Lang-n-Send: Processes That Send Languages

Matteo Cimini
University of Massachusetts Lowell
Lowell, MA, USA
matteo_cimini@uml.edu

We present LANG-N-SEND, a π-calculus that is equipped with language definitions. Processes can define languages in operational semantics, and use them to execute programs. Furthermore, processes can send and receive pieces of operational semantics through channels.

We present a reduction semantics for LANG-N-SEND, and we offer examples that demonstrate some of the scenarios that LANG-N-SEND captures.

1 Introduction

In the last decades, we have seen significant advances in language semantics tools that make it possible for programmers to quickly define and deploy their own programming languages and domain-specific languages, and use them in their programming solutions [6].

It is not too far in the future that it would just be the common practice for programmers to upload, in servers like Amazon AWS, code that does not belong to a programming language that has been fixed beforehand, but rather belongs to a language that has been created on the fly. Programmers would upload both the program and the language in which the program must be evaluated. Reuse is fundamental in this scenario. Libraries of programming languages constructs, as envisioned by Peter Mosses' Component-Based Semantics [14], for example, can become the norm. Servers can provide pieces of languages to clients, which can use them to complete their own language and, in turn, send the language so built to (computing) servers to execute programs.

Current literature does not offer a foundation that directly formalizes this and similar scenarios. In this paper, we present our work towards such a formal foundation.

We present LANG-N-SEND, a π-calculus that is equipped with language definitions. Processes can define languages in operational semantics, add pieces of operational semantics together, and use them to execute programs. Processes can also send and receive pieces of languages through channels. After executing programs, LANG-N-SEND processes can also send the trace of executions to other processes, which in turn can analyze these traces.

We present a reduction semantics for LANG-N-SEND, and we provide some selected examples that demonstrate the scenarios that LANG-N-SEND captures. We have specifically chosen examples that involve the communication of languages among processes. We show the following examples:

- A client that, when entering a sensitive region of code, asks a server to provide the semantics of an interrupt operator, adds it to its language, and only then executes the code.
- A client that defines a language with an interrupt operator, but lets a server decide the semantics of the interruption (whether interrupt or disrupt semantics) by receiving, from the server, the rest of the rules that complete the semantics of the operator.
- A client that lets a server decide whether its language is synchronous or asynchronous by receiving the semantics of the output operator from the server.

M. Carbone and R. Neykova (Eds.): 13th International Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software
EPTCS 356, 2022, pp. 46–56, doi:10.4204/EPTCS.356.5

We believe that LANG-N-SEND represents a first step towards a firm foundation for this type of programming. The next section presents the syntax that LANG-N-SEND uses to define languages. Section 3 presents the syntax of LANG-N-SEND processes. Section 4 presents a reduction semantics. Section 5 provides examples. Section 6 discusses related work, and Section 7 concludes the paper.

2 Syntax for Language Definitions

The syntax of LANG-N-SEND consists of two parts: the syntax for creating languages, and a π-calculus with language definitions. Language definitions can be created in operational semantics. The syntax that we adopt is inspired by [5], and is the following, where *cname* $\in$ CATNAME, $X \in$ META-VAR, *opname* $\in$ OPNAME, and *pn* $\in$ PREDNAME.

$$
\begin{array}{lllll}
\text{Language} & \mathscr{L} & ::= & (G, I) \\
\text{Grammar} & G & ::= & s_1 \cdots s_n \\
\text{Grammar Rule} & s & ::= & \mathit{cname}\, X ::= t_1 \mid \cdots \mid t_n \\
\text{Inference System} & I & ::= & r_1 \cdots r_n \\
\text{Rule} & r & ::= & \dfrac{f_1 \cdots f_n}{f} \\
\text{Formula} & f & ::= & (\mathit{pn}\, t_1 \cdots t_n) \\
\text{Term} & t & ::= & X \mid (\mathit{opname}\, t_1 \cdots t_n)
\end{array}
$$

CATNAME is a set of grammar category names such as *Process*, and *Action*. META-VAR is a set of meta-variables. OPNAME is a set of constructor names such as *par* (for the parallel operator $|$), and *choice* (for the choice operator $+$). PREDNAME is a set of predicate names such as *step* (for reduction rules). As names do not need to be strings, we shall use symbols for constructor and predicate names.

A language has a grammar and an inference rule system. A grammar has multiple grammar rules, each of which defines a category name, and its meta-variable, by providing a series of grammar productions, which are terms. Terms are in abstract syntax tree style, that is, they have a top level constructor applied to a list of terms. We can demonstrate LANG-N-SEND with languages that do not use binders. Therefore, we do not include syntax for binding, though it could be added.

An inference rule system has multiple rules, each of which has a series of formulae as premises, and a formula as conclusion. Formulae, too, are in abstract syntax tree style. Given a language definition, LANG-N-SEND needs to invoke its evaluator to execute programs. As we need a way to locate such evaluator, we fix the following convention: The labeled transition relation is always $\longrightarrow$, and its first argument is always the label of the transition, which is a term. (If reductions do not have labels, they would still use the first argument with a term that is never used).

To make an example, let us consider Basic Process Algebra (BPA [3]) in its finite fragment (no recursion, nor definitions). BPA is formed with actions, the choice operator, and sequential composition. Below are the rules of BPA, where a ranges over actions. Besides transitions of the form $P \xrightarrow{a} P'$, BPA makes use of a predicate $P \longrightarrow^{a} \checkmark$ that says that P takes action a and successfully terminates.

$$
a \longrightarrow^{a} \checkmark \qquad \frac{P_1 \longrightarrow^{a} \checkmark}{P_1 + P_2 \longrightarrow^{a} \checkmark} \qquad \frac{P_2 \longrightarrow^{a} \checkmark}{P_1 + P_2 \longrightarrow^{a} \checkmark}
$$

$$
\frac{P_1 \xrightarrow{a} P_1'}{P_1 + P_2 \xrightarrow{a} P_1'} \qquad \frac{P_2 \xrightarrow{a} P_2'}{P_1 + P_2 \xrightarrow{a} P_2'} \qquad \frac{P_1 \xrightarrow{a} P_1'}{P_1 \cdot P_2 \xrightarrow{a} P_1' \cdot P_2} \qquad \frac{P_1 \longrightarrow^{a} \checkmark}{P_1 \cdot P_2 \xrightarrow{a} P_2}
$$

LANG-N-SEND accommodates BPA as follows. The transition $P \xrightarrow{a} P'$ is encoded as $(\longrightarrow\ a\ P\ P')$. We represent the formula $P \longrightarrow^{a} \checkmark$ with $(checkMark\ (a)\ P)$. Below, *act* is the operator for actions, and *seq* is the sequential operator. We give this language the name *bpa*.

$$bpa \triangleq (Action\ A ::= (a) \mid (b) \mid (c) \qquad (*\ We\ assume\ that\ the\ set\ of\ actions\ is\ \{a,b,c\}\ *)$$
$$Process\ P ::= (act\ A) \mid (+\ P\ P) \mid (seq\ P\ P),$$

$$(checkMark\ A\ (act\ A)) \quad \frac{(checkMark\ A\ P_1)}{(checkMark\ A\ (+\ P_1\ P_2))} \quad \frac{(checkMark\ A\ P_2)}{(checkMark\ A\ (+\ P_1\ P_2))}$$
$$\frac{(\longrightarrow\ A\ P_1\ P_1')}{(\longrightarrow\ A\ (+\ P_1\ P_2)\ P_1')} \quad \frac{(\longrightarrow\ A\ P_2\ P_2')}{(\longrightarrow\ A\ (+\ P_1\ P_2)\ P_2')}$$
$$\frac{(\longrightarrow\ A\ P_1\ P_1')}{(\longrightarrow\ A\ (seq\ P_1\ P_2)\ (seq\ P_1'\ P_2))} \quad \frac{(checkMark\ A\ P_1)}{(\longrightarrow\ A\ (seq\ P_1\ P_2)\ P_2)}\ )$$

3 Syntax for Processes

The syntax of LANG-N-SEND processes P, Q, R, and so on, is defined as follows.

Trace	$\mathfrak{T}$	::=	$tr \mid t \cdots t$
Language Builder	ℓ	::=	$l \mid \mathscr{L} \mid \ell\ \texttt{union}\ \ell$
Processes	P	::=	$\mathbf{0} \mid x(y).P \mid \overline{x}\langle y\rangle.P \mid P \mid P \mid P+P$
			$\mid (\nu x).P \mid\ !P$
(using languages)			$\mid (\ell, \mathfrak{T})>_x t$
(analyzing executions)			$\mid \texttt{isInTrace}(t, \mathfrak{T}) \Rightarrow P\ ;\ P$
(communicating languages)			$\mid x(l).P \mid \overline{x}\langle \ell\rangle.P$
(communicating traces)			$\mid x(tr).P \mid \overline{x}\langle \mathfrak{T}\rangle.P$

LANG-N-SEND contains the elements of the π-calculus [12, 13]. Channels are x, y, z, and so on. We assume a set LANG-VAR of variables l for languages, and a set TRACE-VAR of variables tr for traces. These sets are distinct from each other, and from the set of channels.

Language builder expressions ℓ build a language $\mathscr{L}$, i.e., they ultimately evaluate to a language $\mathscr{L}$. This category can be extended with sophisticated language manipulations. We keep our syntax with the minimal set of operations that are enough to demonstrate our approach. Thus, we have included only a union operation for languages. `union` adds new grammar productions and inference rules to a language. For example, *bpa* `union` $(Process\ P ::= (loopOnNil\ P))\ (\longrightarrow\ A\ (loopOnNil\ (nil))\ (loopOnNil\ (nil))))$ returns the language with the same grammar productions for *Action*, and with the additional production $(loopOnNil\ P)$ for P. Also, the rule above for *loopOnNil* is added to the rules of *bpa*.

LANG-N-SEND extends the processes of the π-calculus with the following constructors. A process $(\ell, \mathfrak{T})>_x t$ is a *program execution*. This process executes the program t according to the operational semantics defined in the language ℓ. In particular, we query the language for reductions that are provable with $\longrightarrow$. We also keep track of the trace of executions. Traces are sequences of labels. As we use terms to represent labels, our traces $\mathfrak{T}$ are sequences of terms. We assume that a program execution starts with an empty sequence of labels, which we denote with [] to avoid a confusing empty space in our examples. A reduction step of t carries a label, and we append it to $\mathfrak{T}$. Then, $\mathfrak{T}$ contains all the labels of all the steps

of t, that is, $\mathfrak{T}$ is a trace of the execution of t. When the execution of t terminates, the trace is sent over the channel x.

To make an example:

$(bpa, [])>_x\ (seq\ (act\ (a))\ (seq\ (act\ (b))\ (act\ (c))))$ reduces to

$(bpa, (a))>_x\ (seq\ (act\ (b))\ (act\ (c)))$ which reduces to

$(bpa, (a)\ (b))>_x\ (act\ (c))$ which reduces to $\overline{x}\langle (a)\ (b)\rangle.\mathbf{0}$.

Notice that, in BPA, this last c does not take a transition, but $c \longrightarrow^{c} \checkmark$ holds. We could account for this with a straightforward modification of BPA that performs the last action as a labeled transition, but we prefer to use its original formulation.

A process $\texttt{isInTrace}(t, \mathfrak{T}) \Rightarrow P\ ;\ Q$ checks whether the label t is one of the labels in the trace $\mathfrak{T}$. In that case, this process behaves as P, otherwise it behaves as Q.

A process $x(l).P$ is a *language input prefix*. This process receives a language on the channel x, binds it to l, and continues as P. A process $\overline{x}\langle \ell \rangle.P$ is a *language output prefix*. This process evaluates ℓ to a language $\mathscr{L}$, sends it over the channel x, and continues as P. Similarly, a process $x(tr).P$ is a *trace input prefix* and receives traces. A process $\overline{x}\langle \mathfrak{T} \rangle.P$ is a *trace output prefix* and sends traces.

4 A Reduction Semantics for LANG-N-SEND

Figure 1 shows the reduction semantics of LANG-N-SEND. Structural congruence $\equiv$ is standard. The reduction relation for the processes of LANG-N-SEND is $\longrightarrow$. This relation relies on two auxiliary relations: the *language building reduction relation* $\longrightarrow_{\mathsf{lan}}$, and the *program reduction relation* $\longrightarrow_{\mathsf{exe}}$. Below we describe the rules of Figure 1.

Rule (COMM) is standard. Rule (COMM-LANG) handles the communication of languages. In this rule, $\longrightarrow^{*}_{\mathsf{lan}}$ is the reflexive and transitive closure of $\longrightarrow_{\mathsf{lan}}$. We evaluate ℓ to a language $\mathscr{L}$, and only then we perform the passing. Rule (COMM-TRACE) handles the communication of traces. Substitution $P\{\mathscr{L}/l\}$ substitutes the free occurrences of l in P with $\mathscr{L}$. Substitution $P\{\mathfrak{T}/tr\}$ substitutes the free occurrences of tr in P with $\mathfrak{T}$. Both substitutions are capture-avoiding, and their definition is straightforward, so we do not show it.

Rule (EXEC) handles program executions when the language is available, that is, it has been evaluated to some $\mathscr{L}$. This rule simply relies on $\longrightarrow_{\mathsf{exe}}$. Rule (EXEC-CTX) evaluates ℓ with $\longrightarrow_{\mathsf{lan}}$-reductions.

Rules (IS-IN-TRACE1) and (IS-IN-TRACE2) define the behavior of `isInTrace`. This process takes a step to P if the label is in $\mathfrak{T}$, and takes a step to Q otherwise.

Rule (UNION) performs the union of two languages using the operation $\cup_{\mathsf{snx}}$. This operation adds new grammar productions and inference rules to a language in the way that we have seen. This operation has been previously defined in [5]. (We discuss related work in Section 6.) Rules (UNION-CTX1) and (UNION-CTX2) evaluate the first and second argument of `union`, respectively.

Rule (PROGRAM-STEP) handles program executions $(\mathscr{L}, \mathfrak{T})>_x t$. This rule is responsible for executing t according to the operational semantics of $\mathscr{L}$. To do so, we should query the inference rule system in $\mathscr{L}$. However, $\mathscr{L}$ contains *syntax that represents* an inference system. We adopt the solution used in [5]: we translate the language into a higher-order logic program with $[\![\mathscr{L}]\!]^{\mathsf{lp}}$, and we use the provability relation $\models$ of logic programs to check whether a step from t is provable for some target t' and some label t''. The translation $[\![\mathscr{L}]\!]^{\mathsf{lp}}$ to logic programs is easy, and has been described in [5]. The way this translation works was not novel in there either, as it has been demonstrated previously that inference systems of the like map well into logic programs [11, 17]. The provability relation $\models$ comes directly

from the semantics of higher order logic programs, which can be found in [11]. Rule (PROGRAM-STEP) also appends t'' to the trace recorded in the program execution.

Rule (PROGRAM-END) detects that a step is not provable for t. Then, the execution of t is terminated, and we send the trace over the channel x.

Notice that t may fail to prove a step for several reasons, including that t is stuck because of missing reduction rules in an ill-defined language. Programmers are responsible for giving well-designed languages, as LANG-N-SEND does not check that.

Reduction Semantics $\boxed{P \equiv P,\ P \longrightarrow P,\ \ell \longrightarrow_{\mathsf{lan}} \ell,\ P \longrightarrow_{\mathsf{exe}} P}$

$$P \mid \mathbf{0} \equiv P \qquad P \mid Q \equiv Q \mid P \qquad (P \mid Q) \mid R \equiv P \mid (Q \mid R) \qquad !P \equiv P \mid !P$$

$$(\nu x).\mathbf{0} \equiv \mathbf{0} \qquad (\nu x).(\nu y).P \equiv (\nu y).(\nu x).P \qquad (\nu x).(P \mid Q) \equiv (\nu x).P \mid Q, \textit{ if } x \textit{ is not a free name of } Q$$

$$\frac{P_1 \longrightarrow P_1'}{P_1 \mid P_2 \longrightarrow P_1' \mid P_2} \qquad \frac{P \longrightarrow P'}{(\nu x).P \longrightarrow (\nu x).P'} \qquad \frac{P \equiv P' \quad P' \longrightarrow Q' \quad Q' \equiv Q}{P \longrightarrow Q}$$

(COMM)
$$x(y).P \mid \overline{x}\langle z\rangle.Q \longrightarrow P\{z/y\} \mid Q$$

(COMM-LANG)
$$\frac{\ell \longrightarrow^{*}_{\mathsf{lan}} \mathscr{L}}{x(l).P \mid \overline{x}\langle \ell\rangle.Q \longrightarrow P\{\mathscr{L}/l\} \mid Q}$$

(COMM-TRACE)
$$x(tr).P \mid \overline{x}\langle \mathfrak{T}\rangle.Q \longrightarrow P\{\mathfrak{T}/tr\} \mid Q$$

(EXEC)
$$\frac{(\mathscr{L},\mathfrak{T})\!>_x t \longrightarrow_{\mathsf{exe}} P}{(\mathscr{L},\mathfrak{T})\!>_x t \longrightarrow P}$$

(EXEC-CTX)
$$\frac{\ell \longrightarrow_{\mathsf{lan}} \ell'}{(\ell,\mathfrak{T})\!>_x t \longrightarrow (\ell',\mathfrak{T})\!>_x t}$$

(IS-IN-TRACE1)
$$\frac{\mathfrak{T} = t_1 \cdots t_n \qquad 1 \le i \le n}{\mathtt{isInTrace}(t_i,\mathfrak{T}) \Rightarrow P\,;\,Q \longrightarrow P}$$

(IS-IN-TRACE2)
$$\frac{\mathfrak{T} = t_1 \cdots t_n \qquad t \neq t_i, \textit{ for all } 1 \le i \le n}{\mathtt{isInTrace}(t,\mathfrak{T}) \Rightarrow P\,;\,Q \longrightarrow Q}$$

(UNION)
$$\mathscr{L}_1 \,\mathtt{union}\, \mathscr{L}_2 \longrightarrow_{\mathsf{lan}} \mathscr{L}_1 \cup_{\mathsf{snx}} \mathscr{L}_2$$

(UNION-CTX1)
$$\frac{\ell_1 \longrightarrow_{\mathsf{lan}} \ell_1'}{\ell_1 \,\mathtt{union}\, \ell_2 \longrightarrow_{\mathsf{lan}} \ell_1' \,\mathtt{union}\, \ell_2}$$

(UNION-CTX2)
$$\frac{\ell_2 \longrightarrow_{\mathsf{lan}} \ell_2'}{\ell_1 \,\mathtt{union}\, \ell_2 \longrightarrow_{\mathsf{lan}} \ell_1 \,\mathtt{union}\, \ell_2'}$$

(PROGRAM-STEP)
$$\frac{[\![\mathscr{L}]\!]^{\mathsf{lp}} \models (t \longrightarrow^{t''} t')}{(\mathscr{L},\mathfrak{T})\!>_x t \longrightarrow_{\mathsf{exe}} (\mathscr{L},\mathfrak{T}\,t'')\!>_x t'}$$

(PROGRAM-END)
$$\frac{[\![\mathscr{L}]\!]^{\mathsf{lp}} \not\models (t \longrightarrow^{t''} t')}{(\mathscr{L},\mathfrak{T})\!>_x t \longrightarrow_{\mathsf{exe}} \overline{x}\langle \mathfrak{T}\rangle.\mathbf{0}}$$

Figure 1: Reduction semantics of LANG-N-SEND.

5 Examples

Server Provides a Disrupt Operator In this example, *server* is a server that offers two services: *task* and *quitOnFailureTask*. These tasks are executed with BPA processes. However, *quitOnFailureTask* is critical, and should stop if a mistake is detected. BPA does not have a way to model disruptions. Therefore, at the moment of executing *quitOnFailureTask* (and only in that case), *server* requests the piece of operational semantics of the disrupt operator of LOTOS [9], adapted for BPA in [2], and adds it

to the language *bpa* from Section 2. Intuitively, $P \blacktriangleright Q$ means that P can be disrupted by Q. This process behaves as P, though at any point, non-deterministically, Q can start its computation, which discards P forever. We repeat the inference rules for $\blacktriangleright$ ([2]).

$$\frac{P_1 \longrightarrow^a \checkmark}{P_1 \blacktriangleright P_2 \longrightarrow^a \checkmark} \qquad \frac{P_1 \xrightarrow{a} P_1'}{P_1 \blacktriangleright P_2 \xrightarrow{a} P_1' \blacktriangleright P_2}$$

$$\frac{P_2 \xrightarrow{a} P_2'}{P_1 \blacktriangleright P_2 \xrightarrow{a} P_2'} \qquad \frac{P_2 \longrightarrow^a \checkmark}{P_1 \blacktriangleright P_2 \longrightarrow^a \checkmark}$$

We define the LANG-N-SEND counterpart of $\blacktriangleright$ in two parts. *almostDisrupt* contains the first row of the rules above. These rules define the behavior of $\blacktriangleright$ insofar the preempted process is concerned. *disruptRules* contains the second row of rules, which are for the preempting process. Then, *disrupt* contains the union of the two, and is the piece of language with the full definition of $\blacktriangleright$.

$$\begin{aligned}
&\mathit{almostDisrupt} \triangleq (\mathit{Process}\ P ::= (\blacktriangleright\ P\ P),\\
&\qquad \frac{(\mathit{checkMark}\ A\ P_1)}{(\mathit{checkMark}\ A\ (\blacktriangleright\ P_1\ P_2))} \qquad \frac{(\longrightarrow\ A\ P_1\ P_1')}{(\longrightarrow\ A\ (\blacktriangleright\ P_1\ P_2)\ (\blacktriangleright\ P_1'\ P_2))}\ )\\
&\mathit{disruptRules} \triangleq (\ \frac{(\longrightarrow\ A\ P_2\ P_2')}{(\longrightarrow\ A\ (\blacktriangleright\ P_1\ P_2)\ P_2')} \qquad \frac{(\mathit{checkMark}\ A\ P_2)}{(\mathit{checkMark}\ A\ (\blacktriangleright\ P_1\ P_2))}\ )\\
&\mathit{disrupt} \triangleq \mathit{almostDisrupt}\ \texttt{union}\ \mathit{disruptRule}
\end{aligned}$$

Below, the process *disruptOperatorProvider* is a server, different from *server*, that provides the *disrupt* piece of language over the channel *getDisrupt*. The code for *server* is also below. We assume that *bpa_program*, a term, is a BPA process to be executed for *quitOnFailureTask*, and that *bpa_sorry* is the BPA process that can non-deterministically preempt *bpa_program*. For readability, we use $\blacktriangleright$ in infix notation. The process for *task* is irrelevant, and we chose $(\mathit{act}\ (a))$.

$$\begin{aligned}
&\mathit{disruptOperatorProvider} \triangleq !(\overline{\mathit{getDisrupt}}\langle \mathit{disrupt}\rangle)\\
&\mathit{server} \triangleq\ !(\ \mathit{task}(x).(\mathit{bpa}, [])>_x (\mathit{act}\ (a))\\
&\qquad\qquad +\\
&\qquad\qquad \mathit{quitOnFailureTask}(x).\mathit{getDisrupt}(l).(\mathit{bpa}\ \texttt{union}\ l, [])>_x (\mathit{bpa_program} \blacktriangleright \mathit{bpa_sorry})\ )\\
&\mathit{system} \triangleq (\mathit{server} \mid \mathit{disruptOperatorProvider} \mid \mathit{client}_1 \mid \mathit{client}_2\ \ldots\ \mid \mathit{client}_n)
\end{aligned}$$

Suppose that *bpa_sorry* performs the action (sorry). We can detect whether *bpa_program* has been disrupted with `isInTrace`. The second branch of the choice operator of *server* would be

$$\begin{aligned}
&\mathit{quitOnFailureTask}(x).\mathit{getDisrupt}(l).(\nu x).\\
&\qquad ((\mathit{bpa}\ \texttt{union}\ l, [])>_x (\mathit{bpa_program} \blacktriangleright \mathit{bpa_sorry}) \mid x(tr).\texttt{isInTrace}((\mathit{sorry}), tr) \Rightarrow P_1\ ;\ P_2)
\end{aligned}$$

Here, *server* creates a private channel x over which the trace is sent. We assume that P_1 and P_2 are two processes that *server* cares to execute depending on whether (sorry) is in the trace or not.

Server Decides Disrupt vs Interrupt In this example, the server *disruptOperatorProvider* is called *quitModeProvider*. It takes in input a channel (such as *quitOnFailureTask*), and non-deterministically

decides whether to provide the disrupt operator or the interrupt operator $\triangleright$ from [2]. The process $P \triangleright Q$ means that P can be interrupted by Q. Differently from the disrupt operator, which completely discards P when Q takes over, the interrupt operator resumes P after Q terminates.

bpa_program uses *one* operator whose underlying semantics is given by *quitModeProvider*. We fix the symbol for this operator to be $\blacktriangleright$. Therefore, when *quitModeProvider* gives the interrupt semantics, it does so by giving the rules of $\triangleright$ for the symbol $\blacktriangleright$. The piece of language for the preempted process, *almostDisrupt*, is the same for $\blacktriangleright$ and $\triangleright$. The rules for the preempting process are the following ([2]).

$$\frac{P_2 \xrightarrow{a} P_2'}{P_1 \triangleright P_2 \xrightarrow{a} P_2' \cdot P_1} \qquad \frac{P_2 \longrightarrow^{a} \checkmark}{P_1 \triangleright P_2 \xrightarrow{a} P_1}$$

Below, *interruptRules* contains the LANG-N-SEND counterpart of these rules, though defined for the symbol $\blacktriangleright$, as explained above. When we add *interruptRules* to *almostDisrupt* we obtain the full definition of the interrupt operator (given as $\blacktriangleright$), which we call *interrupt*.

$$\mathit{interruptRules} \triangleq \left(\frac{(\longrightarrow\ A\ P_2\ P_2')}{(\longrightarrow\ A\ (\blacktriangleright\ P_1\ P_2)\ (\mathit{seq}\ P_2'\ P_1))} \qquad \frac{(\mathit{checkMark}\ A\ P_2)}{(\longrightarrow\ A\ (\blacktriangleright\ P_1\ P_2)\ P_1)} \right)$$

$$\mathit{interrupt} \triangleq \mathit{almostDisrupt}\ \texttt{union}\ \mathit{interruptRules}$$

$$\mathit{quitModeProvider} \triangleq\ !\mathit{whatTask}(y).(\ \overline{\mathit{getQuitMode}}\langle \mathit{interrupt}\rangle\ +\ \overline{\mathit{getQuitMode}}\langle \mathit{disrupt}\rangle\ )$$

$$\begin{aligned}\mathit{server} \triangleq\quad & !(\mathit{task}(x).(\mathit{bpa}, [])\texttt{>}_x\ (\mathit{act}\ (a)) \\ & \quad + \\ & \quad \mathit{quitOnFailureTask}(x).\overline{\mathit{whatTask}}\langle \mathit{quitOnFailureTask}\rangle.\mathit{getQuitMode}(l). \\ & \qquad\qquad (\mathit{bpa}\ \texttt{union}\ l, [])\texttt{>}_x\ (\mathit{bpa\_program} \blacktriangleright \mathit{bpa\_sorry})\ )\end{aligned}$$

$$\mathit{system} \triangleq (\mathit{server} \mid \mathit{quitModeProvider} \mid \mathit{client}_1 \mid \mathit{client}_2\ \ldots\ \mid \mathit{client}_n)$$

Server Decides Synchronous vs Asynchronous In this example, the process *client* executes a CCS process called *ccs_program*. However, *client* requests the semantics of the output prefix operator from the server *outputProvider*, which decides, non-deterministically, whether *ccs_program* must be executed synchronously or asynchronously. There is a syntactic difference between the synchronous output $\overline{a}.P$ and the asynchronous output $\overline{a}$ (with no continuation process). As *ccs_program* is fixed, we settle to use the more general output form $\overline{a}.P$, though its semantics will be given by the server.

We define a partial CCS with inaction, input prefix, output prefix, a one-channel restriction operator $P\backslash a$, and the parallel operator. The semantics of the output prefix, however, is not given. As we do not have negative premises in rules, we define $P\backslash a$ by cases. For simplicity, we only include channels x and y.

$$\begin{aligned}&\mathit{Channel}\ a ::= x \mid y \\ &\mathit{Label}\ L ::= \tau \mid a \mid \overline{a} \\ &\mathit{Process}\ P ::= \mathbf{0} \mid a.P \mid \overline{a}.P \mid\ \mid P \mid\ \mid P\backslash a\end{aligned}$$

$$a.P \xrightarrow{a} P \qquad \frac{P \xrightarrow{\tau} P'}{P\backslash a \xrightarrow{\tau} P'\backslash a}$$

$$\frac{P \xrightarrow{y} P'}{P\backslash x \xrightarrow{y} P'\backslash x} \qquad \frac{P \xrightarrow{\overline{y}} P'}{P\backslash x \xrightarrow{\overline{y}} P'\backslash x} \qquad \frac{P \xrightarrow{x} P'}{P\backslash y \xrightarrow{x} P'\backslash y} \qquad \frac{P \xrightarrow{\overline{x}} P'}{P\backslash y \xrightarrow{\overline{x}} P'\backslash y}$$

$$\frac{P_1 \xrightarrow{L} P_1'}{P_1 \mid P_2 \xrightarrow{L} P_1' \mid P_2} \qquad \frac{P_2 \xrightarrow{L} P_2'}{P_1 \mid P_2 \xrightarrow{L} P_1 \mid P_2'}$$

$$\frac{P_1 \xrightarrow{a} P_1' \quad P_2 \xrightarrow{\overline{a}} P_2'}{P_1 \mid P_2 \xrightarrow{\tau} P_1' \mid P_2'} \qquad \frac{P_1 \xrightarrow{\overline{a}} P_1' \quad P_2 \xrightarrow{a} P_2'}{P_1 \mid P_2 \xrightarrow{\tau} P_1' \mid P_2'}$$

Below, *partialCCS* contains the LANG-N-SEND counterpart of the partial CCS defined above. Inaction is called *nil*, input prefix is called *in*, output prefix is called *out*, the restriction operator is called *res*, and the parallel operator is called *par*.

$$\begin{aligned}
&partialCCS \triangleq \\
&\qquad (Channel\ a ::= (x) \mid (y) \\
&\qquad Label\ L ::= (tau) \mid (in\ a) \mid (out\ a) \\
&\qquad Process\ P ::= (nil) \mid (in\ a\ P) \mid (out\ a\ P) \mid (res\ a\ P) \mid (par\ P\ P),
\end{aligned}$$

$$(\longrightarrow\ (in\ a)\ (in\ a\ P)\ P) \qquad \frac{(\longrightarrow\ (tau)\ P\ P')}{(\longrightarrow\ (tau)\ (res\ a\ P)\ (res\ a\ P'))}$$

$$\frac{(\longrightarrow\ (in\ (y))\ P\ P')}{(\longrightarrow\ (in\ (y))\ (res\ (x)\ P)\ (res\ (x)\ P'))} \qquad \frac{(\longrightarrow\ (out\ (y))\ P\ P')}{(\longrightarrow\ (out\ (y))\ (res\ (x)\ P)\ (res\ (x)\ P'))}$$

$$\frac{(\longrightarrow\ (in\ (x))\ P\ P')}{(\longrightarrow\ (in\ (x))\ (res\ (y)\ P)\ (res\ (y)\ P'))} \qquad \frac{(\longrightarrow\ (out\ (x))\ P\ P')}{(\longrightarrow\ (out\ (x))\ (res\ (y)\ P)\ (res\ (y)\ P'))}$$

$$\frac{(\longrightarrow\ L\ P_1\ P_1')}{(\longrightarrow\ L\ (par\ P_1\ P_2)\ (par\ P_1'\ P_2))} \qquad \frac{(\longrightarrow\ L\ P_2\ P_2')}{(\longrightarrow\ L\ (par\ P_1\ P_2)\ (par\ P_1\ P_2'))}$$

$$\frac{(\longrightarrow\ (in\ A)\ P_1\ P_1') \quad (\longrightarrow\ (out\ A)\ P_2\ P_2')}{(\longrightarrow\ (tau)\ (par\ P_1\ P_2)\ (par\ P_1'\ P_2'))} \qquad \frac{(\longrightarrow\ (out\ A)\ P_1\ P_1') \quad (\longrightarrow\ (in\ A)\ P_2\ P_2')}{(\longrightarrow\ (tau)\ (par\ P_1\ P_2)\ (par\ P_1'\ P_2'))}\ )$$

To complete *partialCCS* with synchronous output, we add the usual rule for output prefix. To complete *partialCCS* with asynchronous output, we add 1) the asynchronous output $\overline{a}$ to the grammar, added as $(out'\ a)$ below, 2) its reduction rule $\overline{a} \xrightarrow{\overline{a}} \mathbf{0}$, and 3) the rule $\overline{a}.P \xrightarrow{\tau} (\overline{a} \mid P)$[1]:

synchOutput $\triangleq$ ($(\longrightarrow\ (out\ a)\ (out\ a\ P)\ P)$) *(* this rule is* $\overline{a}.P \xrightarrow{a} P$ **)*

asynchOutput $\triangleq$ ($Process\ P ::= (out'\ a)$,
$(\longrightarrow\ (out\ a)\ (out'\ a\ )\ (nil))$ $(\longrightarrow\ (tau)\ (out\ a\ P)\ (par\ (out'\ a)\ P))$).

We give the definitions of *outputProvider*, *client*, and *system* below. When *ccs_program* is the process $(\overline{x}.y.\mathbf{0} \mid \overline{y}.\mathbf{0})\backslash x$, whether a communication over the channel y happens or not depends on whether *outputProvider* sends *synchOutput* or *asynchOutput*.

outputProvider $\triangleq$!($\overline{getOutput}\langle synchOutput\rangle + \overline{getOutput}\langle asynchOutput\rangle$)

client $\triangleq$ $getOutput(l).(partialCCS\ \texttt{union}\ l, [])\texttt{>}_x\ ccs\_program$

system $\triangleq$ *client* | *outputProvider*

[1]Notice that a simple rule like $\overline{a}.P \xrightarrow{\tau} \overline{a}.\mathbf{0} \mid P$ is problematic, as the rule applies to $\overline{a}.\mathbf{0}$, as well, replicating forever. Also notice that this τ-transition does not resolve a choice, as *partialCCS* does not contain $+$.

6 Related Work

Our closest related work is [5]. Such work offers a λ-calculus with first-class languages. We would like to characterize precisely the differences between this paper and that work. This paper embeds language definitions in the context of the π-calculus rather than the λ-calculus. The syntax for languages, the language union operator, and the translation to logic programs are from [5]. The operator for program executions, and rule (PROGRAM-STEP) are inspired by [5], but there are several differences in that [5] does not allow for labeled transitions, and does not keep track of the trace of the execution. Moreover, as a consequence of this latter remark, [5] does not have operations such as `isInTrace`, nor any other operation for analyzing executions. Furthermore, [5] imposes that languages have a notion of values, and successful program executions terminate ending up with a value. This hardly applies to process algebras. [5] does not make any example of concurrent scenarios. All the examples in this paper are new.

Semantics engineering tools allow programmers to define their own programming languages [7, 18, 20]. Language workbenches [6] go even further in that direction, and can automatically generate many components for the languages being defined, such as editors with syntax colouring, highlighting, completion, and reference resolution, and they assist in code generation, as well as other phases. However, we are not aware of systems that allow pieces of languages to be sent and received.

Multi-language operational semantics has been studied in several works. Matthews and Findler provide a seminal work of this field [10]. Recent works in multi-language semantics are [15, 16, 19]. All these works apply to two languages selected beforehand, and do not handle arbitrary languages specified by users. Furthermore, they do not offer a formal semantics of processes that communicate languages.

7 Conclusion

We have presented LANG-N-SEND, a π-calculus that is equipped with language definitions. Processes can define languages, and use them to execute programs. Moreover, processes can send and receive pieces of languages through channels. We have presented a reduction semantics for LANG-N-SEND.

We have offered examples that show that LANG-N-SEND can express concurrent scenarios that are not typical, where processes add language features based on semantics sent by servers, and where they obtain which semantics their operators adopt from servers. We believe that LANG-N-SEND represents a first step to a firm foundation for this type of programming.

In the future, we would like to extend LANG-N-SEND. Indeed, we see LANG-N-SEND as a minimal foundational calculus that accommodates the communication of languages. We purposely did not include operations that, in fact, are interesting in this context. We plan to extend LANG-N-SEND with more operations on languages, such as removing rules, and renaming operators, as well as more complex features such as converting languages from substitution-based to environment-based, among others.

Adding binders to our language definitions does not seem to be problematic. [5] has made that addition to model a λ-calculus as language definition. We plan to use binders to make examples with the π-calculus and its variants as LANG-N-SEND language definitions.

We plan to add more operations that query traces more precisely, such as counting labels in traces, and checking whether some labels appear before others. We also plan to add primitive operations for slicing the traces received [1], and we plan to add monitors to program executions [4].

LANG-N-SEND does not allow for the term of a terminated program execution to be sent. We have not included this feature because we believe that it enables rather complex dynamics, and we wanted to confine our examples to the already interesting scenarios that sending/receiving languages allow. We

plan to explore the sending of terms after execution as future work.

We plan to study more examples such as servers that decide the semantics of the parallel operator for client processes (CCS style, only interleaving and no communication, or the synchronous CSP parallel composition [8], for instance). Another example is that of servers that decide the semantics of the choice operator, such as internal vs external, among other possibilities.

Finally, we would like to study an appropriate notion of bisimilarity equivalence in this context.

References

[1] Hiralal Agrawal & Joseph R. Horgan (1990): *Dynamic Program Slicing*. In: *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, PLDI '90, Association for Computing Machinery, New York, NY, USA, pp. 246–256, doi:10.1145/93542.93576.

[2] Jos C. M. Baeten & Jan A. Bergstra (2000): *Mode transfer in process algebra*. *Computing Science Reports* 00-01, Technische Universiteit Eindhoven.

[3] Jan A. Bergstra & Jan W. Klop (1984): *Process Algebra for Synchronous Communication*. *Information and Control* 60(1-3), pp. 109–137, doi:10.1016/S0019-9958(84)80025-X.

[4] Ian Cassar, Adrian Francalanza, Luca Aceto & Anna Ingólfsdóttir (2017): *A Survey of Runtime Monitoring Instrumentation Techniques*. In: *Proceedings Second International Workshop on Pre- and Post-Deployment Verification Techniques, PrePost@iFM 2017, Torino, Italy, 19 September 2017*, pp. 15–28, doi:10.4204/EPTCS.254.2.

[5] Matteo Cimini (2021): *A Calculus for Multi-language Operational Semantics*. In: *Software Verification - 13th International Conference, VSTTE 2021, New Haven, CT, USA, October 18-19, 2021, and 14th International Workshop, NSV 2021, Los Angeles, CA, USA, July 18-19, 2021, Revised Selected Papers*, pp. 25–42, doi:10.1007/978-3-030-95561-8_3.

[6] Sebastian Erdweg, Tijs Storm, Markus Völter, Meinte Boersma, Remi Bosman, WilliamR. Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabriël D. P. Konat, PedroJ. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad A. Vergu, Eelco Visser, Kevin Vlist, Guido H. Wachsmuth & Jimi Woning (2013): *The State of the Art in Language Workbenches*. In Martin Erwig, Richard F. Paige & Eric Wyk, editors: *Software Language Engineering*, *Lecture Notes in Computer Science* 8225, Springer, pp. 197–217, doi:10.1007/978-3-319-02654-1_11.

[7] Matthias Felleisen, Robert Bruce Findler & Matthew Flatt (2009): *Semantics Engineering with PLT Redex*, 1st edition. The MIT Press.

[8] C.A.R. Hoare (1985): *Communicating Sequential Processes*. Prentice-Hall International Series in Computer Science, Prentice Hall. Available at http://www.usingcsp.com/cspbook.pdf.

[9] ISO/IEC (1989): *LOTOS — a formal description technique based on the temporal ordering of observational behaviour*. ISO IS 8807, doi:10.3403/00230466U.

[10] Jacob Matthews & Robert Bruce Findler (2007): *Operational Semantics for Multi-Language Programs*. In: *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '07, Association for Computing Machinery, New York, NY, USA, p. 3–10, doi:10.1145/1190216.1190220.

[11] Dale Miller & Gopalan Nadathur (2012): *Programming with Higher-Order Logic*, 1st edition. Cambridge University Press, New York, NY, USA, doi:10.1017/CBO9781139021326.

[12] Robin Milner, Joachim Parrow & David Walker (1992): *A calculus of mobile processes, I*. *Information and Computation* 100(1), pp. 1–40, doi:10.1016/0890-5401(92)90008-4.

[13] Robin Milner, Joachim Parrow & David Walker (1992): *A calculus of mobile processes, II*. *Information and Computation* 100(1), pp. 41–77, doi:10.1016/0890-5401(92)90009-5.

[14] Peter D. Mosses (2008): *Component-Based Description of Programming Languages*. In: *Visions of Computer Science - BCS International Academic Conference, Imperial College, London, UK, 22-24 September 2008*, pp. 275–286, doi:10.14236/ewic/VOCS2008.23.

[15] Daniel Patterson & Amal Ahmed (2017): *Linking Types for Multi-Language Software: Have Your Cake and Eat It Too*. In Benjamin S. Lerner, Rastislav Bodík & Shriram Krishnamurthi, editors: *2nd Summit on Advances in Programming Languages (SNAPL 2017)*, *Leibniz International Proceedings in Informatics (LIPIcs)* 71, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, pp. 12:1–12:15, doi:10.4230/LIPIcs.SNAPL.2017.12.

[16] Daniel Patterson, Jamie Perconti, Christos Dimoulas & Amal Ahmed (2017): *FunTAL: Reasonably Mixing a Functional Language with Assembly*. In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, Association for Computing Machinery, New York, NY, USA, pp. 495–509, doi:10.1145/3062341.3062347.

[17] Frank Pfenning & Carsten Schürmann (1999): *System Description: Twelf - A Meta-Logical Framework for Deductive Systems*. In: *Proceedings of the 16th International Conference on Automated Deduction: Automated Deduction*, CADE-16, Springer-Verlag, Berlin, Heidelberg, pp. 202–206, doi:10.1007/3-540-48660-7_14.

[18] Grigore Rosu & Traian F. Şerbănuţă (2010): *An overview of the K semantic framework*. *The Journal of Logic and Algebraic Programming* 79(6), pp. 397–434, doi:10.1016/j.jlap.2010.03.012.

[19] Gabriel Scherer, Max S. New, Nick Rioux & Amal Ahmed (2018): *Fabulous Interoperability for ML and a Linear Language*. In: *Foundations of Software Science and Computation Structures - 21st International Conference, FOSSACS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*, pp. 146–162, doi:10.1007/978-3-319-89366-2_8.

[20] Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar & Rok Strniša (2007): *Ott: Effective Tool Support for the Working Semanticist*. In: *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, ICFP '07, ACM, New York, NY, USA, pp. 1–12, doi:10.1145/1291151.1291155.