Symbolic Bisimulation for a Higher-Order Distributed Language with Passivation[⋆]
(Extended Abstract)

Vasileios Koutavas and Matthew Hennessy

Trinity College Dublin
{Vasileios.Koutavas,Matthew.Hennessy}@scss.tcd.ie

Abstract. We study the behavioural theory of a higher-order distributed calculus with private names and locations that can be passivated. For this language, we present a novel Labelled Transition System where higher-order inputs are symbolic agents that can perform a limited number of transitions, capturing the nature of passivation. Standard first-order weak bisimulation over this LTS coincides with contextual equivalence, and provides the first useful proof technique without a universal quantification over contexts for an intricate distributed language.

1 Introduction

Higher-order concurrency naturally arises from the combination of functional and concurrent programming. In many concurrency scenarios, processes have the ability to exchange values over communication channels; in languages with functional characteristics, besides constants of base type, these values include code in the form of function closures of higher types.

The behaviour of processes in simple higher-order concurrency has been studied in the setting of Higher-Order π-calculus (HOπ) [11] and CHOCS [1]. The former work showed that higher-order systems can be translated and studied in first-order π-calculus [12]. The translation is based on the notion of a *trigger*, a simple value representing a function which, when run within a process, triggers the execution of the function in another part of the system. This translation gave rise to *normal bisimulation*, a first-order bisimulation method in which the observer need only examine a process using finite trigger values, enabling simple proofs of equivalence. This proof method is both sound and complete with respect to a natural contextual equivalence, called *barbed congruence* [12, 4, 6].

For *distributed systems*, however the approach of the trigger translation is generally not applicable [15, 7]. The intuition is that in many distributed scenarios, the observable runtime behaviour of a higher-order value depends on the location in which it is run. One of the simplest extensions to HOπ where this *location-dependent behaviour* becomes apparent is when *passivation* is added to the language, as in HOπP [7] which uses transparent locations whose contents

[⋆] This research was supported by SFI project SFI 06 IN.1 1898.

can be passivated and restarted in a different context. This simple construct is sufficient to demonstrate the intricacies arising when communication is location dependent; for example, processes can become temporarily isolated from the external observer, or indeed from other components of the system, essentially encoding *communication barriers*. The resulting behavioural complexity is emphasised by the results for HOπP, which show that extensions of triggers to include arbitrary finite values do not capture contextual equivalence [7, Sec. 6].

The purpose of this paper is to demonstrate that contextual equivalence for distributed systems exhibiting location-dependent behaviour can be captured by a first-order bisimulation semantics in which triggers are replaced by simple *symbolic agents*. We give the first sound and complete first-order bisimulation technique for equivalence in a higher-order distributed language with passivation and private names, which avoids universal quantification over contexts.

The starting point is the *labelled transition system* (LTS) semantics in previous work for HOπ [6], encoding Sangiorgi's triggers as symbolic constants. These constants represent the actual higher-order values transmitted between the observer and the system under observation. This limits the size and complexity of the resulting LTS as these constants can only be subsequently used by the observer to run code produced by the system, and by the system to signal the execution of observer-generated symbolic code. However, for location-dependent behaviour the repertoire of symbolic constants has to be enlarged to what we call *symbolic agents*, a small collection of probes designed to facilitate two kinds of observations capturing the nature of passivation and more generally location-dependent behaviour. The first is to discover if locations in systems where agents are running can communicate with other locations and the observer. The other is to examine the system when system-emitted code runs at agent locations.

The language we consider is a minor variation of HOπP in which, because of lazy scope extrusion of π-calculus names, contextual equivalence can distinguish between systems solely on the basis of their free names [7, Sec. 2.4]. This infelicity is avoided in a variant called HOπPn [10] in which π-calculus restriction is replaced by name allocation; however, this sacrifices expressiveness since basic programming constructs such as recursion and internal choice are not programmable (see Thm. 3.4). In this paper we opt for a passivation language HOPass which, like HOπPn, avoids the complications with free names of HOπP, but also *can* encode useful programming constructs. Our language essentially adds CCS-style local communication ports to HOπPn solely for the purpose of programmability.

Both HOπP and HOπPn have coinductive characterisations of contextual equivalence, in terms of *weak context bisimulation* [7] and *weak environmental bisimulation* [10], respectively. However, the former does not provide a viable proof technique for equivalence because of a significant universal quantification over contexts. The latter also contains a similar quantification; however, powerful *up-to techniques* [13] can certainly help with constructing witness bisimulations.

The symbolic agent LTS in this paper avoids any quantification over contexts and provides a viable proof technique relying only on standard (weak) first-order

$$
\begin{array}{lrcl}
 & a,\dots,t \in \mathsf{GName} & p_{\mathrm{L}} \in \mathsf{LPort} & x,y,z \in \mathsf{Var} \\
 & u,v \in \mathsf{GName} \cup \mathsf{LPort} & & \widehat{u},\widehat{v} \in \mathsf{Var} \cup \mathsf{GName} \cup \mathsf{LPort} \\
\mathsf{Val}: & V ::= a \mid \lambda P & & \widehat{V} \in \mathsf{Var} \cup \mathsf{Val} \\
\mathsf{Proc}: & \multicolumn{3}{l}{P,Q ::= \mathbb{0} \mid \widehat{u}!\widehat{V}.P \mid \widehat{u}?(x{:}T).P \mid (P \mid P) \mid \mathtt{if}\ \widehat{V} = \widehat{V}\ \mathtt{then}\ P\ \mathtt{else}\ P} \\
 & \multicolumn{3}{l}{\quad \mid \ \mathtt{new}\ x.P \mid P\backslash p_{\mathrm{L}} \mid \mathtt{run}\ \widehat{V} \mid \widehat{u}[\![P]\!]} \\
\mathsf{Sys}: & M,N ::= P \mid \nu a.M & \mathsf{Type}: & T ::= \mathsf{Nm} \mid \mathsf{Pr}
\end{array}
$$

Fig. 1. Syntax of HOPass

bisimulation. The usefulness of first-order techniques have been demonstrated for HOπ [12, 4–6], and are equally useful for HOPass. Additionally, our proof technique reduces the size of bisimulations in proofs of equivalence by minimising the number of symbolic transitions that need to be considered. Moreover, we believe that our symbolic agent semantics can be adapted to other distributed languages with location-dependent behaviour, including HOπP.

We continue with the description of HOPass (Sect. 2) and contextual equivalence (Sect. 3). We then explain the intuitions of our LTS (Sect. 4) and detail its symbolic agent transitions (Sect. 5). The sound and complete bisimulation technique and an example equivalence are given in Sect(s). 6 and 7, respectively.

2 The Language HOPass

The abstract syntax of HOPass is shown in Fig. 1. *Generated names* (GName) are used for general communication channels between processes, and *local ports* (LPort) for programming via CCS-style locally scoped communication. *Values* (Val) are the objects transmitted over channels which can be first-order generated names of type Nm or higher-order *code thunks* of type Pr. Terms in HOPass are constructed in two levels: the inner level of *processes* (Proc) and the outer level of *systems* (Sys). A process can be one of the usual π-calculus inert ($\mathbb{0}$), output ($c!V.P$), input ($c?(x{:}T).P$), parallel ($P \mid Q$), and conditional process. Because channels can carry two types of values, we use the type annotation T at input processes and a simple dynamic type system to rule out stuck processes (see [6]).

Processes can also create at runtime a fresh generated name ($\mathtt{new}\ x.P$), and restrict local ports which are CCS channels used for programmability ($P\backslash p_{\mathrm{L}}$). We will only reason about *closed processes* with no free local ports or variables but can have free generated names. Finally, a process can execute a code thunk ($\mathtt{run}\ V$) and run a process within a location u ($u[\![P]\!]$). As we will see, P can reduce inside u and can be passivated by an input on u. A system takes the form $\nu n_1 \dots \nu n_i.P$, consisting of a single process P with a number of bound names. We use the Barendregt convention for bound generated names. Free generated names and local ports are given by fn$(-)$ and flp$(-)$, respectively; we use $(- \sharp -)$ to mean "have disjoint names and ports".

The reduction semantics of systems, $M \to N$, is defined in terms of labelled transitions of processes $P \xrightarrow{\lambda} N$ (Fig. 2). A process transition is annotated with a label indicating an output ($u!V$), input ($u?V$), internal transition (τ), or fresh

Process Transitions

RPin
$$\frac{\vdash_{\mathsf{V}} V : T}{u?(x{:}T).P \xrightarrow{u?V} P\{V/x\}}$$

RPout
$$\frac{\mathrm{flp}(V) = \emptyset}{c!V.P \xrightarrow{c!V} P}$$

RPcommL
$$\frac{P \xrightarrow{u!V} P' \qquad Q \xrightarrow{u?V} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'}$$

RPnew
$$\frac{n \sharp P}{\texttt{new}\, x.P \xrightarrow{\texttt{new}\, n} P\{n/x\}}$$

RPpass
$$\frac{\mathrm{flp}(V) = \emptyset}{c[\![P]\!] \xrightarrow{c!\lambda P} \mathbf{0}}$$

RPmatch
$$\frac{}{\texttt{if}\; a = a\; \texttt{then}\; P\; \texttt{else}\; Q \xrightarrow{\tau} P}$$

RPport
$$\frac{P \xrightarrow{\lambda} Q \qquad p_{\mathrm{L}} \sharp \lambda}{P \backslash p_{\mathrm{L}} \xrightarrow{\lambda} Q \backslash p_{\mathrm{L}}}$$

RPoutPort
$$\frac{}{p_{\mathrm{L}}!V.P \xrightarrow{p_{\mathrm{L}}!V} P}$$

RPmismatch
$$\frac{a \neq b}{\texttt{if}\; a = b\; \texttt{then}\; P\; \texttt{else}\; Q \xrightarrow{\tau} Q}$$

RPrun
$$\frac{}{\texttt{run}\, \lambda P \xrightarrow{\tau} P}$$

RPloc
$$\frac{P \xrightarrow{\lambda} Q}{u[\![P]\!] \xrightarrow{\lambda} u[\![Q]\!]}$$

RPparL
$$\frac{P \xrightarrow{\lambda} P' \qquad \mathrm{new}(\lambda) \sharp Q}{P \mid Q \xrightarrow{\lambda} P' \mid Q}$$

System Reductions

RSnew
$$\frac{P \xrightarrow{\texttt{new}\, n} P'}{P \to \nu n.P'}$$

RSτ
$$\frac{P \xrightarrow{\tau} P'}{P \to P'}$$

RSν
$$\frac{M \to M'}{\nu n.M \to \nu n.M'}$$

Fig. 2. Reduction semantics of HOPass (omitting symmetric rules and RPpassPort)

name generation ($\texttt{new}\, n$). Output (RPout) and passivation (RPpass) over generated names transmit closed values; over local ports (RPoutPort, RPpassPort) they can transmit values with free local ports, enabling the encoding of useful programming idioms (see Thm. 3.4) while avoiding the extrusion of local ports. Input (RPin) receives values of the appropriate type; here $\vdash_{\mathsf{V}} \lambda P : \mathsf{Pr}$ (for any P) and $\vdash_{\mathsf{V}} n : \mathsf{Nm}$, and $P\{V/x\}$ is capture-avoiding substitution. A new generated name is fresh because of the side-conditions in RPnew and RPparL; in the latter rule $\mathrm{new}(\lambda)$ denotes $\{n\}$, when $\lambda = \texttt{new}\, n$, and $\emptyset$ otherwise. The rest are standard rules for running a code thunk (RPrun), communication (RPcommL), equality testing (RPmatch, RPmismatch), and propagating transitions over evaluation contexts (RPparL, RPloc, RPport). System reductions simply bind freshly generated names and propagate internal process transitions. In the following we will use usual syntactic abbreviations from CCS and π-calculus.

3 Contextual Equivalence

In this paper we study *barbed congruence* [8], the contextual equivalence associated with weak bisimulation, which is reduction-closed, preserves weak barbs, and is a congruence. A *weak barb* is the ability of a system to perform an output on a free channel after a number of reductions.

Definition 3.1 (Weak Barb). *M has a weak barb b, written as $M \Downarrow_b$, when $M \to^* \nu\widetilde{n}.P$ and $P \xrightarrow{b!V} Q$, for some $\widetilde{n}$, P, Q, and V with $b \sharp \widetilde{n}$.*

We use standard, single-hole contexts derived from the language grammar, adding a process hole. We use the Barendregt convention only for bound names whose scope does not extend over the hole. We write $K[P]$ to mean the system obtained by replacing the hole of K with process P, and $K[M]$ to mean $\nu\widetilde{n}.\nu\widetilde{m}.K[P]$ when $K = \nu\widetilde{n}.K$ and $M = \nu\widetilde{m}.P$. Due to the convention for the bound names $\widetilde{m}$ we have $\widetilde{m} \sharp K, \widetilde{n}$; however, $\widetilde{n}$ may appear in P.

Definition 3.2 (Contextual Equivalence $(\cong_{\text{cxt}})$). *The relation* $(\cong_{\text{cxt}})$ *on systems is the largest relation such that if* $M \cong_{\text{cxt}} M'$ *then*

1. *For all* b, $M \Downarrow_b$ *iff* $M' \Downarrow_b$.
2. *If* $M \to N$ *then there exists* N' *such that* $M' \to^* N'$ *and* $N \cong_{\text{cxt}} N'$.
3. *If* $M' \to N'$ *then there exists* N *such that* $M \to^* N$ *and* $N \cong_{\text{cxt}} N'$.
4. *For any context* K_{S}, $K_{\mathsf{S}}[M] \cong_{\text{cxt}} K_{\mathsf{S}}[M']$.

If we remove passivation from this language we obtain a language similar to HOπ [11]. It is known that for such a language $(\cong_{\text{cxt}})$ coincides with the version of contextual equivalence which only requires preservation of the relation under parallel contexts (cf. Thm. 3.2). As the following example shows, this is not the case in the presence of passivation.

Example 3.3 (Passivation). Consider the systems

$$M_{3.3} = a!(\lambda b!).\mathbb{0} \qquad\qquad M'_{3.3} = \texttt{new}\, k.\, (a!(\lambda k!).\, *(k?b!))$$

These systems are indistinguishable if we consider only parallel contexts. The intuition is that both systems output a code thunk to any parallel context. The former outputs $\lambda b!$, becoming $\mathbb{0}$, and the latter outputs $\lambda k!$, leaving behind $*(k?b!)$, a replicated process defined in Thm. 3.4. In the case of $M_{3.3}$, the parallel context can essentially only run $\lambda b!$ producing a b-barb (possibly multiple times). Because k is never revealed to the context, in the case of $M'_{3.3}$ whenever the parallel context runs $\lambda k!$ it will again trigger a b-barb. Thus, no parallel context is able to induce an observable difference between $M_{3.3}$ and $M'_{3.3}$.

However, $(\cong_{\text{cxt}})$ considers contexts that run $M_{3.3}$ and $M'_{3.3}$ in a location l, enabling the passivation of process $*(k?b!)$ and distinguishing the behaviour of the two systems. The distinguishing context $K_{3.3} = l[\![\, [\cdot] \,]\!] \mid a?(x).\, l?.\, \texttt{run}\, x$ can input the code from channel a, passivate location l, and run the received code. Thus, $K_{3.3}[M_{3.3}] \Downarrow_b$ but $K_{3.3}[M'_{3.3}] \not\Downarrow_b$ because the latter reduces to $\nu k.k!$. □

Example 3.4 (Derived programming constructs). Consider an extension of HOPass with standard operation of internal choice $(- \oplus -)$ with the non-deterministic reduction semantics: $P \oplus Q \to P$ and $P \oplus Q \to Q$. In this extended language the definition of $(\cong_{\text{cxt}})$ (Thm. 3.2) still applies. We can implement the internal choice operator correctly using local ports. Using the bisimulation technique developed in Sect. 6, one can show that

$$P \oplus Q \;\cong_{\text{cxt}}\; (p_{\text{L}}! \mid p_{\text{L}}?.P \mid p_{\text{L}}?.Q)\backslash p_{\text{L}}$$

An implementation using generated names

$$P \oplus_{\mathtt{new}} Q \overset{\text{def}}{=} \mathtt{new}\, x.(x! \mid x?.P \mid x?.Q)$$

would be incorrect because in general $P \oplus Q \not\cong_{\text{cxt}} P \oplus_{\mathtt{new}} Q$. To see this consider a particular instance when P, Q are $a!$, $b!$, respectively; we show that $a! \oplus_{\mathtt{new}} b! \not\cong_{\text{cxt}} a! \oplus b!$. The idea is to place the processes in a location $l[\![\;]\!]$ which can be passivated and duplicated. Consider $R_l = l[\![a! \oplus b!]\!]$. If we run R_l in parallel with process $K_1 = l?(x).\,(l'[\![\mathtt{run}\, x]\!] \mid l'[\![\mathtt{run}\, x]\!])$ the code in location l will be passivated and duplicated giving us $R_{l'} \mid R_{l'}$. Moreover, process $K_2 = l'?.a?.b?.c!$ blocks when run in parallel with $R_{l'} \mid R_{l'}$. By putting the above processes together we have $R_l \mid K_1 \mid K_2 \not\Downarrow_c$.

However, when $R'_l = l[\![a! \oplus_{\mathtt{new}} b!]\!]$ runs, a fresh name k is generated and we obtain $\nu k.l[\![k! \mid k?.a! \mid k?.b!]\!]$. Thus $R'_l \mid K_1 \mid K_2 \Downarrow_c$ because it can evolve to

$$\begin{array}{lllll}
 & \nu k.\ l'[\![k! \mid k?.a! \mid k?.b!]\!] & \mid\ l'[\![k! \mid k?.a! \mid k?.b!]\!] & \mid & l'?.a?.b?.c! \\
\to^* & \nu k.\ l'[\![\quad k?.a! \mid k?.b!]\!] & \mid\ l'[\![\quad a! \mid \quad b!]\!] & \mid & l'?.a?.b?.c! \\
\to & \nu k. & \ l'[\![\quad a! \mid \quad b!]\!] & \mid & \quad a?.b?.c! \to c!
\end{array}$$

Local ports can also be used to implement other standard programming constructs, such as various forms of recursion. Consider the operator $*(P)$ (omitted from our language) with reduction semantics $*(P) \to P \mid *(P)$. This operator can be encoded correctly using local ports and higher-order communication:

$$\mathsf{Rec}(P) \overset{\text{def}}{=} (\delta(p_{\text{L}}) \mid p_{\text{L}}!\lambda(P \mid \delta(p_{\text{L}})).\mathbb{0})\backslash p_{\text{L}} \qquad \delta(p_{\text{L}}) \overset{\text{def}}{=} p_{\text{L}}?(x{:}\mathsf{Pr}).(\mathtt{run}\, x \mid p_{\text{L}}!x.\mathbb{0})$$

Again, an encoding $\mathsf{Rec}_{\mathtt{new}}$ using generated names would not be correct. The process $l[\![\mathsf{Rec}_{\mathtt{new}}(a!)]\!] \mid K_1 \mid l'?.a?.c!$ *can* reduce to $\nu k.l[\![\delta(k)]\!] \mid a?.c! \not\Downarrow_c$, but $l[\![*(a!)]\!] \mid K_1 \mid l'?.a?.c!$ cannot reduce to a system that does not have a barb on c.

Using only generated names, as in HOπPn [10] discussed in the introduction, internal choice and general recursion are not encodable ($*(P)$ is a primitive). □

Example 3.5. In the last example of this section we show that passivation of observer-generated code is observable; we will return to this example when motivating our LTS for HOPass (Thm. 4.1). Let $M_{3.5} = a?(x).*(l[\![\mathtt{run}\, x]\!])$ and $M'_{3.5} = a?(x).*(\mathtt{run}\, x \mid l!)$. In HOPass these two systems are distinguished by contextual equivalence. This is achieved by the context testing if instances of the code bound to x are passivated after an output on l. For example consider the context $K_{3.5} = [\cdot] \mid a!(\lambda b?.c!).b!.l?$; we have: $K_{3.5}[M_{3.5}] \to^* l[\![c!]\!] \mid *(l[\![\mathtt{run}(\lambda b?.c!)]\!]) \mid l? = N_{3.5} \not\Downarrow_b \Downarrow_c$. This can be matched by $K_{3.5}[M'_{3.5}]$ by performing at least the reductions $K_{3.5}[P'_{3.5}] \to^* c! \mid l! \mid *(\mathtt{run}(\lambda b?.c!) \mid l!) \mid l? = N'_{3.5} \not\Downarrow_b \Downarrow_c$. However, $N_{3.5} \not\cong_{\text{cxt}} N'_{3.5}$ because the passivation of l, $N_{3.5} \to *(l[\![\mathtt{run}(\lambda b?.c!)]\!]) \not\Downarrow_c$, cannot be matched by $N'_{3.5}$. □

4 From HOπ to HOPass

The goal of this paper is to give a first-order, symbolic LTS for HOPass in which standard weak bisimilarity fully captures contextual equivalence; i.e., weak bisimulation is sound and complete with respect to ($\cong_{\mathrm{cxt}}$). Moreover, we seek an LTS with a *small set of transitions* to simplify bisimulation proofs. In this section we motivate this new LTS by starting from a previous first-order LTS for a version of HOπ [6]—essentially HOPass without locations—and examining the additional observational power needed to add to the LTS in order to achieve a sound bisimulation for HOPass.

As in our previous work, there are two basic ingredients to this LTS. The first is the extension of the syntax with *symbolic higher-order inputs* that the observer provides to the system. We thus extend the syntax of processes with *symbolic agents* $\alpha \in \mathsf{Agent}$ representing observer-generated processes.

$$\mathcal{P} ::= \ldots \mid \alpha \qquad \mathsf{EProc} \qquad\qquad \mathcal{V} ::= \ldots \mid \lambda\mathcal{P} \qquad \mathsf{EValue}$$

The second ingredient in the construction of the LTS is the explicit recording of the knowledge of the observer interrogating a system using a *knowledge environment* Δ. Any value sent from the system to the observer is recorded in Δ; if this value is a code thunk it has a unique associated *output index* $\kappa \in \mathsf{OIdx}$. These constants provide an indirect way for referring to outputs and only appear in Δ and on transitions of the LTS; they do not appear in processes and values. We extend the freshness operator $\sharp$ to α's and κ's. A knowledge environment Δ has the following components:

1. $\mathrm{names}(\Delta) \subset_{\mathsf{fin}} \mathsf{Name}$: a finite set of names known to the observer;
2. $\mathrm{agents}(\Delta) \subset_{\mathsf{fin}} \mathsf{Agent}$: a finite set of observer-generated symbolic agents;
3. $\mathrm{fun}(\Delta) \in \mathsf{OIdx} \rightarrow_{\mathsf{fin}} \mathsf{EValue}$: a finite function mapping output indices to system-generated code thunks.

Our LTS contains transitions over configurations of the form:

$$\mathcal{C} ::= \nu\widetilde{a}\langle \Delta \triangleright \mathcal{P}\rangle \qquad \mathsf{Conf}$$

The names in the vector $\widetilde{a}$ are generated by the system but are not known to the observer; they can appear in $\mathcal{P}$ and in the codomain of $\mathrm{fun}(\Delta)$. We consider only configurations that are *well-formed*. That is, configurations $\nu\widetilde{a}\langle \Delta \triangleright \mathcal{P}\rangle$ recording all names and constants used in $\mathcal{P}$ and in the codomain of $\mathrm{fun}(\Delta)$, whose names $\widetilde{a}$ are distinct pairwise and with respect to $\mathrm{names}(\Delta)$. We also identify configurations up to alpha-renaming of $\widetilde{a}$.

As with our LTS for HOπ, our LTS for HOPass will contain one higher-order input rule in which the input value is a fresh $\lambda\alpha$ and two higher-order output rules that that extend $\mathrm{fun}(\Delta)$ (one for output and one for passivation):

$$\begin{array}{llll}
\langle \Delta \triangleright c?(x{:}\mathsf{Pr}).\mathcal{P}\rangle \xrightarrow{c?\lambda\alpha} \langle \Delta, \alpha \triangleright \mathcal{P}\{\lambda\alpha/x\}\rangle & \text{if } \alpha \sharp \Delta & (\textsc{TIn-}\mathsf{Pr}) \\
\langle \Delta \triangleright c!\lambda\mathcal{P}.\mathcal{Q}\rangle \xrightarrow{c!\kappa} \langle \Delta, \kappa\mapsto\lambda\mathcal{P} \triangleright \mathcal{Q}\rangle & \text{if } \kappa \sharp \Delta & (\textsc{TOut-}\mathsf{Pr}) \\
\langle \Delta \triangleright c[\![\mathcal{P}]\!]\rangle \xrightarrow{c!\kappa} \langle \Delta, \kappa\mapsto\lambda\mathcal{P} \triangleright \mathbb{0}\rangle & \text{if } \kappa \sharp \Delta & (\textsc{Tpass})
\end{array}$$

The LTS also has fairly standard rules for internal steps, to propagate transitions over evaluation contexts, and for first-order input and output; we omit these transitions in this extended abstract (see [6]). Transitions in this LTS are labelled either with τ (internal transition), with one of the I/O actions over generated names $\mu ::= c?n \mid c?\lambda\alpha \mid c!n \mid c!\kappa$, or, purely for producing internal transitions, with the corresponding I/O actions over local ports.

Two important transitions in the LTS for HOπ [6] are those implementing the notion of triggers. When a HOπ system runs a symbolic input $\lambda\alpha$ (removing the λ) it enables the transition

$$\nu\widetilde{a}\langle\Delta \triangleright \alpha\rangle \xrightarrow{\mathtt{run}\,\alpha} \nu\widetilde{a}\langle\Delta \triangleright \mathbb{0}\rangle \tag{1}$$

indicating the execution of some code within the observer. Conversely, the observer can run at any point system code stored under the index κ:

$$\nu\widetilde{a}\langle\Delta \triangleright \mathcal{P}\rangle \xrightarrow{\mathtt{run}\,\kappa} \nu\widetilde{a}\langle\Delta \triangleright \mathcal{P} \mid \mathtt{run}\,\Delta(\kappa)\rangle \tag{2}$$

Similarly, in HOPass we need to give the observer the ability to run system-generated code and detect the execution of observer code. However, the above two transitions are not adequate to give us soundness of weak bisimulation. In the rest of this section we give example *in*equivalent processes that can be distinguished by ($\cong_{\text{cxt}}$) and motivate sufficient additions to the LTS of symbolic agents we have described so far. The following section contains the precise definitions of these additions and the relevant bisimulation.

We first show that (1) is no longer adequate in the presence of passivation. The observer should not just forget α after it has been run once; instead it requires the power to repeatedly *ping* α to ensure that the code implicitly represented by this symbolic agent is still alive and can communicate.

Example 4.1 (Example 3.5 revisited). Let us reconsider the systems $M_{3.5} = a?(x).{*}(l[\![\mathtt{run}\,x]\!])$. and $M'_{3.5} = a?(x).{*}(\mathtt{run}\,x \mid l!)$, which we have already seen are distinguished by ($\cong_{\text{cxt}}$) using the context $K_{3.5} = [\cdot] \mid a!(\lambda b?.c!).b!.l?$ testing whether an output $c!$ is possible after the sequence of reductions $a?(\lambda b?.c!), b?, l?$.

The LTS transitions we have seen so far cannot perform such a test; $M_{3.5}$ and $M'_{3.5}$ are not distinguishable in the current LTS. Let us see how we might try to mimic the distinguishing tests performed by $K_{3.5}$. This context first sends in on the channel a the actual code $\lambda b?.c!$ but our transitions are only allowed to send in a symbolic agent $\lambda\alpha$. Next, $K_{3.5}$ communicates with the sent code on b—in the LTS this can only be translated to a transition of the form (1) above. Then, it passivates l and tries to communicate on c with the sent code. The passivation of l can be performed in the LTS, but we cannot translate the communication on c because the previous use of (1) has replaced α with $\mathbb{0}$.

What is required is the ability to repeatedly check if the symbolic agent α is still alive; i.e., that the system has not introduced a communication barrier between α and the observer by passivating the former. In our LTS for HOPass the single use transition (1) will be replaced by a more general *ping* transition $\alpha \rightsquigarrow \alpha'$ which "updates" an α at evaluation position within the system to a fresh α'. The observer can keep performing this indefinitely. In effect, in our new LTS α's represent the *state* of symbolic agents inside configurations, which changes after an agent performs a transition. The transition $\alpha \rightsquigarrow \alpha'$ is a symbolic communication of an agent at state α with the observer, updating the state to a fresh α', which can be probed further (see Thm. 5.1). $\Box$

These *ping* transitions $\alpha \rightsquigarrow \alpha'$ discover *communication barriers* between symbolic agents and the external observer throughout the interrogation of a system. However, they are not sufficient to discover all of the intricacies of HOPass. Transparent locations together with passivation and reactivation can be used to create communication barriers *between* parts of the system, creating a form of *opaque locations*. We believe the ability to encode communication barriers captures the essence of location-dependent behaviour expressible in HOPass.

To crystallise this phenomenon we introduce a *trampoline* operator ($\bowtie$) which can be encoded in HOPass (as well as in HOπP [7]). Consider the process $P \bowtie Q$ with reduction rules:

$$P \bowtie Q \xrightarrow{\lambda} P' \bowtie Q \quad \text{if} \quad P \xrightarrow{\lambda} P' \qquad\qquad P \bowtie Q \xrightarrow{\lambda} P \bowtie Q' \quad \text{if} \quad Q \xrightarrow{\lambda} Q'$$

Essentially, $P \bowtie Q$ represents a communication barrier between the processes P and Q: they can communicate with their environment but not with each other. Provided $p_{\text{L}}, {p_{\text{L}}}' \sharp P, Q$, trampoline can be encoded in a fully-abstract manner:

$$P \bowtie Q \cong_{\text{cxt}} \Big(p_{\text{L}}[\![P]\!] \mid {p_{\text{L}}}'!(\lambda Q).\mathbb{0} \mid *\big(p_{\text{L}}?(x).{p_{\text{L}}}'?(y).(p_{\text{L}}[\![\texttt{run}\, y]\!] \mid {p_{\text{L}}}'!x.\mathbb{0})\big)\Big) \backslash p_{\text{L}}, {p_{\text{L}}}'$$

We prove an instance of this equivalence in Sect. 7 and use this operator extensively when motivating further symbolic transitions.

Example 4.2 (Communication barriers). Let us consider the systems $M_{4.2} = a?(x).(\texttt{run}\, x \bowtie b!)$ and $M'_{4.2} = a?(x).(\texttt{run}\, x \mid b!)$. These systems can be distinguished by $K_{4.2} = [\cdot] \mid a!(\lambda b?c!).\mathbb{0}$ because $K_{4.2}[M_{4.2}] \not\Downarrow_c$ whereas $K_{4.2}[M'_{4.2}] \Downarrow_c$. However, combinations of all LTS transitions we discussed so far are not able to distinguish them. We need to give agents the ability to communicate with the system, which in the next section we achieve by adding a transition $\mu/\alpha \rightsquigarrow \alpha'$ with which an agent transition $\alpha \rightsquigarrow \alpha'$ synchronises with a parallel action μ.

We also need a synchronisation transition between two running symbolic agents in order to observe the different behaviour of $N_{4.2} = a?(x).b?(y).(\texttt{run}\, x \bowtie \texttt{run}\, y)$ and $N'_{4.2} = a?(x).b?(y).(\texttt{run}\, x \mid \texttt{run}\, y)$. These are distinguished by $K'_{4.2} = [\cdot] \mid a!(\lambda c!).b!(\lambda c?.d!).\mathbb{0}$ since $K'_{4.2}[N_{4.2}] \not\Downarrow_d$ whereas $K'_{4.2}[N'_{4.2}] \Downarrow_d$. However, they are not distinguished by the transitions discussed so far. They will be distinguishable with a new synchronisation action $\alpha_1|\alpha_2 \rightsquigarrow \alpha'_1|\alpha'_2$, signalling that two agents α_1 and α_2 can communicate and become α'_1 and α'_2 (see Thm. 5.2). $\Box$

As we discussed, in HOπ the observer can use the transition $\texttt{run}\, \kappa$ (2) to run system code indexed by κ in the knowledge environment of a configuration; this code runs in parallel with the system after the transition. Because again of the communication barriers encodable in HOPass, code run in parallel with the system may exhibit different behaviour than if it were run at the position of an agent; in the presence of passivation we need a more general symbolic transition.

Moreover, an adequate LTS for HOPass needs to enable the passivation of the entire system and any code executed by the observer. The former is motivated by the fact that $(\cong_{\text{cxt}})$ is closed under the context $a?(x).P \mid a![\cdot].\mathbb{0}$; thus, any related systems $M \cong_{\text{cxt}} M'$, possibly obtained after a number of reductions and context closures of $(\cong_{\text{cxt}})$, can be entirely passivated and reused as code thunks in P. The need for the latter is motivated in the next example.

To allow the observer to passivate running code we will use a set of special location names only as a namespace for observer-generated locations. We call these *abstract locations* $\gamma \in \mathsf{Aloc}$, and extend the syntax of HOPass once more to include such locations; we adjust $(\sharp)$ to abstract locations and record γ's in knowledge environments: $\mathcal{P} ::= \ldots \mid \gamma\llbracket\mathcal{P}\rrbracket$. Intuitively, $\gamma\llbracket\mathcal{P}\rrbracket$ represents an agent that is currently running in a new location γ process $\mathcal{P}$, obtained from the observer's knowledge environment.

Remark 4.3. We use abstract locations instead of ordinary fresh names to *limit* the possible LTS transitions: these names need not be used as inputs or elsewhere in the system, considerably simplifying proofs of equivalence. □

Example 4.4 (Code execution and passivation). Consider the systems

$$M_{4.4} = \texttt{new}\, t.a!(\lambda t?.c!).b?(x).(\texttt{run}\, x \bowtie t!) \quad M'_{4.4} = \texttt{new}\, t.a!(\lambda t?.c!).b?(x).(\texttt{run}\, x \mid t!)$$

distinguished by the context $K_{4.4} = [\cdot] \mid a?(y).b!y.\mathbb{0}$ which simply relays a value from a to b. In $K_{4.4}[M_{4.4}]$ this leads to system $\nu t.((t?.c!) \bowtie t!) \not\Downarrow_c$; however, $K_{4.4}[M'_{4.4}]$ reduces to $\nu t.((t?.c!) \mid t!) \Downarrow_c$. Thus, this context distinguishes the two systems by running $(\lambda t?.c!)$ at the only position of $M_{4.4}$ where communication with the $t!$ is impossible (in the LHS of the $\bowtie$). Contexts that do not cause the execution of $(\lambda t?.c!)$ at that position cannot distinguish $M_{4.4}$ from $M'_{4.4}$. The transitions we have discussed so far encode observations made by such contexts and therefore fail to distinguish the two systems.

To see that, we consider the interrogation of a configuration where $\lambda M_{4.4}$ is in Δ, from which the observer, using the previously discussed transitions, can only reach configurations of the form $\nu\widetilde{t}\langle \Delta \triangleright \prod \alpha_i \bowtie t_i! \rangle$, with $\Delta(\kappa_i) = \lambda t_i?.c!$. The only way for the observer to run a κ_i *without* enabling a communication on t_i (and thus an observable $c!$ transition) is to run κ_i at α_i (as the context $K_{4.4}$ above did). The preceding LTS transitions do not capture such a move. Therefore we introduce a separate transition $\alpha \rightsquigarrow \gamma\llbracket\kappa\rrbracket$ which replaces a symbolic agent α with $\gamma\llbracket\Delta(\kappa)\rrbracket$, for a fresh γ.

Now consider the systems:

$$N_{4.4} = \texttt{new}\, t.a!(\lambda t?c!).b!(\lambda t!).\mathbb{0} \qquad N'_{4.4} = \texttt{new}\, t.a!(\lambda t!).b!(\lambda t?c!).\mathbb{0}$$

To distinguish them, the observer needs to input both code thunks on a and b, run both, and passivate one of them after they communicate on t. The last move is not possible with the transitions we have seen so far. A context that performs this scenario and distinguishes $N_{4.4}$ from $N'_{4.4}$ is $K'_{4.4} = [\cdot] \mid a?(x).b?(y).(l\llbracket\texttt{run}\, x\rrbracket \mid \texttt{run}\, y \mid l?.c?.d!)$. We have $K'_{4.4}[N_{4.4}] \not\Downarrow_d$ but $K'_{4.4}[N'_{4.4}] \Downarrow_d$.

In our new LTS, transitions $a!\kappa_1, b!\kappa_2, \alpha_1 \rightsquigarrow \gamma_1\llbracket\kappa_1\rrbracket, \alpha_2 \rightsquigarrow \gamma_2\llbracket\kappa_2\rrbracket$ let the observer receive and run the code emitted from the systems (provided there are

Trun-κ

$$\frac{\Delta(\kappa) = \lambda\mathcal{P} \qquad \gamma \sharp \Delta}{\langle \Delta \rhd \alpha \rangle \xrightarrow{\alpha \rightsquigarrow \gamma\llbracket\kappa\rrbracket} \langle \Delta, \gamma \rhd \gamma\llbracket\mathcal{P}\rrbracket \rangle}$$

Tpass-γ

$$\frac{\kappa, \alpha \sharp \Delta}{\langle \Delta \rhd \gamma\llbracket\mathcal{P}\rrbracket \rangle \xrightarrow{\gamma\llbracket\kappa\rrbracket \rightsquigarrow \alpha} \langle \Delta, \alpha, \kappa{\mapsto}\lambda\mathcal{P} \rhd \alpha \rangle}$$

TioL@α

$$\frac{\langle \Delta \rhd \mathcal{P} \rangle \xrightarrow{\alpha_1 \rightsquigarrow \alpha_2} \langle \Delta' \rhd \mathcal{P}' \rangle \qquad \langle \Delta' \rhd \mathcal{Q} \rangle \xrightarrow{\mu} \langle \Delta'' \rhd \mathcal{Q}' \rangle}{\langle \Delta \rhd \mathcal{P} \mid \mathcal{Q} \rangle \xrightarrow{\mu/\alpha_1 \rightsquigarrow \alpha_2} \langle \Delta'' \rhd \mathcal{P}' \mid \mathcal{Q}' \rangle}$$

Tsync

$$\frac{\langle \Delta \rhd \mathcal{P} \rangle \xrightarrow{\alpha_1 \rightsquigarrow \alpha_2} \langle \Delta' \rhd \mathcal{P}' \rangle \qquad \langle \Delta' \rhd \mathcal{Q} \rangle \xrightarrow{\alpha_3 \rightsquigarrow \alpha_4} \langle \Delta'' \rhd \mathcal{Q}' \rangle}{\langle \Delta \rhd \mathcal{P} \mid \mathcal{Q} \rangle \xrightarrow{\alpha_1|\alpha_3 \rightsquigarrow \alpha_2|\alpha_4} \langle \Delta'' \rhd \mathcal{P}' \mid \mathcal{Q}' \rangle}$$

Tsig

$$\frac{\alpha_2 \sharp \Delta}{\langle \Delta \rhd \alpha_1 \rangle \xrightarrow{\alpha_1 \rightsquigarrow \alpha_2} \langle \Delta, \alpha_2 \rhd \alpha_2 \rangle}$$

Textr@α

$$\frac{\nu\widetilde{a}\langle \Delta, n \rhd \mathcal{P} \rangle \xrightarrow{c!n/\alpha_1 \rightsquigarrow \alpha_2} \nu\widetilde{b}\langle \Delta' \rhd \mathcal{P}' \rangle \qquad c \neq n}{\nu n, \widetilde{a}\langle \Delta \rhd \mathcal{P} \rangle \xrightarrow{c!n/\alpha_1 \rightsquigarrow \alpha_2} \nu\widetilde{b}\langle \Delta' \rhd \mathcal{P}' \rangle}$$

Fig. 3. LTS: symbolic agent transitions (omitting symmetric rules)

running symbolic agents α_1 and α_2). However, to enable further passivation of this code, we introduce transitions of the form $\gamma_1\llbracket\kappa_{\mathrm{fr}}\rrbracket \rightsquigarrow \alpha_{\mathrm{fr}}$ which let the observer passivate the code running in γ_1, replacing it with a fresh symbolic agent α_{fr}, and indexing it by a fresh κ_{fr} in the knowledge environment. With the addition of this last LTS transition the observer can distinguish the above systems in the same way as $K'_{4.4}$ does (see Thm. 5.3 for details). □

5 First-Order Symbolic Agent Transitions

The previous section briefly described internal (τ) and communication (μ) transitions and focused on motivating a set of new symbolic (ζ) transitions for an adequate LTS for HOPass. Here we give the precise rules of the new ζ-transitions, all of which describe a *limited symbolic execution* of agents running in a configuration. An agent is an observer-generated process represented simply by $\alpha \in \mathsf{PConst}$, or an abstract location $\gamma\llbracket\mathcal{P}\rrbracket$ running a single system-generated code thunk ($\gamma \in \mathsf{LConst}$). The result is an LTS with *first-order* transitions ($\eta ::= \tau \mid \mu \mid \zeta$) simplifying bisimulation proofs.

Observer transitions are generated by the rules shown in Fig. 3 and are annotated with one of the following labels:

$$\zeta \;::=\; \alpha \rightsquigarrow \alpha \;\bigm|\; \mu/\alpha \rightsquigarrow \alpha \;\bigm|\; \alpha|\alpha \rightsquigarrow \alpha|\alpha \;\bigm|\; \alpha \rightsquigarrow \gamma\llbracket\kappa\rrbracket \;\bigm|\; \gamma\llbracket\kappa\rrbracket \rightsquigarrow \alpha$$

These transitions encode symbolic moves performed by agents, visible to the overall observer, in order to *reconfigure* agents and interrogate the system. They fall naturally into two groups, the first concerned with *communication barriers* and the second with code *execution* and *passivation*. The first three involve communication barriers.

Ping: $\alpha_1 \rightsquigarrow \alpha_2$**.** This transition, produced by rule Tsig, allows the observer to determine if an agent α_1 is running. There is a communication barrier between the observer and α_1 only if α_1 is not running in the configuration. As a result of this transition, a running instance of α_1 is replaced by a fresh α_2, distinguishing it from other instances of α_1 in the configuration. In this way this transition can be used to distinguish the processes in Thm. 4.1.

Example 5.1 (Thm. 4.1 continued). Let us see how $M_{3.5}$ and $M'_{3.5}$ can be distinguished. Consider an observer examining configurations such as $\langle \Delta \triangleright M_{3.5} \rangle$ and $\langle \Delta \triangleright M'_{3.5} \rangle$, where $\Delta = \{a, l\}$. After a transition $a?\lambda\alpha_1$ we get the configurations $\langle \Delta, \alpha_1 \triangleright *(l[\![\mathtt{run}\,\lambda\,\alpha_1]\!]) \rangle$ and $\langle \Delta, \alpha_1 \triangleright *(\mathtt{run}\,\lambda\,\alpha_1 \mid l!) \rangle$. After τ-transitions and transitions $\alpha_1 \rightsquigarrow \alpha_2$ and $l!\kappa$ we get $\langle \Delta, \widetilde{\alpha}, \kappa{\mapsto}\lambda\alpha_2 \triangleright *(l[\![\mathtt{run}\,\lambda\,\alpha_1]\!]) \rangle$ and $\langle \Delta, \widetilde{\alpha}, \kappa{\mapsto}\lambda\mathbb{0} \triangleright \alpha_2 \mid *(\mathtt{run}\,\lambda\,\alpha_1 \mid l!) \rangle$. The latter configuration has an $\alpha_2 \rightsquigarrow \alpha_3$ transition but the former does not. □

I/O from a Symbolic Agent: $\mu/\alpha_1 \rightsquigarrow \alpha_2$. Because of the communication barriers encodable in HOPass, an agent may or may not be running at the same time as an observable μ action in a configuration. This transition, due to rule TioL@α and its symmetric one, allows the observer to detect this situation and distinguish systems $M_{4.2}$ and $M'_{4.2}$ in Thm. 4.2. As with standard name output, a name output detected by an agent can extrude a private name, moving it from the list of bound names into the knowledge environment (rule Textr@α). Note the chaining of the knowledge environments in the two premises of the rule that accumulates the effects of the two transitions in the final Δ''.

Agent Synchronisation: $\alpha_1|\alpha_3 \rightsquigarrow \alpha_2|\alpha_4$. For the same reason as above, this transition allows the observer to detect whether two symbolic agents are simultaneously running and can thus communicate. As before, the effects of the two transitions in the premises (i.e., the extension of Δ with fresh α_2 and α_4) are accumulated in the final Δ'' by chaining. Such a transition, generated by Tsync, can be used to distinguish the systems $N_{4.2}$ and $N'_{4.2}$ in Thm. 4.2.

Example 5.2 (Thm. 4.2 continued). An observer can distinguish configurations such as $\langle \Delta \triangleright M_{4.2} \rangle$ and $\langle \Delta \triangleright M'_{4.2} \rangle$ because the latter can perform the transition sequence $a?\alpha_1, b!/\alpha_1 \rightsquigarrow \alpha_2$ but the former cannot. Similarly, the observer can distinguish $\langle \Delta \triangleright N_{4.2} \rangle$ from $\langle \Delta \triangleright N'_{4.2} \rangle$ because the latter can perform the transition sequence $a?\alpha_1, b?\alpha_3, \alpha_1|\alpha_3 \rightsquigarrow \alpha_2|\alpha_4$ but the former cannot. □

We now detail the symbolic transitions concerned with code execution and passivation. These only use fresh abstract locations γ, not generated names, simplifying the LTS and the construction of witness bisimulations. We also ensure that at each abstract location γ only one system-generated process is executing at any time, further simplifying the LTS.

Code Execution: $\alpha \rightsquigarrow \gamma[\![\kappa]\!]$. With this transition (due to Trun-κ) the observer sends a system-generated code thunk, indexed in the knowledge environment by κ, to the location of a running agent α to be executed. The agent originates from a higher-order input transition (rule TIn-Pr in Sect. 4) and the thunk from a higher-order system output (rule Tout-Pr), before or after the input. After the transition, α is replaced by a fresh abstract location γ in which the code in $\Delta(\kappa)$ runs (and only that).

Abstract Location Passivation: $\gamma\llbracket\kappa\rrbracket \rightsquigarrow \alpha$. This transition (due to Tpass-γ) allows the observer to passivate an abstract location γ, which has previously been introduced by the symbolic transition just described, $\alpha \rightsquigarrow \gamma\llbracket\kappa\rrbracket$.

Example 5.3 (Thm. 4.4 revisited). We have seen that systems $N_{4.4}$ and $N'_{4.4}$ are not contextually equivalent. Here we show how an observer can use the above two symbolic transitions to distinguish them, when examining the configurations $\langle \Delta_1 \triangleright \alpha_1 \mid \alpha_2 \rangle$ and $\langle \Delta'_1 \triangleright \alpha_1 \mid \alpha_2 \rangle$, where $\Delta_1 = \Delta, \kappa{\mapsto}\lambda N_{4.4}$, $\Delta'_1 = \Delta, \kappa{\mapsto}\lambda N'_{4.4}$, and $\Delta = \{a, b, c, \widetilde{\alpha}\}$. After transition $\alpha_1 \rightsquigarrow \gamma_1\llbracket\kappa\rrbracket$ we get

$$\nu t\langle \Delta_1, \gamma_1 \triangleright \gamma_1\llbracket N_{4.4}\rrbracket \mid \alpha_2 \rangle \qquad \nu t\langle \Delta'_1, \gamma_1 \triangleright \gamma_1\llbracket N'_{4.4}\rrbracket \mid \alpha_2 \rangle$$

and after a sequence of weak transitions $a!\kappa_1, b!\kappa_2, \gamma_3\llbracket\kappa_3\rrbracket \rightsquigarrow \alpha_3$:

$$\nu t\langle \Delta_{\kappa_1,\kappa_2} \triangleright \alpha_3 \mid \alpha_2 \rangle \qquad \nu t\langle \Delta_{\kappa_2,\kappa_1} \triangleright \alpha_3 \mid \alpha_2 \rangle$$

where $\Delta_{x,y} = \Delta_1, \widetilde{\alpha}, \widetilde{\gamma}, (x{\mapsto}\lambda t?c!), (y{\mapsto}\lambda t!), (\kappa_3{\mapsto}\lambda\mathbb{0})$. The observer can now run both κ_1 and κ_2 (with the transitions $\alpha_3 \rightsquigarrow \gamma_3\llbracket\kappa_1\rrbracket$ and $\alpha_2 \rightsquigarrow \gamma_2\llbracket\kappa_2\rrbracket$) and obtain:

$$\nu t\langle \Delta_{\kappa_1,\kappa_2}, \widetilde{\gamma} \triangleright \gamma_3\llbracket t?c!\rrbracket \mid \gamma_2\llbracket t!\rrbracket \rangle \qquad \nu t\langle \Delta_{\kappa_2,\kappa_1}, \widetilde{l} \triangleright \gamma_3\llbracket t!\rrbracket \mid \gamma_2\llbracket t?c!\rrbracket \rangle$$

Only the left configuration can now perform a weak sequence of transitions $(\gamma_2\llbracket\kappa_3\rrbracket \rightsquigarrow \alpha_4), c!$. Hence the original systems are differentiated by the observer. Note that the passivation of γ_1 in this example re-introduced a new symbolic agent α_3 in its place, allowing the observer to continue the interrogation of the configuration. This is why Tpass-γ introduces a new constant in our LTS. □

6 Weak Bisimulation Theory

We employ the standard bisimulation theory, applied to the LTS of configurations generated by our first-order agent semantics outlined in the previous sections. This is then restricted to a *subset* of configurations. Weak bisimilarity over this subset is sound and complete with respect to contextual equivalence (Thm. 3.2). We use the standard notation $(\overset{\eta}{\Rightarrow})$ to mean the reflexive transitive closure of $(\xrightarrow{\tau})$, when $\eta = \tau$, and $(\overset{\tau}{\Rightarrow}\xrightarrow{\eta}\overset{\tau}{\Rightarrow})$ otherwise.

Definition 6.1 (Weak Bisimulation). $\mathbb{R}$: $\mathsf{Conf} \times \mathsf{Conf}$ *is a weak bisimulation when for all $\mathcal{C}_1 \mathbin{\mathbb{R}} \mathcal{C}'_1$ the following condition and its converse are satisfied: If $\mathcal{C}_1 \xrightarrow{\eta} \mathcal{C}_2$ then there exists $\mathcal{C}'_2$ such that $\mathcal{C}'_1 \overset{\eta}{\Rightarrow} \mathcal{C}'_2$ and $\mathcal{C}'_1 \mathbin{\mathbb{R}} \mathcal{C}'_2$.*

The largest weak bisimulation, *weak bisimilarity* $(\approx)$, is the union of all weak bisimulations; it is straightforward to show that this is an equivalence relation.

We have deliberately restricted the number and form of symbolic transitions, so as to facilitate the description of witness bisimulations when proving systems equivalent. For example there is no direct way in which the observer can execute at top-level code received from the system, indexed by a κ; this was even necessary in the simpler language of HOπ [6]. In the current framework, the observer interrogating a system, needs to have already executing within the system symbolic agents, represented either by occurrences of α's or γ's. Because of this, we let the observer interrogate a system $\nu\widetilde{a}.P$ by transitions in the agent LTS by starting in the configuration $\nu\widetilde{a}\big\langle \varDelta, \kappa{\mapsto}\lambda P \triangleright \prod \alpha_i \big\rangle$. Here each symbolic agent α_i allows the observer to initiate the interrogation, by executing one of the symbolic actions from Fig. 3. In fact only *two* such symbolic agents are necessary.

Theorem 6.2 (Soundness and Completeness of $(\approx)$). *Let $M = \nu\widetilde{m}.P$, $N = \nu\widetilde{n}.Q$ be closed systems. Then for $\varDelta = \{\widetilde{c}, \widetilde{\alpha}\} \supseteq \mathrm{fn}(M), \mathrm{fn}(N), \alpha_1, \alpha_2$: $M \cong_{\mathrm{cxt}} N$ iff $\nu\widetilde{m}\big\langle \varDelta, \kappa{\mapsto}\lambda P \triangleright \alpha_1 \mid \alpha_2 \big\rangle \approx \nu\widetilde{n}\big\langle \varDelta, \kappa{\mapsto}\lambda Q \triangleright \alpha_1 \mid \alpha_2 \big\rangle$.*

7 Example Equivalence

The encoding in HOPass of internal choice $(\oplus)$, replication $(*(\,))$, and the trampoline operator $(\bowtie)$ are fully abstract. Here we prove an instance of the last: in HOPass extended with $(\bowtie)$ and replication, $M = a?(x,y).(\mathtt{run}\,x \bowtie \mathtt{run}\,y) \cong_{\mathrm{cxt}} a?(x,y).(\mathtt{run}\,x \bowtie_{\mathsf{enc}} \mathtt{run}\,y) = M'$, where $(\bowtie_{\mathsf{enc}})$ is the encoding on page 175. Soundness of $(\approx)$ holds for the extended language. Thus, from Thm. 6.2, it suffices to show $\big\langle \varDelta, \kappa{\mapsto}\lambda M \triangleright \alpha_1 \mid \alpha_2 \big\rangle \approx \big\langle \varDelta, \kappa{\mapsto}\lambda M' \triangleright \alpha_1 \mid \alpha_2 \big\rangle$. To reduce the size of the proof we make use of a standard *up-to beta steps* technique, similar to that in our previous work for HOπ [6], and observe that internal $\mathtt{run}$ transitions and all transitions involving communication on p_{L} and $p_{\mathrm{L}}{}'$ in $(\bowtie_{\mathsf{enc}})$ are beta steps. We construct the following relation on well-formed configurations and prove it is a weak bisimulation up to beta steps by induction on the construction and enumeration of the possible LTS transitions.

$$\begin{gathered}\big\langle \varDelta, \widetilde{\kappa}_1{\mapsto}\lambda M, \kappa_{21}{\mapsto}\lambda\alpha_{11} \bowtie \alpha_{12}, \ldots \kappa_{2m}{\mapsto}\lambda\alpha_{1m} \bowtie \alpha_{2m} \triangleright \alpha_3 \mid \alpha_4 \big\rangle \\ \qquad \mathbb{R}\ \big\langle \varDelta, \widetilde{\kappa}_1{\mapsto}\lambda M', \kappa_{21}{\mapsto}\lambda\alpha_{11} \bowtie_{\mathsf{enc}} \alpha_{12}, \ldots \kappa_{2m}{\mapsto}\lambda\alpha_{1m} \bowtie_{\mathsf{enc}} \alpha_{2m} \triangleright \alpha_3 \mid \alpha_4 \big\rangle \\ (\mathcal{C}, \gamma)\{\{\gamma[\![\mathtt{run}\,\mathcal{C}(\kappa)/\alpha]\!]\}\}\ \mathbb{R}\ (\mathcal{C}', \gamma)\{\{\gamma[\![\mathtt{run}\,\mathcal{C}'(\kappa)/\alpha]\!]\}\} \text{ if } \mathcal{C}\ \mathbb{R}\ \mathcal{C}' \\ (\mathcal{C}, \widetilde{\alpha})\{\{\alpha_1 \bowtie \alpha_2/M\}\}\ \mathbb{R}\ (\mathcal{C}', \widetilde{\alpha})\{\{\alpha_1 \bowtie_{\mathsf{enc}} \alpha_2/M\}\} \text{ if } \mathcal{C}\ \mathbb{R}\ \mathcal{C}'\end{gathered}$$

Here $\{\{\mathcal{P}/\mathcal{Q}\}\}$ replaces *one* occurrence of $\mathcal{Q}$ with $\mathcal{P}$ in a configuration, and $(\mathcal{C}, \varDelta)$ extends the knowledge environment of $\mathcal{C}$ with a fresh $\varDelta$; $\mathcal{C}(\kappa)$ denotes the code indexed by κ in the environment of $\mathcal{C}$. In the base case of the construction, related knowledge environments contain an arbitrary number of indices to the initial systems ($\kappa_{1i}{\mapsto}\lambda M$ and $\kappa_{1i}{\mapsto}\lambda M'$), as well as an arbitrary number of $\kappa_{2i}{\mapsto}\lambda\alpha_{1i} \bowtie \alpha_{2i}$ and $\kappa_{2i}{\mapsto}\lambda\alpha_{1i} \bowtie_{\mathsf{enc}} \alpha_{2i}$, where the α_{1i}'s are not necessarily pairwise distinct (similarly for the α_{2i}'s).

8 Conclusions

We presented the first first-order bisimulation proof technique for a distributed language with passivation and private names, which is sound and complete with respect to weak barbed congruence, the contextual equivalence associated with weak bisimulation. In our language, code behaviour is location-dependent, the usual encodings of useful systems are possible, and unnecessary complexities with free names are avoided. We believe that our technique can be adapted to other distributed languages with location-dependent behaviour, and indeed to HOπP [7], a passivation language where free names are observable for which the only available proof technique is context bisimulation. Normal and environmental bisimulation are sound only for sublanguages of HOπP [7, 9], and the latter technique is sound and complete for a language with generative names [10]. These language variations, however, cannot express many useful systems, such as those with internal choice, replication, or higher-order values containing input processes. In other languages with passivation, context bisimulation is only sound for the weak case [2, 3] or sound and complete for only the strong case [14]. Unlike context and environmental bisimulation, our proof technique avoids any universal quantification over contexts. This is achieved by a labelled transition system in which higher-order input values are replaced by abstract agents which can perform limited symbolic transitions within systems, simplifying proofs.

References

1. Amadio, R.M., Dam, M.: Reasoning about higher-order processes. In: Mosses, P.D., Nielsen, M. (eds.) CAAP 1995, FASE 1995, and TAPSOFT 1995. LNCS, vol. 915, pp. 202–216. Springer, Heidelberg (1995)
2. Castagna, G., Vitek, J., Zappa Nardelli, F.: The seal calculus. Information and Computation 201(1), 1–54 (2005)
3. Godskesen, J.C., Hildebrandt, T.: Extending howes method to early bisimulations for typed mobile embedded resources with local names. In: Sarukkai, S., Sen, S. (eds.) FSTTCS 2005. LNCS, vol. 3821, pp. 140–151. Springer, Heidelberg (2005)
4. Jeffrey, A., Rathke, J.: Contextual equivalence for higher-order pi-calculus revisited. LMCS 1(1:4) (2005)
5. Koutavas, V., Hennessy, M.: A testing theory for a higher-order cryptographic language. In: Barthe, G. (ed.) ESOP 2011. LNCS, vol. 6602, pp. 358–377. Springer, Heidelberg (2011), `http://dx.doi.org/10.1007/978-3-642-19718-5_19`
6. Koutavas, V., Hennessy, M.: First-order reasoning for higher-order concurrency. Computer Languages, Systems & Structures 38(3), 242–277 (2012)
7. Lenglet, S.: Schmitt A, and Stefani J.-B. Characterizing contextual equivalence in calculi with passivation. Information and Computation 209(11), 1390–1433 (2011)
8. Milner, R., Sangiorgi, D.: Barbed bisimulation. In: Kuich, W. (ed.) ICALP 1992. LNCS, vol. 623, pp. 685–695. Springer, Heidelberg (1992)
9. Piérard, A., Sumii, E.: Sound bisimulations for higher-order distributed process calculus. In: Hofmann, M. (ed.) FOSSACS 2011. LNCS, vol. 6604, pp. 123–137. Springer, Heidelberg (2011)
10. Pierard, A., Sumii, E.: A higher-order distributed calculus with name creation. In: LICS, pp. 531–540. IEEE Computer Society (June 2012)
11. Sangiorgi, D.: Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms. PhD thesis, University of Edinburgh (1992)
12. Sangiorgi, D.: From pi-calculus to higher-order pi-calculus–and back. In: Gaudel, M.-C., Jouannaud, J.-P. (eds.) CAAP 1993, FASE 1993, and TAPSOFT 1993. LNCS, vol. 668, pp. 151–166. Springer, Heidelberg (1993)
13. Sangiorgi, D.: On the bisimulation proof method. MSCS 8(5), 447–479 (1998)
14. Schmitt, A., Stefani, J.-B.: The kell calculus: A family of higher-order distributed process calculi. In: Priami, C., Quaglia, P. (eds.) GC 2004. LNCS, vol. 3267, pp. 146–178. Springer, Heidelberg (2005)
15. Vivas, J.-L., Dam, M.: From higher-order π-calculus to π-calculus in the presence of static operators. In: Sangiorgi, D., de Simone, R. (eds.) CONCUR 1998. LNCS, vol. 1466, pp. 115–130. Springer, Heidelberg (1998)