A Session Subtyping Tool

Lorenzo Bacchiani, Mario Bravetti, Julien Lange, Gianluigi Zavattaro

A Session Subtyping Tool

Lorenzo Bacchiani[1] Mario Bravetti[2] Julien Lange[3] Gianluigi Zavattaro[2]

[1] University of Bologna, Italy
[2] Department of Computer Science and Engineering & Focus Team, INRIA University of Bologna, Italy
[3] Royal Holloway, University of London, Egham, UK

Abstract. Session types are becoming popular and have been integrated in several mainstream programming languages. Nevertheless, while many programming languages consider asynchronous FIFO channel communication, the notion of subtyping used in session type implementations is the one defined by Gay and Hole for synchronous communication. This might be because there are several notions of asynchronous session subtyping, these notions are usually undecidable, and only recently sound (but not complete) algorithmic characterizations for these subtypings have been proposed. But the fact that the definition of asynchronous session subtyping and the theory behind related algorithms are not easily accessible to non-experts may also prevent further integration. The aim of this paper, and of the tool presented therein, is to make the growing body of knowledge about asynchronous session subtyping more accessible, thus promoting its integration in practical applications of session types.

1 Introduction

In recent years, session types have been integrated into several mainstream programming languages (see, e.g., [15,25,26,19,24,1,23]) where they specify the pattern of interactions that each endpoint must follow, i.e., a communication protocol. All of these practical applications show a good level of maturity of the session type theory, but there are still some limitations. In particular, the notion of subtyping considered in such tools usually assumes synchronous communication channels, while, in many cases, communication takes place over asynchronous point-to-point FIFO channels (where outputs are non-blocking). In this setting, the emitted messages are stored inside channels, and there may be an arbitrary delay between an output (on an endpoint) and the corresponding input (on the opposite endpoint). The impact on session subtyping of these aspects related with asynchronous communication has been initially studied in [21,20,12], but the notions of subtyping proposed therein were subsequently proved to be undecidable [6,18]. Only recently, sound (but not complete) algorithms for asynchronous session subtyping have been proposed [7,5,9]. However, the theory behind asynchronous session types (see [11] for a gentle introduction) and related algorithms is rather intricate and this could limit their dissemination in the research community, as well as their adoption in practical applications.

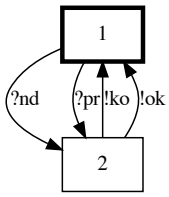
The aim of this paper, and of the tool that we introduce, is to make the growing body of knowledge about asynchronous session subtyping more accessible. More precisely, we present in an uniform and intuitive way various notions of (a)synchronous session subtyping that were presented in the literature following different formalisms, e.g., types in [9] or communicating finite-state machines in [5]. Our tool integrates several algorithms for checking subtyping that can be invoked from an easy-to-use Python GUI. This interface allows the user to input, using standard session type syntax, two types: the candidate subtype and supertype. The tool automatically generates the graphical representation of these session types as communicating finite-state machines [3]. It is also possible to execute on them the desired subtyping algorithm(s). The tool has been implemented in a modular way, and it is possible to easily include several subtyping algorithms, simply by customizing a JSON configuration file. In the current version, we consider: two algorithms from [17] for synchronous session subtyping (based on Gay and Hole's [14] and Kozen et al.'s [16] algorithms), a sound algorithm for checking (orphan message free) asynchronous session subtyping [5], and a sound algorithm for checking fair asynchronous session subtyping [9]. The implementations of these algorithms, besides returning a verdict about subtyping of the two types, also return a graphical representation of the so-called *subtyping simulation game*: i.e., the procedure to check that each relevant input/output action that can be performed by the candidate subtype has a corresponding matching action in the candidate supertype. This graphical representation is helpful to understand the reason behind the given verdict. The original command line Haskell implementations of the algorithms in [17,5,9] have been adapted and integrated by: (*i*) uniformizing their graphical notation/colors (e.g., *inner/outer* states represented as rectangles, with the initial one being thicker, error ones being red, content of outer ones being blue, etc. . .), (*ii*) reimplementing the synchronous algorithm so to also generate the simulation graph, (*iii*) completely rewriting, in the fair asynchronous algorithm, the controllability check (existance of a compliant peer, see Section 2.1) and (*iv*) error detection with generation of red states for all algorithms, (*v*) pre-transforming inputted types with a Python ANTLR4 parser that produces a common raw syntax.

Synopsis. Section 2 recalls basic notions about session subtyping using tool-simulated examples and Section 3 describes the functionalities of the tool. Finally, in Section 4 we conclude the paper.

The tool sources/binaries are available at [2].

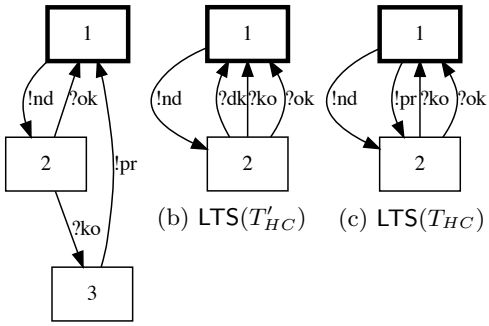
2 Session Subtyping

We first recall the syntax of session types and their automata representation in the style of communicating finite-state machines (CFSM) [3]. We then show how our tool can generate simulation graphs for supported session subtyping relations: synchronous [17], asynchronous [5] and fair asynchronous subtyping [9]. In the asynchronous cases automata are assumed to communicate over unbounded FIFO channels as for CFSMs.



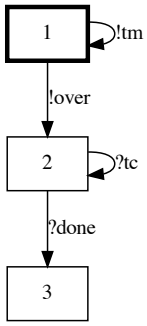
(a) $\mathsf{LTS}(T_{HS})$

Fig. 1. Hospital server.



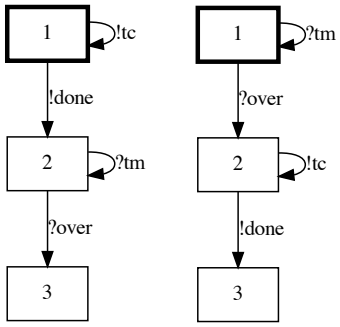
(a) $\mathsf{LTS}(T''_{HC})$ (b) $\mathsf{LTS}(T'_{HC})$ (c) $\mathsf{LTS}(T_{HC})$

Fig. 2. Hospital clients.



(a) $\mathsf{LTS}(T_{SS})$

Fig. 3. Satellite protocol server.



(a) $\mathsf{LTS}(T'_{SC})$ (b) $\mathsf{LTS}(T_{SC})$

Fig. 4. Satellite protocol clients.

2.1 Session Types and their Automata Representation

The formal syntax of two-party session types is given below. Notice that we follow the simplified notation used in, e.g., [13,6,9], which abstracts away from data carried by messages (payloads). This is done in order to focus on the key aspects of the session subtyping problem (as we will see co/contra-variance of output/input and output anticipation): passing data or channels (delegation) are features that we deem orthogonal to such a problem.

Definition 1 (Session Types). *Given a set of label names $\mathcal{L}$, ranged over by l, the syntax of two-party session types is given by the following grammar:*

$$T \;::=\; \oplus\{l_i;T_i\}_{i\in I} \;\mid\; \&\{l_i;T_i\}_{i\in I} \;\mid\; \mu\mathbf{X}.T \;\mid\; \mathbf{X} \;\mid\; \mathbf{end}$$

where $I\neq\emptyset$ and $\forall i\neq j\in I.\ l_i\neq l_j$.

Type $\oplus\{l_i;T_i\}_{i\in I}$ represents an internal choice among *outputs*, specifying that the chosen label name $l_i\in\mathcal{L}$ is sent and, then, continuation T_i is executed. $\&\{l_i;T_i\}_{i\in I}$ represents, instead, an external choice among *inputs*, specifying that,

once a label name $l_i \in \mathcal{L}$ is received, continuation T_i takes place. Types $\mu\mathbf{X}.T$ and $\mathbf{X}$ denote standard recursion constructs. We assume recursion to be guarded, i.e., in $\mu\mathbf{X}.T$ the recursion variable $\mathbf{X}$ occurs only after receving or sending a label. Type **end** denotes the end of the interaction. Session types are closed, i.e., all recursion variables $\mathbf{X}$ occur under the scope of a corresponding binder $\mu\mathbf{X}.T$.

In the tool we graphically represent the behaviour of a session type T as a Labeled Transition System (LTS), see, e.g., Figures 1, 2, 3 and 4. Following the notation of CFSMs, we denote a LTS by $(Q, q_0, \rightarrow)$, with Q being a set of states, q_0 the initial state and $\rightarrow$ a transition relation over $Q \times (\{!,?\} \times \mathcal{L}) \times Q$, with label "$!\,l$" representing output on l and label "$?\,l$" representing input on l.

We use $\mathsf{LTS}(T)$ to denote the LTS of type T. Let $\mathcal{T}$ be the set of all session types T. We define transition relation $\longrightarrow \subseteq \mathcal{T} \times (\{!,?\} \times \mathcal{L}) \times \mathcal{T}$, as the least transition set satisfying the following rules

$$\oplus\{l_i; T_i\}_{i \in I} \xrightarrow{!\,l_i} T_i \quad i \in I \qquad \&\{l_i; T_i\}_{i \in I} \xrightarrow{?\,l_i} T_i \quad i \in I \qquad \frac{T\{\mu\mathbf{X}.T/\mathbf{X}\} \xrightarrow{\ell} T'}{\mu\mathbf{X}.T \xrightarrow{\ell} T'}$$

with label ℓ ranging over $\{!,?\} \times \mathcal{L}$. Notice that a state **end**, called *termination state*, has no outgoing transitions. Given a session type T we define $\mathsf{LTS}(T)$ as being $(Q_T, T, \rightarrow_T)$, where: Q_T is the set of terms T' which are reachable from T according to $\longrightarrow$ relation and $\rightarrow_T$ is defined as the restriction of $\longrightarrow$ to $Q_T \times (\{!,?\} \times \mathcal{L}) \times Q_T$.

Notice that in general an LTS may express more behaviours than the ones described by session types: it can include non-deterministic and mixed choices, i.e. choices including both inputs and outputs. Here we only consider LTSs $(Q, q_0, \rightarrow)$ such that $\exists T \in \mathcal{T}.\, \mathsf{LTS}(T) = (Q, q_0, \rightarrow)$.

Example 1. As an example of session types we consider the Hospital server from [5]:

$$T_{HS} = \mu\mathbf{X}.\,\&\{nd; \oplus\{ko; \mathbf{X},\ ok; \mathbf{X}\},\ pr; \oplus\{ko; \mathbf{X},\ ok; \mathbf{X}\}\}$$

Figure 1 shows $\mathsf{LTS}(T_{HS})$ as produced by our tool. The server T_{HS} expects to receive two types of messages: *nd* (next patient data) or *pr* (patient report). Then it may send either *ok* or *ko*, indicating whether the evaluation of received data was successful or not, and it loops.

We now define the *dual* of a session type T, written $\overline{T}$. $\overline{T}$ is inductively obtained from T as follows: $\overline{\oplus\{l_i; T_i\}_{i \in I}} = \&\{l_i; \overline{T}_i\}_{i \in I}$, $\overline{\&\{l_i; T_i\}_{i \in I}} = \oplus\{l_i; \overline{T}_i\}_{i \in I}$, $\overline{\mathbf{end}} = \mathbf{end}$, $\overline{\mathbf{X}} = \mathbf{X}$, and $\overline{\mu\mathbf{X}.T} = \mu\mathbf{X}.\overline{T}$. For example, the dual of the Hospital server T_{HS} is:

$$\overline{T_{HS}} = \mu\mathbf{X}.\,\oplus\{nd; \&\{ko; \mathbf{X},\ ok; \mathbf{X}\},\ pr; \&\{ko; \mathbf{X},\ ok; \mathbf{X}\}\}$$

Example 2. We now consider examples of session types that are clients of the Hospital service: an "ideal" client T_{HC} and two specific ones T'_{HC} and T''_{HC}, respectively.

$$T_{HC} = \overline{T_{HS}} = \mu\mathbf{X}.\,\oplus\{nd; \&\{ko; \mathbf{X},\ ok; \mathbf{X}\},\ pr; \&\{ko; \mathbf{X},\ ok; \mathbf{X}\}\}$$
$$T'_{HC} = \mu\mathbf{X}.\,\oplus\{nd; \&\{ko; \mathbf{X},\ ok; \mathbf{X},\ dk; \mathbf{X}\}\}$$
$$T''_{HC} = \mu\mathbf{X}.\,\oplus\{nd; \&\{ko; \mathbf{X},\ ok; \oplus\{pr; \mathbf{X}\}\}\}$$

Figure 2 shows $\mathsf{LTS}(T''_{HC})$, $\mathsf{LTS}(T'_{HC})$ and $\mathsf{LTS}(T_{HC})$,[4] as produced by our tool.

The "ideal" client T_{HC} is simply the dual of the Hospital server: first it may send two types of messages *nd* or *pr*, then it expects to receive either *ok* or *ko*. In general, a client that is *compliant* with the Hospital server is a type such that: (*i*) each message sent by the client (resp. server) can be received by the server (resp. client), and (*ii*) neither the server nor the client blocks in a receive. For example, the client T'_{HC}, a slightly modified version of T_{HC} that may send *nd* only and expects to receive also *dk* (don't know) besides *ok* and *ko* (i.e., it applies covariance of outputs and contravariance of inputs, see [17]) is still compliant with the Hospital server. Hence we say that T'_{HC} is a subtype of T_{HC}.

Under asynchronous communication client compliance is relaxed by requiring that all messages that are sent are *eventually* received. For example, in this setting, client T''_{HC} (that may anticipate output *nd* w.r.t. inputs) is also a compliant client, see [5], hence T''_{HC} is an asynchronous subtype of T_{HC}.

Notice that, both for synchronous and asynchronous communication (see [7]), it holds, for any session type T, T': T' subtype of T implies $\overline{T}$ subtype of $\overline{T'}$ (closure under duality). As we will see, our tool automatically handles the generation of the *dual subtyping problem* ($\overline{T}$ subtype of $\overline{T'}$) from T' subtype of T by exchanging and dualizing inputted types.

Example 3. As another example, we consider clients of the Satellite protocol from [9]: an "ideal" client (the dual of the Satellite protocol server T_{SS} whose LTS is depicted in Figure 3) and a specific one; here denoted with T_{SC} and T'_{SC}, respectively.

$$T_{SC} = \overline{T_{SS}} = \mu\mathbf{X}.\ \&\{tm; \mathbf{X},\ over; \mu\mathbf{Y}.\ \oplus\{tc; \mathbf{Y},\ done; \mathbf{end}\}\}$$
$$T'_{SC} = \mu\mathbf{X}.\ \oplus\{tc; \mathbf{X},\ done; \mu\mathbf{Y}.\ \&\{tm; \mathbf{Y},\ over; \mathbf{end}\}\}$$

Figure 4 shows $\mathsf{LTS}(T'_{SC})$ and $\mathsf{LTS}(T_{SC})$, as produced by our tool. The "ideal" client T_{SC} may receive a number of telemetries (*tm*), followed by a message *over*. In the second phase, the client sends a number of telecommands (*tc*), followed by a message *done*. Under *fair* asynchronous communication client T'_{SC} (with phases exchanged) is also compliant with the server, i.e. T'_{SC} a fair asynchronous subtype of T_{SC}, see [9]. Compared to asynchronous communication considered in Example 2, here client compliance entails that, under *fairness assumption* (i.e. communication loops with some exit are assumed to be eventually escaped), both the client and the server must reach successfull termination with no messages left to be consumed in the FIFO channels.

2.2 Synchronous Session Subtyping

In order to establish whether type T' is a synchronous subtype of a type T [17] we can perform synchronous simulation of the (ordered) pair of LTSs $\mathsf{LTS}(T') = (Q', q'_0, \rightarrow')$ and $\mathsf{LTS}(T) = (Q, q_0, \rightarrow)$. Simulation states are pairs (q', q), with

[4] As we will see, the order in which the LTSs are presented reflects the subtyping relation (we will show that T''_{HC} and T'_{HC} are subtypes of T_{HC}) and the positions in which types are inputed in the tool.

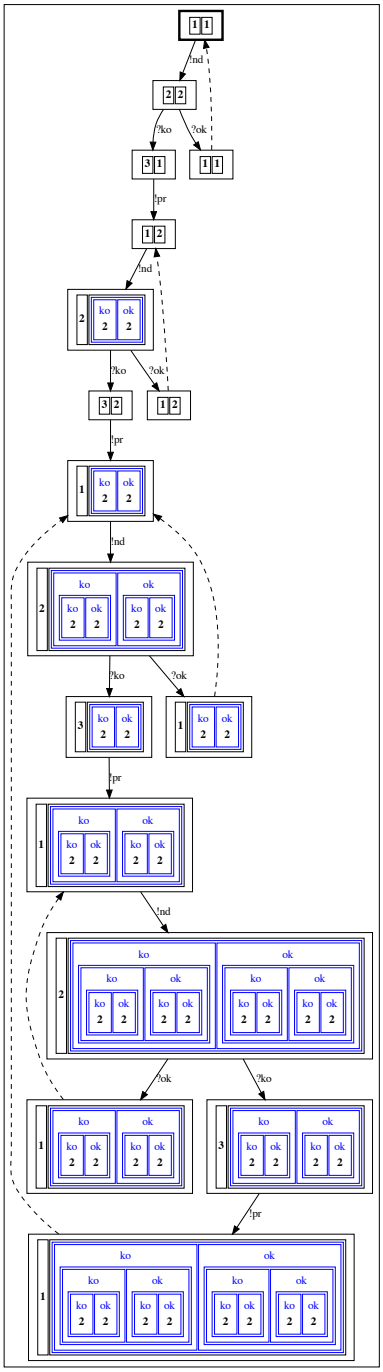


Fig. 5. Asynchronous simulation.

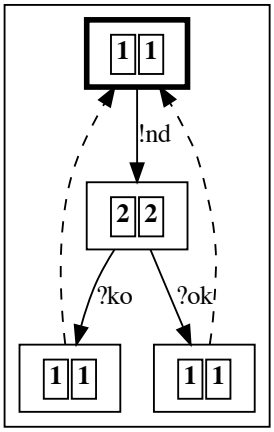


Fig. 6. Synchronous simulation.

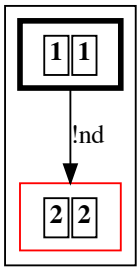


Fig. 7. Failed synchronous simulation.

$q' \in Q'$ and $q \in Q$. The simulation proceeds by starting from state (q'_0, q_0) and by synchronously matching transitions of $\mathsf{LTS}(T')$ and $\mathsf{LTS}(T)$ having the *same* labels (both "$!\,l$" or both "$?\,l$"). For each reached simulation state (q', q) we must have: (i) the set of outputs (resp. inputs) fireable by q' is subset (resp. superset) or equal to the set of outputs (resp. inputs) fireable by q; this enacts covariance (resp. contravariance) of outputs (resp. inputs), (ii) if (q', q) performs no transitions then both q' and q must perform no transitions (successfully terminate).

On the contrary, simulation states (q', q) for which the above constraints are not satisfied are called *failure simulation states* (depicted in red in our tool) and cause synchronous subtyping not to hold.

Example 4. Figure 6 shows the synchronous simulation graph, as produced from our tool, for the pair $\mathsf{LTS}(T'_{HC})$ and $\mathsf{LTS}(T_{HC})$. Notice that our tool builds the simulation graph as a tree: when a pair (q', q) is reached, which was previously traversed (as e.g. for the $(1, 1)$ pair), simulation does not proceed further in that branch and a dashed line is depicted connecting the two copies of (q', q). Notice that, if in T'_{HC} we turn $?ko$ into $?ko1$ (creating a mismatch with the server), T'_{HC} is no longer a synchronous subtype of T_{HC}. This can be seen in Figure 7 where the originated failure simulation state is depicted in red.

We now give the formal definition of synchronous subtyping. We first define set of inputs and set of outputs fireable by a state q as follows: $\mathsf{in}(q) = \{l \mid \exists q'.q \xrightarrow{?\,l} q'\}$ and $\mathsf{out}(q) = \{l \mid \exists q'.q \xrightarrow{!\,l} q'\}$.

Example 5. Consider $\mathsf{LTS}(T_{HC})$ (Figure 2), we have the following:

$$\begin{array}{ll} \mathsf{in}(1) = \emptyset & \mathsf{in}(2) = \{ko, ok\} \\ \mathsf{out}(1) = \{nd, nd\} & \mathsf{out}(2) = \emptyset \end{array}$$

Definition 2 (Synchronous Simulation). *Given set of label names $\mathcal{L}$ and two LTSs $(P, p_0, \rightarrow_1)$ and $(Q, q_0, \rightarrow_2)$, synchronous simulation is defined as a labeled transition system over states of $P \times Q$, i.e. pairs denoted by $p \preccurlyeq q$, with $p \in P$ and $q \in Q$. In particular, the initial state is $p_0 \preccurlyeq q_0$ and the transition relation $\hookrightarrow$, labeled over $\{!,?\} \times \mathcal{L}$, is defined as the minimal relation satisfying rules:*

$$\frac{p \xrightarrow{?\,l}_1 p' \quad q \xrightarrow{?\,l}_2 q' \quad \mathsf{in}(p) \supseteq \mathsf{in}(q)}{p \preccurlyeq q \;\overset{?\,l}{\hookrightarrow}\; p' \preccurlyeq q'}\ (\mathsf{In}) \qquad \frac{p \xrightarrow{!\,l}_1 p' \quad q \xrightarrow{!\,l}_2 q' \quad \mathsf{out}(p) \subseteq \mathsf{out}(q)}{p \preccurlyeq q \;\overset{!\,l}{\hookrightarrow}\; p' \preccurlyeq q'}\ (\mathsf{Out})$$

Formally, a type T' is a synchronous subtype of a type T if the LTS obtained as the synchronous simulation of the pair $\mathsf{LTS}(T')$ and $\mathsf{LTS}(T)$ is such that, for every state $q' \preccurlyeq q$ reachable from the initial simulation state, we have: if $q' \preccurlyeq q$ performs no transitions then both q' and q perform no transitions.

2.3 Asynchronous Session Subtyping

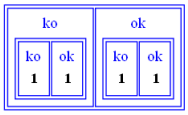
In contrast to synchronous simulation, the asynchronous one gives the possibility of "anticipating", in the right-hand LTS, output transitions w.r.t. input transitions that precede them. This can be shown, using our tool, via an example.

Example 6. Figure 5 shows the asynchronous simulation tree, as produced from our tool, for the pair $\mathsf{LTS}(T''_{HC})$ and $\mathsf{LTS}(T_{HC})$. Simulation of Figure 5 proceeds as follows. For instance, after transitions $!nd$, $?ko$ and $!pr$ (i.e. path "$!nd\,?ko\,!pr$") are synchronously performed by $\mathsf{LTS}(T''_{HC})$ and $\mathsf{LTS}(T_{HC})$, they reach states 1 and 2, respectively. Now, $\mathsf{LTS}(T''_{HC})$ in state 1 can only do output $!nd$, while $\mathsf{LTS}(T_{HC})$ in state 2 can only do inputs. Being asynchronous, the simulation can proceed by calculating the so-called state 2 *input tree* $\mathsf{inTree}(2) = \langle ko : 1,\ ok : 1\rangle$, i.e. the spanning tree from state 2 (constructed considering input transitions only), which has two leaves, both being state 1. Provided that all leaves of $\mathsf{inTree}(2)$ can perform $!nd$, the simulation can proceed by considering $\langle ko:[\,],\ ok:[\,]\rangle$ as an "accumulated" input for the right-hand LTS and by making all states in its leaves evolve by performing the $!nd$ transition. Therefore, after simulation performs $!nd$, $\mathsf{LTS}(T''_{HC})$ and $\mathsf{LTS}(T_{HC})$ reach states 2 and $\langle ko:2,\ ok:2\rangle$, respectively (the state reached by the right-hand LTS is actually an input tree).

In general, input trees (e.g. $\langle ko : 2,\ ok : 2\rangle$ in the example above) are defined in [5] as *input contexts* $\mathcal{A}$ (representing "accumulated" input, e.g. $\langle ko : [],\ ok : []\rangle$ in the example above) with *holes* "[ ]" replaced by LTS states. Their syntax is:

$$\mathcal{A} \quad ::= \quad [\,] \ \mid\ \langle l_i : \mathcal{A}_i\rangle_{i\in I}$$

In the tool we represent input trees by nested boxes. For instance the input tree $\langle ko : \langle ko : 1,\ ok : 1\rangle,\ ok : \langle ko : 1,\ ok : 1\rangle\rangle$ is represented as:



In general, due to input accumulation, represented by an input tree, even if two types are in an asynchronous subtyping relation, the simulation could proceed infinitely without meeting failure simulation states (as it would happen for the pair $\mathsf{LTS}(T''_{HC})$ and $\mathsf{LTS}(T_{HC})$ of Example 6). In our tool we use the algorithm of [5] for checking asynchronous subtyping, which is sound but not complete (in some cases it terminates without returning a decisive verdict). In a nutshell, such an algorithm proceeds as follows. The subtyping simulation terminates when we encounter a failure state (depicted in red in our tool), meaning that the two types are not in the subtyping relation, or when we detect a repetitive behaviour in the simulation (which, we show, can always be found, in case of infinite simulation). In the latter case, we check whether this repetitive behaviour satisfies sufficient conditions (see [5] for details) that guarantee that the subtyping simulation will never encounter failures. If the conditions are satisfied the algorithm concludes that the two types are in the subtyping relation, otherwise a *maybe* verdict is returned.

Therefore, the tool always produces a finite simulation tree: for types that are detected to be subtypes, simulation can stop in a state that is identified (via a dashed transition in our tool) to a previously encountered state, even if they are not identical; see bottommost state of Figure 5 (outgoing dashed transition).

The sufficient conditions checked by the algorithm guarantees that the behaviour beyond such a simulation state is a repetition of the behaviour already observed.

We now give the formal definition of asynchronous subtyping. Given a LTS $(Q, q_0, \to)$, we write $q_0 \xrightarrow{\ell_1 \cdots \ell_k} q_k$ iff there are $q_1, \ldots, q_{k-1} \in Q$ such that $q_{i-1} \xrightarrow{\ell_i} q_i$ for $1 \leq i \leq k$. Given a list of messages $\omega = l_1 \cdots l_k$ $(k \geq 0)$, we write $?\omega$ for the list $?l_1 \cdots ?l_k$ and $!\omega$ for $!l_1 \cdots !l_k$.

Definition 3 (Input Context). *An input context is a term of the grammar*

$$\mathcal{A} \quad ::= \quad [\,]_j \;\mid\; \langle l_i : \mathcal{A}_i \rangle_{i \in I}$$

where: All indices j, denoted by $I(\mathcal{A})$, are distinct and are associated to holes. Moreover, $I \neq \emptyset$ and $\forall i \neq j \in I.\ l_i \neq l_j$.

Holes are, thus, actually indexed so to make it possible to individually replace them. In this way $\mathcal{A}[q_i]^{i \in I(\mathcal{A})}$ denotes the *input tree* obtained by syntactically replacing each hole $[\,]_i$ in $\mathcal{A}$ by a specific state $q_i \in Q$. In the sequel, we use $\mathcal{IT}_Q$ to denote the set of input trees over states $q \in Q$.

Auxiliary functions Given a CSFM $(Q, q_0, \to)$ and a state $q \in Q$, we define:

- $\mathsf{cycle}(\star, q) \iff \exists \omega \in \mathcal{L}^*, \omega' \in \mathcal{L}^+, q' \in Q.\ q \xrightarrow{\star\omega} q' \xrightarrow{\star\omega'} q'$ (with $\star \in \{!, ?\}$),
- the *partial* function $\mathsf{inTree}(\cdot)$ as

$$\mathsf{inTree}(q) = \begin{cases} \bot & \text{if } \mathsf{cycle}(?, q) \\ q & \text{if } \mathsf{in}(q) = \varnothing \\ \langle l_i : \mathsf{inTree}(q_i') \rangle_{i \in I} & \text{if } \mathsf{in}(q) = \{l_i \mid i \in I\} \neq \varnothing \end{cases}$$

  with q_i' being the state such that $q \xrightarrow{?\,l_i} q_i'$.

Predicate $\mathsf{cycle}(\star, q)$ says that, from q, we can reach a cycle with only sends (resp. receives), depending on whether $\star = !$ or $\star = ?$. The partial function $\mathsf{inTree}(q)$, when defined, returns the tree containing all sequences of messages which can be received from q until a final or sending state is reached. Intuitively, $\mathsf{inTree}(q)$ is undefined when $\mathsf{cycle}(?, q)$ as it would return an infinite tree.

Example 7. Consider $\mathsf{LTS}(T_{HC})$ (Figure 2), we have the following:

$\mathsf{inTree}(1) = 1 \qquad \mathsf{inTree}(2) = \langle ko : 1,\ ok : 1 \rangle$

$\mathsf{inTree}(2)$ tool representation

Example 8. Consider the LTS of Figure 8. From state 1 we can reach state 2 with an output. The latter can loop with an output into itself. Hence, we have both $\mathsf{cycle}(!, 1)$ and $\mathsf{cycle}(!, 2)$.

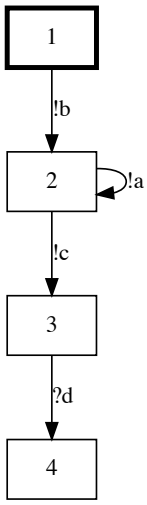


Fig. 8. Output cycle example.

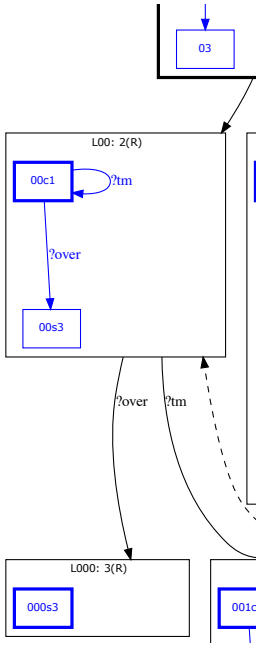


Fig. 9. Fair asynchronous simulation fragment.

Definition 4 (Asynchronous Simulation). *Given set of label names $\mathcal{L}$ and two LTSs: $LTS_1 = (P, p_0, \rightarrow_1)$ and $LTS_2 = (Q, q_0, \rightarrow_2)$, asynchronous simulation is defined as a labeled transition system over states of $P \times \mathcal{IT}_Q$, i.e. pairs denoted by $p \preccurlyeq \mathcal{A}[q_j]^{j \in J}$, with $p \in P$ and $\mathcal{A}[q_j]^{j \in J} \in \mathcal{IT}_Q$. In particular, the initial state is $p_0 \preccurlyeq q_0$ and the transition relation $\hookrightarrow$, labeled over $\{!,?\} \times \mathcal{L}$, is defined as the minimal relation satisfying rules in Definition 2, plus the following ones:*

$$\frac{p \xrightarrow{?\,l_k}_1 p' \qquad k \in I \qquad \mathsf{in}(p) \supseteq \{l_i \mid i \in I\}}{p \preccurlyeq \langle l_i : \mathcal{A}_i[q_{i,j}]^{j \in J_i} \rangle_{i \in I} \xhookrightarrow{?\,l_k} p' \preccurlyeq \mathcal{A}_k[q_{k,j}]^{j \in J_k}} \,(\mathsf{InCtx})$$

$$\frac{\begin{array}{c} p \xrightarrow{!\,l}_1 p' \qquad\qquad \neg\mathsf{cycle}(!, p) \\ \forall j \in J.\big(\mathsf{inTree}(q_j) = \mathcal{A}_j[q_{j,h}]^{h \in H_j} \wedge \forall h \in H_j.(\mathsf{out}(p) \subseteq \mathsf{out}(q_{j,h}) \wedge q_{j,h} \xrightarrow{!\,l}_2 q'_{j,h})\big) \end{array}}{p \preccurlyeq \mathcal{A}[q_j]^{j \in J} \xhookrightarrow{!\,l} p' \preccurlyeq \mathcal{A}[\mathcal{A}_j[q'_{j,h}]^{h \in H_j}]^{j \in J}} \,(\mathsf{OutA})$$

The two additional rules express how inputs are accumulated and consumed by means of input trees in the LTS_2. The first one is applicable when the input tree state of the LTS_2 is non-empty and the state p of the LTS_1 is able to perform a receive action corresponding to any message located at the root of the input tree (contra-variance of receive actions). The second rule allows the LTS_1 to execute some send actions by matching them with send actions that, in the LTS_2, occur after receives. Intuitively, each send action outgoing from state p of the LTS_1

must also be executable from each of the states $q_{j,h}$ of $\mathsf{inTree}(q_j) = \mathcal{A}_j[q_{j,h}]^{h \in H_j}$, with q_j being a leaf of the input tree state $\mathcal{A}[q_j]^{j \in J}$ of the LTS_2 (covariance of send actions). The constraint $\neg\mathsf{cycle}(!, p)$ guarantees that accumulated receive actions will be eventually executed.

2.4 Fair Asynchronous Session Subtyping

Consider again the satellite protocol of Example 3. The asynchronous subtyping of previous Section 2.3 rejects T'_{SC} as a subtype of T_{SC}. Indeed that notion of subtyping allows for anticipation of outputs only when they are preceded by a *bounded* number of inputs. However the outputs of T_{SC} occur after an arbitrary number of inputs. That notion of subtyping requires that all sent messages are consumed along *all* possible computations of the receiver. While in T'_{SC} there is a degenerate execution where the candidate subtype sends an infinite number of tc messages and thus never performs the required inputs.

In contrast, *fair* asynchronous session subtyping [9] relies on the assumption that such degenerate executions cannot occur under the natural assumption the loop of outputs eventually terminates, i.e., only a finite (but unspecified) amount of messages can be emitted.

Concretely, the fair subtyping uses a more expressive notion of input contexts $\mathcal{A}$ that also include recursive constructs. Their syntax becomes:

$$\mathcal{A} \quad ::= \quad [\,] \;\mid\; \langle l_i : \mathcal{A}_i \rangle_{i \in I} \;\mid\; \mu \mathbf{X}.\mathcal{A} \;\mid\; \mathbf{X}$$

These input context can encode the recursive reception of messages in the satellite example and thus identify T'_{SC} as a fair asynchronous subtype of T_{SC}.

Figure 9 shows a fragment of the resulting fair asynchronous simulation tree, as produced from our tool: due to the more complex syntax of input contexts, states now contain a (possibly looping) automaton instead of an input tree.

3 Main Functionalities of the Tool

Besides the classic operations that a text editor allows (e.g. edit, load, save), users can compute the dual of: either a single session type or the entire subtyping problem. To facilitate understanding of session types, the tool offers the possibility to view/save the graphical representation of a given type by means of "Show Image" and "Save Image" respectively. In our tool types are inputted by means of *two text areas*: the leftmost one is used for the candidate subtype and the rightmost one for the candidate supertype. Input types must be expressed with the syntax presented in Definition 1, with "+" standing for "$\oplus$" and "*rec*" standing for "μ". In addition the tool accepts: (*i*) the alternative "raw" syntax $[!\,a\,;T, !\,b\,;T', \dots]$ standing for $\oplus\{a\,;T, b\,;T', \dots\}$ and $[?\,a\,;T, ?\,b\,;T', \dots]$ for $\&\{a\,;T, b\,;T', \dots\}$ (*ii*) the abbreviations $!\,a\,;T$ and $?\,a\,;T$ standing for $\oplus\{a\,;T\}$ and $\&\{a\,;T\}$. A Python parser checks that the inputted types fit the above syntax using the following *EBNF* syntax:

$$\begin{aligned} S \quad &::= OP\,\mathbf{\{}\,\boldsymbol{id}\mathbf{;}\,S\,(\mathbf{,}\;\boldsymbol{id}\mathbf{;}\,S)^*\mathbf{\}} \;\mid\; \mathbf{rec}\,\boldsymbol{id}\mathbf{.}\;S \;\mid\; \boldsymbol{id} \;\mid\; \mathbf{end} \\ &\quad\; \mathbf{!}\,\boldsymbol{id}\mathbf{;}\,S \;\mid\; [\,\mathbf{!}\,\boldsymbol{id}\mathbf{;}\,S\,(\mathbf{,}\,\mathbf{!}\,\boldsymbol{id}\mathbf{;}\,S)^*] \;\mid\; \mathbf{?}\,\boldsymbol{id}\mathbf{;}\,S \;\mid\; [\,\mathbf{?}\,\boldsymbol{id}\mathbf{;}\,S\,(\mathbf{,}\,\mathbf{?}\,\boldsymbol{id}\mathbf{;}\,S)^*] \\ OP \;&::= \mathbf{+} \;\mid\; \mathbf{\&} \end{aligned}$$

where ***id*** is a non-empty sequence of uppercase and lowercase letters possibly followed by trailing numbers.

The core of the tool is the algorithm menu. Users can choose between different subtyping algorithms and possibly set a maximum number of execution steps. The algorithm response can be: "true", "false" or "maybe" (for asynchronous algorithms, due to undecidability, or when the specified number of steps is not enough to determine the subtyping relation), along with the time needed. In addition, it is possible to run all the algorithms to have an overview of the types of relationship that hold. Finally, the "Simulation Result" menu, which is initially disabled, makes it possible to show or save the graphical output of the last performed algorithm.

3.1 Extensibility of the Tool

Our tool (and its GUI) is automatically extensible with new subtyping algorithms by simply modifying its json configuration file that we will detail in the next section. Such a file can also be modified, directly from the GUI, by using the "Algorithm configuration" menu under "Settings". Configuration of a new algorithm is done by providing: its displayed name and the path and calling pattern of its execution command, in the form

```
ExecutableName [flag] [t1] [t2] [steps]
```

The tool will replace [flag][t1][t2][steps] with: the user-selected flags, the pair of session types the user wants to check and the number of steps the algorithm is requested to do. The above order of bracketed elements ([steps] is optional) may change according to the algorithm. The json file also maps algorithm-dependent flag names into tool functionalities by categorizing them. For instance, the default *flag* category includes flags that simply modify the behaviour of algorithms: e.g. the asynchronous one has the `--nofallback` flag that prevents the algorithm from trying to fall back to the dual subtyping problem in case of an initial maybe verdict. Moreover, the *execution flag* category is useful when an executable encloses different (alternative) algorithms, e.g. Gay and Hole (`--gayhole`) and Kozen et al.'s (`--kozen`) algorithms for synchronous subtyping (with one indicated as being the default). Morever, the *visual flag* category includes just the name of the flag causing the algorithm to produce the graphical simulation.

When adding an algorithm to the tool, the following requirements have to be satisfied: they have to support command line execution (with the possibility of taking .txt files as input) and have to fit the "raw" syntax described above. Regarding the algorithm response, the only requirement is that it is printed on the standard output. Finally, to generate the graphical output, it is mandatory that the algorithm creates a .dot file no matter what its name is (since it is specified in dedicated section of the json configuration). It is important to observe that our tool is agnostic to the implementation language of algorithms, since it makes use of their executable version.

3.2 Configuration of Tool Algorithms

The json file presented below is an example of the "algorithms_config.json" currently used by the tool. The *standard_exec* field specifies the default execution flag, e.g. Gay and Hole or Kozen. Moreover, the *simulation_file* field indicates the relative path to the algorithm generated *Graphviz* ".dot" simulation file. Similarly *win, osx* and *linux* point at the folder in which the tool looks for the algorithm binaries for that specific os.

```
[{
   "alg_name": "Async Subtyping",
   "flag": "--nofallback",
   "execution_flag": "",
   "standard_exec": "",
   "visual_flag": "--pics",
   "simulation_file": "tmp/simulation_tree",
   "win": "asynchronous-subtyping\\win\\",
   "osx": "asynchronous-subtyping/osx/",
   "linux": "asynchronous-subtyping/linux/",
   "exec_comm": "Checker [flags] [t1] [t2]"
 },
 {
   "alg_name": "Fair Async Subtyping",
   "flag": "",
   "execution_flag": "",
   "standard_exec": "",
   "visual_flag": "--debug",
   "simulation_file": "tmp/simulation_tree",
   "win": "fair-asynchronous-subtyping\\win\\",
   "osx": "fair-asynchronous-subtyping/osx/",
   "linux": "fair-asynchronous-subtyping/linux/",
   "exec_comm": "Checker [flags] [t1] [t2] [steps]"
 },
 {
   "alg_name": "Sync Subtyping",
   "flag": "",
   "execution_flag": "--gayhole,--kozen",
   "standard_exec": "--gayhole",
   "visual_flag": "--pics",
   "simulation_file": "tmp/simulation_tree",
   "osx": "sync_subtyping/osx/",
   "win": "sync_subtyping\\win\\",
   "linux": "sync_subtyping/linux/",
   "exec_comm": "Checker [flags] [t1] [t2]"
 }]
```

4 Conclusion

In this paper we introduced an integrated extensible GUI-based tool which: applies algorithms for synchronous and (fair) asynchronous session subtyping, and generates graphical simulations showing how underlying algorithms work.

Concerning future work, we plan to use our synchronous subtyping simulation algorithm (with error detection) in the context of type checking for object oriented programming languages where classes are endowed with usage protocols [8]. Indeed, extending the theory of [8] with protocol subtyping, would make it possible to verify correctness also for class inheritance. In particular, we have started integrating our algorithm into the Java checker [22], which is based on [8]. Finally, we plan to extend the syntax of session types managed by our tool, e.g. by including passing of data/channels and, possibly, by also encompassing preemption mechanisms [10,4], which are often used in communication protocols.

References

1. D. Ancona, V. Bono, M. Bravetti, J. Campos, G. Castagna, P. Deniélou, S. J. Gay, N. Gesbert, E. Giachino, R. Hu, E. B. Johnsen, F. Martins, V. Mascardi, F. Montesi, R. Neykova, N. Ng, L. Padovani, V. T. Vasconcelos, and N. Yoshida. Behavioral types in programming languages. *Foundations and Trends in Programming Languages*, 3(2-3):95–230, 2016.
2. L. Bacchiani, M. Bravetti, J. Lange, and G. Zavattaro. Tool source files for Linux, Windows and OSx (and binaries for Windows and OSx). `https://github.com/LBacchiani/session-subtyping-tool`.
3. D. Brand and P. Zafiropulo. On communicating finite-state machines. *J. ACM*, 30(2):323–342, 1983.
4. M. Bravetti. Axiomatizing maximal progress and discrete time. *Log. Methods Comput. Sci.*, 17(1): 1:1–1:44, 2021.
5. M. Bravetti, M. Carbone, J. Lange, N. Yoshida, and G. Zavattaro. A sound algorithm for asynchronous session subtyping and its implementation. *Log. Methods Comput. Sci.*, 17(1): 20:1–20:35, 2021.
6. M. Bravetti, M. Carbone, and G. Zavattaro. Undecidability of asynchronous session subtyping. *Inf. Comput.*, 256:300–320, 2017.
7. M. Bravetti, M. Carbone, and G. Zavattaro. On the boundary between decidability and undecidability of asynchronous session subtyping. *Theor. Comput. Sci.*, 722:19–51, 2018.
8. M. Bravetti, A. Francalanza, I. Golovanov, H. Hüttel, M. Jakobsen, M. Kettunen, and A. Ravara. Behavioural types for memory and method safety in a core object-oriented language. In B. C. d. S. Oliveira, editor, *Programming Languages and Systems - 18th Asian Symposium, APLAS 2020, Fukuoka, Japan, November 30 - December 2, 2020, Proceedings*, volume 12470 of *Lecture Notes in Computer Science*, pages 105–124. Springer, 2020.
9. M. Bravetti, J. Lange, and G. Zavattaro. Fair refinement for asynchronous session types. In S. Kiefer and C. Tasson, editors, *Foundations of Software Science and Computation Structures - 24th Int. Conference, FOSSACS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings*, volume 12650 of *Lecture Notes in Computer Science*, pages 144–163. Springer, 2021.

10. M. Bravetti and G. Zavattaro. On the expressive power of process interruption and compensation. *Math. Struct. Comput. Sci.*, 19(3):565–599, 2009.
11. M. Bravetti and G. Zavattaro. Asynchronous session subtyping as communicating automata refinement. *Softw. Syst. Model.*, 20(2):311–333, 2021.
12. T. Chen, M. Dezani-Ciancaglini, A. Scalas, and N. Yoshida. On the preciseness of subtyping in session types. *Logical Methods in Computer Science*, 13(2), 2017.
13. P. Deniélou and N. Yoshida. Multiparty compatibility in communicating automata: Characterisation and synthesis of global session types. In F. V. Fomin, R. Freivalds, M. Z. Kwiatkowska, and D. Peleg, editors, *Automata, Languages, and Programming - 40th Int. Colloquium, ICALP 2013, Riga, Latvia, July 8-12, 2013, Proceedings, Part II*, volume 7966 of *Lecture Notes in Computer Science*, pages 174–186. Springer, 2013.
14. S. J. Gay and M. Hole. Subtyping for session types in the pi calculus. *Acta Inf.*, 42(2-3):191–225, 2005.
15. R. Hu and N. Yoshida. Hybrid session verification through endpoint API generation. In *Fundamental Approaches to Software Engineering (FASE 2016)*, pages 401–418, 2016.
16. D. Kozen, J. Palsberg, and M. I. Schwartzbach. Efficient recursive subtyping. *Math. Struct. Comput. Sci.*, 5(1):113–125, 1995.
17. J. Lange and N. Yoshida. Characteristic formulae for session types. In M. Chechik and J. Raskin, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 22nd Int. Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*, volume 9636 of *Lecture Notes in Computer Science*, pages 833–850. Springer, 2016.
18. J. Lange and N. Yoshida. On the undecidability of asynchronous session subtyping. In *Foundations of Software Science and Computation Structures (FoSSaCS 2017)*, volume 10203 of *Lecture Notes in Computer Science*, pages 441–457, 2017.
19. S. Lindley and J. G. Morris. Embedding session types in Haskell. In *Haskell 2016*, pages 133–145, 2016.
20. D. Mostrous and N. Yoshida. Session typing and asynchronous subtyping for the higher-order π-calculus. *Inf. Comput.*, 241:227–263, 2015.
21. D. Mostrous, N. Yoshida, and K. Honda. Global principal typing in partially commutative asynchronous sessions. In *Programming Languages and Systems, 18th European Symposium on Programming (ESOP 2009)*, pages 316–332, 2009.
22. J. Mota, M. Giunti, and A. Ravara. Java typestate checker. In *Coordination Models and Languages - 23rd IFIP WG 6.1 Int. Conference, COORDINATION 2021, Held as Part of the 16th Int. Federated Conference on Distributed Computing Techniques, DisCoTec 2021, Valletta, Malta, June 14-18, 2021. Proceedings*, Lecture Notes in Computer Science. Springer, 2021. To appear.
23. R. Neykova, R. Hu, N. Yoshida, and F. Abdeljallal. A Session Type Provider: Compile-time API Generation for Distributed Protocols with Interaction Refinements in F♯. In *CC 2018*. ACM, 2018.
24. D. A. Orchard and N. Yoshida. Effects as sessions, sessions as effects. In *Principles of Programming Languages (POPL 2016)*, pages 568–581, 2016.
25. L. Padovani. A simple library implementation of binary sessions. *J. Funct. Program.*, 27:e4, 2017.
26. A. Scalas and N. Yoshida. Lightweight session programming in scala. In *European Conference on Object-Oriented Programming (ECOOP 2016)*, pages 21:1–21:28, 2016.