APIGenMPST - 1st Semester Report

Telmo Ribeiro
Faculdade de Ciências da Universidade do Porto
Porto, Portugal
telmo.ribeiro@fc.up.pt

December 2023

1 Introduction

The operations performed by distributed systems happen in a highly structured manner.
Such property allows these applications to be abstracted as types through an intuitive syntax, which is then used as a basis of validating programs though an associated discipline. [1]
Multiparty Session Types (MPST) are a formal specification and verification framework for message-passing protocols. [2]

A simple protocol to describe the relay of a task from a master node to two other worker nodes could be described as the following:

Example 1.1:
G = Master>WorkerA:Work ; Master>WorkerB:Work ;
(WorkerA>Master:Done ‖ WorkerB>Master:Done)

Both the syntax and the semantics will be better explained going forward but currently it is needed to understand that we have three roles: **Master**, **WorkerA** and **WorkerB**.
Initially, **Master** delegates Work to **WorkerA**, then **Master** delegates the same task to **WorkerB** and then, in parallel, it will receive a datatype representing that the task as been preformed by each of the workers.

After the protocol has been established it will be put through the MPST's pipeline.
The steps in said pipeline that are already implemented will be explained throughout the report.

MPSTs provide some guarantees such as:
- **Communication safety** where interactions within a session never incur a

communication error.
- **Progress** where channels for a session are used linearly and are deadlock-free in a single session.
- **Session fidelity** where the communication sequence in a session follows the protocol declared in the session type.

All of the implementation so far has been written in Scala.

2 Implementation Choices

In the literature we are often faced with different choices regarding the meaning, scope and expressiveness of the *global types*.
Indeed, throughout the initial phase there was a discussion on choices made like:
- Interaction: Synchronous vs Asynchronous
- **Sequences**: Weak vs Strong
- **Parallel**: Free Interleaving vs No Interleaving
- **Branching**: Plain vs Robust
- **Recursion**: Fixed Point vs Kleene's Closure

There was a debate of which *global type*'s would be picked for implementation and if that was the case, how would such implementation be achieved.
Both problems were sorted through the mentality that we would like to achieve something that was present in the majority of papers so that previous readers could relate to the benefits of such picks as well as make the language the most expressive we could.
For example, when talking about recursion there are usually two ways to go about it, either fixed points or Kleene's closure.
It is known[add_source] that everything that can be express though Kleene's closure can be express though Fixed Points but the other way around is not true.
Therefore, when choosing between those two options, Fixed Points were picked since they would allow to keep the most expressiveness.

When going through the literature, there was a change in scope that was also subject of much debate.
Previous papers tend to talk about the sessions in greater detail and delve into aspects such as the handshakes pre-communication between the processes that implement each role and other synchronization techniques [add_source].
This kind of detail is overlooked by the more recent ones, as such, we also choose to not go in depth when it comes to session creation.

3 Grammar

MPSTs depend on the evaluation of *global types* in order to derive the *local types*, through the projection of the former trough every role partaking in it.
An overlapping of features from either *global* and *local types* are found in the literature [3] and there was an active effort to maintain them in a versatile and expressive manner.
The description of both grammars goes as follows:

Global Type's Grammar:

$$\mathbf{G ::= p{>}q{:}t \mid G_1 + G_2 \mid G_1 \; ; \; G_2 \mid G_1 \parallel G_2 \mid \mu X \; ; \; G \mid X \mid skip}$$

Global type's meanings:
- **p>q:t** is the asynchronous communication from the role **p** to the role **q** of the datatype **t**.
- $\mathbf{G_1 + G_2}$ where **+** is the branching option between G_1 and G_2.
- $\mathbf{G_1 \; ; \; G_2}$ where **;** is the **weak** sequential composition of G_1 and G_2.
- $\mathbf{G_1 \parallel G_2}$ where $\parallel$ is the parallel, with free interleaving, composition of G_1 and G_2.
- μ**X ; G** is the bounding of the fixed point variable **X** to the recursive *global type* **G**.
- **X** is a fixed point variable used in the call of the recursive global type.
- **skip** is an internal *global type* with no meaning except to be skipped over.

Example 2.1:
G = Master>WorkerA:Work ; Master>WorkerB:Work ; (WorkerA>Master:Done ∥ WorkerB>Master:Done)

Local Type's Grammar:

$$\mathbf{L ::= pq!t \mid pq?t \mid L_1 + L_2 \mid L_1 \; ; \; L_2 \mid L_1 \parallel L_2 \mid \mu X \; ; \; L \mid X \mid skip}$$

With the following meanings:
- **pq!t** is the asynchronous send from role **p** to role **q** of the datatype **t**.
- **pq?t** is the asynchronous receive from role **p** to role **q** of the datatype **t**.
- all other *local types* behave in a similar way to their *global type*'s counterpart.

Example 2.2:
$\mathbf{L}_{Master}$ **= MasterWorkerA!Work ; MasterWorkerB!Work ; (MasterWorkerA?Done ∥ MasterWorkerB?Done)**
$\mathbf{L}_{WorkerA}$ **= WorkerAMaster?Work ; WorkerAMaster!Done**
$\mathbf{L}_{WorkerB}$ **= WorkerBMaster?Work ; WorkerBMaster!Done**

Implementation-wise the decision was made to collapse both the *global* and *local type*'s grammar into the *protocol*'s grammar.
Since both share extreme similarities, we can benefit from optimisations like

code reuse by having them under the same structure, when such reuse can not be preformed it is not troublesome to assert that only instructions from either *global* or *local types* are expected.

```
enum Protocol:
    // GlobalTypes //
    case Interaction(agentA: String, agentB: String, message:
    String)
    case RecursionFixedPoint(variable: String, protocolB: Protocol)
    case RecursionCall(variable: String)
    case Sequence(protocolA: Protocol, protocolB: Protocol)
    case Parallel(protocolA: Protocol, protocolB: Protocol)
    case Choice  (protocolA: Protocol, protocolB: Protocol)
    // LocalTypes //
    case Send   (agentA: String, agentB: String, message: String)
    case Receive(agentA: String, agentB: String, message: String)
    // Internal //
    case Skip
end Protocol
```

Listing 1: grammar

4 Parsing

At this point, the goal is to receive a *protocol*'s specification as a *global type* and produce an **abstract syntax tree (AST)** that accurately represents the former.
To this effect, the Scala's library **RegexParsers** was used.
Through the aid of hindsight and deeper search it is now known that, performance-wise, there would be libraries that could provide greater benefits. Still, to this point it has not yet been determined performance issues that would justify the parser's rewriting where another library is used as a basis.
Roles and **datatypes** are currently only allowed to be alphabetic strings, such implementation can be extended quite easily.
The order of operations expressed in the parser tries to satisfy what is usually found on the literature.

Global Type's Order:

$$\mathbf{(\ G\ )} >> \mathbf{p{>}q{:}t} >> \mu\mathbf{X}\ ;\ \mathbf{G} >> \mathbf{X} >> \mathbf{G}_1\ ;\ \mathbf{G}_2 >> \mathbf{G}_1 \parallel \mathbf{G}_2 >> \mathbf{G}_1 + \mathbf{G}_2$$

The following steps on the MPSTs pipeline will be performed on top of the **AST** provided by the parser or by the ones derived through the projection of the former.
Since the **AST** is implemented in a binary fashion, there is room for optimisation to limitless arguments.

Example 3.1:
create an actually AST representation
AST = Sequence(Interaction(Master, WorkerA, Work), Sequence(Interaction(Master, WorkerB, Worker), Parallel(Interaction(WorkerA, Master, Done), Interaction(WorkerB, Master, Done)))

5 Projectability Checking

This step is dedicated to analyse the provided **AST** in order to determine if it can be projected or not.
Projectability or wellformedness, is usually divided in a set of properties that must be maintained in order to comply with the demands stated in the Introduction and assure the overall well function of the protocols.

Projectability

No self-communication:
Forbidding self-communication was a trivially implementation.
Traversing the **AST**, every interaction of the type: **p>q:t** needs to be checked assuring that $\mathbf{p} \neq \mathbf{q}$. [4]

```
1  private def selfCommunication(global: Protocol): Boolean =
2      global match
3          // terminal cases //
4          case Interaction(agentA, agentB, _) => agentA != agentB
5          case RecursionCall(_)               => true
6          // recursive cases //
7          case RecursionFixedPoint(_, globalB) => selfCommunication(
       globalB)
8          case Sequence(globalA, globalB)      => selfCommunication(
       globalA) && selfCommunication(globalB)
9          case Parallel(globalA, globalB)      => selfCommunication(
       globalA) && selfCommunication(globalB)
10         case Choice  (globalA, globalB)      => selfCommunication(
       globalA) && selfCommunication(globalB)
11         // unexpected cases //
12          case _ => throw new RuntimeException("\nExpected:\
       tGlobalType\nFound:\t\tLocalType")
13 end selfCommunication
```

Listing 2: No Self-Communication

No free variables:
Free recursion variables can not be present on the *global type*. That is, a variable **X** in a *global type* **G**, if previously there was a reduction of the *global type*: μ**X ; G**.[5]
A special case is to treat recursion variables that appear in parallel branches as free variables, even when that same parallel was achieved in **G** after μ**X ; G**. [5]
Although the reason for generally forbidding free variables might be trivial, the special case may not be.
Through this action, we make sure that no race conditions arises from different iterations of the same loop.

```
private def freeVariables(variables: Set[String], global: Protocol)
    : Boolean =
    global match
        // terminal cases //
        case Interaction(_, _, _)    => true
        case RecursionCall(variable) => variables.contains(variable
    )
        // recursive cases //
        case RecursionFixedPoint(variable, globalB) =>
    freeVariables(variables + variable, globalB)
        case Sequence(globalA, globalB)  =>
            globalA match
                case RecursionCall(variable) => freeVariables(
    variables, globalA) && freeVariables(variables - variable,
    globalB)
            case _                       => freeVariables(variables
    , globalA) && freeVariables(variables, globalB)
        case Parallel(globalA, globalB)  => freeVariables(Set(),
    globalA) && freeVariables(Set(), globalB)
        case Choice  (globalA, globalB)  => freeVariables(variables
    , globalA) && freeVariables(variables, globalB)
        // unexpected cases //
        case _ => throw new RuntimeException("\nExpected:\
    tGlobalType\nFound:\t\tLocalType")
end freeVariables
```

Listing 3: No Free Variables

Disambiguation:
Disambiguation is concerned with the need of each role to determine what choice's branch was picked.
To meet this criteria it is needed that for each role, its first interaction in both branches are different, either in the roles involved, their actions or their datatype. [5]
The current implementation gets the set of head interactions that a role has in each branch, then it checks if the intersection of those sets are empty, accepting if so or rejecting otherwise.

```
@tailrec
private def roleDisambiguation(globalA: Protocol, globalB: Protocol
    , roles: Set[String], isStillProjectable: Boolean): Boolean =
    if   roles.isEmpty
    then isStillProjectable
    else
        val role: String               = roles.head
        val headGlobalA: Set[Protocol] = headInteraction(globalA,
    role)
        val headGlobalB: Set[Protocol] = headInteraction(globalB,
    role)
        val isProjectable: Boolean     = (headGlobalA intersect
    headGlobalB).isEmpty
        roleDisambiguation(globalA, globalB, roles - role,
    isStillProjectable && isProjectable)
end roleDisambiguation

```

```
private def disambiguation(global: Protocol): Boolean =
    global match
        // terminal cases //
        case Interaction(_, _, _) => true
        case RecursionCall(_)     => true
        // recursive cases //
        case RecursionFixedPoint(_, globalB) => disambiguation(globalB)
        case Sequence(globalA, globalB)      => disambiguation(globalA) && disambiguation(globalB)
        case Parallel(globalA, globalB)      => disambiguation(globalA) && disambiguation(globalB)
        case   Choice(globalA, globalB)      => roleDisambiguation(globalA, globalB, roles(global), true)
        // unexpected cases //
        case _ => throw new RuntimeException("\nExpected:\tGlobalType\nFound:\t\tLocalType")
end disambiguation
```

Listing 4: Disambiguation

Linearity:
The concept and therefore, the implementation of linearity is close to that of disambiguation.
For each parallel *global type* we need to make sure that there are no similar interactions in both branches. [4]
We retrieve the set of all the interactions in both branches a role is involved in, then we make sure that the intersection of both sets are empty, accepting if so or rejecting otherwise.

```
private def linearity(global: Protocol): Boolean =
    global match
        // terminal cases //
        case Interaction(_, _, _) => true
        case RecursionCall(_)     => true
        // recursive cases //
        case RecursionFixedPoint(_, globalB) => linearity(globalB)
        case Sequence(globalA, globalB)      => linearity(globalA) && linearity(globalB)
        case Parallel(globalA, globalB)      =>
            val iterationsA: Set[Protocol] = interactions(globalA)
            val iterationsB: Set[Protocol] = interactions(globalB)
            (iterationsA intersect iterationsB).isEmpty
        case Choice  (globalA, globalB)      => linearity(globalA) && linearity(globalB)
        // unexpected cases //
        case _ => throw new RuntimeException("\nExpected:\tGlobalType\nFound:\t\tLocalType")
end linearity
```

Listing 5: Linearity

6 Projection

When *global types* admit all the **projectability properties** then it can be submitted to the next phase on the MPSTs pipeline.
Projection refers to the step where *local types* are derived from the *global type*. Each projection is the local understanding of the *global type* from the point of view of a given role.

Implementation-wise, the projection phase begins with the collection of all roles from the *global type*.
Then, for each agent, there will be an attempt to translate the *global type* where only the actions related to the agent are aimed to be kept.
When the role is not present in an interaction, either as sender or as receiver, then said interaction is replaced by the internal *local type* **skip**.

```
1  private def projection(global: Protocol, role: String): Protocol =
2      global match
3          // terminal cases //
4          case Interaction(agentA, agentB, message) =>
5              if      role == agentA then Send   (agentA, agentB,
       message)
6              else if role == agentB then Receive(agentB, agentA,
       message)
7              else    Skip
8          case RecursionCall(variable) => RecursionCall(variable)
9          // recursive cases //
10         case RecursionFixedPoint(variable, globalB) =>
       RecursionFixedPoint(variable, projection(globalB, role))
11         case Sequence(globalA, globalB) => Sequence(projection(
       globalA, role), projection(globalB, role))
12         case Parallel(globalA, globalB) => Parallel(projection(
       globalA, role), projection(globalB, role))
13         case Choice  (globalA, globalB) => Choice  (projection(
       globalA, role), projection(globalB, role))
14         // unexpected cases //
15         case _ => throw new RuntimeException("\nExpected:\
       tGlobalType\nFound:\t\tLocalType")
16 end projection
```

Listing 6: Projection

After such replacements are preformed, there will be a cleaning phase where as a side effect all **skips** will be propagated, until it has reached a state that can no longer be simplified, that is, only *local types* that mention the role and are allowed and no **skip** is present.

7 Semantics

Semantics are currently in development.
At the present it is being tested an **operational semantics** where it is allowed *weak sequentialisation.*[6]
That is, in $\mathbf{G}_1$ **;** $\mathbf{G}_2$, it is not mandatory for $\mathbf{G}_1$ to be fully reduced in order for $\mathbf{G}_2$ to start reducing.
In fact, is is only necessary that there are no dependencies from $\mathbf{G}_1$ in $\mathbf{G}_2$, allowing more expressiveness in the MPSTs.
The goal, on the subject of semantics, is to allow multiple ones to be implemented and tested in order to achieve a greater grasp from the MPSTs.

References

[1] K. Honda, N. Yoshida, and M. Carbone, "Multiparty asynchronous session types," 2008.

[2] N. Yoshida and L. Gheri, "A very gentle introduction to multiparty session types," in *International Conference on Distributed Computing and Internet Technology*, pp. 73–93, Springer, 2019.

[3] S.-S. Jongmans and J. Proença, "St4mp: a blueprint of multiparty session typing for multilingual programming," in *International Symposium on Leveraging Applications of Formal Methods*, pp. 460–478, Springer, 2022.

[4] G. Cledou, L. Edixhoven, S.-S. Jongmans, and J. Proença, "Api generation for multiparty session types, revisited and revised using scala 3 (full version)," 2022.

[5] P.-M. Deniélou and N. Yoshida, "Dynamic multirole session types," in *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 435–446, 2011.

[6] L. Edixhoven, S.-S. Jongmans, J. Proença, and I. Castellani, "Branching pomsets: design, expressiveness and applications to choreographies," *Journal of Logical and Algebraic Methods in Programming*, vol. 136, p. 100919, 2024.

[7] A. Scalas and N. Yoshida, "Less is more: multiparty session types revisited," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–29, 2019.