A Sound Algorithm for Asynchronous Session Subtyping

Mario Bravetti
Department of Computer Science / INRIA FoCUS Team, University of Bologna, Italy

Marco Carbone
IT University of Copenhagen, Denmark

Julien Lange
University of Kent, UK

Nobuko Yoshida
Imperial College London, UK

Gianluigi Zavattaro
Department of Computer Science / INRIA FoCUS Team, University of Bologna, Italy

Abstract

Session types, types for structuring communication between endpoints in distributed systems, are recently being integrated into mainstream programming languages. In practice, a very important notion for dealing with such types is that of subtyping, since it allows for typing larger classes of system, where a program has not precisely the expected behavior but a similar one. Unfortunately, recent work has shown that subtyping for session types in an asynchronous setting is undecidable. To cope with this negative result, the only approaches we are aware of either restrict the syntax of session types or limit communication (by considering forms of bounded asynchrony). Both approaches are too restrictive in practice, hence we proceed differently by presenting an algorithm for checking subtyping which is sound, but not complete (in some cases it terminates without returning a decisive verdict). The algorithm is based on a tree representation of the coinductive definition of asynchronous subtyping; this tree could be infinite, and the algorithm checks for the presence of finite witnesses of infinite successful subtrees. Furthermore, we provide a tool that implements our algorithm and we apply it to many examples that cannot be managed with the previous approaches.

2012 ACM Subject Classification Theory of computation → Concurrency

Keywords and phrases Session types, Concurrency, Subtyping, Algorithm

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2019.38

Related Version A full version of this paper is available at `https://arxiv.org/abs/1907.00421`.

Funding H2020-MSCA-RISE Project 778233 (BEHAPI); EPSRC EP/K034413/1, EP/K011715/1, EP/L00058X/1, EP/N027833/1, and EP/N028201/1.

1 Introduction

Session types are behavioural types that specify the structure of communication between the endpoints of a distributed system or the processes of a concurrent program. In recent years, session types have been integrated into several mainstream programming languages (see, e.g., [4, 19, 25, 28–30, 32]) where they specify the pattern of interactions that each endpoint must follow, i.e., a communication protocol. The notion of duality is at the core of theories based on session types, where it guarantees that each send (resp. receive) action is matched by a corresponding receive (resp. send) action, and thus rules out deadlocks and orphan messages. A two-party communication protocol specified as a pair of session types is “correct” (deadlock free, etc) when these types are dual of each other. Unfortunately, in

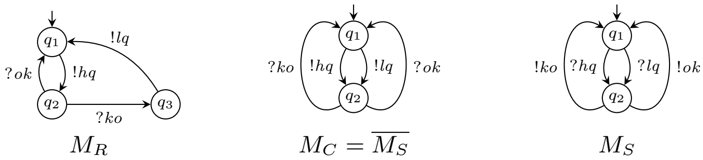


Figure 1 Video streaming example. M_R is the (refined) session type of the client, M_C is a supertype of M_R, and M_S is the session type of the server.

practice, duality is a too strict prerequisite, since it does not provide programmers with the flexibility necessary to build practical implementations of a given protocol. A natural solution for relaxing this rigid constraint is to adopt a notion of (session) subtyping which lets programmers implement refinements of the specification (given as a session type). In particular, an endpoint implemented as program P_2 with type M_2 can always be safely replaced by another program P_1 with type M_1 whenever M_1 is a subtype of M_2 (written $M_1 \preccurlyeq M_2$ in this paper).

The two main known notions of subtyping for session types differ in the type of communication they support: either synchronous (rendez-vous) or asynchronous (over unbounded FIFO channels). *Synchronous session subtyping* only allows for a subtype to implement fewer internal choices (sends), and more external choices (receives), than its supertype. Hence checking whether two types are related can be done efficiently (quadratic time wrt. the size of the types [23]). Synchronous session subtyping is of limited interest in modern programming languages such as Go and Rust, which provide *asynchronous* communication over channels. Indeed, in an asynchronous setting, the programmer needs to be able to make the best of the flexibility given by non-blocking send actions. This is precisely what the *asynchronous session subtyping* offers: it widens the synchronous subtyping relation by allowing the subtype to anticipate send actions, when this does not affect its communication partner. We illustrate the salient points of the asynchronous session subtyping with Figure 1, which depicts the hypothetical session types of the client and server endpoints of a video streaming service, represented as communicating machines – an equivalent formalism [6, 13]. Machine M_S (right) is a server which can deal with two types of requests: it can receive either a message *lq* (low-quality) or a message *hq* (high-quality). After receiving a message of either type, the server replies with *ok* or *ko*, indicating whether the request can be fulfilled, then it returns to its starting state. Machine M_C (middle) represents the type of the client. It is the *dual* of the server M_S (written $\overline{M_S}$), as required in standard two-party session types without subtyping. A programmer may want to implement a slightly improved program which behaves as Machine M_R (left). This version requests high-quality (*hq*) streaming first, and falls back to low-quality (*lq*) if the request is denied (it received *ko*). In fact, machine M_R is an (asynchronous) subtype of machine M_C. Indeed, M_R is able to receive the same set of messages as M_C, each of the sent messages are also allowed by M_C, and the system consisting of the parallel composition of machines M_R and M_S is free from deadlocks and orphan messages. We will use this example in the rest of the paper to illustrate our theory.

Recently, we have proven that checking whether two types are in the asynchronous subtyping relation is, unfortunately, *undecidable* [7, 8, 24]. In order to mitigate this negative result, some theoretical algorithms have been proposed for restricted subclasses of session types. These restrictions can be divided into two main categories: syntactical restrictions, i.e., allowing only one type of non-unary branching (internal or external), or adding bounds to the size of the FIFO communication channels. Both types of restrictions are problematic in practice. Syntactic restrictions disallow protocols featuring both types of internal/external choices, e.g., the machines M_C and M_S in Figure 1 contain (non-unary) external and internal choices. On the other hand, applying a bound to the subtyping relation is generally difficult because (i) it may be undecidable whether such a bound exists, (ii) the channel bounds used in the implementation (if any) might not be known at compile time, and (iii) very simple systems, such as the one in Figure 1, require unbounded communication channels.

In this paper, we give an algorithm that can soundly deal with a much larger class of session types. Rather than imposing syntactical restrictions or bounds, we describe an algorithm whose termination condition is based on a well-quasi order between pairs of candidate subtypes. This condition allows us to construct a finite tree representation of the coinductive definition of asynchronous subtyping, which we use to synthesise intermediate automata. Finally, we give a sufficient condition for asynchronous subtyping based on a compatibility relation between these intermediate automata. This compatibility check is similar to that of subtyping for recursive types [3,16,21,23]. We provide a full implementation of our algorithm and show that it performs well on examples taken from the literature.

2 Communicating Machines and Asynchronous Subtyping

In this section, we recall the definition of two-party communicating machines, that communicate over unbounded FIFO channels (§ 2.1), and define asynchronous subtyping for session types [10, 11], which we adapt to communicating machines, following [8] (§ 2.2).

2.1 Communicating Machines

Let $\mathbb{A}$ be a (finite) alphabet, ranged over by a, b, etc. We let ω, ω', etc. range over words in $\mathbb{A}^*$. The set of send (resp. receive) actions is $Act_! = \{!\} \times \mathbb{A}$, (resp. $Act_? = \{?\} \times \mathbb{A}$). The set of actions is $Act = Act_! \cup Act_?$, ranged over by ℓ, where send action $!a$ puts message a on an (unbounded) buffer, while receive action $?a$ represents the consumption of a from a buffer. We define $dir(!a) \overset{\text{def}}{=} !$ and $dir(?a) \overset{\text{def}}{=} ?$ and let ψ and φ range over Act^*. We write $\cdot$ for the concatenation operator on words.

In this work, we only consider communicating machines which correspond to (two-party) session types. Hence, we focus on deterministic (communicating) finite-state machines, without mixed states (i.e., states that can fire both send and receive actions) as in [13, 14].

▶ **Definition 1** (Communicating Machine). *A communicating machine M is a tuple (Q, q_0, δ) where Q is the (finite) set of states, $q_0 \in Q$ is the initial state, and $\delta \in Q \times Act \times Q$ is a transition relation such that $\forall q, q', q'' \in Q.\ \forall \ell, \ell' \in Act$: **(1)** $(q, \ell, q'), (q, \ell', q'') \in \delta$ implies $dir(\ell) = dir(\ell')$, and **(2)** $(q, \ell, q'), (q, \ell, q'') \in \delta$ implies $q' = q''$. We write $q \xrightarrow{\ell} q'$ for $(q, \ell, q') \in \delta$, omit unnecessary labels, and write $\rightarrow^*$ for the reflexive transitive closure of $\rightarrow$.*

Condition **(1)** enforces directed states, while Condition **(2)** enforces determinism.

Given $M = (Q, q_0, \delta)$, we say that $q \in Q$ is *final*, written $q \not\rightarrow$, iff $\forall q' \in Q.\ \forall \ell \in Act.\ (q, \ell, q') \notin \delta$. A state $q \in Q$ is *sending* (resp. *receiving*) iff q is not final and $\forall q' \in Q.\ \forall \ell \in Act.\ (q, \ell, q') \in \delta.\ dir(\ell) = !$ (resp. $dir(\ell) = ?$). We write $\delta(q, \ell)$ for q' iff $(q, \ell, q') \in \delta$.

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 $\omega = a_1 \cdots a_k$ $(k \geq 0)$, we write $?\omega$ for the list $?a_1 \cdots ?a_k$ and $!\omega$ for $!a_1 \cdots !a_k$. We write $q \xrightarrow{!}{}^* q'$ iff $\exists \omega \in \mathbb{A}.\ q \xrightarrow{!\omega} q'$ and $q \xrightarrow{?}{}^* q'$ iff $\exists \omega \in \mathbb{A}.\ q \xrightarrow{?\omega} q'$ (note that ω may be empty, in which case $q = q'$). Given $\psi \in Act^*$ we write $\mathsf{snd}(\psi)$ (resp. $\mathsf{rcv}(\psi)$) for the largest sub-sequence of ψ which consists only of the messages of send (resp. receive) actions.

2.2 Asynchronous Session Subtyping

Input trees and contexts. In order to formalise the subtyping relation, we use syntactic constructs used to record the input actions that have been anticipated by a candidate supertype, e.g., machine M_2 in Definition 5, as well as the local states it may reach.

▶ **Definition 2** (Input Tree). *An input tree is a term of the grammar:* $T \quad ::= \quad q \mid \langle a_i : T_i \rangle_{i \in I}$.

In the sequel, we use $\mathcal{T}_Q$ to denote the input trees over states $q \in Q$. An input context is an input tree with a "hole" in the place of a sub-term.

▶ **Definition 3** (Input Context). *An input context is a term of* $\mathcal{A} \quad ::= \quad [\,]_j \mid \langle a_i : \mathcal{A}_i \rangle_{i \in I}$, *where all indices* j*, denoted by* $I(\mathcal{A})$*, are distinct and are associated to holes.*

For input trees and contexts of the form $\langle a_i : T_i \rangle_{i \in I}$ and $\langle a_i : \mathcal{A}_i \rangle_{i \in I}$, we assume that $I \neq \emptyset$, $\forall i \neq j \in I.\ a_i \neq a_j$, and that the order of the sub-terms is irrelevant. When convenient, we use set-builder notation to construct input trees or contexts, e.g., $\langle a_i : T_i \mid i \in I \rangle$.

Given an input context $\mathcal{A}$ and an input context $\mathcal{A}_i$ for each i in $I(A)$, we write $\mathcal{A}[\mathcal{A}_i]^{i \in I(A)}$ for the input context obtained by replacing each hole $[\,]_i$ in $\mathcal{A}$ by the input context $\mathcal{A}_i$. We write $\mathcal{A}[T_i]^{i \in I(A)}$ for the input tree where holes are replaced by input trees.

Auxiliary functions. In the rest of the paper we use the following auxiliary functions on communicating machines. Given a machine $M = (Q, q_0, \delta)$ and a state $q \in Q$, we define:

- $\mathsf{cycle}(\star, q) \iff \exists \omega \in \mathbb{A}^*, \omega' \in \mathbb{A}^+, q' \in Q.\ q \xrightarrow{\star\omega} q' \xrightarrow{\star\omega'} q'$ (with $\star \in \{!, ?\}$),
- $\mathsf{in}(q) = \{a \mid \exists q'.q \xrightarrow{?a} q'\}$ and $\mathsf{out}(q) = \{a \mid \exists q'.q \xrightarrow{!a} q'\}$,
- let the *partial* function $\mathsf{inTree}(\cdot)$ be defined as:

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

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 function $\mathsf{in}(q)$ (resp. $\mathsf{out}(q)$) returns the messages that can be received (resp. sent) in q. When defined, $\mathsf{inTree}(q)$ 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 4.** Given M_C (Figure 1), we have $\mathsf{inTree}(q_1) = q_1$ and $\mathsf{inTree}(q_2) = \langle ok : q_1,\ ko : q_1 \rangle$.

Asynchronous subtyping. We present our definition of asynchronous subtyping (following the orphan-message-free version from [11]). Our definition is a simple adaptation[1] of [8, Definition 2.4] (given on syntactical session types) to the setting of communicating machines.

▶ **Definition 5** (Asynchronous Subtyping). *Let* $M_i = (Q_i, q_{0_i}, \delta_i)$ *for* $i \in \{1, 2\}$*.* $\mathcal{R}$ *is an asynchronous subtyping relation on* $Q_1 \times \mathcal{T}_{Q_2}$ *such that* $(p, T) \in \mathcal{R}$ *implies what follows:*

1. *if* $p \not\rightarrow$ *then* $T = q$ *such that* $q \not\rightarrow$*;*
2. *if* p *is a receiving state then*

[1] In definitions for syntactical session types, e.g., [26], input contexts are used to accumulate inputs that precede anticipated outputs; here, having no specific syntax for inputs, we use input trees instead.

a. *if* $T = q$ *then* q *is receiving and* $\forall q' \in Q_2$ *s.t.* $q \xrightarrow{?a} q'$. $\exists p'$ *s.t.* $p \xrightarrow{?a} p' \wedge (p', q') \in \mathcal{R}$*;*
b. *if* $T = \langle a_i : T_i \rangle_{i \in I}$ *then* $\forall i \in I$. $\exists p'$ *s.t.* $p \xrightarrow{?a_i} p' \wedge (p', T_i) \in \mathcal{R}$*;*

3. *if* p *is a sending state then*
a. *if* $T = q$ *then* q *is sending and* $\forall p' \in Q_1$ *s.t.* $p \xrightarrow{!a} p'$. $\exists q'$ *s.t.* $q \xrightarrow{!a} q' \wedge (p', q') \in \mathcal{R}$ *or*
b. *if* $T = \mathcal{A}[q_i]^{i \in I}$ *then let* $\mathcal{A}_i[q_{i,h}]^{h \in H_i} = \mathsf{inTree}(q_i)$ *and if* $p \xrightarrow{!a} p'$ *then* $\forall i \in I. \forall h \in H_i$. $\exists q'_{i,h}$. $q_{i,h} \xrightarrow{!a} q'_{i,h} \wedge (p', \mathcal{A}[\mathcal{A}_i[q'_{i,h}]^{h \in H_i}]^{i \in I}) \in \mathcal{R}$ *and, if* $\mathcal{A}[\mathcal{A}_i[\,]^{h \in H_i}]^{i \in I}$ *is not a single hole, then* $\neg\mathsf{cycle}(!, p)$*.*

M_1 is an asynchronous subtype of M_2, written $M_1 \preccurlyeq M_2$, if there is an asynchronous subtyping relation $\mathcal{R}$ such that $(q_{0_1}, q_{0_2}) \in \mathcal{R}$.

The relation $M_1 \preccurlyeq M_2$ checks that M_1 is a subtype of M_2 by executing M_1 and simulating its execution with M_2. M_1 may fire send actions earlier than M_2, in which case M_2 is allowed to fire these actions even if it needs to fire some receive actions first. These receive actions are accumulated in an input context and are expected to be subsequently matched by M_1. Due to the presence of such an input context, the states reached by M_2 during the computation are represented as input trees. The definition first differentiates the type of state p:

Final. In (1), if M_1 is in a final state, then M_2 is in a final state with an empty input context.

Receiving. Case (2) says that if M_1 is in a receiving state, then either (2a) the input context is empty ($T = q$) and M_1 must be able to receive all messages that M_2 can receive; or, (2b) M_1 must be able to consume all the messages at the root of the input tree.

Sending. Case (3) says that if M_1 is a sending state then either: (3a) the input context is empty ($T = q$) and M_2 must be able to send all messages that M_1 can send; or, (3b) M_2 must be able to send every a that M_1 can send, possibly after some receive actions recorded in each $\mathcal{A}_i[q_{i,h}]^{h \in H_i}$. Note that whichever receiving path M_1 chooses in the continuation, M_2 must be able to simulate it, hence the send action $!a$ should be available at the end of each receiving path. Moreover, whenever there are accumulated inputs, we require that $\mathsf{cycle}(!, p)$ does *not* hold, guaranteeing that subtyping preserves orphan-message freedom, i.e., such accumulated receive actions will be eventually executed.

Observe that Case (2) enforces a form of contra-variance for receive actions, while Case (3) enforces a form of covariance for send actions. In Figure 1, we have $M_R \preccurlyeq M_C$ (see § 3).

3 A Sound Algorithm for Asynchronous Subtyping

Our subtyping algorithm takes two machines M_1 and M_2 and produces three possible outputs: *true*, *false*, or *unknown*, which respectively indicate that $M_1 \preccurlyeq M_2$, $M_1 \not\preccurlyeq M_2$, or that the algorithm was unable to prove one of these two results. The algorithm is in three stages. (1) It builds the *simulation tree* of M_1 and M_2 (see Definition 7) that represents sequences of checks between M_1 and M_2, corresponding to the checks in the definition of asynchronous subtyping. Simulation trees may be infinite, but the function terminates whenever: it reaches a node that cannot be expanded, it visits a node whose label has been seen along the path from the root, or it expands a node whose ancestors validate a termination condition that we formalise in Theorem 13. The resulting tree satisfies one of the following conditions: (i) it contains a leaf that could not be expanded because the node represents an unsuccessful check between M_1 and M_2 (in which case the algorithm returns *false*), (ii) all leaves are successful final configurations, see Condition (1) of Definition 5, in which case the algorithm replies *true*, or (iii) for each leaf n it is possible to identify a corresponding ancestor $\mathsf{anc}(n)$. In this last case the tree and the identified ancestors are passed onto the next stage. (2) The algorithm divides the finite tree into several subtrees rooted at those ancestors that do not have other ancestors above them (see the strategy that we outline on page 10). (3) The final stage analyses whether each subtree is of one of the two following kinds. (i) All the leaves in the subtree have the same label as their ancestors: in this case the subtree contains all the needed subtyping checks. (ii) The subtree is a *witness subtree* (see Definition 20), meaning that all the checks that may be considered in continuations of the finite subtree are guaranteed to be successful as well. If all the identified subtrees are of one of the these two kinds, the algorithm replies *true*. Otherwise, it replies *unknown*.

3.1 Generating Asynchronous Simulation Trees

We first define labelled trees, of which our simulation trees are instances; then, we give the operational rules for generating a simulation tree from a pair of communicating machines.

▶ **Definition 6** (Labelled Tree). *A labelled tree is a tree*[2] $(N, n_0, \hookrightarrow, \mathcal{L}, \Sigma, \Gamma)$*, consisting of nodes* N*, root* $n_0 \in N$*, edges* $\hookrightarrow \subseteq N \times \Sigma \times N$*, and node labelling function* $\mathcal{L} : N \longmapsto \Gamma$*.*

Hereafter, we write $n \overset{\sigma}{\hookrightarrow} n'$ when $(n, \sigma, n') \in \hookrightarrow$ and write $n_1 \xhookrightarrow{\sigma_1 \cdots \sigma_k} n_{k+1}$ when there are $n_1, \ldots, n_{k+1}$, such that $n_i \overset{\sigma_i}{\hookrightarrow} n_{i+1}$ for all $1 \leq i \leq k$. We write $n \hookrightarrow n'$ when $n \overset{\sigma}{\hookrightarrow} n'$ for some σ and the label is not relevant. As usual, we write $\hookrightarrow^*$ for the reflexive and transitive closure of $\hookrightarrow$, and $\hookrightarrow^+$ for its transitive closure. Given an edge label $\sigma \in \Sigma$ and two node labels $\alpha, \beta \in \Gamma$, we use $\alpha \overset{\sigma}{\hookrightarrow} \beta$ as a shorthand for $\forall n.\ \mathcal{L}(n) = \alpha \Rightarrow \exists n'.\ \mathcal{L}(n') = \beta \land n \overset{\sigma}{\hookrightarrow} n'$. Moreover, we reason up-to tree isomorphism, i.e., two labelled trees are equivalent if there exists a bijective node renaming that preserves both node labelling and labelled transitions.

We can then define simulation trees, labelled trees representing all possible configurations reachable by the asynchronous subtyping simulation game.

▶ **Definition 7** (Simulation Tree). *Let* $M_1 = (P, p_0, \delta_1)$ *and* $M_2 = (Q, q_0, \delta_2)$ *be two communicating machines. Their simulation tree, written* $\mathsf{simtree}(M_1, M_2)$*, is the minimal labelled tree* $(N, n_0, \hookrightarrow, \mathcal{L}, Act, P \times \mathcal{T}_Q)$*, with* $\mathcal{L}(n_0) = p_0 \preccurlyeq q_0$*, generated by the following rules:*

$$\frac{p \xrightarrow{?a} p' \quad q \xrightarrow{?a} q' \quad \mathsf{in}(p) \supseteq \mathsf{in}(q)}{p \preccurlyeq q \overset{?a}{\hookrightarrow} p' \preccurlyeq q'}\ \textit{(In)} \qquad \frac{p \xrightarrow{!a} p' \quad q \xrightarrow{!a} q' \quad \mathsf{out}(p) \subseteq \mathsf{out}(q)}{p \preccurlyeq q \overset{!a}{\hookrightarrow} p' \preccurlyeq q'}\ \textit{(Out)}$$

$$\frac{p \xrightarrow{?a_k} p' \quad k \in I \quad \mathsf{in}(p) \supseteq \{a_i \mid i \in I\}}{p \preccurlyeq \langle a_i : \mathcal{A}_i[q_{i,j}]^{j \in J_i} \rangle_{i \in I} \overset{?a_k}{\hookrightarrow} p' \preccurlyeq \mathcal{A}_k[q_{k,j}]^{j \in J_k}}\ \textit{(InCtx)}$$

$$\frac{\begin{array}{c} p \xrightarrow{!a} p' \qquad \neg\mathsf{cycle}(!, p) \\ \forall j \in J. \big(\mathsf{inTree}(q_j) = \mathcal{A}_j[q_{j,h}]^{h \in H_j} \land \forall h \in H_j. (\mathsf{out}(p) \subseteq \mathsf{out}(q_{j,h}) \land q_{j,h} \xrightarrow{!a} q'_{j,h})\big) \end{array}}{p \preccurlyeq \mathcal{A}[q_j]^{j \in J} \overset{!a}{\hookrightarrow} p' \preccurlyeq \mathcal{A}[\mathcal{A}_j[q'_{j,h}]^{h \in H_j}]^{j \in J}}\ \textit{(OutAcc)}$$

Given machines M_1 and M_2, Definition 7 generates a tree whose nodes are labelled by terms of the form $p \preccurlyeq \mathcal{A}[q_i]^{i \in I}$ where p represents the state of M_1, $\mathcal{A}$ represents the receive actions accumulated by M_2, and each q_i represents the state of machine M_2 after each path of accumulated receive actions from the root of $\mathcal{A}$ to the i^{th} hole. Note that we overload the symbol $\preccurlyeq$ used for asynchronous subtyping (Definition 5), however the actual meaning is always made clear by the context. We comment each rule in detail below.

[2] A tree is a connected directed graph without cycles: $\forall n \in N.\ n_0 \hookrightarrow^* n \land \forall n, n' \in N.\ n \hookrightarrow^+ n'.\ n \neq n'$.

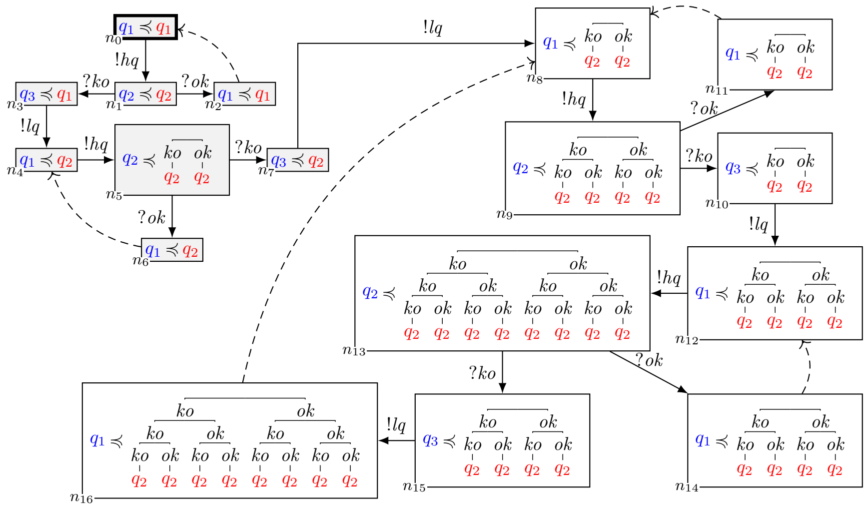


Figure 2 Part of the simulation tree (solid edges only) and candidate tree for $M_R \preccurlyeq M_C$ (Figure 1). The root is circled in thicker line. The node identities are shown at the bottom left of each label.

Rules (In) and (Out) enforce contra-variance of inputs and covariance of outputs, respectively, when no accumulated receive actions are recorded, i.e., $\mathcal{A}$ is a single hole. Rule (In) corresponds to Case (2a) of Definition 5, while rule (Out) corresponds to Case (3a).
Rule (InCtx) is applicable when the input tree $\mathcal{A}$ is non-empty and the state p (of M_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). This rule corresponds to Case (2b) of Definition 5.
Rule (OutAcc) allows M_2 to execute some receive actions before matching a send action executed by M_1. This rule corresponds to Case (3b) of Definition 5. Intuitively, each send action outgoing from state p must also be eventually executable from each of the states q_j (in M_2) which occur in the input tree $\mathcal{A}[q_j]^{j\in J}$. The possible combinations of receive actions executable from each q_j before executing $!a$ is recorded in $\mathcal{A}_j$, using $\mathsf{inTree}(q_j)$. We assume that the premises of this rule only hold when all invocations of $\mathsf{inTree}(\cdot)$ are defined. Each tree of accumulated receive actions is appended to its respective branch of the input context $\mathcal{A}$, using the notation $\mathcal{A}[\mathcal{A}_j[q'_{j,h}]^{h\in H_j}]^{j\in J}$. The premise $\mathsf{out}(p) \subseteq \mathsf{out}(q_{j,h}) \land q_{j,h} \xrightarrow{!a} q'_{j,h}$ guarantees that each $q_{j,h}$ can perform the send actions available from p (covariance of send actions). The additional premise $\neg\mathsf{cycle}(!, p)$ corresponds to that of Case (3b) of Definition 5.

▶ **Example 8.** Figure 2 gives a graphical view of the initial part of the simulation tree $\mathsf{simtree}(M_R, M_C)$. Consider the solid edges only for now. Observe that all branches of the simulation tree are infinite; some traverse nodes with infinitely many different labels, due to the unbounded growth of the input trees (e.g., the one repeatedly performing transitions $!hq{\cdot}?ko{\cdot}!lq$); while others traverse nodes with *finitely* many distinct labels (e.g., the one performing first transitions $!hq{\cdot}?ko{\cdot}!lq$ and then repeatedly performing $!hq{\cdot}?ok$).

We adapt the terminology of [20] and say that a node n of $\mathsf{simtree}(M_1, M_2)$ is a *leaf* if it has no successors. A leaf n is *successful* iff $\mathcal{L}(n) = p \preccurlyeq q$, with p and q final; all other leaves are unsuccessful. A *branch* (a full path through the tree) is *successful* iff it is infinite or finishes with a successful leaf; otherwise it is unsuccessful. Using this terminology, we relate asynchronous subtyping (Definition 5) with simulation trees (Definition 7) in Theorem 9.

▶ **Theorem 9.** *Let $M_1 = (P, p_0, \delta_1)$ and $M_2 = (Q, q_0, \delta_2)$ be two communicating machines. All branches in $\mathsf{simtree}(M_1, M_2)$ are successful if and only if $M_1 \preccurlyeq M_2$.*

3.2 A Simulation Tree-Based Algorithm

Checking whether all branches in $\mathsf{simtree}(M_1, M_2)$ are successful is undecidable. This is a consequence of the undecidability of asynchronous session subtyping [7, 8, 24]. The problem follows from the presence of infinite branches that cannot be algorithmically identified. Our approach is to characterise finite subtrees (called *witness subtrees*) such that all the branches that traverse such finite subtrees are guaranteed to be infinite.

The presentation of our algorithm is in three parts. In Part (1), we give the definition of the kind of *finite* subtree (of a simulation tree) we are interested in (called *candidate* subtrees). In Part (2), we give an algorithm to extract *candidate* subtrees from a simulation tree $\mathsf{simtree}(M_1, M_2)$. In Part (3) we show how to check whether a candidate subtree (which is finite) is a *witness* of infinite branches (hence successful) in the simulation tree.

Part 1. Characterising finite and candidate sub-trees. We define the candidate subtrees of a simulation tree, which are finite subtrees accompanied by an ancestor function mapping each boundary node n to a node located on the path from the root of the tree to n.

▶ **Definition 10** (Finite Subtree). *A finite subtree (r, B) of a labelled tree $S = (N, n_0, \hookrightarrow, \mathcal{L}, \Sigma, \Gamma)$, with r being the subtree root and B the finite set of its leaves (boundary nodes), is the subgraph of S such that:* ***(1)*** *$\forall n \in B.\ r \hookrightarrow^* n$;* ***(2)*** *$\forall n \in B.\ \nexists n' \in B.\ n \hookrightarrow^+ n'$; and* ***(3)*** *$\forall n \in N.\ r \hookrightarrow^* n \implies \exists n' \in B.\ n \hookrightarrow^* n' \vee n' \hookrightarrow^* n$. We use $\mathsf{nodes}(S, r, B) = \{n \in N \mid \exists n' \in B.\ r \hookrightarrow^* n \hookrightarrow^* n'\}$ to denote the (finite) set of nodes of the finite subtree (r, B). Notice that $r \in \mathsf{nodes}(S, r, B)$ and $B \subseteq \mathsf{nodes}(S, r, B)$.*

Condition **(1)** requires that each boundary node can be reached from the root of the subtree. Condition **(2)** guarantees that the boundary nodes are not connected, i.e., they are on different paths from the root. Condition **(3)** enforces that each branch of the tree passing through the root r contains a boundary node.

▶ **Definition 11** (Candidate Subtree). *Let $M_1 = (P, p_0, \delta_1)$ and $M_2 = (Q, q_0, \delta_2)$ be two communicating machines with $\mathsf{simtree}(M_1, M_2) = (N, n_0, \hookrightarrow, \mathcal{L}, Act, P \times \mathcal{T}_Q)$. A* candidate subtree *of $\mathsf{simtree}(M_1, M_2)$ is a finite subtree (r, B) paired with a function $\mathsf{anc} : B \longmapsto \mathsf{nodes}(\mathsf{simtree}(M_1, M_2), r, B) \backslash B$ such that, for all $n \in B$, we have: $\mathsf{anc}(n) \hookrightarrow^+ n$ and there are $p, \mathcal{A}, \mathcal{A}', I, J, \{q_j \mid j \in J\}$ and $\{q_i \mid i \in I\}$ such that*

$$\mathcal{L}(n) = p \preccurlyeq \mathcal{A}[q_i]^{i \in I} \ \wedge\ \mathcal{L}(\mathsf{anc}(n)) = p \preccurlyeq \mathcal{A}'[q_j]^{j \in J} \ \wedge\ \{q_i \mid i \in I\} \subseteq \{q_j \mid j \in J\}$$

A candidate subtree is a finite subtree accompanied by a total function on its boundary nodes. The purpose of function anc is to map each boundary node n to a "similar" ancestor n' such that: n' is a node (different from n) on the path from the root r to n (recall that we have $r \notin B$) such that the two labels of n' and n share the same state p of M_1, and the states of M_2 (that populate the holes in the leaves of the input context of the boundary node) are a subset of those considered for the ancestor. We write $img(\mathsf{anc})$ for $\{n \mid \exists n' \in B.\ \mathsf{anc}(n') = n\}$, i.e., $img(\mathsf{anc})$ is the set of ancestors of a given candidate subtree.

▶ **Example 12.** Figure 2 depicts a finite subtree of $\mathsf{simtree}(M_R, M_C)$. The anc function is represented by the dashed edges from boundary nodes to ancestors. We can distinguish distinct candidate subtrees in Figure 2, for instance one rooted at n_0 and with boundary $\{n_2, n_6, n_{11}, n_{14}, n_{16}\}$, another one rooted at n_8 and with boundary $\{n_{11}, n_{14}, n_{16}\}$.

Part 2. Identifying candidate subtrees. We now describe how to generate a finite subtree of the simulation tree, from which we extract candidate subtrees. Since simulation trees are potentially infinite, we need to identify termination conditions (i.e., conditions on nodes that become the boundary of the generated finite subtree).

We first need to define the auxiliary function $\mathsf{extract}(\mathcal{A}, \omega)$, which checks the presence of a sequence of messages ω in an input context $\mathcal{A}$, and extracts the residual input context.

$$\mathsf{extract}(\mathcal{A}, \omega) = \begin{cases} \mathcal{A} & \text{if } \omega = \epsilon \\ \mathsf{extract}(\mathcal{A}_i, \omega') & \text{if } \omega = a_i \cdot \omega', \mathcal{A} = \langle a_j : \mathcal{A}_j \rangle_{j \in J}, \text{ and } i \in J \\ \bot & \text{otherwise} \end{cases}$$

Our termination condition is formalised in Theorem 13 below. This result follows from an argument based on the finiteness of the states of M_1 and of the sets of states from M_2 (which populate the holes of the input contexts in the labels of the nodes in the simulation tree). We write $\mathsf{minHeight}(\mathcal{A})$ for the smallest $\mathsf{height}_i(\mathcal{A})$, with $i \in I(\mathcal{A})$, where $\mathsf{height}_i(\mathcal{A})$ is the length of the path from the root of the input context $\mathcal{A}$ to the i^{th} hole.

▶ **Theorem 13.** *Let $M_1 = (P, p_0, \delta_1)$ and $M_2 = (Q, q_0, \delta_2)$ be two communicating machines with $\mathsf{simtree}(M_1, M_2) = (N, n_0, \hookrightarrow, \mathcal{L}, Act, P \times \mathcal{T}_Q)$. For each infinite path $n_0 \hookrightarrow n_1 \hookrightarrow n_2 \cdots \hookrightarrow n_i \hookrightarrow \cdots$ there exist $i < j < k$, with*

$$\mathcal{L}(n_i) = p \preccurlyeq \mathcal{A}_i[q_h]^{h \in H_i} \qquad \mathcal{L}(n_j) = p \preccurlyeq \mathcal{A}_j[q'_h]^{h \in H_j} \qquad \mathcal{L}(n_k) = p \preccurlyeq \mathcal{A}_k[q''_h]^{h \in H_k}$$

s.t. $\{q'_h \mid h \in H_j\} \subseteq \{q_h \mid h \in H_i\}$ and $\{q''_h \mid h \in H_k\} \subseteq \{q_h \mid h \in H_i\}$; and, for $n_i \overset{\psi}{\hookrightarrow} n_j$:

i) $\mathsf{rcv}(\psi) = \omega_1 \cdot \omega_2$ *with ω_1 s.t. $\exists t, z.$* $\mathsf{extract}(\mathcal{A}_i, \omega_1) = [\,]_t \wedge \mathsf{extract}(\mathcal{A}_k, \omega_1) = [\,]_z$, *or*

ii) $\mathsf{minHeight}(\mathsf{extract}(\mathcal{A}_i, \mathsf{rcv}(\psi))) \leq \mathsf{minHeight}(\mathsf{extract}(\mathcal{A}_k, \mathsf{rcv}(\psi)))$.

Intuitively, the theorem above says that for each infinite branch in the simulation tree, we can find special nodes n_i, n_j and n_k such that the set of states in $\mathcal{A}_j$ (resp. $\mathcal{A}_k$) is included in that of $\mathcal{A}_i$ and the receive actions in the path from n_i to n_j are such that: either (i) only a precise prefix of such actions will be taken from the receive actions accumulated in n_i and n_k or (ii) all of them will be taken from the receive actions in which case n_k must have accumulated more receive actions than n_i. Case (i) deals with infinite branches with only finite labels (hence finite accumulation) while case (ii) considers those cases in which there is unbounded accumulation along the infinite branch.

Based on Theorem 13, the following *algorithm* generates a finite subtree of $\mathsf{simtree}(M_1, M_2)$:

> Compute, initially starting from the root, the branches[3] of $\mathsf{simtree}(M_1, M_2)$ stopping when one of the following types of node is encountered: a leaf, or a node n with a label already seen along the path from the root to n, or a node n_k (with the corresponding node n_i) as those described by the above Theorem 13.

▶ **Example 14.** Consider the finite subtree in Figure 2. It is precisely the finite subtree identified as described above: we stop generating the simulation tree at nodes n_2, n_6, n_{11}, and n_{14} (because their labels have been already seen at the corresponding ancestors n_0, n_4, n_8, and n_{12}) and n_{16} (because of the ancestors n_8 and n_{12} such that n_8, n_{12} and n_{16} correspond to the nodes n_i, n_j and n_k of Theorem 13).

[3] The order nodes are generated is not important (our implementation uses a DFS approach, cf. §4).

When the computed finite subtree contains an unsuccessful leaf, we can immediately conclude that the considered communicating machines are not related. Otherwise, we extract smaller finite subtrees (from the subtree) that are potential candidates to be subsequently checked.

We define the anc function as follows: for boundary nodes n with an ancestor n' such that $\mathcal{L}(n) = \mathcal{L}(n')$ we define $\mathsf{anc}(n) = n'$; for boundary nodes n_k (with the corresponding node n_i) as those described by in Theorem 13 we define $\mathsf{anc}(n_k) = n_i$. The extraction of the finite subtrees is done by characterising their roots (and taking as boundary the reachable boundary nodes): let $P = \{n \in \mathit{img}(\mathsf{anc}) \mid \exists n'.\ \mathsf{anc}(n') = n \wedge \mathcal{L}(n) \neq \mathcal{L}(n')\}$, the set of such roots is $R = \{n \in P \mid \nexists n' \in P.\ n' \hookrightarrow^{+} n\}$.

Intuitively, to extract subtrees, we restrict our attention to the set P of ancestors with a label different from their corresponding boundary node (corresponding to branches that can generate unbounded accumulation). We then consider the forest of subtrees rooted in nodes in P without an ancestor in P. Notice that for successful leaves we do not define anc; hence, only extracted subtrees without successful nodes have a completely defined anc function. These are candidate subtrees that will be checked as described in the next step.

▶ **Example 15.** Consider the finite subtree in Figure 2. Following the strategy above we extract from it the candidate subtree rooted at n_8 (white nodes), with boundary $\{n_{11}, n_{14}, n_{16}\}$. Note that each ancestor node above n_8 has a label identical to its boundary node.

Part 3. Checking whether the candidate subtrees are witnesses of infinite branches. The final step of our algorithm consists in verifying a property on the identified candidate subtrees which guarantees that all branches traversing the root of the candidate subtree are infinite, hence successful. A candidate subtree satisfies this property when it is also a *witness subtree*, which is the key notion (Definition 20) presented in this third part.

In order for a subtree to be a witness, we require that any behaviour in the simulation tree going beyond the subtree is the infinite repetition of the behaviour already observed in the considered finite subtree. This infinite repetition is only possible if whatever receive actions are accumulated in the input context $\mathcal{A}$ (using Rule (OutAcc)) are eventually executed by the candidate subtype M_1 in Rule (InCtx). The compatibility check between what receive actions can be accumulated and what is eventually executed is done by first synthesising a pair of intermediate automata from a candidate subtree, that represent the possible (repeated) accumulation of the candidate supertype M_2 and the possible (repeated) receive actions of the candidate subtype M_1, and then by checking that these automata are *compatible*. For convenience, we define these intermediate automata as a system of (possibly) mutually recursive equations, which we call a *system of input tree equations*.

▶ **Definition 16** (Input Tree Equations). *Given a set of variables $\mathcal{V}$, ranged over by X, an input tree definition is a term of the grammar* $E \ ::= \ X \mid \langle a_i : E_i\rangle_{i\in I} \mid \langle E_i\rangle_{i\in I}$.
A system of input tree equations is a tuple $\mathcal{G} = (\mathcal{V}, X_0, \mathbf{E})$ consisting of a set of variables $\mathcal{V}$, an initial variable $X_0 \in \mathcal{V}$, and with $\mathbf{E}$ consisting of exactly one input tree definition $X \stackrel{def}{=} E$, with $E \in \mathcal{T}_{\mathcal{V}}$, for each $X \in \mathcal{V}$, where $\mathcal{T}_{\mathcal{V}}$ denotes the input tree definitions on variables $\mathcal{V}$.

Given an input tree definition of the form $\langle a_i : E_i\rangle_{i\in I}$ or $\langle E_i\rangle_{i\in I}$, we assume that $I \neq \emptyset$, $\forall i \neq j \in I.\ a_i \neq a_j$, and that the order of the sub-terms is irrelevant. Whenever convenient, we use set-builder notation to construct an input tree definition, e.g., $\langle E_i \mid i \in I\rangle$. In an input tree equation, the construct $\langle a_i : E_i\rangle_{i\in I}$ represents the capability of accumulating (or actually executing) the receive actions on each message a_i then behaving as in E_i. The construct $\langle E_i\rangle_{i\in I}$ represents a *silent choice* between the different capabilities E_i.

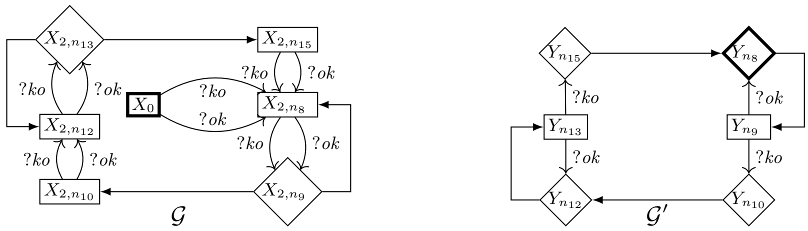


Figure 3 Graphical view of the input tree equations for $M_R \preccurlyeq M_C$ (Figure 1). The starting variables are X_0 and Y_{n_8}. Silent choices are diamond-shaped nodes, other nodes are rectangles.

▶ **Definition 17** (Input Tree Compatibility). *Given two systems of input tree equations $\mathcal{G} = (\mathcal{V}, X_0, \mathbf{E})$ and $\mathcal{G}' = (\mathcal{V}', X'_0, \mathbf{E}')$, such that $\mathcal{V} \cap \mathcal{V}' = \emptyset$, we say that $\mathcal{G}$ is* compatible *with $\mathcal{G}'$, written $\mathcal{G} \sqsubseteq \mathcal{G}'$, if there exists a relation $\mathcal{R} \subseteq \mathcal{T}_\mathcal{V} \times \mathcal{T}'_\mathcal{V}$ s.t. $(X_0, X'_0) \in \mathcal{R}$ and:*

- *if $(X, E) \in \mathcal{R}$ then $(E', E) \in \mathcal{R}$ with $X \overset{def}{=} E'$;*
- *if $(E, X) \in \mathcal{R}$ then $(E, E') \in \mathcal{R}$ with $X \overset{def}{=} E'$;*
- *if $(\langle E_i \rangle_{i\in I}, E) \in \mathcal{R}$ then $\forall i \in I.\ (E_i, E) \in \mathcal{R}$;*
- *if $(E, \langle E_i \rangle_{i\in I}) \in \mathcal{R}$ then $\forall i \in I.\ (E, E_i) \in \mathcal{R}$;*
- *if $(\langle a_i : E_i \rangle_{i\in I}, \langle a_j : E'_j \rangle_{j\in J}) \in \mathcal{R}$ then $I \subseteq J$ and $\forall i \in I.\ (E_i, E'_i) \in \mathcal{R}$.*

Intuitively, $\mathcal{G} \sqsubseteq \mathcal{G}'$ verifies the compatibility between $\mathcal{G}$, which represents the receive actions that can be accumulated in the context $\mathcal{A}$, and $\mathcal{G}'$, which represents the receive actions that can be actually executed. The first two items of Definition 17 let a variable be replaced by its definition. The next two items explore all the successors of silent choices. The last item guarantees that all the receive actions accumulated in $\mathcal{G}$, cf. Rule (OutAcc), can be actually matched by receive actions in $\mathcal{G}'$, cf. Rule (InCtx).

▶ **Example 18.** A graphical representations of two systems of input tree equations is in Figure 3. We have $\mathcal{G} \sqsubseteq \mathcal{G}'$ since all non-silent choices have the same outgoing transitions.

Before giving the definition of witness subtree, we introduce a few auxiliary functions on which it relies. Given $\omega \in \mathbb{A}^*$, and a state $q \in Q$, we define $\mathsf{accTree}(q, \omega)$ as follows:

$$\mathsf{accTree}(q,\omega) = \begin{cases} [q]_k \text{ with } k \text{ fresh} & \text{if } \omega = \epsilon \\ \mathcal{A}[\mathsf{accTree}(q'_i, \omega')]^{i\in I} & \text{if } \omega = a \cdot \omega', \mathcal{A}[q_i]^{i\in I} = \mathsf{inTree}(q), \forall i \in I.\ q_i \xrightarrow{!a} q'_i \\ \bot & \text{otherwise} \end{cases}$$

Function $\mathsf{accTree}(q, \omega)$ is a key ingredient of the witness subtree definition as it allows for the construction of the accumulation of receive actions (represented as an input tree) that is generated from a state q mimicking the sequence of send actions sending the messages in ω.

We use the auxiliary function $\mathsf{minAcc}(n, q, \psi)$ below to ensure that the effect of performing the transitions from an ancestor to a boundary node is that of increasing (possibly non-strictly) the accumulated receive actions. Here, n represents a known lower bound for the length of a sequence of receive actions accumulated in an input context $\mathcal{A}$, i.e., the length of a path from the root of $\mathcal{A}$ to one of its holes. Assuming that this hole contains the state q, the function returns a lower bound for the length of such a sequence of accumulated receive actions after the transitions in ψ have been executed. Formally, given a natural number n, a sequence of action $\psi \in Act^*$, and a state $q \in Q$ we define this function as follows:

$$\mathsf{minAcc}(n, q, \psi) = \begin{cases} n & \text{if } \psi = \epsilon \\ \mathsf{minAcc}(n-1, q, \psi') & \text{if } \psi = ?a \cdot \psi' \wedge n > 0 \\ \min_{i \in I} \mathsf{minAcc}(n + \mathsf{height}_i(\mathcal{A}), q_i, \psi') & \text{if } \psi = !a \cdot \psi' \wedge \mathsf{accTree}(q, a) = \mathcal{A}[q_i]^{i \in I} \\ \bot & \text{otherwise} \end{cases}$$

▶ **Example 19.** Consider the transitions from node n_7 to n_9 in Figure 2. There are two send actions $!lq$ and $!hq$ that cannot be directly fired from state q_2 which is a receiving state; the effect is to accumulate receive actions. Such an accumulation is computed by $\mathsf{accTree}(q_2, lq \cdot hq) = \langle ko : \langle ko : q_2, ok : q_2 \rangle, ok : \langle ko : q_2, ok : q_2 \rangle\rangle$. For this sequence of transitions, the effect on the (minimal) length of the accumulated receive actions can be computed by $\mathsf{minAcc}(0, q_2, !lq \cdot !hq) = 2$; meaning that before executing the sequence of transitions $!lq \cdot !hq$ state q_2 has not accumulated receive actions in front, while at the end an input context with minimal depth 2 is generated as accumulation.

We finally give the definition of witness subtree.

▶ **Definition 20** (Witness Subtree). *Let $M_1 = (P, p_0, \delta_1)$ and $M_2 = (Q, q_0, \delta_2)$ be two communicating machines with $\mathsf{simtree}(M_1, M_2) = (N, n_0, \hookrightarrow, \mathcal{L}, Act, P \times \mathcal{T}_Q)$. A candidate subtree of $\mathsf{simtree}(M_1, M_2)$ with root r and boundary B is a* witness *if the following holds:*

1. *For all $n \in B$, given ψ such that $\mathsf{anc}(n) \overset{\psi}{\hookrightarrow} n$, we have $|\mathsf{rcv}(\psi)| > 0$.*
2. *For all $n \in img(\mathsf{anc})$ and $n' \in img(\mathsf{anc}) \cup B$ such that $n \overset{\psi}{\hookrightarrow} n'$, $\mathcal{L}(n) = p \preccurlyeq \mathcal{A}[q_i]^{i \in I}$, and $\mathcal{L}(n') = p' \preccurlyeq \mathcal{A}'[q_j]^{j \in J}$, we have that $\forall i \in I$:*
   a. *$\{q_h \mid h \in H \text{ s.t. } \mathsf{accTree}(q_i, \mathsf{snd}(\psi)) = \mathcal{A}''[q_h]^{h \in H}\} \subseteq \{q_j \mid j \in J\}$;*
   b. *if $n' \in B$ then $\mathsf{minAcc}(\mathsf{minHeight}(\mathcal{A}), q_i, \psi) \geq \mathsf{minHeight}(\mathcal{A})$.*
3. *$\mathcal{G} \sqsubseteq \mathcal{G}'$ where*
   a. *$\mathcal{G} = (\{X_0\} \cup \{X_{q,n} \mid q \in Q, n \in \mathsf{nodes}(S, r, B) \setminus B\}, X_0, \mathbf{E})$ with $\mathbf{E}$ defined as follows:*
      i. *$X_0 \overset{def}{=} T\{X_{q,r}/q \mid q \in Q\}$, with $\mathcal{L}(r) = p \preccurlyeq T$*
      ii. *$X_{q,n} \overset{def}{=}$*

$$\begin{cases} \langle X_{q,tr(n')} \mid \exists a.n \overset{?a}{\hookrightarrow} n' \rangle & \text{if } \exists a.n \overset{?a}{\hookrightarrow} \\ \langle \mathcal{A}[X_{q'_i,tr(n')}]^{i \in I} \mid \exists a.n \overset{!a}{\hookrightarrow} n' \wedge \mathsf{inTree}(q) = \mathcal{A}[q_i]^{i \in I} \wedge \forall i \in I. q_i \overset{!a}{\longrightarrow} q'_i \rangle & \text{otherwise} \end{cases}$$

   b. *$\mathcal{G}' = (\{Y_n \mid n \in \mathsf{nodes}(S, r, B) \setminus B\}, Y_r, \mathbf{E}')$ with $\mathbf{E}'$ defined as follows:*

$$Y_n \overset{def}{=} \begin{cases} \langle Y_{tr(n')} \mid n \overset{!a}{\hookrightarrow} n' \rangle & \text{if } \exists n'.n \overset{!a}{\hookrightarrow} n' \\ \langle a : Y_{tr(n')} \mid n \overset{?a}{\hookrightarrow} n' \rangle & \text{if } \exists n'.n \overset{?a}{\hookrightarrow} n' \end{cases} \qquad \text{with } tr(n) = n, \text{ if } n \notin B; \; tr(n) = \mathsf{anc}(n), \text{ otherwise.}$$

Condition (1) requires the existence of a receive transition between an ancestor and a boundary node. This implies that if the behaviour beyond the witness subtree is the repetition of behaviour already observed in the subtree, then there cannot be send-only cycles. Condition (2a) requires that the transitions from ancestors to boundary nodes (or to other ancestors) are such that they include those behaviour that can be computed by the accTree function. We assume that this condition does not hold if $\mathsf{accTree}(q_i, \mathsf{snd}(\psi)) = \bot$ for any $i \in I$; hence the states q_i of M_2 in an ancestor are able to mimic all the send actions performed by M_1 along the sequences of transitions in the witness subtree starting from the considered ancestor. Condition (2b) ensures that by repeating transitions from ancestors to boundary nodes, the accumulation of receive actions is, overall, increasing. In other words, the rate at which accumulation is taking place is higher than the rate at which the context is reduced by Rule (InCtx). Condition (3) checks that the receive actions that can be accumulated by M_2(represented by $\mathcal{G}$) and those that are expected to be actually executed by M_1 (represented by $\mathcal{G}'$) are compatible. In $\mathcal{G}$, there is an equation for the root node and for each pair consisting of a local state in M_2 and a node n in the witness subtree. The equation for the root node is given in (3(a)i), where we simply transform an input context into an input tree definition. The other equations are given in (3(a)ii), where we use the partial function $\mathsf{inTree}(q)$. Each equation represents what can be accumulated by starting from node n (focusing on local state q). In $\mathcal{G}'$, there is an equation for each node n in the witness subtree, as defined in (3b) There are two types of equations depending on type of transitions outgoing from node n. A send transition leads to silent choices, while receive transitions generate corresponding receive choices.

▶ **Example 21.** The candidate subtree rooted at n_8 in Figure 2 satisfies Definition 2. (1) Each path from an ancestor to a boundary node includes at least one receive action. (2a) For each sequence of transitions from an ancestor to a boundary node (or another ancestor) the behaviour of the states of M_2, as computed by the $\mathsf{accTree}$ function, has already been observed. (2b) For each sequence of transitions from an ancestor to a boundary node, the rate at which receive actions are accumulated is higher than or equal to the rate at which they are removed from the accumulation. (3) The systems of input tree equations $\mathcal{G}$ (3a) and $\mathcal{G}'$ (3b) are given in Figure 3, and are compatible, see Example 18.

We conclude by stating our main result; given a simulation tree with a witness subtree with root r, all the branches in the simulation tree traversing r are infinite (hence successful).

▶ **Theorem 22.** *Let $M_1 = (P, p_0, \delta_1)$ and $M_2 = (Q, q_0, \delta_2)$ be two communicating machines with $\mathsf{simtree}(M_1, M_2) = (N, n_0, \hookrightarrow, \mathcal{L}, Act, P \times \mathcal{T}_Q)$. If $\mathsf{simtree}(M_1, M_2)$ has a witness subtree with root r then for every node $n \in N$ such that $r \hookrightarrow^* n$ there exists n' such that $n \hookrightarrow n'$.*

Hence, we can conclude that if the candidate subtrees of $\mathsf{simtree}(M_1, M_2)$ identified following the strategy explained in Part (2) are also witness subtrees, then we have $M_1 \preccurlyeq M_2$.

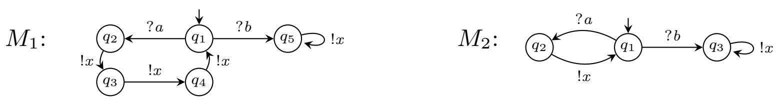
▶ Remark 23. When our algorithm finds a successful leaf, a previously seen label, or a witness subtree in each branch then the machines are in the subtyping relation. If an unsuccessful leaf is found (while generating the initial finite subtree as described in Part (2)), then the machines are *not* in the subtyping relation. In all other cases, the algorithm is unable to give a decisive verdict (i.e., the result is *unknown*). There are two possible causes for an unknown result: either (i) it is impossible to extract a forest of candidate subtrees (i.e., there are successful leaves below some ancestor) or (ii) some candidate subtree is not a witness.

4 Evaluation, Related Work, and Conclusions

Evaluation. To evaluate the cost and applicability of our algorithm, we have produced a faithful implementation of it, which constructs the simulation tree in a depth-first search manner, while recording the nodes visited in different branches to avoid re-computing several times the same subtrees. We have run our tool on 174 tests which were either taken from the literature on asynchronous subtyping [10, 24], or handcrafted to test the limits of our approach. All of these tests terminate under a second. Out of these tests, 92 are *negative* (the types are not in the subtyping relation) and our tool gives the expected result (*false*) for all of them. The other 82 tests are *positive* (the types are in the subtyping relation) and our tool gives the expected result (*true*) for all but 8 tests, for which it returns *unknown*. All of these 8 examples feature complex accumulation patterns that our theory cannot recognise, e.g., Example 24. The implementation and our test data are available on our GitHub repository [5].

The implementation includes an additional optimisation which performs a check for $\overline{M_2} \preccurlyeq \overline{M_1}$ (relying on a previous result showing that $M_1 \preccurlyeq M_2 \iff \overline{M_2} \preccurlyeq \overline{M_1}$ [7, 24]) when the result of checking $M_1 \preccurlyeq M_2$ is *unknown*.

▶ **Example 24.** Given the machines below, $\mathsf{simtree}(M_1, M_2)$ contains infinitely many nodes with labels of the form: $q_1 \preccurlyeq \langle a : \langle a : \langle a : \langle \cdots \rangle , b : q_3 \rangle , b : q_3 \rangle , b : q_3 \rangle$.



Each of these nodes has two successors, one where $?a$ is fired (the machines stay in the larger loop), and one where $?b$ is fired (the machines move to their smaller loop). The machines can always enter this send-only cycle, hence Condition (1) of Definition 20 never applies.

Related work. Gay and Hole [16, 17] introduced (synchronous) subtyping for session types and show it is decidable. Mostrous et al. [27] adapted the notion of session subtyping to asynchronous communication, by introducing delayed inputs. Later, Chen et al. [10, 11] provided an alternative definition prohibiting orphan messages, we used this definition in this work. Recently, asynchronous subtyping was shown to be undecidable by encoding it as an equivalent question in the setting of Turing machines [24] and queue machines [7]. Recent work [7, 8, 24] investigated restrictions to achieve decidability, these restrictions are either on the size of the FIFO channels or syntactical. In the latter case, we recall the single-out and single-in restrictions, i.e., where all output (respectively input) choices are singletons.

The relationship between communicating machines and (multiparty asynchronous) session types has been studied in [13, 14]. Communicating machines are Turing-complete, hence most of their properties are undecidable [6]. Many variations have been introduced in order to recover decidability, e.g., using (existential or universal) bounds [18], restricting to different types of topologies [22, 31], or using bag or lossy channels instead of FIFO queues [1, 2, 9, 12].

Conclusions and future work. We have proposed a sound algorithm for checking asynchronous session subtyping, showing that it is still possible to decide whether two types are related for many nontrivial examples. Our algorithm is based on a (potentially infinite) tree representation of the coinductive definition of asynchronous subtyping; it checks for the presence of finite witnesses of infinite successful subtrees. We have provided an implementation and applied it to examples that cannot be recognised by previous approaches. Although the (worst-case) complexity of our algorithm is rather high (the termination condition expects to encounter a set of states already encountered, of which there may be exponentially many), our implementation shows that it actually terminates under a second for machines of size comparable to typical communication protocols used in real programs, e.g., Go programs feature between three and four communication primitives per channel and whose branching construct feature two branches, on average [15].

As future work, we plan to enrich our algorithm to recognise subtypes featuring more complex accumulation patterns, e.g., Example 24. Moreover, due to the tight correspondence with safety of communicating machines [24], we plan to investigate the possibility of using our approach to characterise a novel decidable subclass of communicating machines.

References

1. Parosh Aziz Abdulla, Ahmed Bouajjani, and Bengt Jonsson. On-the-Fly Analysis of Systems with Unbounded, Lossy FIFO Channels. In *CAV 1998*, pages 305–318, 1998. doi:10.1007/BFb0028754.
2. Parosh Aziz Abdulla and Bengt Jonsson. Verifying Programs with Unreliable Channels. In *(LICS 1993)*, pages 160–170, 1993. doi:10.1109/LICS.1993.287591.
3. Roberto M. Amadio and Luca Cardelli. Subtyping Recursive Types. *ACM Trans. Program. Lang. Syst.*, 15(4):575–631, 1993. doi:10.1145/155183.155231.
4. Davide Ancona, Viviana Bono, Mario Bravetti, Joana Campos, Giuseppe Castagna, Pierre-Malo Deniélou, Simon J. Gay, Nils Gesbert, Elena Giachino, Raymond Hu, Einar Broch Johnsen, Francisco Martins, Viviana Mascardi, Fabrizio Montesi, Rumyana Neykova, Nicholas Ng, Luca Padovani, Vasco T. Vasconcelos, and Nobuko Yoshida. Behavioral Types in Programming Languages. *Foundations and Trends in Programming Languages*, 3(2-3):95–230, 2016. doi:10.1561/2500000031.
5. The Authors. A sound algorithm for asynchronous session subtyping. https://github.com/julien-lange/asynchronous-subtyping, 2019.
6. Daniel Brand and Pitro Zafiropulo. On Communicating Finite-State Machines. *J. ACM*, 30(2):323–342, 1983. doi:10.1145/322374.322380.
7. Mario Bravetti, Marco Carbone, and Gianluigi Zavattaro. Undecidability of asynchronous session subtyping. *Inf. Comput.*, 256:300–320, 2017.
8. Mario Bravetti, Marco Carbone, and Gianluigi Zavattaro. On the boundary between decidability and undecidability of asynchronous session subtyping. *Theor. Comput. Sci.*, 722:19–51, 2018. doi:10.1016/j.tcs.2018.02.010.
9. Gérard Cécé, Alain Finkel, and S. Purushothaman Iyer. Unreliable Channels are Easier to Verify Than Perfect Channels. *Inf. Comput.*, 124(1):20–31, 1996. doi:10.1006/inco.1996.0003.
10. Tzu-Chun Chen, Mariangiola Dezani-Ciancaglini, Alceste Scalas, and Nobuko Yoshida. On the Preciseness of Subtyping in Session Types. *Logical Methods in Computer Science*, 13(2), 2017. doi:10.23638/LMCS-13(2:12)2017.
11. Tzu-Chun Chen, Mariangiola Dezani-Ciancaglini, and Nobuko Yoshida. On the Preciseness of Subtyping in Session Types. In *PPDP 2014*, pages 146–135. ACM Press, 2014.
12. Lorenzo Clemente, Frédéric Herbreteau, and Grégoire Sutre. Decidable Topologies for Communicating Automata with FIFO and Bag Channels. In *CONCUR 2014*, pages 281–296, 2014. doi:10.1007/978-3-662-44584-6_20.
13. Pierre-Malo Deniélou and Nobuko Yoshida. Multiparty Session Types Meet Communicating Automata. In *ESOP 2012*, pages 194–213, 2012. doi:10.1007/978-3-642-28869-2_10.
14. Pierre-Malo Deniélou and Nobuko Yoshida. Multiparty Compatibility in Communicating Automata: Characterisation and Synthesis of Global Session Types. In *ICALP 2013*, pages 174–186, 2013. doi:10.1007/978-3-642-39212-2_18.
15. N. Dilley and J. Lange. An Empirical Study of Messaging Passing Concurrency in Go Projects. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 377–387, February 2019. doi:10.1109/SANER.2019.8668036.
16. Simon J. Gay and Malcolm Hole. Types and Subtypes for Client-Server Interactions. In *ESOP 1999*, pages 74–90, 1999. doi:10.1007/3-540-49099-X_6.
17. Simon J. Gay and Malcolm Hole. Subtyping for session types in the pi calculus. *Acta Inf.*, 42(2-3):191–225, 2005. doi:10.1007/s00236-005-0177-z.
18. Blaise Genest, Dietrich Kuske, and Anca Muscholl. On Communicating Automata with Bounded Channels. *Fundam. Inform.*, 80(1-3):147–167, 2007. URL: http://content.iospress.com/articles/fundamenta-informaticae/fi80-1-3-09.
19. Raymond Hu and Nobuko Yoshida. Hybrid Session Verification Through Endpoint API Generation. In *FASE 2016*, pages 401–418, 2016. doi:10.1007/978-3-662-49665-7_24.
20. Petr Jancar and Faron Moller. Techniques for Decidability and Undecidability of Bisimilarity. In *CONCUR 1999*, pages 30–45, 1999. doi:10.1007/3-540-48320-9_5.

21 Dexter Kozen, Jens Palsberg, and Michael I. Schwartzbach. Efficient Recursive Subtyping. *Mathematical Structures in Computer Science*, 5(1):113–125, 1995. `doi:10.1017/S0960129500000657`.

22 Salvatore La Torre, P. Madhusudan, and Gennaro Parlato. Context-Bounded Analysis of Concurrent Queue Systems. In *TACAS 2008*, pages 299–314, 2008. `doi:10.1007/978-3-540-78800-3_21`.

23 Julien Lange and Nobuko Yoshida. Characteristic Formulae for Session Types. In *TACAS*, volume 9636 of *Lecture Notes in Computer Science*, pages 833–850. Springer, 2016.

24 Julien Lange and Nobuko Yoshida. On the Undecidability of Asynchronous Session Subtyping. In *FoSSaCS*, volume 10203 of *Lecture Notes in Computer Science*, pages 441–457, 2017.

25 Sam Lindley and J. Garrett Morris. Embedding session types in Haskell. In *Haskell 2016*, pages 133–145, 2016. `doi:10.1145/2976002.2976018`.

26 Dimitris Mostrous and Nobuko Yoshida. Session typing and asynchronous subtyping for the higher-order π-calculus. *Inf. Comput.*, 241:227–263, 2015. `doi:10.1016/j.ic.2015.02.002`.

27 Dimitris Mostrous, Nobuko Yoshida, and Kohei Honda. Global Principal Typing in Partially Commutative Asynchronous Sessions. In *ESOP 2009*, pages 316–332, 2009. `doi:10.1007/978-3-642-00590-9_23`.

28 Rumyana Neykova, Raymond Hu, Nobuko Yoshida, and Fahd Abdeljallal. A Session Type Provider: Compile-time API Generation for Distributed Protocols with Interaction Refinements in F$\sharp$. In *CC 2018*. ACM, 2018.

29 Dominic A. Orchard and Nobuko Yoshida. Effects as sessions, sessions as effects. In *POPL 2016*, pages 568–581, 2016. `doi:10.1145/2837614.2837634`.

30 Luca Padovani. A simple library implementation of binary sessions. *J. Funct. Program.*, 27:e4, 2017. `doi:10.1017/S0956796816000289`.

31 Wuxu Peng and S. Purushothaman. Analysis of a Class of Communicating Finite State Machines. *Acta Inf.*, 29(6/7):499–522, 1992. `doi:10.1007/BF01185558`.

32 Alceste Scalas and Nobuko Yoshida. Lightweight Session Programming in Scala. In *ECOOP 2016*, pages 21:1–21:28, 2016. `doi:10.4230/LIPIcs.ECOOP.2016.21`.