for graphs structures. We are in particular interested in nested recursion inside another recursion. For this purpose we assume that all the labels for recursion are distinct.

We present the scenarii by giving both the protocol and the graph, and for each of them we explain our expectations. In the last section, we summarize our results.

7.2 Test cases

We present in this section only the complex cases. The basic ones can be find in previous chapters.

7.2.1 Parallel and Recursion

We first consider the case of a recursion inside one thread of a parallel construct. We want to check that there will be no join node on the graph. This thread will indeed never terminate, so it could make the two other ones waiting for ever if we join them. Such a global protocol could be:

```
global protocol TestParallelAndRecursion ( role A, role B, role C, role D ) {
  parallel{
    Msg1 from A to B within 1s;
  } and {
    rec Loop {
      Msg from A to C within 1s;
      continue Loop;
    }
  } and {
    Msg2 from A to D within 1s;
  }
}
```

Figure 7.1 is obtained from this global protocol.

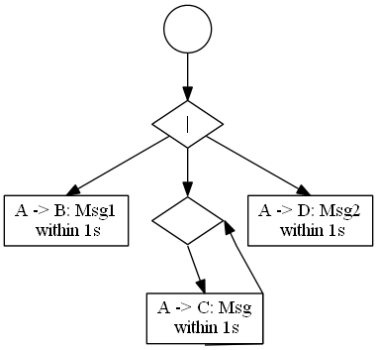


Figure 7.1: Representation of a recursion inside a parallel construct

7.2.2 Choice and Recursion

We now consider the case of a recursion inside one branch of a choice construct. We want to check that there will be a merge node on the graph that merges the branches which have no recursion inside. One of these branches will indeed terminate, so we could want to continue with other interactions. Such a global protocol could be:

```
global protocol TestChoiceAndRecursion ( role A, role B, role C, role D ) {
  choice at A {
    Msg1 from A to B within 3h;
  } or {
  rec Loop {
    Msg from A to C within 2min;
    continue Loop;
    }
  }  or {
    Msg2 from A to D within 5min;
  }
}
```

Figure 7.2 is obtained from this global protocol.

7.2.3 Recursion and Choice

We consider here the case of a choice inside a recursion, where some branches have a continue construct and others do not have one. We want to check that there will be a merge node on the graph that merges the branches which have no continue inside. One of these branches will indeed terminate, so we could want to have other interactions. We also want to check that the recursion appears properly on the graph. Such a global protocol could be:

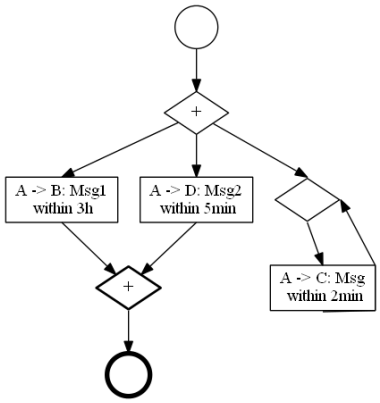


Figure 7.2: Representation of a recursion inside a choice construct

```
global protocol TestRecursionAndChoice ( role A, role B, role C, role D ) {
  rec Loop {
    choice at A {
       Msg1 from A to B within 3h;
    } or {
       Msg from A to C within 2min;
      continue Loop;
    }  or {
      Msg2 from A to D within 5min;
    }
  }
}
```

Figure 7.3 is obtained from this global protocol.

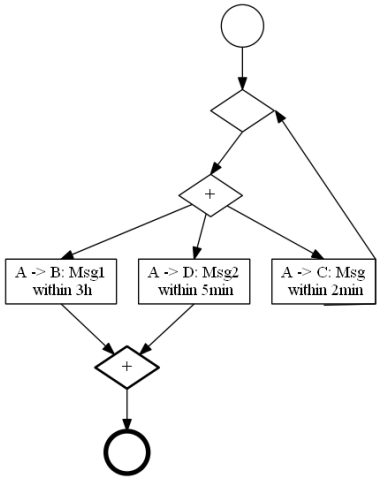


Figure 7.3: Representation of a continue inside only one branch

7.2.4 Double Recursion and Choice

We consider finally the same case as the previous with a nested recursion inside the choice construct. Therefore we have two different recursions. We want to check that there will be a merge node on the graph for the branches which have no continue inside. We also want to check that the recursion appears properly on the graph and that there is no mismatch between the two recursion. Such a global protocol could be:

```
global protocol TestDoubleRecursionAndChoice ( role A, role B, role C, role D ) {
 rec Loop {
  choice at A {
    Msg1 from A to B within 3h;
    } or {
      Msg from A to C within 2min;
           continue Loop;
    } or {
    Msg2 from A to D within 5min;
    rec Loop2{
      Msg3 from D to A within 1s;
      continue Loop2;
      }
    }
  }
}
```

Figure 7.4 is obtained from this global protocol.

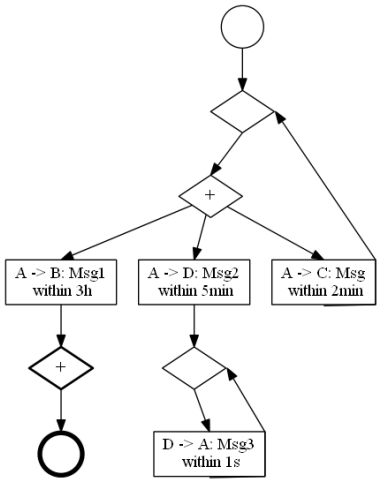


Figure 7.4: Representation of a nested recursion inside a choice

7.3 Result

We have tested these cases in both ways: this means from the global protocol in Scribble we obtained the graph, and from the graph we obtained the global protocol in Scribble. We were therefore able to check the conformance of the representations. We were also interested in the design of the graph.

The features we wanted to check about merge, join and recursion, in particular in nested constructs, as specified for each case in the previous section, were all satisfied. We have found neither errors, nor mismatches for well-formed protocols.

For each case we have also checked the layout of the graph. We want to ensure that the layout, automatically managed by the Dot language, is clear enough. We have established several criteria for this purpose:

- General impression,
- Order of the nodes,
- Clarity of the labels.

In case of a missing character in the program in Scribble, for instance in this protocol,

```
global protocol TestMessage ( role A, role U) {
 PollMode(InstrumentId,int) from U to A within 2s
}
```

we even obtain an error message like this: `line 3:0 missing ';' at u'}'`.

We are now ensured that every well-formed global protocol can match a well-formed graph, and vice-versa. Besides large expressiveness, we satify the criteria we aimed at:

- Simplicity,
- Readability,
- Clear correspondence between graphs and global protocols in Scribble

We conclude that our implementations conforms our expectations.

Conclusion

Achievements

We present here a summary of what we have achieved during the project.

We aimed at developing a graph representation for global protocols in the context of multiparty sessions. Therefore we have conducted research to analyse precisely the needs relative to such a representation and the state of the art in this field. Besides several ways of representation, we studied the theory about session types from binary session types to generalised multiparty session types. Based on this analysis we were able to propose graphical notations to solve our issue. The principles underpinning our choice for the graphical objects were as follows: (1) simplicity, (2) readability and (3) intuitive understanding. We have then performed the implementation of a tool to establish a correspondence between graphs, using these earlier mentioned notations, and global protocols in Scribble, as they are currently defined. It is now possible to obtain a graph representing a given global protocol in Scribble and also to obtain a global protocol in Scribble from a given graph. We assumed for both ways that the input is well-formed.

Furthermore, in order to make our contribution to the session theory, we proceeded with an extension relative to global protocols. Several significant features were still to be encompassed in the theory, such as timeouts. Therefore we chose to handle time in our graphs and consequently in protocols. For this purpose, we draw our inspiration from timed automata. As well as timeouts for message passing, we introduced delays and other time constraints in our work in order to gain in expressiveness. Along with the design of the graphical notations, we performed the extension of our previous implementation to support these changes and studied some well-formedness properties we could state for this new approach.

Consequently we have achieved to obtain a correspondence between graphs, representing global protocols of complex interactions between several processes within a time scheme, and these protocols in the Scribble language.

The challenging parts of the project

Implementation side

As far as the implementation is concerned, the project was quite challenging. We chose to develop our tool in Python for coherence reasons as it is the language currently used for similar work in the team. It was a good opportunity to learn this language, widely-used by engineers in industry. Already knowing the Java language, the first basics were easy to assimilate. What was more difficult was to use ANTLRWorks with Python. Understanding the principles underlying a project with ANTLR was necessary to then try to work with it using the Python language. By default, ANTLRWorks uses Java as programming language, i.e. the files are generating as Java files. Then, ANTLR was extended to support also Python. However the latest version of ANTLRWorks does not support Python yet, but it supports an interesting run tool: it is possible with an input file to obtain the representation of both the parsing tree and the Abstract Syntax Tree. Therefore, we can easily follow and debug our work. To solve this issue, we used two different versions of ANTLRWorks: one for debugging and testing, the other one for generating Python files.

Furthermore, to gain flexibility in the choices for the implementation, we have only developed a few lines of code within ANTLRWorks. The remaining was performed directly in an Eclipse environment after the generation of the files with ANTLRWorks.

Theory side

As far as theory is concerned, the first challenge was to understand session types. The basics in Pi-calculus from the Model of Concurrent Computation course were helpful to understand the common communication features of sessions. Then we focused on the notion of time in several linked pieces of work: finite state machines, automata, extended Pi-calculus, Petri-nets, etc. This means a lot of background reading to understand the purpose of the different papers. These deep studies were indeed needed in order to be able to analyse and compare these different approaches.

Possible improvements

The first point we can make here, linked to the previous section, is about the choice of the programming language. As we have encountered difficulties in using Python with ANTLRWorks, we could improve the work by using Java. It allows indeed simplifying the use of ANTLR. However, we have to balance this point with the reasons of the choice of Python: (1) other parts of the software tool chain are already in Python, and (2) Python is widely-used in the engineering community. Another alternative could be to develop our own parser directly in an Eclipse environment. Nevertheless we will not have the testing tool provided by ANTLRWorks.

Another point concerns the design of the graph. Work can be done on creating a user interface for the drawing. It will make the design of the graph more interactive. Then we meet another issue, which is the choice of the library for the graphical notations. We chose to use a library in Python for graphic objects that was not specially developed for drawing graphs for communication protocols, whereas BPMN notations were. The latter allows using more complex pictograms. Therefore we could use BPMN tools for the drawing and generate XML scripts for BPMN choreographies rather than Pydot object. For the other way of the correspondence we could define a parser to obtain a protocol from the script of a choreography. However, this will mean to lose the simplicity and readability we aimed at, unless we define some restrictions in the use of BPMN notations. A better solution would be to create a new diagram, called global protocol, that uses our notations with the BPMN design.

To continue improve our work, we should also consider more use cases from the reality. It allows indeed checking whether our notations can express every well-formed global protocol and be used by engineers.

Future work

From what we have achieved during this project, several pieces of work can be performed.

As far as graphs are concerned, we have assumed that the graphs written by the user were well-formed. So we could develop a tool to check the well-formedness of graphs before proceeding with generating protocols in Scribble. The verification will be done against the well-formedness conditions we defined. Furthermore, we have not included some features defined in Scribble in our work, in particular delegation. This means that first we have to find a suitable graphical notation and then extend the implementation.

As far as Scribble is concerned, future work can be done on the expressiveness of Scribble in order to have a real correspondence with generalised multiparty session types. Some situations are indeed expressible with generalised multiparty session types but not in Scribble as merge and join are supported by the former: for instance the merge of two out of three branches in a choice construction. This part will be quite challenging as we need to keep the simplicity and readability of the language Scribble.

Furthermore, to follow the idea of a tool chain from graph to end-point programs, we could merge this project with another one, as a collaboration work. Some work is indeed currently done to create an Eclipse plugin for the Scribble development. Given a global protocol, it type checks it, rewrites it in a prettier way, with good indentation for instance, if needed, and generates the local protocols by projection. We could easily imagine inserting our work to add the possibility of giving either a global protocol or a graph representation of it. In this way, we get, as well as the local protocols, the global one and its graph representation.

Finally, as we have begun some work on timed global protocol, we conjectured some results that are still to prove: temporal satisfiability condition, deadlock freedom and progress. Our work raised several

issues about properties that need to be cleared. We need also to define the way to implement clocks. For the moment, we assumed that clocks were global information, this means that every process can have access to every clocks defined in the protocol. This allows avoiding problems about asynchrony of time. Nevertheless we may want to define also local clocks, or restricted clocks. Furthermore, we have to change the monitor to work with time. For this purpose, the major issue will be to be able to check time. One solution could be to add clocks in the queue. Work can therefore be performed to formally describe everything and make it sound.

Including our work in the tool chain that generates the end-point programs from a global scenario is indeed the major application of our work. It will also contribute to other applications handling global protocol, in particular future work on timed global protocols.

Concluding remarks

The project was a good opportunity to experience how to conduct research. After two terms of lectures, it allows applying our knowledge and explore more deeply an area of computing. Academic research aims not only at trying new concepts and exploring unknown subjects, but also at developing tools that will have a real application in the engineering world. It was indeed the case for our project: by working on a new way of representing global protocols, our goal was to obtain a tool that will be widely used by engineers when designing the communication features of physical systems.

Bibliography

[1] BPMN 2.0 notations. http://www.bpmb.de/images/BPMN2_0_Poster_EN.pdf.

[2] Business Process Model Notation. http://www.bpmn.org/.

[3] Graphviz, Graph Visualization Software. http://www.graphviz.org/.

[4] Pydot, Python interface to Graphviz's Dot language. http://code.google.com/p/pydot/.

[5] 'safety assurance framework for distributed systems through multiparty session types.

[6] Scribble developpement tool site. http://www.jboss.org/scribble.

[7] Scribble Language Reference. http://www.doc.ic.ac.uk/~rhu/scribble/langref.html.

[8] W3C - Web Services Choreography Working Group. http://www.w3.org/2002/ws/chor/.

[9] R. Alur. Timed automata. In *Computer Aided Verification*, volume 1633 of *Lecture Notes in Computer Science*, pages 688–688. Springer, 1999.

[10] R. Alur and D.L. Dill. A theory of timed automata. *Theoretical computer science*, 126(2):183–235, 1994.

[11] J. Bengtsson and W. Yi. Timed automata: Semantics, algorithms and tools. *Lectures on Concurrency and Petri Nets*, 3098:87–124, 2004.

[12] M. Berger. *Towards Abstractions for Distributed Systems*. PhD thesis, Imperial College, Deptartment of Computing, 2002.

[13] M. Berger and N. Yoshida. Timed, Distributed, Probabilistic, Typed Processes. In *Proceedings of the 5th Asian conference on Programming languages and systems*, volume 4807 of *Lecture Notes in Computer Science*, pages 158–174. Springer-Verlag, 2007.

[14] L. Bettini, M. Coppo, L. D'Antoni, M. De Luca, M. Dezani-Ciancaglini, and N. Yoshida. Global Progress in Dynamically Interleaved Multiparty Sessions. 5201:418–433, 2008.

[15] L. Bocchi, K. Honda, E. Tuosto, and N. Yoshida. A theory of design-by-contract for distributed multiparty interactions. *CONCUR 2010-Concurrency Theory*, 6269:162–176, 2010.

[16] G. Ciobanu and C. Prisacariu. Timers for distributed systems. *Electronic Notes in Theoretical Computer Science*, 164(3):81–99, 2006.

[17] P.M. Deniélou and N. Yoshida. Multiparty Session Types Meet Communicating Automata. In *ESOP*, volume 7211 of *Lecture Notes in Computer Science*, pages 194–213, 2012.

[18] J. Desel and J. Esparza. *Free choice Petri nets*, volume 40. Cambridge Univ Pr, 1995.

[19] A.S. Henriksen, L. Nielsen, T.T. Hildebrandt, N. Yoshida, and F. Henglein. Trustworthy Pervasive Healthcare Services via Multiparty Session Types. To appear, 2012.

[20] K. Honda, A. Mukhamedov, G. Brown, T.C. Chen, and N. Yoshida. Scribbling interactions with a formal foundation. volume 6536 of *Lecture Notes in Computer Science*, pages 55–75. Springer, 2011.

[21] K. Honda, V. Vasconcelos, and M. Kubo. Language primitives and type discipline for structured communication-based programming. volume 1381 of *Lecture Notes in Computer Science*, pages 122–138. Springer, 1998.

[22] R. Hu, N. Yoshida, and K. Honda. Session-Based Distributed Programming in Java. volume 5142 of *Lecture Notes in Computer Science*, pages 516–541. Springer, 2008.

[23] Tom Hvitved. *Contract Formalisation and Modular Implementation of Domain-Specific Languages*. PhD thesis, Department of Computer Science, University of Copenhagen, 2012.

[24] Tom Hvitved, Felix Klaedtke, and Eugen Zălinescu. A trace-based model for multiparty contracts. *Journal of Logic and Algebraic Programming*, 81(2):72–98, 2012. Formal Languages and Analysis of Contract-Oriented Software (FLACOS'10).

[25] J.Y. Lee and J. Zic. On modeling real-time mobile processes. In *Australian Computer Science Communications*, volume 24, pages 139–147. Australian Computer Society, Inc., 2002.

[26] H.A. López, C. Olarte, and J.A. Pérez. Towards a unified framework for declarative structured communications. *Programming Language Approaches to Concurrency and CommunicationcEntric Software (PLACES'2009), EPTCS*, 17:1–15, 2010.

[27] F. Maggi, M. Montali, M. Westergaard, and W. van der Aalst. Monitoring business constraints with linear temporal logic: an approach based on colored automata. volume 6896 of *Lecture Notes in Computer Science*, pages 132–147. Springer, 2011.

[28] N. Ng, N. Yoshida, and K. Honda. Multiparty Session C: Safe Parallel Programming with Message Optimisation. In *TOOLS (50)*, volume 7304 of *Lecture Notes in Computer Science*, pages 202–218. Springer, 2012.

[29] N. Saeedloei and G. Gupta. Timed Pi-calculus. *University of Texas at Dallas technical report*, 2008.

[30] W.M.P. Van der Aalst et al. The application of Petri nets to workflow management. *Journal of Circuits Systems and Computers*, 8:21–66, 1998.

[31] W. VANDERAALST. *Timed Coloured Petri Nets and their Application to Logistics*. PhD thesis, Technische Universiteit Eindhoven.

[32] N. Yoshida and V.T. Vasconcelos. Language Primitives and Type Discipline for Structured Communication-Based Programming Revisited: Two Systems for Higher-Order Session Communication. *Electronic Notes in Theoretical Computer Science*, 171(4):73–93, 2007.

List of Figures

List of Tables

Appendix A

Details of the examples exposed in the report

We give here the details of some basic examples presented in the report. We give Abstract Syntax Trees obtained from global protocols and the Python code to obtain the graph in the other way.

A.1 Message

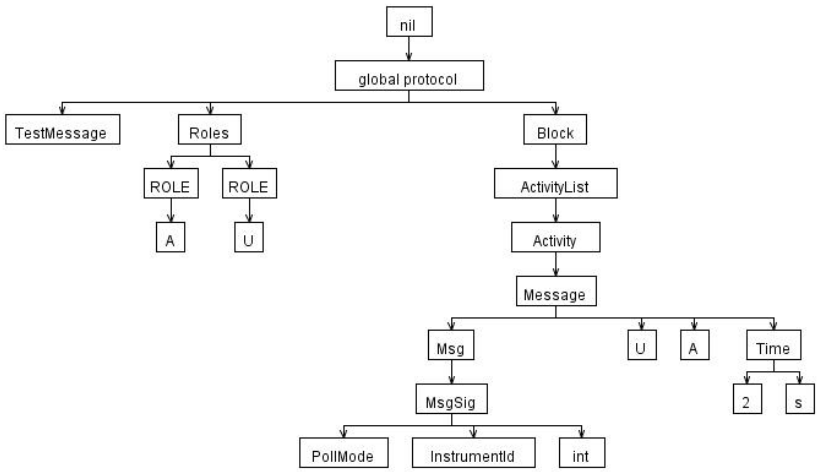


Figure A.1: Abstract Syntax Tree for the message case

```
start = extensions.Start()
msg = extensions.Message(1,"U","A","PollMode < InstrumentId, int >",2)
end = extensions.End()
graph.add_Snode(start)
graph.add_Snode(msg)
graph.add_Snode(end)
ed = extensions.SEdge(start,msg)
ed2 = extensions.SEdge(msg,end)
graph.add_Sedge(ed)
graph.add_Sedge(ed2)
```

Code in Python

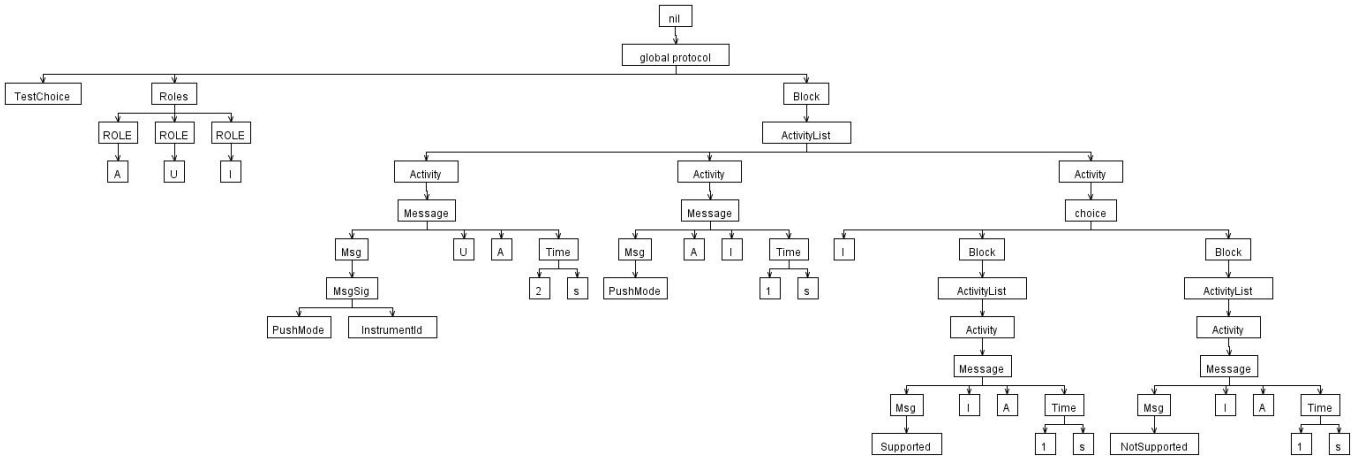


Figure A.2: Abstract Syntax Tree for the choice case

A.2 Choice

```
start = extensions.Start()
msg = extensions.Message(1,"U","A","PushMode < InstrumentId >",2)
msg2 = extensions.Message(2,"A","I","PushMode",1)
choice = extensions.Choice(5,"I")
msg3 = extensions.Message(3,"I","A","Supported",1)
msg4 = extensions.Message(4,"I","A","NotSupported",1)
merge = extensions.Merge(6)
end = extensions.End()
graph.add_Snode(start)
graph.add_Snode(msg)
graph.add_Snode(msg2)
graph.add_Snode(choice)
graph.add_Snode(msg3)
graph.add_Snode(msg4)
graph.add_Snode(merge)
graph.add_Snode(end)
ed = extensions.SEdge(start,msg)
ed3 = extensions.SEdge(msg,msg2)
ed4 = extensions.SEdge(msg2,choice)
ed5 = extensions.SEdge(choice,msg3)
ed6 = extensions.SEdge(choice,msg4)
ed7 = extensions.SEdge(msg3,merge)
ed8 = extensions.SEdge(msg4,merge)
ed2 = extensions.SEdge(merge,end)
graph.add_Sedge(ed)
graph.add_Sedge(ed3)
graph.add_Sedge(ed4)
graph.add_Sedge(ed5)
graph.add_Sedge(ed6)
graph.add_Sedge(ed7)
graph.add_Sedge(ed8)
graph.add_Sedge(ed2)
```

Code in Python

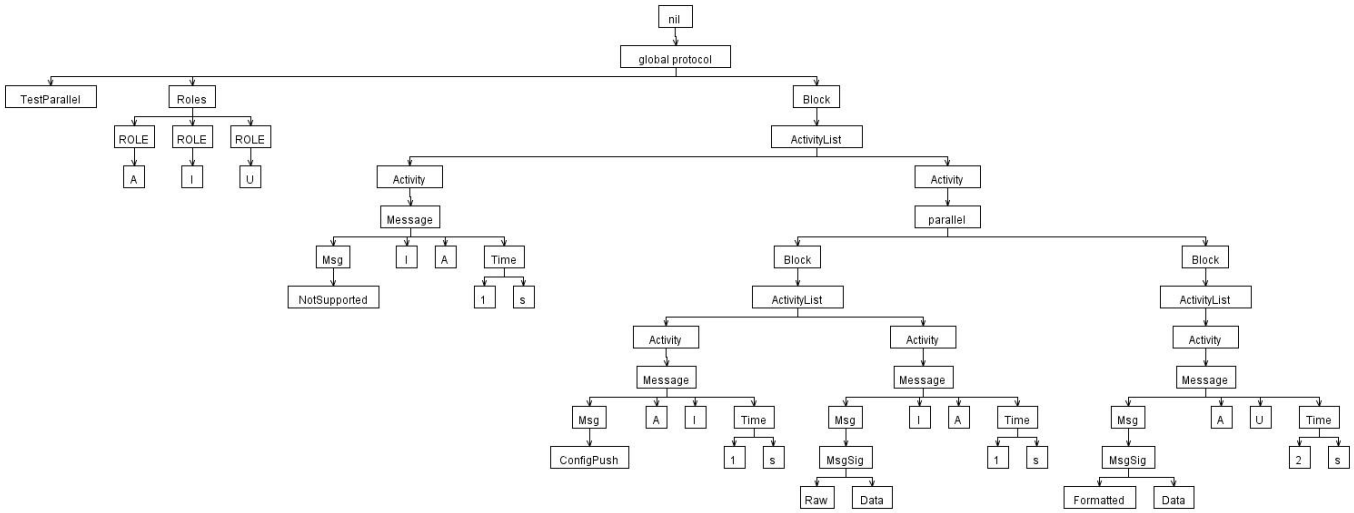


Figure A.3: Abstract Syntax Tree for the parallel case

A.3 Parallel

```
start = extensions.Start()
msg = extensions.Message(1,"I","A","NotSupported",1)
msg2 = extensions.Message(2,"A","I","ConfigPush",1)
fork = extensions.Fork(5)
msg3 = extensions.Message(3,"I","A","Raw < Data >",1)
msg4 = extensions.Message(4,"A","U","Formatted < Data >",2)
join = extensions.Join(6)
end = extensions.End()
graph.add_Snode(start)
graph.add_Snode(msg)
graph.add_Snode(msg2)
graph.add_Snode(fork)
graph.add_Snode(msg3)
graph.add_Snode(msg4)
graph.add_Snode(join)
graph.add_Snode(end)
ed = extensions.SEdge(start,msg)
ed3 = extensions.SEdge(msg,fork)
ed4 = extensions.SEdge(fork,msg2)
ed5 = extensions.SEdge(msg2,msg3)
ed6 = extensions.SEdge(fork,msg4)
ed7 = extensions.SEdge(msg3,join)
ed8 = extensions.SEdge(msg4,join)
ed2 = extensions.SEdge(join,end)
graph.add_Sedge(ed)
graph.add_Sedge(ed3)
graph.add_Sedge(ed4)
graph.add_Sedge(ed5)
graph.add_Sedge(ed6)
graph.add_Sedge(ed7)
graph.add_Sedge(ed8)
graph.add_Sedge(ed2)
```

Code in Python

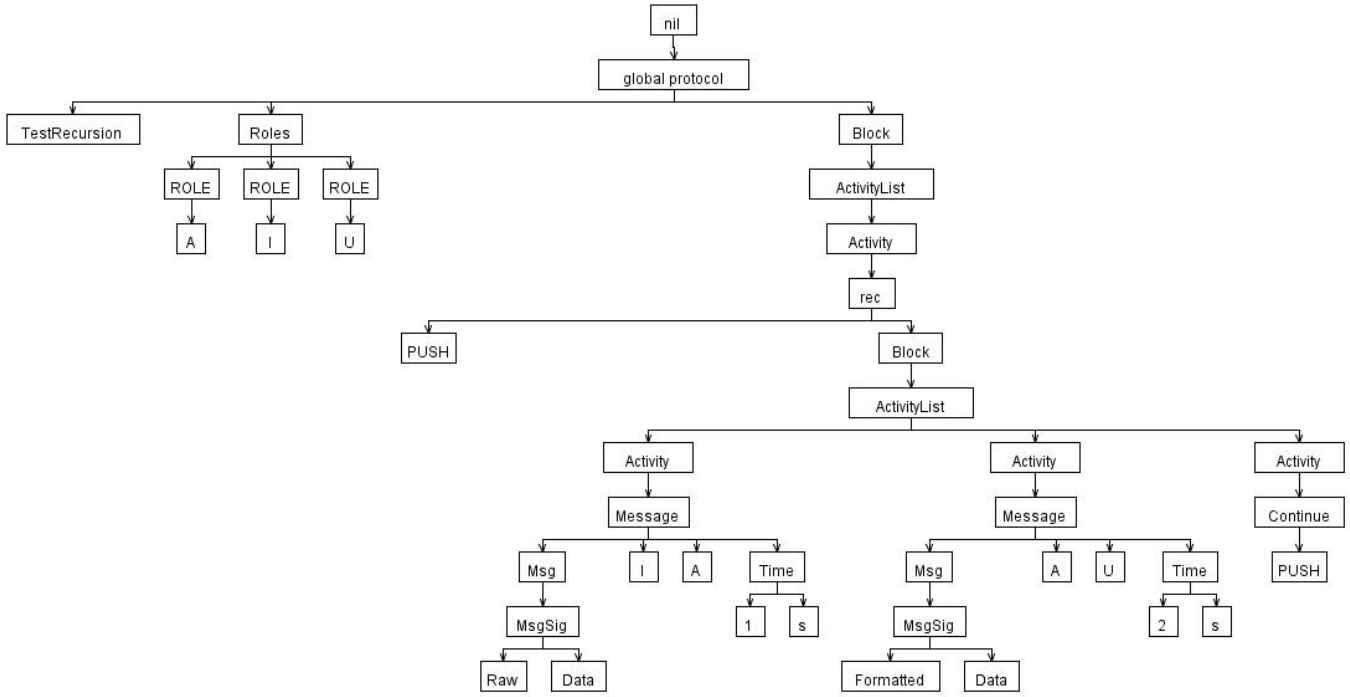


Figure A.4: Abstract Syntax Tree for the recursion case

A.4 Recursion

```
start = extensions.Start()
msg = extensions.Message(1,"I","A","Raw < Data >",1)
msg2 = extensions.Message(2,"A","U","Formatted < Data >",2)
rec = extensions.Rec(3,"PUSH")
graph.add_Snode(start)
graph.add_Snode(msg)
graph.add_Snode(msg2)
graph.add_Snode(rec)
ed = extensions.SEdge(start,rec)
ed3 = extensions.SEdge(rec,msg)
ed4 = extensions.SEdge(msg,msg2)
ed2 = extensions.SEdge(msg2,rec)
graph.add_Sedge(ed3)
graph.add_Sedge_with_ports(ed,'n','s')
graph.add_Sedge(ed4)
graph.add_Sedge_with_ports(ed2,'e','s')
```

Code in Python

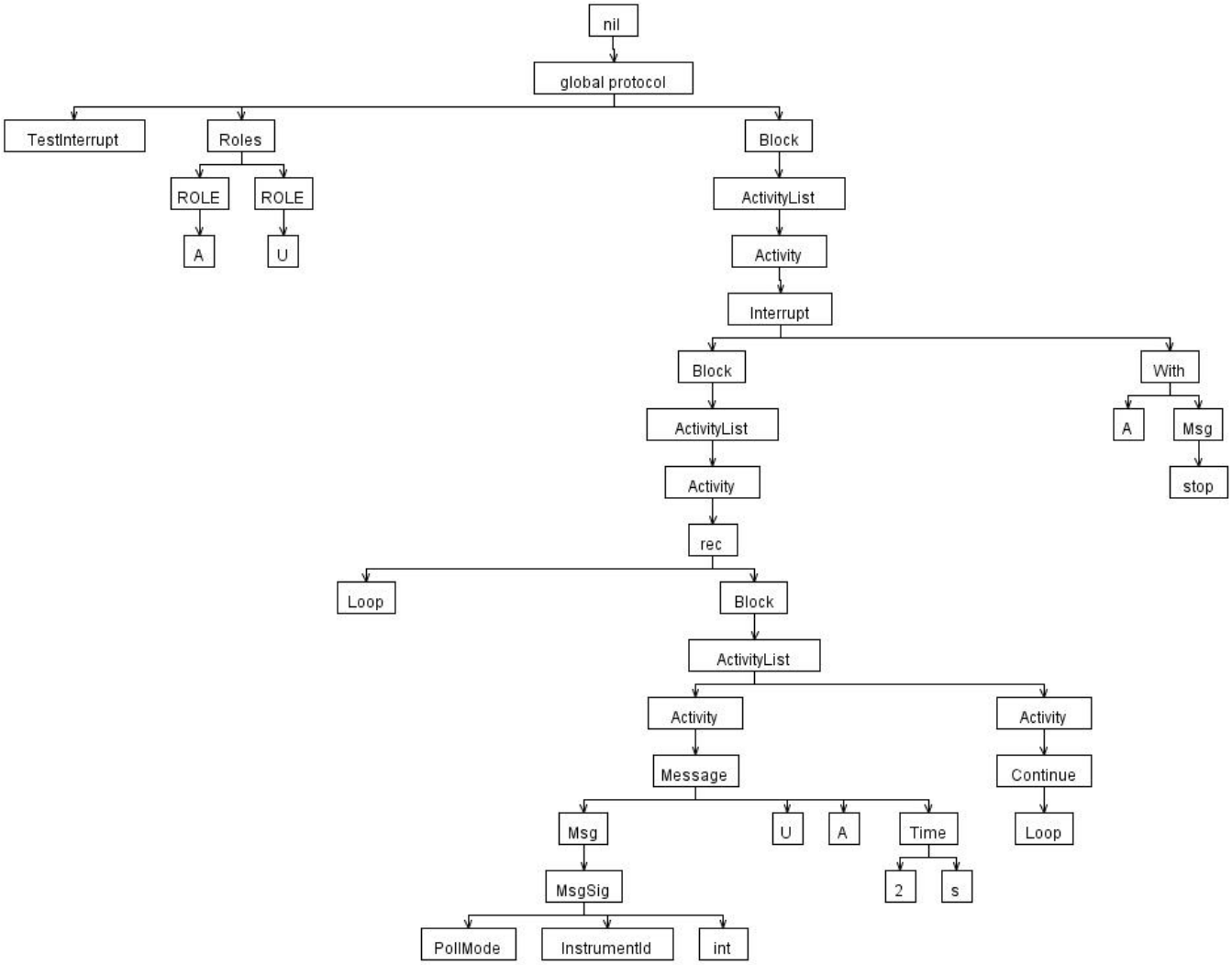


Figure A.5: Abstract Syntax Tree for the interruption case

A.5 Interruption

```
start = extensions.Start()
inter = extensions.Interrupt(0, "by A with stop")
msg = extensions.Message(1,"U","A","PollMode<InstrumentId,int>","2s")
rec = extensions.Rec(2, "Loop")
graph.add_Snode(start)
graph.add_Snode(msg)
graph.add_Snode(inter)
graph.add_Snode(rec)
ed = extensions.SEdge(start,inter)
ed3 = extensions.SEdge(inter,rec)
ed4 = extensions.SEdge(rec,msg)
ed2 = extensions.SEdge(msg,rec)
graph.add_Sedge(ed)
graph.add_Sedge_with_ports(ed3,'n','s')
graph.add_Sedge(ed4)
graph.add_Sedge_with_ports(ed2,'e','s')
```

Code in Python

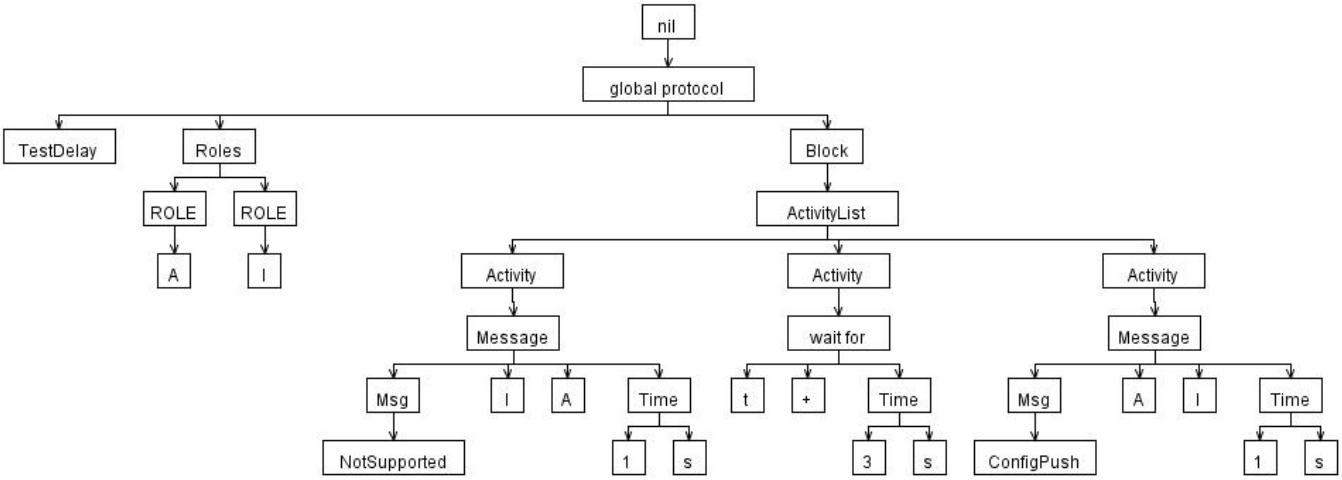


Figure A.6: Abstract Syntax Tree for the delay case

A.6 Delay

```
start = extensions.Start()
msg = extensions.Message(1,"I","A","NotSupported",1)
delay = extensions.Delay(3,"t+3")
msg2 = extensions.Message(2,"A","I","ConfigPush",1)
end = extensions.End()
graph.add_Snode(start)
graph.add_Snode(msg)
graph.add_Snode(delay)
graph.add_Snode(msg2)
graph.add_Snode(end)
ed = extensions.SEdge(start,msg)
ed3 = extensions.SEdge(msg,delay)
ed4 = extensions.SEdge(delay,msg2)
ed2 = extensions.SEdge(msg2,end)
graph.add_Sedge(ed)
graph.add_Sedge(ed3)
graph.add_Sedge(ed4)
graph.add_Sedge(ed2)
```

Code in Python

Appendix B

Implementation

B.1 ANTLR grammar: the complete file

```
grammar global_scribble_graph;

options {
        output=AST;
        backtrack=true;
        language=Python;
        }

tokens {
        Message;
        Block;
        ActivityList;
        Activity;
        Constraint;
        Parallel='parallel';
        Choice='choice';
        Rec='rec';
        Continue;
        Interrupt;
        With;
        Roles;
        ROLE;
        PROTOCOL='global protocol';
        SIG = 'sig';
        MsgSig;
        Msg;
        Before = 'before';
        After = 'after';
        Is = 'is';
        Wait='wait for';
        }
```

```
/*------------------------------------------------------------------
 * PARSER RULES
 *------------------------------------------------------------------*/

protocolDef
        :       PROTOCOL protocolName ('<' parameters '>' )? rolesDef  protocolBody -> ^(PROTOCOL protocolName rolesDef protocolBody) ;

protocolName
        :       ID;

parameters
        :       SIG identifier  ( ',' SIG identifier )* ;

rolesDef:       '(' roleDef (',' roleDef)* ')' -> ^(Roles (roleDef)+);

roleDef :       'role' roleName -> ^(ROLE roleName);

roleName:       ID;

protocolBody
        :       block;

block   :       '{' activityList '}' -> ^(Block activityList);


timeConstraint: ID After date ';' -> ^(After ID date)
                | ID Before date ';' -> ^(Before ID date)
                | ID Is date ';' -> ^(Is ID date);

time    :        NO Unit;

date    :       hour min sec day '/' month '/' year ;
```

```
activityList
        :       ( activity)*->^(ActivityList activity* );

activity:       timeConstraint? interaction-> ^(Activity interaction timeConstraint?)
                | timeConstraint? choice-> ^(Activity choice timeConstraint? )
                | timeConstraint? parallel-> ^(Activity parallel timeConstraint?)
                | timeConstraint? recursion-> ^(Activity recursion timeConstraint?)
                | timeConstraint? cont-> ^(Activity cont timeConstraint?)
                | timeConstraint? interruptible->^(Activity interruptible timeConstraint?)
                | timeConstraint? delay -> ^(Activity delay timeConstraint?);

delay   :       Wait ID Symbol time -> ^(Wait ID Symbol time)
                | Wait ID Is date -> ^(Wait ID date);

interaction
        :       message 'from' roleName 'to' roleName 'within' time ';' -> ^(Message message roleName roleName time);

choice
        :       Choice 'at' roleName block ( 'or'  block )*
                -> ^(Choice roleName block+ );

parallel:
                Parallel block ( 'and'  block )+  -> ^(Parallel block (block)+);

recursion
        :       Rec identifier block-> ^(Rec identifier block) ;

identifier
        :       ID;

cont    :       'continue' identifier ';'-> ^(Continue identifier);
```

```
interruptible
]       :       'interruptible' block withMessage-> ^(Interrupt block withMessage);

withMessage
]       :       'by' roleName 'with' message (','by' roleName 'with' message )*-> ^(With (roleName message)+);

]message :      ID -> ^(Msg ID)
}               | messageSignature -> ^(Msg  messageSignature );

messageSignature
]       :       messageOperator  '('  payloadType ( ',' payloadType )*  ')' -> ^(MsgSig messageOperator (payloadType)+ );

messageOperator
]       :       ID;

payloadType
]       :       ID;

]hour   :       NO 'h';

]min    :       NO 'min';

]sec    :       NO 's';

]day    :       DAY;

]month  :       MONTH;

]year   :       NO;

/*------------------------------------------------------------------
 * LEXER RULES
 *------------------------------------------------------------------*/

]ID : ('a'..'z'|'A'..'Z'|'_')('a'..'z'|'A'..'Z'|'0'..'9'|'_')* ;

]NO     :       ('1'..'9')('0'..'9')*;

]Symbol :       ('+'|'-'|'*'|'/');

]Unit   :       ('ns'|'ms'|'s'|'min'|'h'|'days'|'months'|'years');

]DAY    :       ('1'..'9')|'10'|'11'|'12'|'13'|'14'|'15'|'16'|'17'|'18'|'19'|'20'|'21'|'22'|'23'|'24'|'25'|'26'|'27'|'28'|'29'|'30'|'31';

]MONTH  :       ('1'..'9'|'10'|'11'|'12');

]WHITESPACE : ( '\t' | ' ' | '\r' | '\n'| '\u000C' )+    { $channel = HIDDEN; } ;
```

B.2 Examples of code

We present here the code for the method `toScribble` in the class SGraph, derived from Dot:

```
def toScribble(self):
        protocoldcl = "global protocol "+self.get_name()+"("
        roles_list = self.obj_dict['roles']
        for i in range(0 ,len(roles_list)):
            protocoldcl = protocoldcl+"role "+roles_list[i]
            if i<len(roles_list)-1:
                protocoldcl = protocoldcl+", "
        protocoldcl=protocoldcl+")"

        # assumption: there is only one start node
        start = self.get_Snode('start')[0]

        return protocoldcl+start.toScribble(self)
```

and the method `toScribble` in the class Fork, derived from SNode:

```
def toScribble(self, graph, indent):
        flag = False
        node = None
        next_nodes = self.next_nodes(graph)
        code = self.addIndent(indent)
        code = code + "par{"
        c0, n0 = next_nodes[0].toScribble(graph, indent+1)
        if n0 is None:
            flag = True
```

```
        else:
            node = n0
        code = code + c0
        code = code + self.addIndent(indent)+"}"
        for i in range(1,len(next_nodes)):
            c, n = next_nodes[i].toScribble(graph, indent+1)
            if not flag and n is None:
                flag = True
                node = None
            code = code +self.addIndent(indent)+ "and{"
            code = code + c
            code = code +self.addIndent(indent)+ "}"
        if node is None:
            return code, None
        else:
            co,no = node.toScribble(graph, indent)
            return code+co , no
```

B.3 User guide

In order to use our implementation, developed in the project global_scribble_graph, you should follow the steps describe hereunder.

First, you need to install Pydot as a Python package. This can be found in: `http://code.google.com/p/pydot/downloads/list`. Furthermore, Pydot requires also:

- pyparsing in order to load DOT files (`http://pyparsing.wikispaces.com/`),
- Graphviz to render the graphs ([3]).

Second, you need to install also ANTLR. The Python runtime environment can be found in `http://www.antlr.org/download/Python`.

After these two installations the Python browser looks like in Figure B.1.

Third, you need to import the project global_scribble_graph_project.

To generate a graph from a global protocol in Scribble (.spr file), you need to specify the file and then run the main method in scribbleToGraph.py in the Source package of the project.

To generate a global protocol in Scribble from a graph, you need to first create the graph. You can use for this purpose the file graphCreation.py in the Source package. Then you need to run the main method in graphToScribble.py in the Source package as well.

If you use ANTLRWorks to generate the files, ANTLRWorks 1.2.2 allows generating the files in Python, ANTLRWorks 1.3 allows debugging grammars. Do not use other updated versions, it may not work.

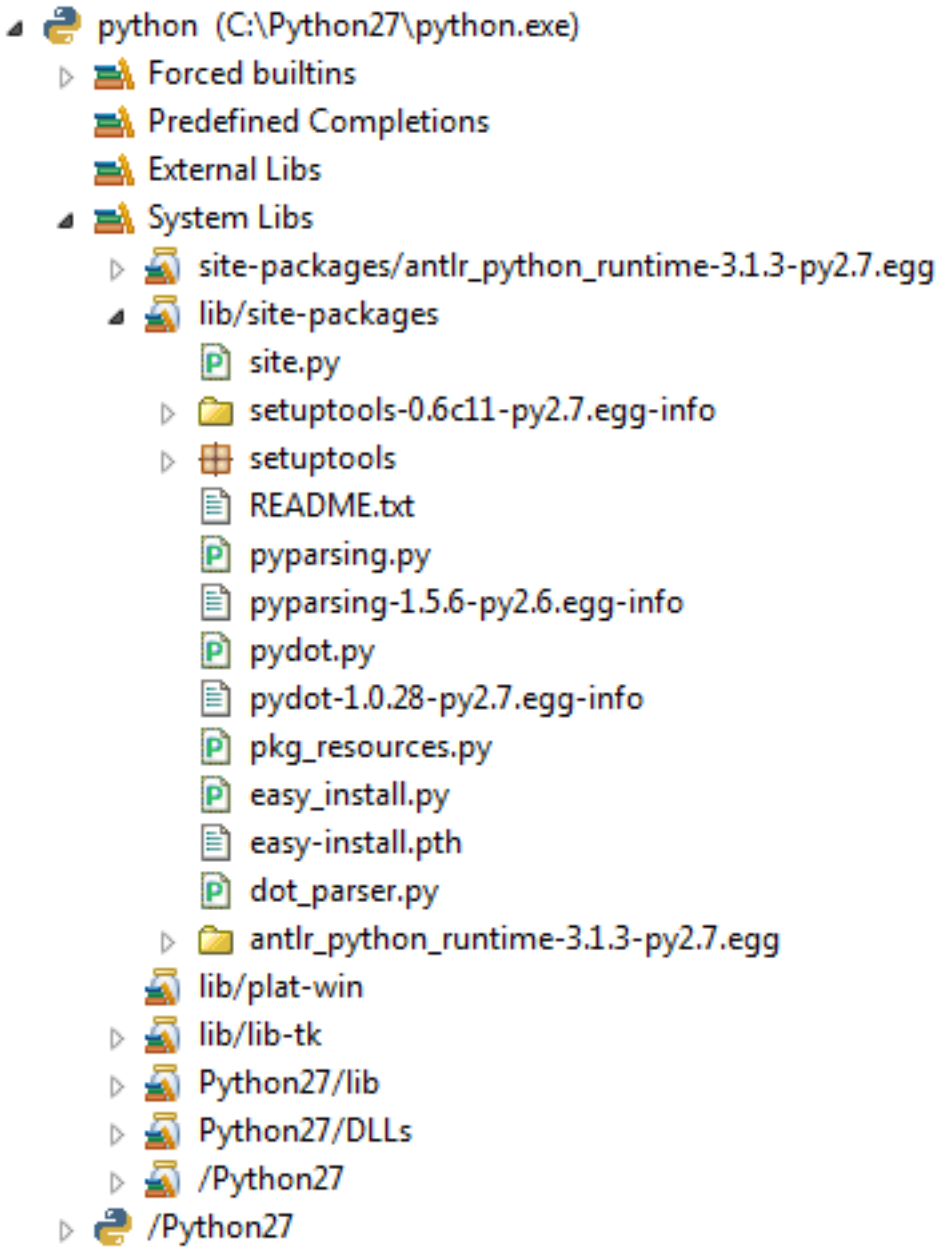


Figure B.1: Python's package

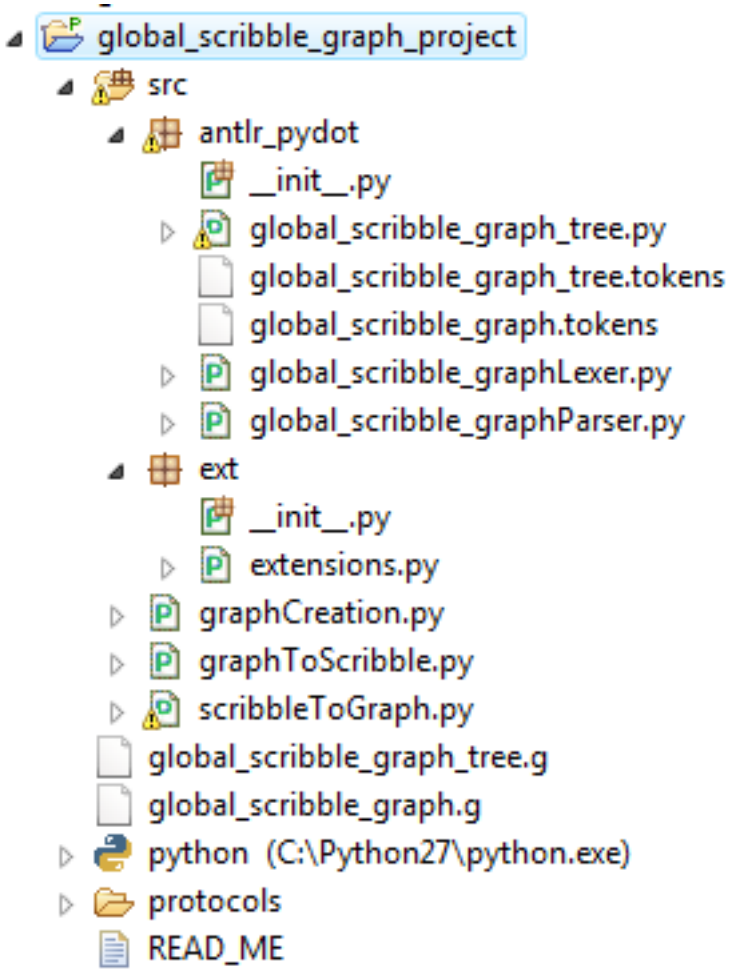


Figure B.2: Structure of the project