A Modular Toolkit for Distributed Interactions

Julien Lange Emilio Tuosto
jlange@mcs.le.ac.uk emilio@mcs.le.ac.uk
Department of Computer Science, University of Leicester, UK

We discuss the design, architecture, and implementation of a toolkit which supports some theories for distributed interactions. The main design principles of our architecture are *flexibility* and *modularity*. Our main goal is to provide an easily extensible workbench to encompass current algorithms and incorporate future developments of the theories. With the help of some examples, we illustrate the main features of our toolkit.

1 Introduction

With the emergence of distributed systems, communication has become one of the most important elements of today's programming practise. Nowadays, distributed applications typically build up from (existing) components that are glued together (sometimes dynamically) to form more complex pieces of software. It is hence natural to model such applications as units of computation interacting through suitable communication models. An intricacy of communication-centred applications is that interactions are distributed. Here, the acceptation of distribution has to be taken in a very general sense since interactions are physically and logically distributed; as a matter of fact, components may run remotely and, for instance, components may belong to different administrative domains.

In order to ensure predictable behaviours of communication-centred applications, it is necessary that software development is based on solid methodologies. Besides the theoretical results that allow us to analyse systems and prove their properties, it is also desirable to provide practitioners with a set of tools to support them in addressing the most common problems (e.g. avoiding synchronisation bugs).

In recent years, session types have appeared as an effective mathematical foundation for designing and reasoning on distributed interactions. For instance, dyadic session types [13] have been proposed as a structuring method and a formal basis for the verification of distributed interactions of two participants (e.g., in client-server architectures). Multiparty sessions [14] generalise session types to support more than two participants; they have been used in [9] to statically compute upper bounds of the size of buffers used for asynchronous communications in global interactions. Moreover, dynamic multiparty sessions have been studied in [6] and [7]. On top of multiparty sessions, in [4], a theory of design-by-contract for distributed interactions has been introduced. Basically, session types are extended with *assertions* acting as pre-/post-conditions or invariants of interactions.

Our main objective is to describe the design principles of the architecture of a modular toolkit which puts in practise the theories of distributed interactions based on session types. We have developed a toolkit that accommodates a few main requirements:

- Firstly, the toolkit provides a workbench for theoretical studies so to permit (*i*) to experiment with potentially more realistic examples and (*ii*) to possibly combine several of these methodologies.
- Secondly, our toolkit is easily extensible so to allow researchers to explore new directions as the theory of distributed interactions develops. For instance, the use of the tool revealed some issues

K. Honda and A. Mycroft (Eds.): Programming Language
Approaches to Concurrency and communication-cEntric
Software 2010 (PLACES'10)
EPTCS 69, 2011, pp. 92–110, doi:10.4204/EPTCS.69.7

in the design of global interaction à-la [4] and triggered the design of additional algorithms to help the programmer design correct distributed protocols [5].

- Finally, albeit being a prototype for research, our toolkit shapes the basic implementations that can be used in more realistic frameworks for the development of communication-centric software.

Arguably, most of the research around session types has been mainly devoted to give a precise description of verification and validation frameworks. In fact, only very few and ad-hoc implementations have been developed (e.g., [15, 18, 21, 22]).

The main contribution of this paper is the description of the design choice and the implementation of a modular toolkit that features the main algorithms necessary to analyse systems using theories based on session types. The most interesting features of our framework are illustrated by means of a few examples. The toolkit, its documentation, and a few examples are available at [17].

Synopsis § 2 gives background information and a motivating example. § 3 gives more details on the design principles of our toolkit, its architecture and implementation. § 4 gives illustrative examples of the tool's features. § 5 highlights its main advantages. § 6 compares our work to other implementation of ST. § 7 concludes and highlights our future plans.

2 Background and motivating example

We briefly describe the distinguished aspects common to several theories of distributed interactions. The design principles of our toolkit hinge on some key elements of session types that uniformly apply to several theoretical frameworks. The key ingredients of the theories of distributed interactions based on session types are described below.

Sessions are sets of *structured* interactions which correspond, for instance, to a complex communication protocol. Typically sessions are conceived as "correct executions" of a set of distributed interactions, namely those executions that run from the *session initialisation* to its termination. The basic idea is that a computation consists of several concurrent sessions that involve some participants. A main concern is that participants acting in different sessions do not interfere. For instance, a desirable property to enforce is that a message sent in a session from A and meant for B is not received by another participant C; however, other relevant properties can be considered as, for instance, progress properties of sessions which guarantee that participants are not stuck because of communications errors.

Interaction primitives basically include communication mechanisms à-la π-calculus that deal with sessions as first-class values. Another kind of interaction primitives often present features a *select/branch* mechanism which resembles a simplified form of method invocation. For instance, communication interaction and select/branch in the global calculus [14] notation are

$$A \to B : k\langle \mathtt{sort} \rangle \qquad \text{and} \qquad A \to B : k\{l_i : G_i\}_{i \in J}$$

In the former, participant A sends a message of type `sort` to B on the channel k; in the latter A selects one of the labels l_i (sending it on k) and, correspondingly, B executes its i^{th} branch G_i.

Communication primitives typically permit *delegation*, namely the fact that sessions can be exchanged so to allow a process to delegate to another process the continuation of the computation.

Typing disciplines guarantee properties of computations. For example, in dyadic session types [13] the *duality* principle guarantees that, in a session, the actions of a participant have to be complemented by the other participant (or its delegates). Among the properties checked by type systems, *progression* and some form of correctness properties are paramount. For instance, in [4] a well-typed system is guaranteed to respect the contract specified by its assertions and, once projected, the program is guaranteed to be free from "communication-errors".

Type systems are sometime subject to *well-formedness* conditions. For instance, global types in [14] have to be linear in order to be projected onto *local types*.

We illustrate some theoretical aspects with an example adapted from [14] to the *global assertions* in [4]. Intuitively global assertions may be thought of as global types decorated with formulae of a (decidable) logic. The following is a global assertion[1] specifying a protocol with two buyers (B_1 and B_2) and a seller (S). The buyers B_1 and B_2 want to purchase a book from S by combining their money.

$$
\begin{array}{rlllll}
G = & B_1 \rightarrow S & : & s\langle t : \mathtt{string}\rangle & (\!|\textbf{assert}\ t \neq \text{""}|\!). & (1)\\
& S \rightarrow B_1 & : & b_1\langle q : \mathtt{int}\rangle & (\!|\textbf{assert}\ q > 0|\!). & (2)\\
& B_1 \rightarrow B_2 & : & b_2\langle c : \mathtt{int}\rangle & (\!|\textbf{assert}\ 0 < c \leq q|\!). & (3)\\
& B_2 \rightarrow S & : & s\{\mathsf{ok}\ (\!||\!) : D, \mathsf{quit}\ (\!||\!) : \mathit{end}\} & & (4)
\end{array}
$$

The session G above describes the interactions among B_1, B_2, and S after a session initialisation is performed[2]; such initialisation will assign a role to each participant, namely each participant will act either as the first buyer B_1, or the second one B_2, or else the seller S. Each one of the interactions (1÷4) is decorated with an assertion of the form $(\!|\textbf{assert}\ \phi|\!)$ stating a condition ϕ on the variables of the protocol ($(\!||\!)$ abbreviates $(\!|\textbf{assert}\ \mathsf{true}|\!)$). Basically, G can be considered as a global type decorated with logical formulae.

In (1), B_1 and S interact (through s) and exchange the book title t; the assertion decorating (1) states that t is not the empty string which means that B_1 guarantees $t \neq$ "" while S relies on such assumption. In (2), S gives B_1 a quote q; similarly to (1), the assertion $(\!|\textbf{assert}\ q > 0|\!)$ constraints the price to a positive value and it constitutes an obligation for S and an assumption for B_1. In (3), B_1 tells B_2 its non-negligible contribution c to the purchase (as B_1 guarantees $(\!|\textbf{assert}\ 0 < c \leq q|\!)$). In the last step, B_2 may refuse (selecting label quit) or accept[3] the deal (selecting label ok); in the former case the protocol just finishes, otherwise it continues as:

$$D = B_2 \rightarrow S : s\langle a : \mathtt{string}\rangle (\!|\textbf{assert}\ a \neq \text{""}|\!).S \rightarrow B_2 : b_2\langle d : \mathtt{date}\rangle (\!||\!)$$

namely B_2 and S exchange delivery address and date.

Linearity is a (typically decidable) property ensuring that communications on a common channel are ordered temporally. Linear types can be *projected* so to obtain the *local types* for each participant. In order to have effective algorithms, the theoretical framework requires the decidability of the logic for expressing assertions as well as the *well-assertedness* of global assertions. Informally, a global assertion is well-asserted when (a) each possible choice a sender makes that satisfy the assertion of its interaction is not making later senders unable to fulfil their contracts (*temporal satisfiability*) and (b) participants state assertions only on known variables (*history sensitivity*).

Well-asserted and linear global assertions can be projected, similarly to global types, so to obtain local types, namely the interactions as perceived from the point of view of each participant. Unlike

[1] In this paper we deviate from the syntax adopted in [4] for assertions.

[2] The session initialisation is not described in the global types or global assertions; it is an operation executed by the processes implementing the type G.

[3] For simplicity, it is not specified how B_2 takes the decision; this can easily be done with suitable assertions on c and q.

for global types though, projections of global assertions must also "split" assertions in rely/guarantee propositions to be assigned to each participant. The projections of our example are:

$$pB_1 = s!\langle t : \texttt{string}\rangle\ (\!|\textbf{assert}\ t \neq \text{""}|\!);\ b_1?\langle q : \texttt{int}\rangle\ (\!|\textbf{assert}\ q > 0 \wedge t \neq \text{""}|\!);\ b_2!\langle \texttt{c:int}\rangle\ (\!|\textbf{assert}\ q > 0 \wedge 0 < c \leq q|\!)$$

$$pB_2 = b_2?\langle \texttt{c:int}\rangle\ (\!|\textbf{assert}\ \phi|\!);\ s \oplus \{\mathsf{ok}\ (\!||\!) : s!\langle a : \texttt{string}\rangle\ (\!|\textbf{assert}\ a \neq \text{""}|\!);\ b_2?\langle d : \texttt{date}\rangle\ (\!|\textbf{assert}\ \phi \wedge a \neq \text{""}|\!),\ \mathsf{quit}\ (\!||\!) : \mathit{end}\}$$

$$pS = s?\langle \texttt{string}\rangle\ (\!|\textbf{assert}\ t \neq \text{""}|\!);\ b_1!\langle q : \texttt{int}\rangle\ (\!|\textbf{assert}\ q > 0|\!);\ s\&\{\mathsf{ok}\ (\!|\textbf{assert}\ \psi|\!) : s?\langle a : \texttt{string}\rangle\ (\!|\textbf{assert}\ \psi \wedge a \neq \text{""}|\!); b_2!\langle d : \texttt{date}\rangle\ (\!||\!),\ \mathsf{quit}\ (\!|\textbf{assert}\ \psi|\!) : \mathit{end}\}$$

where $\phi = \exists q : \mathsf{int}, t : \mathsf{string} |\ 0 < c \leq q\ \wedge\ q > 0\ \wedge\ t \neq \text{""}$, and
$\psi = \exists c : \mathsf{int} |\ 0 < c \leq q\ \wedge\ q > 0\ \wedge\ t \neq \text{""}$

The behavioural types pB_1, pB_2, and pS above characterise classes of processes that are "well-behaved" with respect to the global interactions. For instance, let us consider the process cB_1 below.

```
cB1 = ā[2,3](s,b1,b2).                               // Session initialisation
      s!⟨''The art of computer programming''⟩;     // Send title to Seller
      b1?(quote);                                   // Receive quote from Seller
      b2!⟨quote/2⟩                                  // Send contribution to Buyer2
```

The process cB_1 starts a session on a declaring to act as the first buyer of the global assertion G above; this is done by the action $\bar{a}[2,3](s,b_1,b_2)$ that will synchronise with the other two participants (denoted by 2 and 3) using the session channels s, b_1, and b_2. It can be proved[4] that cB_1 has type pB_1 which guarantees that cB_1 has a correct interaction with any two other processes having type pB_2 and pS. The rest of the process is an instance of the type pB_1 detailing the behaviour of the first buyer.

3 Toolkit design and implementation

3.1 Objectives

The objective of this work is to describe the architecture and the implementation of a modular toolkit implementing algorithms as those described in § 2. The toolkit we developed supports the following development methodology (see [23] for a concrete realisation). A team of software architects writes a global description of the distributed interactions which specifies the intended behaviour of the whole system. The global description is checked and projected onto each participant. Then, each part of the system is developed (possibly independently) by a group of programmers. Finally, the pieces of programs are checked, validated, and possibly monitored during the execution. This methodology is supported by the theories drafted in § 2 whereby

1. global descriptions are given by global types and global assertions,
2. projections yield the parts of the systems to be developed, and

[4] We consider here a trivial process for simplicity; there are more complex cases where, for instance, the first buyer delegates the interactions with the second buyer to another process.

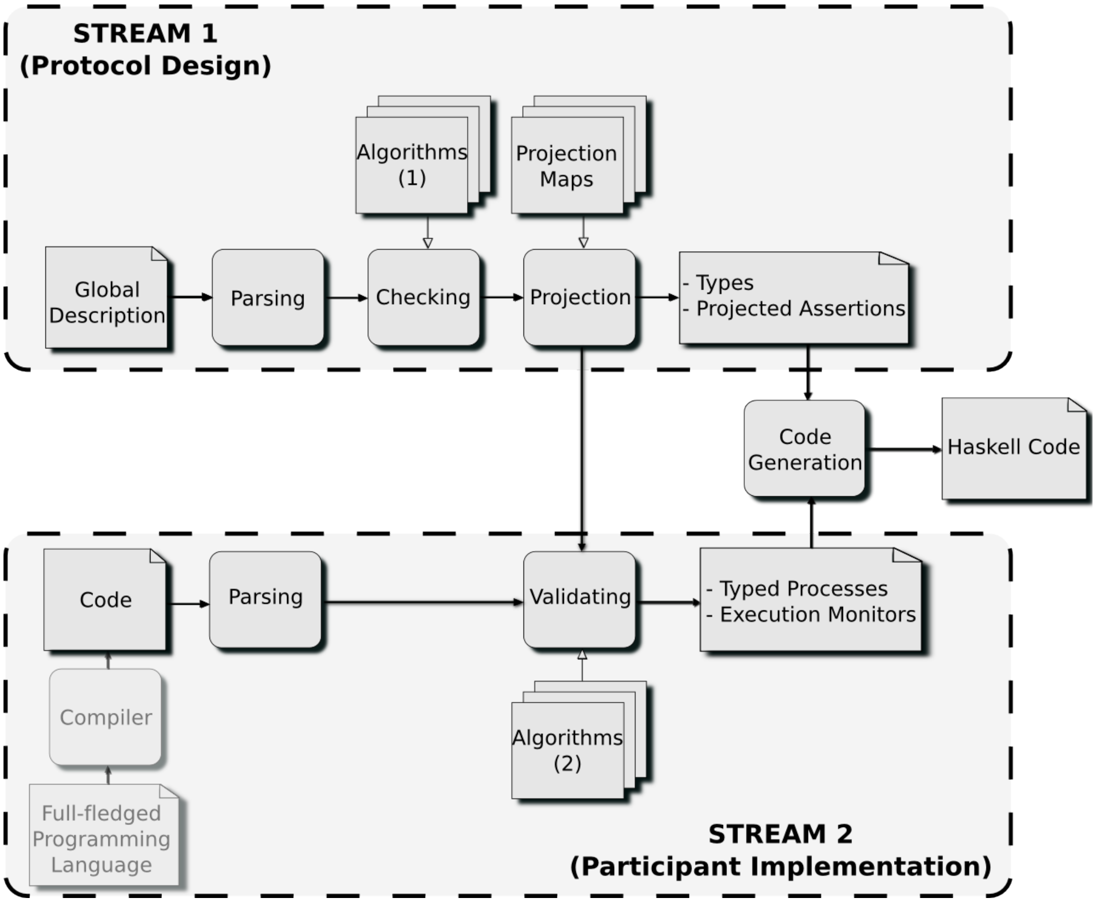

Figure 1: Architecture

3. compliance of code with the specification is obtained by typing systems (to be matched against the projection).

It is therefore possible to statically verify properties of designs/implementations and to automatically generate monitors that control the execution in untrusted environments. Our main driver is that the architecture has to easily allow our toolkit to be adapted to changes and advancements in the theories. For instance, it has to consistently integrate the two (equivalent) projection algorithms described in [4], or be parametric wrt the logic used in the assertion predicates. Note that our approach distinguishes itself from other works such as [15, 18, 21, 22] by focusing on the tools accompanying the theories and not on the integration of ST in a programming language.

3.2 Architecture

The architecture of our workbench is illustrated in Figure 1 and consists of two main streams, STREAM 1 and STREAM 2; both streams' output are used for the *code generation* activity which combines behavioural types and processes to generate safe Haskell code. The two streams correspond to design of protocols, on the one hand, and participants design and implementation, on the other hand. The input of STREAM 1 is a global description of the interactions while the input of STREAM 2 is the "program code" of each participant of the system, written in a dialect of the π-calculus[5]. In STREAM 1, global descriptions are parsed, checked, and then projected onto each participant. In STREAM 2, the code of participants is parsed, typed, and then validated against the local types obtained in STREAM 1 by projecting the global interactions. We give a walk-through of the architecture to illustrate the main components of the toolkit.

STREAM 1**.** Taking a global description, such as G in § 2, a parser constructs an abstract tree of the distributed interactions, while interacting with the user if syntax errors are detected in the description. The checking module applies a series of algorithms (see (1) in Figure 1) on the tree to check that some properties are guaranteed. At least, the following algorithms are executed.

- *One-time unfolding*: recursive types are unfolded one time according to the equi-recursive view of recursive types. One-time unfolding is necessary before checking for linearity, whose definition relies on type equality.
- *Linearity check*: this is necessary to ensure that there is no races on the communication channels.
- *Well-assertedness*: if the global interactions are decorated with assertions, it has to be checked that they are well-asserted (cf. § 2) in order to project them and obtain the local types as described in [4].

Each algorithm notifies the user in case the description does not satisfy the properties, and accordingly, stop the execution of the process. If the global description is "valid", the projections (like pB_1, pB_2, and pS in § 2) can be calculated. This is done according to the function defined in [4, §4] which builds up on the projection operation in [14, §4.2].

STREAM 2**.** A program code (written in a π-calculus-like language, such as cB_1 in § 2) is parsed to check for syntax errors and to build an abstract tree, similarly to STREAM 1. Then, the following steps are applied on the tree.

- A typing algorithm infers the type of the processes according to a set of typing rules.
- If the processes are asserted, a validator checks the interaction predicates for satisfiability.
- The inferred types (and possibly the assertions) are compared with the projections.

As in STREAM 1, each step of the stream notifies the user in case errors are detected. In particular, mismatches between a participant's projection and its corresponding inferred types are explicitly output.

Code generation. In order to illustrate a practical use of our toolkit, we have developed a prototypical translator which generates Haskell code from the π-calculus-like code representing well-typed participants of a global interaction. On successful completion, STREAM 1 and STREAM 2 produce two outputs that are compatible. More precisely, safe Haskell code can be generated from the verified π-calculus-like code given in STREAM 2 and, possibly, execution monitors[6] can be integrated from the projections computed in STREAM 1. This is possible due to the fact that inputs have passed all the checks (i.e. processes are validated against the projections obtained by the global interactions).

Further details on the code generation are given in § 3.3.

[5]The program code may be obtained by compiling programs written in full-fledged programming languages extended with session types like [15]. However, this feature is not yet available in the toolkit.

[6]Monitors can be automatically generated from global assertions, but they are not yet part of the toolkit.

3.3 Implementation

An implementation of the toolkit has been developed in Haskell. Haskell has been chosen because a functional language allows us to keep the implementation close to the underlying theories and is more suitable for a large class of algorithm in the toolkit. For instance, the typing and projection algorithms can be straightforwardly implemented by exploiting the pattern matching featured by Haskell. Moreover, Haskell provides a convenient means to build a modular architecture; in fact, each component of Figure 1 is implemented in a different module. In addition, support of first-class functions allows for the re-use of different functions in many different contexts (for instance, to realise the parametricity of the toolkit wrt the assertion logic and use the same typing algorithm for binary and multiparty sessions).

In the following, we discuss the main implementation details of the current version of the toolkit.

Parsing. Stable parsing tools are available for Haskell. The parsers were built using Alex [1] and Happy [12]. A basic attribute grammar takes care of checking conformance of the code (e.g. basic type checking of the participants implementations is done at parsing time). From the code input, it generates an abstract syntax tree (encoded in Haskell types) which is then given to the next algorithm (*linearity* check in the multiparty case and *typing* in the binary case). The languages for global description and local processes accepted by the toolkit are very similar to the ones defined in [4, 14]. The main differences are as follows.

- Session request $\overline{a}[2..n](\tilde{s})$ and session acceptance $a[2](\tilde{s})$ are respectively written as

  $$\texttt{init:a[}P_1 \ldots P_n\texttt{]}(\tilde{s}) \quad \text{and} \quad \texttt{join:a[}P_i\texttt{]}(\tilde{s})$$

  where P_i are participant identifiers. Note that the first participant identifier in the session request primitive is the initiator's.

- The language adopted in the toolkit to represent processes requires that each branch construct is identified by a string which is then used as a prefix for the corresponding label selections; the syntax for the branch/selection constructs is respectively

  `channel&id{...}` and `channel$ [assertion] id.label`

  This allows us to simplify the typing algorithm. Indeed, without such identifiers, it would be more complex to infer which branching construct a label is referring to. We illustrate this with an example; consider the following process

  $$\begin{array}{lll} \texttt{k} \;\; \texttt{\&} \; \texttt{id} & \{ & \texttt{t}_1: \;\; \texttt{P}_1\texttt{; s \$ [-] l}_1 \texttt{ ; Q}_1 \\ & & \texttt{t}_2: \;\; \texttt{P}_2\texttt{; s \$ [-] l}_2 \texttt{ ; Q}_2 \\ & & \texttt{t}_3: \;\; \texttt{P}_3\texttt{; s \$ [-] l}_3 \texttt{ ; Q}_3 \quad \} \end{array}$$

  which branches on channel `k`; if the ith branch is selected, a label is sent on channel `s` after executing a process $\texttt{P}_i$ (assuming $\texttt{P}_i$ does not have interactions on `s`) and finally finishes with $\texttt{Q}_i$.

  Let the type of the processes interacting on `s` be

  $$\texttt{s} \oplus \{\, \texttt{l}_1 : \texttt{T}_1, \quad \texttt{l}_2 : \texttt{T}_2, \quad \texttt{l}_3 : \texttt{T}_2 \,\}$$

  To type channel `s` in the branches of the process, the algorithm needs a way to realise that all the labels $\texttt{l}_i$ belong to the same branching construct. Since label selection can be done at any place in a process (e.g. in the branches of an `if-then-else` or a branching construct) and typing is done separately in the branches, one needs a way of gathering all the labels of a same group. Using an identifier for each branching construct and using it as prefix in label selections allows the algorithm to directly know which branching construct a `select` is referring to.

- As in [4], recursive definitions for local processes take two kinds of parameters: expressions and communication channels. For example,

$$\mathtt{mu\,t}(\mathtt{e}_1 \ldots \mathtt{e}_n; \mathtt{k}_1 \ldots \mathtt{k}_m)(\mathtt{p}_1 : \mathtt{s}_1 \ldots \mathtt{p}_n : \mathtt{s}_n; \mathtt{k}'_1 \ldots \mathtt{k}'_m).\mathtt{P}$$

  is a recursive definition with n "value" parameters and m communication channel parameters. It is required that the type of each expression $\mathtt{e}_\mathtt{i}$ matches $\mathtt{s}_\mathtt{i}$. The initialisation parameters $\mathtt{e}_\mathtt{i}$ and $\mathtt{k}_\mathtt{i}$ specify the initial value for the formal parameters $\mathtt{p}_\mathtt{i}$ and $\mathtt{k}'_\mathtt{i}$. Each channel used in $\mathtt{P}$ must be one the $\mathtt{k}'_\mathtt{i}$s.

Well-assertedness. The logic used for the assertion is based on the Presburger arithmetic [8], since a decidable logic was necessary to develop an effective algorithm for checking the well-assertedness condition on global assertions. We have adopted a convenient API [20] implementing the Presburger arithmetic in Haskell. The well-assertedness algorithm analyses which participants know which variables to ensure *history sensitivity* and tests the satisfiability of the assertions as defined in [4] to ensure *temporal satisfiability* (cf. § 2). A key part of the algorithm is the verification of temporal-satisfiability in recursive definitions. It is implemented as follows. Firstly, the algorithm checks whether the invariant is satisfied by the assertions encountered previously, then it stores the formal parameters and the invariant of the recursion, and goes on with the verification of the continuation's temporal satisfiability. When a recursive call is encountered, the satisfiability of the corresponding invariant is tested in the new context (i.e. the actual parameters of the recursive call substitute the formal parameters of the recursive definition).

Note that because of the use of the API for the Presburger arithmetic, the assertions one may write are quite limited, i.e. only predicates involving integers and booleans are supported at the moment. The well-assertedness algorithm was developed apart of the Presburger arithmetic API, to ease future changes in the underlying logic used in the assertions. In case users do not want to assert their global description, all the assertions can be replaced by `[-]` which stands for `True` in the language we defined.

Projection. The projection of a global description is done participant-wise. The algorithm outputs a list of pairs (participant identifier and end-point type). If the description is asserted, then the projected assertions are computed at the same time. The output of this step is given to the typing algorithm.

Typing algorithm. The typing algorithm has been designed to be as flexible as possible. As an example, we use the same core algorithm for both binary and multiparty session typing. To make this possible, the typing algorithm is abstracted away from two functions, the *compatibility* and *composition* operation on type environments. The former operation is used for testing compatibility between two typings while the composition operation is used to compose two typing environments.[7] To type concurrent

[7] In the binary case, two typing environments Δ_0 and Δ_1 are *compatible* if for every channel typed in Δ_0 and Δ_1, their types are the *dual* of each other. In the multiparty case, Δ_0 and Δ_1 are *compatible* if they type different participants for common channels.

For dyadic session types, if a channel k is typed in two compatible environments Δ_0 and Δ_1, then the type of k in the *composition* $\Delta_0 \circ \Delta_1$ becomes $\perp$ (i.e. the interactions are internal). For multiparty session types, *composition* consists, basically, of the union of typing environments.

branches (i.e. possibly representing different participants), the algorithm first computes the type of each branch. If the types of all branches can be successfully obtained, the algorithm composes them using the parameterised operations (provided that the obtained types are compatible).

The core of the algorithm consists of a depth-first traversal of the abstract syntax tree. Each time a session initiation primitive like $\texttt{init:a}[P_1 \ldots P_n]\texttt{(}\tilde{k}\texttt{)}$ or $\texttt{join:a}[P_i]\texttt{(}\tilde{k}\texttt{)}$ is found, the algorithm types the channels $\tilde{k}$ (called *current channels*) in the rest of the tree according to the typing rules specified in the theory. Using pattern matching, it is straightforward to implement such rules so to maintain a strong connection between theory and implementation.

An interesting rule is the one for typing recursive definitions. When the algorithm encounters a recursive definition, e.g.

$$\texttt{mu}\,\texttt{t}(\tilde{\texttt{e}};\tilde{\texttt{k}})(\tilde{\texttt{s}}:\tilde{\texttt{S}};\tilde{\texttt{h}}).\texttt{P}$$

it first verifies that the types of the expressions in $\tilde{\texttt{e}}$ match the declaration of the formal parameters (i.e. $\tilde{\texttt{S}}$). The recursion variable, the formal parameters and the continuation are stored in the environment. Then, it continues the typing of $\texttt{P}$ where the current channels corresponding to the channels in $\tilde{\texttt{k}}$ are replaced by the corresponding channels in $\tilde{\texttt{h}}$.

Once a recursive call is encountered, e.g.

$$\texttt{t}(\tilde{\texttt{e}'};\tilde{\texttt{k}'})$$

the algorithm ensures, that $\texttt{P}[\tilde{\texttt{e}'}/\tilde{\texttt{s}}][\tilde{\texttt{k}'}/\tilde{\texttt{h}}]$ type checks, using the information previously stored in the environment.

In the multiparty case, when a session has been fully typed, its type is compared to the corresponding projection. This is done using a *refinement* relation that allows the tool to accept processes with types that specify a more refined behaviour that the one in the projection. For instance, a process may select less labels, weaken the predicate for branching and reception, or strengthen the predicates for selection and sending.

Code generation. The code generation is straightforward and exploits Haskell's `Chan` objects for communication channels. In Haskell, `Chan` is part of the concurrency libraries provided and is an abstract type for unbounded FIFO channels.

Figure 2 shows some example of generated code for send, receive, branching and recursion respectively.

- Send is simply translated into a call to `writeChan` which writes a new value on the specified channel. All values are serialised[8] using `show` as channels accepts only one type of value per instance.
- Receive is translated to a call to `readChan`, which reads data from a channel. When retrieving values from Haskell channels, it is necessary to cast back the string to its actual type (i.e. `read t'::(Int)`, in the example). This is needed as a Haskell compiler may not be able to infer the type of the value received. Remarkably, in our case the type is known in advance since the session was typed. The `let...in` construct of Haskell is quite useful since it allows us to bind the receive value to a new variable without having to take into account possible renaming. Indeed, nested

[8] Note that serialisation in Haskell is supported through the inheritance of the `Show` and `Read` classes. Every type inheriting the `Show` class has to implement the function `show`, from which a string representation of the object can be generated. Dually, a type which inherits `Read` defines a function `read` which can extract the data from the string representation. It is the case that `Show` and `Read` can be inherited for most of user defined types.

Send:

```
s!(``The Art...'')(t: string)[-];
```

```
writeChan s ( show (``The Art...''));
```

Receive:

```
b1?(q: int)[-];
(...)
```

```
t' <- readChan b1;
let (q) = read t'::(Int)
in
do {(...)}
```

Branch:

```
s&id{
[-] ok: (...)
[-] quit: (...)
}
```

```
let brvarid' = read brvarid::String in
case brvarid' of
``idok'' -> do {(...)}
``idquit'' -> do {(...)}
```

Recursion:

```
mu t(e1,e2;c1,c2)(p1:int,p2:bool;k1,k2).
(...)
```

```
let t p1 p2 k1 k2 =
(...)
in
t e1 e2 c1 c2
```

Figure 2: Example of generated code.

`let...in` blocks declaring the same variable names are allowed and the scope of the binding corresponds to the one used in our language.

- For branching blocks, one first reads one label on the channel, then a `case` construct implements the actual branching.
- Recursive definition are translated into a `let...in` block where the type variable `t` is defined as a function with formal parameters $\mathtt{p}_1$, $\mathtt{p}_2$, $\mathtt{k}_1$ and $\mathtt{k}_2$ and body as the continuation of the recursive definition. In the `in` part the new function is instantiated with actual parameters $\mathtt{e}_1$, $\mathtt{e}_2$, $\mathtt{c}_1$ and $\mathtt{c}_2$.

In order to make the execution of the generated code observable it is possible to make the toolkit print information each time an action such as send, receive, select and branch is invoked by a participant.

4 Examples

In this section, we give two examples of distributed protocols. The first example is taken from § 2 and illustrates how the toolkit can be used. The second example gives a more realistic protocol, including a recursive definition, which is verified by the toolkit.

4.1 Buyer-Seller protocol

We start with the example given in Section 2. Figure 3 shows the input file given to the toolkit. The first part (lines 1 - 9) represents the global description of the two-buyer protocol (G in § 2), while the second part (lines 11 - 34) gives an implementation of the participants (Seller, Buyer1, and Buyer2) in our π-calculus dialect. These processes are meant to be executed in parallel.

Buyer1 (lines 13 - 17, in Figure 3) sends the title of a book, receives its price and, then, sends its contribution to Buyer2. Buyer2 (lines 19 - 24) receives the contribution that Buyer1 is willing to make. If it is under 100, it confirms the sale to Seller and sends its address. Seller (lines 26 - 33) receives a book title, sends the book's price to Buyer1 and then wait for Buyer2 to confirm, or not, the sale.

```
1  Global[buyerex]:
2  B1 -> S: s (t: string)[-].
3  S -> B1: b1 (q: int)[q > 0].
4  B1 -> B2: b2 (c: int)[0 < c and c <= q].
5  B2 -> S: s::id{
6     [-] ok: B2 -> S: s (a: string)[-].
7             S -> B2: b2 (d: date)[-].end,
8     [-] quit: end
9     }
10
11 Process:
12 {
13 init: buyerex[B1, B2, S](s, b1, b2).
14       s!(”The Art of Computer Programming”)(t: string)[-];
15       b1?(q: int)[q > 0];
16       b2!(q)(c: int)[0 < c and c <= q];
17       end
18 |
19 join: buyerex[B2](s, b1, b2).
20       b2?(c: int)[0 < c];
21       if (c < 100)
22       then s$ [-] id.ok; s!(”my address”)(a: string)[-];
23                          b2?(d: date)[Exists q: int (0 < c and c <= q)];end
24       else s$ [-] id.quit; end
25 |
26 join: buyerex[S](s, b1, b2).
27       s?(t: string)[-];
28       b1!(99)(q: int)[q > 50];
29       s&id{
30        [-] ok:   s?(a: string)[-];
31                  b2!(11/12/2010)(d: date)[-];end,
32        [-] quit: end
33       }
34 }
```

Figure 3: Buyer-Seller protocol.

The interactions are decorated with the assertions presented in § 2. However because of the limitation imposed by the API for the Presburger arithmetic, it is currently not possible to define assertion on strings, such as `t` $\neq$ ””.

In Figure 3, notice that Seller guarantees a stronger condition for the sending on $b1$ (see line 28) compared to its counterpart in the global description (line 4); this is made possible by the refinement relation defined on local assertions (cf. [4]).

When given the content of Figure 3 as input, the implementation outputs the text given in Figure 4 and generates Haskell code. The toolkit first signals (lines 1- 2) that the parsing was successful and the global description is well-asserted (and linear). Then, the projections of the protocol (lines 6 - 18) on each of the participants are given ($B1$, $B2$ and S standing for Buyer1, Buyer2 and Seller, respectively). Finally, the types of the processes, prefixed by the session headings, are printed (lines 22 - 32), which match the projections output before. Notice that the predicates in the projection of Seller (line 13) and in the type of its implementation (line 29) are compatible since $q > 50 \implies q > 0$.

When there is a mismatch between the projections and the types inferred from the participants implementation, the toolkit shows the problematic projection and inferred type. For instance, if one changes the first interaction of Buyer1 (line 14 in Figure 3) to `s!(112)(t:int)[-];`, the tool outputs:

```
Parse Successful.
WellAsserted? [True]

Projections:
*buyerex:
[s!<t: string>[true];b1?<q: int>[q > 0];b2!<c: int>[0 < c && c <= q];end]@B1

[b2?<c: int>[Exist q: int st. (0 < c && c <= q && q > 0)];s${
  [true]ok: s!<a: string>[true];b2?<d: date>[Exist q: int st. (0 < c && c <= q && q > 0)];end,
  [true]quit: end
}]@B2

[s?<t: string>[true];b1!<q: int>[q > 0];s&{
  [Exist c: int st. (0 < c && c <= q && q > 0)]
      ok: s?<a: string>[Exist c: int st. (0 < c && c <= q && q > 0)];b2!<d: date>[true];end,
  [Exist c: int st. (0 < c && c <= q && q > 0)]
      quit: end
}]@S


Types:
buyerex[s, b1, b2]: s!<t: string>[true];b1?<q: int>[q > 0];b2!<c: int>[0 < c && c <= q];end@B1

buyerex[s, b1, b2]: b2?<c: int>[0 < c];s${
  [true]ok: s!<a: string>[true];b2?<d: date>[Exist q: int st. (0 < c && c <= q)];end,
  [true]quit: end
}@B2

buyerex[s, b1, b2]: s?<t: string>[true];b1!<q: int>[q > 50];s&{
  [true]ok: s?<a: string>[true];b2!<d: date>[true];end,
  [true]quit: end
}@S
```

Figure 4: Output for Buyer-Seller.

```
Local type doesn't match projection for B1!
Type:        s!<t:int> [true];b1?<q:int>[q>0];b2!<c:int>[0<c && c<=q];end
Projection:  s!<t:string>[true];b1?<q:int>[q>0];b2!<c:int>[0<c && c<=q];end
```

In addition, if we set the book's price to `0` in Seller, i.e. we change line 28 to `b1!(0)(q:int)[q>50];` in Figure 3. The tool signals that the assertion is not satisfiable:

```
[Typing-Send] Assertion not satisfiable:  true => 0 > 50.
```

Meaning that in the current assertion environment (which is empty, i.e. equals `true`), it is not true that the sent value guarantees the assertion[9].

4.2 A guessing game protocol

We use a protocol resembling a simple game where a Generator (`G`) chooses a natural number which has to be discovered in less that 10 attempts by a Player (`P`), according to the hints given by a Server (`S`). The code representing the protocol and implementing the participants is given in Figure 5. The first part (lines 1-9) of Figure 5 represents the global description of the game. First, `G` chooses a number $n > 0$ and sends it to `S`. Then, `P` sends a first attempt (line 3) to `S`. The recursive definition has two parameters: r,

[9] In the near future, such error messages will accompanied by a line number.

```
Global [a]:
G -> S: k (n: int)[n > 0].
P -> S: l (x: int)[x > 0].
mu rec (x,0) (r:int @ {P,S}, cpt: int @{P,S}) [r > 0 and cpt <= 10].
S -> P:h::id {
     [r > n and cpt < 10] less: P -> S: l (y: int)[y > 0]. rec(y,cpt+1),
     [r < n and cpt < 10] greater: P -> S: l (z: int)[z > 0]. rec(z,cpt+1),
     [r == n] win: end,
     [cpt >= 10] lose: end }

Process:
{
init: a[G,S,P](k,h,l).
      k!(15)(n: int)[n > 0]; end
|
join: a[S](k,h,l).
      k?(n: int)[n > 0];
      l?(x: int)[x > 0];
      mu rec (x,0;h,l) (r: int, cpt: int; hf,lf).
      if(cpt >= 10)
      then hf $ [cpt >= 10] id.lose; end
      else if (r > n)
           then hf $ [r > n and cpt < 10] id.less;
                lf?(y: int)[y > 0 and r > n and x > 0 and n > 0 and r > 0 and cpt < 10];
                rec(y,cpt+1;hf,lf)
           else if (r < n)
                then hf $ [r < n and cpt < 10] id.greater;
                     lf?(z: int)[z > 0 and r < n and x > 0 and n > 0 and r > 0 and cpt < 10];
                     rec(z,cpt+1;hf,lf)
                else hf $ [r == n] id.win; end
|
join: a[P](k,h,l).
      l!(11)(x: int)[x > 0];
      mu rec (x,0;h,l) (r: int, cpt:int; hf,lf).
      hf&id{
       [r > 1] less: lf!(r - 1)(y: int)[y > 0]; rec(y,cpt+1;hf,lf),
       [r >= 0] greater: lf!(r + 1)(z: int)[z > 0]; rec(z,cpt+1;hf,lf),
       [-] win: end,
       [-] lose: end }
}
```

Figure 5: Protocol description and implementation of a Guessing game.

the current attempt, initially assigned with value x, and `cpt` the attempt counter, initially assigned with value 0. Depending on whether r is less than, greater than or equal to n, `S` sends the corresponding label to `P`. If `P` guesses the correct number in less that 10 attempts, `P` wins and the session ends. Otherwise, the session ends after 10 attempts, and `P` looses.

The second part of Figure 5 gives an implementation of each participants. In our example, `G` chooses always 15 for n and `S` is faithful to the global description (i.e. it is not lying to `P`). The participant `P` always starts with 11 as its first guess, then if `S` says *less* (resp. *greater*), `P` tries the number minus (resp. plus) one. Remark that in both recursive definitions of participant `S` and `P`, two formal parameters are used for the communication channels, i.e. `hf` and `lf` (alternatively, one could have used the names `h` and `l` again).

The code in Figure 5 can be given as input to the toolkit, which then returns the content of Figure 6. Lines 4-22 of Figure 6 are the projections for participants `G`, `P`, and `S` computed from the global descrip-

```
Parse Successful.
WellAsserted? [True]
Projections:
*a:
[k!<n: int>[n>0];end]@G

[l!<x: int>[x>0];mu rec(x, 0){r: int, cpt: int}[r>0 && cpt<=10].h&{
[Exist n: int st. (r>n && cpt<10 && x>0 && n>0 && r>0 && cpt<=10)]
   less: l!<y: int>[y>0];rec(y, cpt+1),
[Exist n: int st. (r<n && cpt<10 && x>0 && n>0 && r>0 && cpt<=10)]
   greater: l!<z: int>[z>0];rec(z, cpt+1),
[Exist n: int st. (r = n && x>0 && n>0 && r>0 && cpt<=10)]win: end,
[Exist n: int st. (cpt >= 10 && x>0 && n>0 && r>0 && cpt<=10)]lose: end
}]@P

[k?<n: int>[n>0];l?<x: int>[x>0 && n>0];mu rec(x, 0){r: int, cpt: int}[r>0 && cpt<=10].h${
[r>n && cpt<10]
   less: l?<y: int>[y>0 && r>n && cpt<10 && x>0 && n>0 && r>0 && cpt<=10];rec(y, cpt+1),
[r<n && cpt<10]
   greater: l?<z: int>[z>0 && r<n && cpt<10 && x>0 && n>0 && r>0 && cpt<=10];rec(z, cpt+1),
[r = n]win: end,
[cpt >= 10]lose: end }]@S

Types:
a[k, h, l]: k!<n: int>[n>0];end@G

a[k, h, l]: k?<n: int>[n>0];l?<x: int>[x>0];mu rec(x, 0){r: int, cpt: int}[true].h${
[cpt >= 10]lose: end,
[r>n && cpt<10]
   less: l?<y: int>[y>0 && r>n && x>0 && n>0 && r>0 && cpt<10];rec(y, cpt+1),
[r<n && cpt<10]
   greater: l?<z: int>[z>0 && r<n && x>0 && n>0 && r>0 && cpt<10];rec(z, cpt+1),
[r = n]win: end }@S

a[k, h, l]: l!<x: int>[x>0];mu rec(x, 0){r: int, cpt: int}[true].h&{
[r>1]  less: l!<y: int>[y>0];rec(y, cpt+1),
[r >= 0] greater: l!<z: int>[z>0];rec(z, cpt+1),
[true]   win: end,
[true]   lose: end }@P
```

Figure 6: Output for Guessing game.

tion of the guessing game. Remarkably, the projection for `G` is not a recursive type since `G` is not involved in the recursion. The rest of Figure 6 consists of the local types for `G`, `S`, and `P`. These have been inferred from the participants implementation. Notice that the invariants of recursive definitions in local types are set to `true`. This is allowed since each local type must be a refinement of its corresponding projection, which is *well-asserted* (i.e. it is guaranteed that the invariant is respected). The toolkit also generates Haskell code implementing the participants, this can be compiled and ran as it is.

5 On featuring modularity

In this section we describe how modularity is featured in our implementation. We mainly envisage four possible degrees of modularity discussed below.

Notation. All inputs and outputs of the implementation (e.g. global assertions, projections, etc.) are encoded in Haskell data types that specify an abstract syntax of the supported languages. This allows

II	$n_1 < n_2$ and
	$n_i = p_i \to p : k_i$ (i=1,2)
		`dep_ii :: Prefix -> Prefix -> Bool`
		`dep_ii (Prefix p1 p k1) (Prefix p2 q k2)`
		`| k1 /= k2 = (q == p)`
		`dep_ii (Prefix p1 p k1) (Prefix p2 q k2)`
		`| k1 == k2 = (p1 == p2) && (p == q)`
		`dep_ii _ _ = False`

IO	$n_1 < n_2$,
	$n_1 = p_1 \to p : k_1$ and
	$n_2 = p \to p_2 : k_2$
		`dep_io :: Prefix -> Prefix -> Bool`
		`dep_io (Prefix p1 p k1) (Prefix q p2 k2)`
		`| k1 /= k2 = (q == p)`
		`dep_io _ _ = False`

OO	$n_1 < n_2$,
	$n_i = p \to p_i : k_i$ (i=1,2)
		`dep_oo :: Prefix -> Prefix -> Bool`
		`dep_oo (Prefix p p1 k1) (Prefix q p2 k2)`
		`| k1 == k2 = (q == p)`
		`dep_oo _ _ = False`

where each dep_** p_1 p_2 assumes that $p_1 < p_2$.

Figure 7: Dependency relations implementation.

to possibly support other notations than the ones originally considered. Notably, the implementation exhibits four data structures to/from which other languages can be translated: *global assertions*, *end-point assertions* (projections), *π-calculus dialect* (participants implementation), *assertion logic*.

Languages. An important requirement of our modular approach, is that it has to feature the parametrisation of the implementation with respect to the languages used to describe the distributed interactions and the associated type systems. For instance, the theory described in [4] abstracts from the actual logical language used to express asserted interactions. Notably, depending on the chosen language, ad-hoc optimisations can be applied.

Algorithms. The tool consists of several algorithms that can be used in a modular way (i.e. the users will be able to choose which algorithms they need). For instance, several algorithms are described in [4, §3.3] to check well-assertedness of assertions; in fact, depending on the adopted logic several formulae manipulation could be applied. Notably, the well-assertedness notion defined in [4] could be replaced by equivalent ones which exploit optimisations on logical formulae. In this way, one could use the simple algorithms in theoretical experimentation on simple scenarios, while more efficient algorithms could be used when considering realistic cases.

Theory. Since the toolkit is developed in a functional language, it allows the theory to be straightforwardly mapped into the programming language. This means that, most of the time, when one wants to change a rule or a definition this can be done by changing only a few lines of code. We illustrate this with an example. The definition of the *dependency relations* ([14, §3.3]) is translated as shown in Figure 7. In the conclusions of [14], the authors comment the adaptation of the theory to support synchrony. Following their idea, this could be done by taking into account output-output dependencies between different names and adding a new dependency from output to input. In our implementation this change could be implemented simply by a few modifications of the code in Figure 7. In particular, we would relax the condition `k1 == k2` in `OO` and add a new `dep_OI` function for output-input dependencies, similar to the other rules.

6 Related work

Implementations of session types. A few other implementations of the theories based on session types exist. We describe them in the following and compare them with our work.

Hu et al. [15] present an implementation of an extension of Java to support distributed programming based on binary session types. The implementation consists of three main components: an extension of the language to specify protocols, a pre-processor to translate the specification to Java and a runtime library which implements the communication channels and runtime checks. Neubauer et al. [18] propose an encoding of session types in Haskell (in terms of type classes with functional dependencies). This implementation is quite limited, e.g. it is restricted to one channel. Pucella et al. [21] proposes an implementation of session types in Haskell which infers protocols automatically. They also claim that their approach can be applied to other polymorphic, typed languages such as ML and Java. Sackman et al. [22] propose a full fledged implementation of binary session types in Haskell in the form of an API. The type inference systems of [21] and [22] are based on Haskell's type system, i.e. they do not directly implement the typing rules defined in the theories for session types.

Note that our work tackled the theories for *multi*party sessions, while all the other implementations based on session types consider only two-party sessions.

Another difference between these works and ours is that we have mainly focused on the tools accompanying the theories and not on the integration of session types in a programming language. The part of our work which can be directly compared with these implementations is our Haskell code generator. However in its current state, it is only a proof-of-concept of the usefulness of the verification tool that are situated upstream in the toolkit. If we were to generate code for more realistic applications, we would, e.g., use network connections instead of Haskell `Chan`. This would notably require a good runtime library to support, for instance, delegation of channels.

Other theories. We believe that our implementation could also encompass other theories for distributed interaction. For instance, a sub-typing relation such as the one defined by Gay et al. [11] could be easily added to the toolkit by using the *refinement relation* in a new version of the two parameterisable functions for the binary typing algorithm. In particular, the *compatibility* function should be relaxed so to enable compatibility of types. Similarly, to support *union types* such as in [3], it would requires to add a few constructs to the accepted language (i.e. to allow the specification of types such as "`int ∨ real`") and a new compatibility function which allows a type containing a union of types to be compatible with one having one element of it.

On the other hand, theories such as the one developed for the *conversation types* [7] would be much harder to integrate in our toolkit. For instance, the notion of (nested) conversation would require to adapt the language and most of the typing rules. In addition, the form of the types in [7] is quite different from most of the other theories that we have focused on. However, provided a good mapping between the theory of multiparty session types [14] and conversation types in terms of channels and conversations, we believe that an adapted version of *apartness* and *merge* relations could be part of the verifications done in the *compatibility* function. These two relations are used to test the compatibility of two conversation types. This would enable, to some extent, the verification of the compatibility of participants without having a global description.

Applications. The toolkit aims to support a development methodology of communication-centric software based on formal theories of distributed interactions where global and local "views" are used to verify properties of systems. It is worth mentioning that a similar methodology has recently been adopted in the SAVARA project [23] where *global* and *local* models are used in the development process to validate requirements against implementations. Noteworthy, SAVARA combines state of the art design techniques with session types and provides an open environment where tools based on formal theories can be integrated. We are considering the integration of (part of) our toolkit in SAVARA. In particular, our toolkit could be used to project the choreography model onto individual services. Namely, SAVARA uses WS-CDL [16] to represent the choreography model, from which it can generates WS-BPEL [2] implementations of individual services and BPMN [24] diagrams that may be used to guide the implementation.

We believe the integration of our tool in SAVARA is feasible and would require the following three main components. Firstly, a mapping from WS-CDL to global assertions. This should be quite straightforward as the global types are very similar to WS-CDL. However, the support for assertions may require more work as it would demand an extension of WS-CDL with pre-/post-conditions on messages. Secondly, we need to translate the projections output by our tool to BPEL, BPMN and/or another language, such as the ones used in SAVARA to design/implement services. Finally, we need some mechanisms to type check the conformance of services against a choreography. This means that we need a tool which translates the (partly) implemented services to a language compliant with (the abstract syntax of) our π-calculus-like language.

Technically, these three mappings should be relatively easy to implement as it amounts to transform an XML tree to Haskell data types (and vice-versa). However, careful attention is needed when including the assertions in the notations supported by SAVARA.

7 Conclusions

We have described the architecture and the main implementation aspects of a toolkit for distributed interactions. The distinguished design principles of the architecture are flexibility and modularity, to meet the future changes in the theories underlying the toolkit.

The toolkit relies on Cooper's decision procedure for the Presburger Arithmetic [8], which has a super-exponential complexity [19] on the size of the logic formulae. This means that the current version of the algorithm might take a (very) long time to check protocol with very long predicates. However, all the algorithms of the toolkit itself have a polynomial complexity on the size of the global descriptions and participant implementations. In addition, the toolkit is designed in such a way that the underlying logic can be changed easily. Therefore, if a more efficient logic API appears it can easily replace the current one. To give an idea of the current efficiency of the toolkit, it takes more or less one second to process the examples of Section 4, but a global description decorated with a 80 line-long assertion (including quantifiers) takes about 11 minutes.[10]

Our toolkit is currently under development and we are considering several ways of enhancing it. We are considering of using Haskell as basis for the input languages. For instance, one could use Haskell as the language for expressions used, notably, in call such as $s!(\text{"}\mathit{myString}\text{"})$. In addition, Haskell types could be used as basis for the types permitted by the tool.

Extending the input languages to Haskell might reveal a delicate extension. In fact, on the one hand this would provide the possibility of expressing interesting predicates for the assertions, on the other hand, global assertions require the logic used to assert interactions to be decidable. In fact, increasing

[10]All the examples mentioned in this paper have been ran in Ubuntu 10.04 with GHC 6.12.1, on an Intel Core 2 Duo @ 3.16 Ghz machine.

the expressivity of the input language by allowing Haskell types might compromise the decidability of the logic.

We intend to study the feasibility of a more realistic code generator, which uses network connections (i.e. distributed Haskell) instead of FIFO channels; and produces assertion monitor to ensure runtime checking of assertions.

Another interesting implementation perspective would be to integrate the algorithms featured by our toolkit in a full-fledged programming language (e.g. Java, similarly to [15]). For example, we conjecture that global assertions could be implemented in two phases. Firstly, a language-independent part could take care of the verification and validation tools which guarantee the good behaviour of programs (i.e. the implementation of the toolkit described here). Secondly, a language-dependent part could extend a programming language by developing an API which implements the communication primitives (session initiation, value passing, branch/select and delegation); while a translator to an abstract language (such as the π-calculus-like language we use) links the API to our toolkit (see faded boxes in Figure 1).

The toolkit turned out to be a remarkably useful tool to identify which part of the theory relies on the programmers. In particular, while designing tests for the toolkit, we noticed that making a global description *well-asserted* was often non-trivial. This led us to design algorithms [5] which, if applicable, solve well-assertedness problems automatically, and give indications to the programmers on where problems originate. We plan to add these algorithms to the toolkit in the future.

References

[1] *Alex: A lexical analyser generator for Haskell*. `http://www.haskell.org/alex`.

[2] Alexandre Alves, Assaf Arkin, Sid Askary, Ben Bloch, Francisco Curbera, Yaron Goland, Neelakantan Kartha, Sterling, Dieter König, Vinkesh Mehta, Satish Thatte, Danny van der Rijn, Prasad Yendluri & Alex Yiu (2006). *Web Services Business Process Execution Language Version 2.0*. OASIS Committee Draft. Available at `http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf`.

[3] Lorenzo Bettini, Sara Capecchi, Mariangiola Dezani-Ciancaglini, Elena Giachino & Betti Venneri (2008): *Session and Union Types for Object Oriented Programming*. *LNCS* , pp. 659–680Available at `http://dx.doi.org/10.1007/978-3-540-68679-8_41`.

[4] Laura Bocchi, Kohei Honda, Emilio Tuosto & Nobuko Yoshida (2010): *A Theory of Design-by-Contract for Distributed Multiparty Interactions*. In Gastin & Laroussinie [10], pp. 162–176. Available at `http://dx.doi.org/10.1007/978-3-642-15375-4_12`.

[5] Laura Bocchi, Julien Lange & Emilio Tuosto (2011): *Amending Contracts for Choreographies*. In: *ICE*. To appear.

[6] Roberto Bruni, Ivan Lanese, Hernán Melgratti & Emilio Tuosto (2008): *Multiparty sessions in SOC*. In: Doug Lea & Gianluigi Zavattaro, editors: *COORDINATION'08*, *LNCS* 5052, Springer, pp. 67–82. Available at `http://www.di.unipi.it/~bruni/publications/multiparty.ps.gz`.

[7] Luís Caires & Hugo Torres Vieira (2009): *Conversation Types*. In: *ESOP'09*, Springer, Berlin, Heidelberg, pp. 285–300. Available at `http://dx.doi.org/10.1007/978-3-642-00590-9_21`.

[8] D. C. Cooper (1972): *Theorem proving in arithmetic without multiplication*. *Machine Intelligence* 7 , pp. 91–99Available at `http://citeseerx.ist.psu.edu/showciting?cid=697241`.

[9] Pierre-Malo Deniélou & Nobuko Yoshida (2010): *Buffered Communication Analysis in Distributed Multiparty Sessions*. In Gastin & Laroussinie [10], pp. 343–357. Available at `http://dx.doi.org/10.1007/978-3-642-15375-4_24`.

[10] Paul Gastin & Francoise Laroussinie, editors (2010): *CONCUR 2010 – Concurrency Theory*, *Lecture Notes in Computer Science* 6269. Springer.

[11] Simon Gay & Malcolm Hole (1999): *Types and Subtypes for Client-Server Interactions*. In: *Proceedings of the 1999 European Symposium on Programming, number 1576 in Lecture Notes in Computer Science*, Springer, pp. 74–90. Available at http://dx.doi.org/10.1007/3-540-49099-X_6.

[12] *Happy: The Parser Generator for Haskell*. http://www.haskell.org/happy.

[13] Kohei Honda, Vasco T. Vasconcelos & Makoto Kubo (1998): *Language Primitives And Type Discipline For Structured Communication-Based Programming*. In: *In ESOP, volume 1381 of LNCS*, Springer, pp. 122–138. Available at http://dx.doi.org/10.1007/BFb0053567.

[14] Kohei Honda, Nobuko Yoshida & Marco Carbone (2008): *Multiparty asynchronous session types*. In: *POPL*, ACM, New York, NY, USA, pp. 273–284. Available at http://doi.acm.org/10.1145/1328438.1328472.

[15] Raymond Hu, Nobuko Yoshida & Kohei Honda (2008): *Session-based distributed programming in Java*. *ECOOP, Springer LNCS* 5142, pp. 516–541. Available at http://dx.doi.org/10.1007/978-3-540-70592-5_22.

[16] Nickolas Kavantzas, David Burdett, Greg Ritzinger, Tony Fletcher, Yves Lafon & Charlton Barreto (2005). *Web Services Choreography Description Language Version 1.0*. World Wide Web Consortium, Candidate Recommendation CR-ws-cdl-10-20051109. Available at http://www.w3.org/TR/2005/CR-ws-cdl-10-20051109.

[17] Julien Lange. *VOSENID: A Modular Toolkit for Distributed Interactions*. http://www.cs.le.ac.uk/people/jl250/tools.

[18] Matthias Neubauer & Peter Thiemann (2004): *An Implementation of Session Types*. In: *In PADL, volume 3057 of LNCS*, Springer, pp. 56–70. Available at http://dx.doi.org/10.1007/978-3-540-24836-1_5.

[19] Derek C. Oppen (1978): *A $2^{2^{2^{pn}}}$ upper bound on the complexity of Presburger Arithmetic*. *Journal of Computer and System Sciences* 16(3), pp. 323 – 332. Available at http://www.sciencedirect.com/science/article/B6WJ0-4B4RN0K-D0/2/0b53fc6b4ea1e8d83d4a7cacef229610.

[20] *presburger: Cooper's decision procedure for Presburger arithmetic*. http://hackage.haskell.org/package/presburger.

[21] Riccardo Pucella & Jesse A. Tov (2008): *Haskell session types with (almost) no class*. In: *Haskell '08: Proceedings of the first ACM SIGPLAN symposium on Haskell*, ACM, New York, NY, USA, pp. 25–36. Available at http://doi.acm.org/10.1145/1411286.1411290.

[22] Matthew Sackman & Susan Eisenbach (2008): *Session Types in Haskell: Updating Message Passing for the 21st Century*. Technical Report, Imperial College London, Department of Computing. Available at http://hdl.handle.net/10044/1/5918.

[23] *SAVARA and the "Testable Architecture" Methodology*. http://www.jboss.org/savara.

[24] S.A. White (2004): *Introduction to BPMN*. Technical Report, Object Management Group. Available at http://www.bpmn.org/Documents/Introduction_to_BPMN.pdf.