

Supporting Conversation Types in a Distributed Programming Language

Luísa Lourenço
Advisor: Prof. Dr. Luís Caires

Departamento de Informática
Faculdade de Ciências e Tecnologia
Universidade Nova de Lisboa

July 21, 2009

Motivation

Software Analysis

We usually want to ensure applications are error-free during their execution.

This can be achieved through the use of type systems.

A type system is a syntactic method that attributes types to program's terms in such way that assures the program has no ill behaviour during execution time.

There are many type systems for sequential programming but few for service-oriented programming.

Motivation

Software Analysis

Why is it important to have a type theory for service-oriented applications?

Motivation

Software Analysis

Why is it important to have a type theory for service-oriented applications?

Because these applications are executed in a distributed environment so

- There is no centralized control
- High level of concurrency and parallelism
- Message-based communication

Motivation

Software Analysis

Why is it important to have a type theory for service-oriented applications?

Therefore they are prone to new kind of errors like transmitted messages being received in the wrong order, which can:

- Lead to interaction's protocols violation
- A partner to wrongly act upon a out-of-order message

Thus the conception of new type systems for service-oriented applications is of great importance.

Goals

This work's goals consists in design and implementation of:

- A proof of concept distributed programming language based on the notion of *conversation*
- A typechecking algorithm, with subtyping, for the correspondent type system

Background

Process Calculi

Process calculi offer a powerful tool to reason about the interactions between concurrent processes by means of a simple modeling language with algebraic properties that enables us to verify equivalence of processes.

The basic unit of a process calculus is a *process* which consists in an agent that can interact with other processes through communication using channels.

Background

Calculus for Multiparty Conversation

A variety of process calculi emerged to reason about distributed computing, from broader calculi to represent concurrent and distributed systems like

- CCS [Milner, 1980]
- π -calculus [Milner, 1999]

to more specific calculi to model service-oriented systems

- Multiparty Session π -calculus [HondaYoshidaCarbone, 2008]
- Conversation Calculus [CairesVieira, 2009]
(more in Vieira's forthcoming PhD thesis)

Background

Calculus for Multiparty Conversation

In particular, service-oriented calculi are based on two approaches: multi-session based interactions and conversation access point based interactions.

The difference consists in how the interactions happening in a *conversation* are modelled:

- for multi-session approach: interactions happen through shared channels that are distributed at service invocation time to all participants;
- for conversation access point approach: there is a conversation access point where all the interactions of a conversation takes place.

Background

Type Systems for Distributed Systems

Type systems for concurrent and distributed systems have evolved in the past years, first by reasoning about what kind of values one expects from communication channels to describing the behaviour a program has with respect to what messages it receives or sends.

Background

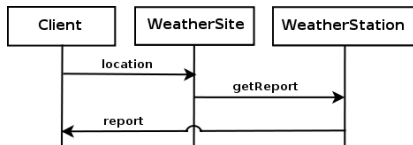
Type Systems for SOC

More recently, type theory was developed for service-oriented systems:

- for multi-session based approach: a global description of all the interactions between conversations is given through a global type whilst a local type describes each partner's local behaviour;
- for conversation access point approach: an unification of a global and local description of behaviour is achieved by a conversation type to each partner that can be unified into a more general conversation type describing the expected contract for that service.

Weather Forecast Service

An Example of Use of Our Language



The Weather Forecast Service, when invoked, awaits for the client's location, asks the associated weather station to join the on-going conversation and requests a weather report. After the weather station joins the conversation, it will generate a weather report and send it directly to the client.

Weather Forecast Service

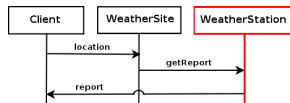
An Example of Use of Our Language

Using our language we would obtain:

```

WeatherStationCode ≜
site WeatherStation {
  def weatherReport as {
    receive(getReport);
    send(report, generatedReport);
  }
};;

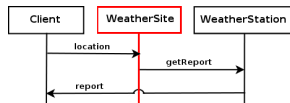
```



Weather Forecast Service

An Example of Use of Our Language

Using our language we would obtain:



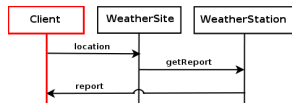
```

WeatherSiteCode ≙
site WeatherSite {
  def forecastWeather as {
    val loc = receive(location);
    join weatherReport in WeatherStation as {
      send(getReport);
    }
  }
}
};;
  
```

Weather Forecast Service

An Example of Use of Our Language

Using our language we would obtain:



ClientCode \triangleq

```
invoke forecastWeather in WeatherSite as {
  send(location, my_location);
  val my_weather_report = receive(report);
};;
```

Weather Forecast Service

An Example of Use of Our Language

Now with conversation types:

WeatherSiteCode \vdash

```
WeatherSite:[forecastWeather](location?(String);
                                getReportτ();
                                report!(String)
                                ),
```

```
WeatherStation:[weatherReport]( getReport!() )
```

WeatherStationCode \vdash

```
WeatherStation:[weatherReport](getReport?();
                                report!(String)
                                )
```

Weather Forecast Service

An Example of Use of Our Language

Now with conversation types:

```
ClientCode ⊢ WeatherSite : [forecastWeather](location!(String);  
                               report?(String)  
                               )
```

Expected Challenges

One of the expected challenges during the execution of this work consists in the adaptation and implementation of the merge relation in the typechecking algorithm.

Workplan

What has be done so far

At this point, we have defined the syntax and semantics of our language as well as some of the functionalities the runtime support should offer in order for all to work as intended.

We also have defined most of the typing rules for our type system, namely the typing rules for service primitives.

Workplan

- 1 Language's implementation: (1st September - late October)
 - Interpreter;
 - Distributed Runtime Support;
 - Typechecking algorithm, including the definition and implementation of subtyping and merge relations.
- 2 Validation of the implementation through: (late October - mid-January)
 - Production of examples;
 - Formalization of the type system and correctness proofs for the typechecking algorithm w.r.t. the type system. (tentative)
- 3 Finalization of dissertation's writing and preparation for its presentation.
(mid-January - late-February)

This work is supported by a research grant from **EU Project SENSORIA**.