The Buffered π-Calculus: A Model for Concurrent Languages[⋆]

Xiaojie Deng[1], Yu Zhang[2], Yuxin Deng[1], and Farong Zhong[3]

[1] BASICS, Department of Computer Science and Engineering, Shanghai Jiao Tong University, Shanghai, China
[2] State Key Laboratory for Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China
[3] Department of Computer Science, Zhejiang Normal University, Zhejiang, China

Abstract. Message-passing based concurrent languages are widely used in developing large distributed and coordination systems. This paper presents the buffered π-calculus — a variant of the π-calculus where channel names are classified into buffered and unbuffered: communication along buffered channels is asynchronous, and remains synchronous along unbuffered channels. We show that the buffered π-calculus can be fully simulated in the polyadic π-calculus with respect to strong bisimulation. In contrast to the π-calculus which is hard to use in practice, the new language enables easy and clear modeling of practical concurrent languages. We encode two real-world concurrent languages in the buffered π-calculus: the (core) Go language and the Core Erlang. Both encodings are fully abstract with respect to weak bisimulations.

Keywords: process calculus, formal model, full abstraction

1 Introduction

Concurrent programming languages become popular in recent years thanks to the large demand of distributed computing and the pervasive exploitation of multi-processor architectures. Unlike the shared-memory concurrency model, which is now mainly used on multi-processor platforms, message passing based concurrent languages are particularly popular in developing large distributed, coordination systems. Indeed, quite a few real-world concurrent languages are intensively used in industry. The most well-known languages are probably Erlang, developed by Ericsson [1], and the much younger language Go, developed by Google [6]. Both languages achieve their asynchronous communication via order-preserving message passing.

On the other side, the π-calculus [11, 14] has shown its success in modeling and verifying both specifications and implementations. Its asynchronous variant

[⋆] Partially supported by Natural Science Foundation of China (61173033, 61033002, 61161130530, 61100053) and the Opening Fund of Top Key Discipline of Computer Software and Theory in Zhejiang Provincial Colleges at Zhejiang Normal University.

[3, 8] is a good candidate as the target formal model. Despite the fact that it is called asynchronous, communication in the asynchronous π-calculus is however synchronous. It is shown in [2] that the communication modelled by the asynchronous π-calculus is equivalent to message passing via bags — senders put messages into some bags, and receivers may get arbitrary messages from these bags. This result indicates that additional effort should be made to respect the order of the messages, which is adopted in the implementation of many concurrent languages.

In view of this, we may expect a formal model where asynchronous communication is supported natively. In fact, our primary goal is to achieve a formal model by which we can easily define a formal semantics of Go and do verification on top of it. The developers of Go claim that the concurrency feature of Go is rooted in CSP [7], while we show that the π-calculus should be an appropriate model for Go as CSP does not support a channel passing mechanism.

In the spirit of the name passing mechanism of the π-calculus and the channel type of the Go language, we extend the π-calculus by introducing a special kind of names, each associated with a first-in-first-out buffer. We call these names *buffered names*. Communication along buffered names is asynchronous, while that along unbuffered (normal) names remains synchronous. We call this variant language the buffered π-calculus, and abbreviate it as the π_b-calculus.

We develop the π_b-calculus by defining its operational semantics as a labelled transition system and supplying an encoding into the polyadic π-calculus. We also present translations of the languages Go and Erlang into the π_b-calculus and show that the model is sufficient and relatively easier for modeling real-world concurrent languages.

Beauxis *et al* introduced the $\pi_{\mathfrak{B}}$-calculus in order to study the asynchronous nature of the asynchronous π-calculus [2]. Their asynchronous communication is achieved via explicit use of buffers. In case that the buffers are ordered structures such as queues or stacks, the asynchronous communication modelled by $\pi_{\mathfrak{B}}$ differs from that by the asynchronous π-calculus. While communication in the $\pi_{\mathfrak{B}}$-calculus is always asynchronous, we keep both synchronous and asynchronous communication in the π_b-calculus, through different types of names.

Encoding programming languages in process calculus have been studied by many researchers. Milner defines the semantics of a non-trivial parallel programming language by a translation into CCS in [9]. In [15], a translation from a parallel object oriented language to the minimal π-calculus is presented. The correctness of the translation is justified by the operational correspondence between units and their encodings. Our treatments to the Go language follows the approach in [15]. In addition, we show a full abstraction theorem, namely equivalent Go programs are translated into equivalent π_b processes.

For functional languages, Noll and Roy [12] presented an initial translation mapping from a Core Erlang [4] to the asynchronous π-calculus. Later on they [13] improved the translation by revising the non-deterministic encoding of pattern matching based expressions, and by adding the encoding for tuples. Their translations, however, are not sound in the sense that the order of messages is not always respected. By modelling the mailbox structure explicitly by buffered names in the π_b-calculus, we obtain a more accurate encoding which is fully abstract with respect to weak bisimulation.

The rest of the paper is structured as follows. Section 2 presents the syntax and semantics of the π_b-calculus and a simple encoding in the polyadic π-calculus [10]. We show that this encoding preserves the strong bisimulation relation. In Section 3 we define a formal semantics for Go and present an encoding of Go in the π_b-calculus. Due to page limit, the encoding of Erlang is supplied in the full version of the current paper [5], as well as many definitions and proofs. Finally, Section 4 concludes the paper.

2 The π_b-Calculus

We assume an infinite set $\mathcal{N}$ of names, ranged over by a, b, c, d, x, y. Processes are defined by the following grammar:

$$P, Q, \ldots := \sum_{i\in I} \pi_i.P_i \;\Big|\; P|Q \;\Big|\; (\nu c : n)P \;\Big|\; (\nu c)P \;\Big|\; !P$$

where $\pi = c(x) \mid \overline{c}\langle d\rangle \mid \tau$.

Most of the syntax is standard: $\sum_{i\in I} \pi_i.P_i$ is the guarded choice (I is finite), which behaves nondeterministically as one of its components $\pi_j.P_j$ for some $j \in I$; composition $P|Q$ acts as P and Q running in parallel; $!P$ is the replication of process P; Prefixes $c(x)$ and $\overline{c}\langle d\rangle$ are input and output along name c; and τ is the silent action. We write $\mathbf{0}$ for the empty guarded choice, it is the process which can do nothing.

The π_b-calculus extends the π-calculus in the fact that names can be buffered or unbuffered. Unbuffered names are names in the π-calculus, and buffered names have the buffer attribute specified by a *buffer store*. A buffer store, denoted by $\mathcal{B}$, is a partial function from buffered names to pairs (n, l), where n is a positive integer representing the capacity of the buffer, and l is a list of names in the buffer, with the same order. Both $(\nu c)P$ and $(\nu c : n)P$ are called *new processes*. The (standard) new process $(\nu c)P$ specifies that c (whether buffered or unbuffered) is a local name in P. The extended new process $(\nu c : n)P$ creates a local buffered name c, whose associated buffer has the capacity n for asynchronous communication inside P. Notice that $(\nu c)P$ only says that the name c is local and does not imply that c is unbuffered — c can be a buffered name whose buffer is already created in the buffer store.

Input process $c(x).P$ and output process $\overline{c}\langle d\rangle.P$ can communicate with each other along name c when they run in parallel. If c is an unbuffered name, the communication is synchronous and happens as in the π-calculus: the object d is passed from the output side to the input side. If c is a buffered name, then the communication becomes asynchronous: the output process simply puts d into the buffer of c if it is not full and continues, or blocks if the buffer is full; the input process retrieves the oldest value from the buffer of c if it is not empty and continues, or blocks if the buffer is empty.

As usual, we write $\tilde{c}$ for a sequence of names, and abbreviate $(\nu c_1)\ldots(\nu c_n)P$ to $(\nu c_1 \ldots c_n)P$. A name x is bound if it appears in input prefix, otherwise it is free. We write $P\{\tilde{c}/\tilde{x}\}$ for the process resulting from simultaneously substituting c_i for each free x_i in P. The newly created name c in $(\nu c : n)P$ or $(\nu c)P$ are local names. A name is global if it is not localized by any new operator. We use $ln(P)$ and $gn(P)$ for the set of local names and global names occurring in P.

Throughout the development of the paper, we assume the following *De Barendregt name convention*: *Local names are different from each other and from global names.* For instance, we shall never consider processes like $\overline{a}\langle c\rangle.(\nu a)P$ or $(\nu a)(\nu a)P$. We note that this convention is dispensable and we simply adopt it to make the presentation of the calculus simple and clean. One can also remove the convention and use syntactic rules to manage name conflicts, but dealing with names in buffers can be very subtle.

A process can send a local name into a buffer. The fact that a name stored in buffers is local must be tracked, because it may affect the name scope when another process retrieves this name from the buffer. The convention also works for buffer stores. We shall discuss more on this when defining the operational semantics. Inside a buffer store, a value of the form (νc) indicates that the name c was sent into the buffer when it was local. Given a buffer store $\mathcal{B}$, we write $gn(\mathcal{B}(b))$ for the set of global names that occur in b's buffer, and $gn(\mathcal{B}) = \bigcup_{b\in \mathtt{dom}(\mathcal{B})} gn(\mathcal{B}(b))$. Similarly $ln(\mathcal{B}(b))$ and $ln(\mathcal{B})$ for local names in $\mathcal{B}(b)$ and $\mathcal{B}$. The buffer store $\mathcal{B}\{c/d\}$ is obtained by substituting c for each d in $\mathcal{B}$.

We say a process Q is guarded in P, if every occurrence of Q in P is within some prefix process. Intuitively, a guarded process cannot affect the behavior of its host process until the action induced by its guarding prefix is performed. New operators are guarded in P if all new processes are guarded in P.

The structural congruence $\equiv_{\mathcal{B}}$ with respect to the buffer store $\mathcal{B}$ is defined as the smallest congruence relation over processes satisfying the following laws:

1. $P \equiv_{\mathcal{B}} Q$, if Q is obtained from P by renaming bound names, or local names not occurring in $\mathcal{B}$.
2. $P \mid Q \equiv_{\mathcal{B}} Q \mid P; P \mid (Q \mid R) \equiv_{\mathcal{B}} (P \mid Q) \mid R; P \mid \mathbf{0} \equiv_{\mathcal{B}} P$.
3. $!P \equiv_{\mathcal{B}} P \mid !P$.
4. $(\nu c)(\nu d)P \equiv_{\mathcal{B}} (\nu d)(\nu c)P$.
5. $(\nu c)\mathbf{0} \equiv_{\mathcal{B}} \mathbf{0}$, if $c \notin ln(\mathcal{B})$; $(\nu c)(P \mid Q) \equiv_{\mathcal{B}} (\nu c)P \mid Q$, if $c \notin ln(\mathcal{B}) \wedge c \notin gn(Q)$.

Structural congruence allows us to pull unguarded new operators to the "outermost" level.

Buffer store $\mathcal{B}$ is *valid* for process P if each local name of $\mathcal{B}$ appears in some new operator occurring at the outermost level of P, i.e., for every $c \in ln(\mathcal{B})$, $P \equiv_{\mathcal{B}} (\nu c)P'$ for some P'.

2.1 Operational Semantics

The (early) transition semantics of π_b is given in terms of a labelled transition system generated by the rules in Table 1. The transition rules are of the form $P, \mathcal{B} \xrightarrow{\alpha} P', \mathcal{B}'$, where P, P' are processes, $\mathcal{B}, \mathcal{B}'$ are buffer stores and α is an action, which can be one of the forms: silent action τ, free input $c(d)$, free output $\overline{c}\langle d\rangle$ or bound output $\overline{c}\langle \nu d\rangle$. We write $n(\alpha)$ for the set of names occurring in α.

$$\mathtt{IU}\ \frac{c \notin \mathtt{dom}(\mathcal{B})}{c(x).P,\mathcal{B} \xrightarrow{c(d)} P\{d/x\},\mathcal{B}} \qquad \mathtt{OU}\ \frac{c \notin \mathtt{dom}(\mathcal{B})}{\overline{c}\langle d\rangle.P,\mathcal{B} \xrightarrow{\overline{c}\langle d\rangle} P,\mathcal{B}} \qquad \textsc{Open}\ \frac{P,\mathcal{B}\{c/\nu c\} \xrightarrow{\overline{d}\langle c\rangle} P',\mathcal{B}'}{(\nu c)P,\mathcal{B} \xrightarrow{\overline{d}\langle \nu c\rangle} P',\mathcal{B}'}$$

$$\mathtt{IB}\ \frac{\mathcal{B}(b) = (n,[d] :: l)}{b(x).P,\mathcal{B} \xrightarrow{\tau} P\{d/x\},\mathcal{B}[b \mapsto (n,l)]} \qquad \mathtt{OB}\ \frac{\mathcal{B}(b) = (n,l);\ |l| < n}{\overline{b}\langle d\rangle.P,\mathcal{B} \xrightarrow{\tau} P,\mathcal{B}[b \mapsto (n,l :: [d])]}$$

$$\mathtt{IBG}\ \frac{\mathcal{B}(b) = (n,l);\ |l| < n;\ b \notin ln(P)}{P,\mathcal{B} \xrightarrow{b(d)} P,\mathcal{B}[b \mapsto (n,l :: [d])]} \qquad \mathtt{OBG}\ \frac{\mathcal{B}(b) = (n,[d] :: l);\ b \notin ln(P)}{P,\mathcal{B} \xrightarrow{\overline{b}\langle d\rangle} P,\mathcal{B}[b \mapsto (n,l)]}$$

$$\textsc{Sum}\ \frac{j \in I;\ \pi_j.P_j,\mathcal{B} \xrightarrow{\alpha} P',\mathcal{B}'}{\sum_{i\in I}\pi_i.P_i,\mathcal{B} \xrightarrow{\alpha} P',\mathcal{B}'} \qquad \textsc{Com}\ \frac{P,\mathcal{B} \xrightarrow{c(d)} P',\mathcal{B};\ Q,\mathcal{B} \xrightarrow{\overline{c}\langle d\rangle} Q',\mathcal{B};\ c \notin \mathtt{dom}(\mathcal{B})}{P \mid Q,\mathcal{B} \xrightarrow{\tau} P' \mid Q',\mathcal{B}}$$

$$\textsc{Par}\ \frac{P,\mathcal{B} \xrightarrow{\alpha} P',\mathcal{B}';\ \text{new operators are guarded in } P \mid Q}{P \mid Q,\mathcal{B} \xrightarrow{\alpha} P' \mid Q,\mathcal{B}'}$$

$$\textsc{New}\ \frac{P,\mathcal{B}\{c/\nu c\} \xrightarrow{\alpha} P',\mathcal{B}';\ c \notin n(\alpha)}{(\nu c)P,\mathcal{B} \xrightarrow{\alpha} (\nu c)P',\mathcal{B}'\{\nu c/c\}} \qquad \textsc{Stru}\ \frac{P \equiv_{\mathcal{B}} P';\ P',\mathcal{B} \xrightarrow{\alpha} Q',\mathcal{B}';\ Q' \equiv_{\mathcal{B}'} Q}{P,\mathcal{B} \xrightarrow{\alpha} Q,\mathcal{B}'}$$

$$\textsc{NewB}\ (\nu b : n)P,\mathcal{B} \xrightarrow{\tau} (\nu b)P,\mathcal{B}[b \mapsto (n,[\ ])]$$

Table 1. Transition Rules of π_b

These rules are compatible with the transition rules for the π-calculus. IU and OU are rules for unbuffered names and synchronous communication is specified by Com. IB and OB define the asynchronous communication along buffered names: $b(x).P$ performs a τ action by receiving the "oldest" name d from b's buffer, while $\overline{b}\langle d\rangle.P$ performs a τ action by inserting d into b's buffer. Communication along buffered names is asynchronous because it involves two transitions (IB and OB) and other actions may occur between them.

IBG and OBG indicate that a buffer store itself may have actions. If b is a global buffered name, that is (νb) does not occur in P, then we can insert names to or receive names from b's buffer directly. In New and Open, the substitutions on the buffer store are for the sake of validity. NewB is the rule for the extended new process. After creating an empty buffer for b, the capacity parameter n is dropped, leaving the new operator indicating that b is a local name.

The Par rule describes how processes can progress asynchronously, which typically happens with buffered names. However, unlike in the π-calculus, where we have open/close rules to manage name scope extension, in the π_b-calculus, it is hard (perhaps impossible) to define an appropriate close rule because when a local name is exported to a buffer, it becomes hard to track which process will retrieve the name so as to determine the name scope. For instance, consider the process $P_1|P_2|P_3$ where $P_1 = (\nu a)\overline{b}\langle a\rangle.P_1'$, $P_2 = b(y).\cdots$, $P_3 = b(z).\cdots$ and a valid buffer store $\mathcal{B} = [b \mapsto (2,[\ ])]$. In the π_b-calculus, P_1 inserts the local a into b's buffer by a τ action, then it can possibly be received by P_2 or P_3, hence tracking the scope of a becomes very hard. Our solution here is to prevent processes from inserting local names into buffers when they are running in parallel with other processes. For processes like the above example, we extend the scope of a to the entire process by structural congruence laws and obtain a process in the form $(\nu a)(\overline{b}\langle a\rangle.P'_1|P_2|P_3)$ thanks to the name convention. This avoids the scope problem.

We have adopted the name convention which simplifies the definition of the labeled transition system. Dealing with names with buffers is subtle and the transition rules without the name convention are presented in [5].

The following proposition says that transition rules preserve buffer validity:

Proposition 1. *If $\mathcal{B}$ is valid for process P and we have the transition $P, \mathcal{B} \xrightarrow{\alpha} P', \mathcal{B}'$, then $\mathcal{B}'$ is valid for P'.*

As in the π-calculus, strong bisimulation over the set of π_b processes can be defined as follows.

Definition 2. *A symmetric binary relation $\mathcal{R}$ over π_b processes is a* bisimulation, *if whenever $(P, \mathcal{B}_P)\mathcal{R}(Q, \mathcal{B}_Q)$ and $(P, \mathcal{B}_P) \xrightarrow{\alpha} (P', \mathcal{B}'_P)$,*

$$\exists(Q', \mathcal{B}'_Q)\ .\ (Q, \mathcal{B}_Q) \xrightarrow{\alpha} (Q', \mathcal{B}'_Q) \wedge (P', \mathcal{B}'_P)\mathcal{R}(Q', \mathcal{B}'_Q)$$

Strong bisimilarity $\dot{\sim}$ is the largest strong bisimulation over the set of π_b processes. $(P, \mathcal{B}_P)$ and $(Q, \mathcal{B}_Q)$ are strongly bisimilar, written as $(P, \mathcal{B}_P) \dot{\sim} (Q, \mathcal{B}_Q)$, if they are related by some strong bisimulation.

2.2 Encoding in the Polyadic π-Calculus

We demonstrate an encoding of the π_b-calculus in the polyadic π-calculus.

Intuitively, a π_b name c is encoded into a pair of π names (c_1, c_2) by the injective *name translation function* N. In the name pair, c_1 is called the *input name* and c_2 the *output name* of c. In addition, input and output names for unbuffered names are identical, but not for buffered names. The two translation names of buffered name b are exactly the names along which a buffer process modelling the buffer of b receives and sends values.

The *translation function* $[\![\cdot]\!]$ takes a π_b process and a valid buffer store as parameters and returns a single π process. The encoding of a buffer store is a composition of buffer processes each representing a buffered name's buffer. For processes, the encoding differs from the original process in the new operators and prefixes. A new operator is encoded into two new operators localizing the pair of translation names. The encoding of input prefix $c(x)$ is also an input prefix but the subject is c's input name c_1, while the encoding of output prefix $\overline{c}\langle d\rangle$ has the output name c_2 as the subject. Finally, in the encoding of an extended new process $(\nu b : n)P$, a buffer process representing b's buffer is added.

The formal definition of the translation, including the buffer process and the translation function, are presented in the companion technical report.

The following lemma shows that transitions of a π_b process can be simulated by its encoding, and no more transition is introduced by the encoding.

Lemma 3. $(P, \mathcal{B}) \xrightarrow{\alpha} (P', \mathcal{B}')$ *if and only if* $[\![P, \mathcal{B}]\!] \xrightarrow{M(\alpha)} [\![P', \mathcal{B}']\!]$.

where M is a bijection relating actions of π_b-calculus to actions of π-calculus. It follows that the encoding preserves strong bisimulation.

Theorem 4. *$(P, \mathcal{B}_P) \dot{\sim} (Q, \mathcal{B}_Q)$ if and only if $[\![P, \mathcal{B}_P]\!] \dot{\sim} [\![Q, \mathcal{B}_Q]\!]$.*

3 The Go Programming Language

The Go programming language is a general purpose language developed by Google to support easy and rapid development of large distributed systems. This section presents a formal operational semantics of the (core) Go language and a fully abstract encoding in the π_b-calculus.

The syntax of a core of Go is presented as follows:

$$\begin{array}{lrl}
\text{Types :} & \mathtt{t} ::= & \mathtt{int} \mid \mathtt{chan}\ \mathtt{t} \\
\text{Expressions :} & e, e_1, e_2, \ldots ::= & x \mid n \mid ch \mid \mathtt{make}(\mathtt{chan}\ \mathtt{t}, n) \mid \texttt{<-}e \\
\text{Statements :} & s, s_1, s_2, \ldots ::= & \mathtt{nil} \mid x = e \mid e_1\texttt{<-}e_2 \mid s_1; s_2 \mid \mathtt{go}\ f(e_1 \ldots e_n) \\
 & & \mid \mathtt{select}\ \{c_1 \ldots c_n\} \\
\text{where} & c_1, c_2, \ldots ::= & \mathtt{case}\ x = \texttt{<-}e : s \mid \mathtt{case}\ e_1\texttt{<-}e_2 : s
\end{array}$$

The *channel type*, coupled with the concept called *Go-routine*, constitutes the core of Go's concurrency system. Channel types are of the form `chan t`, where `t` is called the *element type*. Channels (ch) are first-class values of this language, and they are created by the make expression $\mathtt{make}(\mathtt{chan}\ \mathtt{t}, n)$, where `chan t` specifies the channel type and the integer n specifies the size of the channel buffer. Notice that n must be non-negative and if it is zero, the created channel will be a synchronous channel.

Go-routines are similar to OS threads but much cheaper. A Go-routine is launched by the statement $\mathtt{go}\ f(v_1 \ldots v_n)$. The function body of f will be executed in parallel with the program that executes the `go` statement. When the function completes, this Go-routine terminates and its return value is discarded.

Communication among Go-routines is achieved by sending and receiving operations on channels. Sending statement $ch\texttt{<-}v$ sends v to channel ch, while receiving $\texttt{<-}ch$, regarded as an expression in Go, receives a value from ch. Communication via unbuffered channels are synchronous. Buffered (non-zero sized) channels enable asynchronous communication. Sending a value to a buffered channel can proceed as long as its buffer is not full and receiving from a buffered channel can proceed as long as its buffer is not empty.

`select` statements introduce non-deterministic choice, but their clauses refer to only communication operations. A `select` statement randomly selects a clause whose communication is "ready" (able to proceed), completes the selected communication, then proceeds with the corresponding clause statement.

Without loss of generality, we stipulate that a Go program is a set of function declarations, each of the form $\mathtt{func}\ f(x_1 \ldots x_n)\ \{s\}$. A Go program must specify a main function, which we shall refer to as f_{start} in the sequel, as the entry point — running a Go program is equivalent to executing $\mathtt{go}\ f_{start}(\ldots)$ with appropriate arguments. For the sake of simplicity, we only consider function calls in go statements and we assume that all functions do not return values and their bodies contain no local variables other than function arguments.

3.1 Operational Semantics

The structural operational semantics of Go is defined by a two-level labelled transition system: the local transition system specifies the execution of a single Go-routine in isolation, and the global transition system describes the behavior of a running Go program.

We first define the evaluation of expressions. An expression configuration is a triple $\langle e, \sigma, \delta_c \rangle$, where e is the expression to be evaluated, σ is the *local store* mapping local variables to values, and δ_c is the *channel store* mapping channels to triples (n, l, g), where n is the capacity of the channel's buffer, l is a list of values in the channel buffer, and g is a tag indicating whether the channel is local (0) or global (1). The transition rules between expression configurations $\overset{\alpha}{\mapsto}_g$ are defined as follows, where actions can be either silent action τ, or $\mathtt{r}(ch, v)$ denoting receive action. We often omit τ from silent transitions.

$$\textsc{Var}\ \langle x, \sigma, \delta_c \rangle \mapsto_g \langle \sigma(x), \sigma, \delta_c \rangle \qquad \texttt{RvE}\ \frac{\langle e, \sigma, \delta_c \rangle \overset{\alpha}{\mapsto}_g \langle e', \sigma, \delta_c' \rangle}{\langle \texttt{<-}e, \sigma, \delta_c \rangle \overset{\alpha}{\mapsto}_g \langle \texttt{<-}e', \sigma, \delta_c' \rangle}$$

$$\texttt{RvU}\ \frac{\delta_c(ch) = (0, [\,], g)}{\langle \texttt{<-}ch, \sigma, \delta_c \rangle \xmapsto{\mathtt{r}(ch,v)}_g \langle v, \sigma, \delta_c \rangle} \qquad \texttt{RvB}\ \frac{\delta_c(ch) = (n, [v] :: l, g);\ n > 0}{\langle \texttt{<-}ch, \sigma, \delta_c \rangle \mapsto_g \langle v, \sigma, \delta_c[ch \mapsto (n, l, g)] \rangle}$$

$$\textsc{Mak}\ \frac{ch \notin \mathtt{dom}(\delta_c)}{\langle \mathtt{make}(\mathtt{chan}\ \mathtt{t}, n), \sigma, \delta_c \rangle \mapsto_g \langle ch, \sigma, \delta_c[ch \mapsto (n, [\,], 0)] \rangle}$$

$\textsc{Var}$ retrieves the value of x from local store σ. $\textsc{Mak}$ creates a fresh local channel ch. Other rules concern receiving from channels. Once the channel expression is fully evaluated, the real receive begins following rules RvU and RvB. The value received from an unbuffered channel is indicated in the label, while the value received from a buffered channel is the "oldest" value of the channel's buffer.

The local transition system defines transition rules between local configurations. A local configuration is a tuple $\langle s, \sigma, \delta_c \rangle$, where s is the statement to be executed, σ is the *local store* and δ_c is the *channel store*. Each Go-routine has its own local store, but the channel store is shared by all Go-routines of a running program. Some of the local transition rules are presented as follows. Two additional actions can occur in local transition rules: $\mathtt{s}(ch, v)$ for message sending over channels and $\mathtt{g}(f, v_1 \dots v_n)$ for Go-routine creation.

$$\textsc{SdU}\ \frac{\delta_c(ch) = (0, [\,], g)}{\langle ch\texttt{<-}v, \sigma, \delta_c \rangle \xhookrightarrow{\mathtt{s}(ch,v)}_g \langle \mathtt{nil}, \sigma, \delta_c \rangle}$$

$$\textsc{SdB}\ \frac{\delta_c(ch) = (n, l, g);\ n > 0;\ |l| < n}{\langle ch\texttt{<-}v, \sigma, \delta_c \rangle \hookrightarrow_g \langle \mathtt{nil}, \sigma, \delta_c[ch \mapsto (n, l :: [v], g)] \rangle}$$

$$\textsc{Go}\ \langle \mathtt{go}\ f(v_1 \dots v_n), \sigma, \delta_c \rangle \xhookrightarrow{\mathtt{g}(f, v_1 \dots v_n)}_g \langle \mathtt{nil}, \sigma, \delta_c \rangle$$

Rules SdU and SdB capture the behavior of sending over unbuffered and buffered channels respectively. Sending a value v over an unbuffered channel ch carries a sending label $\mathtt{s}(ch, v)$, while sending over buffered channels is silent and can proceed as long as the target channel buffer is not full. The Go rule says that a go statement does nothing locally and can always proceed with a transition with the g label — the label is here simply for notifying the global configuration to generate corresponding Go-routines. Subexpression evaluation in Go is strict and leftmost.

Global transitions happen between global configurations which contain information of all running Go-routines. A global configuration, denoted by $\Lambda, \Lambda_1 \ldots$, is defined as a tuple $\langle \Gamma, \delta_c \rangle$, where Γ is a multi-set of statement/local store pairs (s, σ), of all running Go-routines, and δ_c is the channel store. A global transition takes the form $\delta_f \vdash \langle \Gamma_1, \delta_{c_1} \rangle \xrightarrow{\alpha}_g \langle \Gamma_2, \delta_{c_2} \rangle$, where δ_f is a mapping from function names to function definitions. A Go program will start from an initial configuration $\langle \{(s_{start}, \sigma_{start})\}, \delta_{init} \rangle$, where s_{start} is the body of the main function *start*, σ_{start} is the local store of *start*, and δ_{init} is the initial channel store. The global transition rules are listed in Table 2. A global action can be either τ, $\mathtt{r}(ch, v)$ or $\mathtt{s}(ch, v)$.

$$\mathtt{Loc}\ \dfrac{\langle s, \sigma, \delta_c \rangle \hookrightarrow_g \langle s', \sigma', \delta_c' \rangle}{\delta_f \vdash \langle \Gamma \cup \{(s, \sigma)\}, \delta_c \rangle \rightarrow_g \langle \Gamma \cup \{(s', \sigma')\}, \delta_c' \rangle}$$

$$\mathtt{Com}\ \dfrac{\langle s_1, \sigma_1, \delta_c \rangle \xhookrightarrow{\mathtt{r}(ch,v)}_g \langle s_1', \sigma_1, \delta_c \rangle;\ \langle s_2, \sigma_2, \delta_c \rangle \xhookrightarrow{\mathtt{s}(ch,v)}_g \langle s_2', \sigma_2, \delta_c \rangle}{\delta_f \vdash \langle \Gamma \cup \{(s_1, \sigma_1), (s_2, \sigma_2)\}, \delta_c \rangle \rightarrow_g \langle \Gamma \cup \{(s_1', \sigma_1), (s_2', \sigma_2)\}, \delta_c \rangle}$$

$$\mathtt{LGo}\ \dfrac{\langle s, \sigma, \delta_c \rangle \xhookrightarrow{\mathtt{g}(f, v_1 \ldots v_m)}_g \langle s', \sigma, \delta_c \rangle;\ \delta_f(f) = (\mathtt{func}\ f(x_1 \ldots x_m)\ \{s_f\})}{\delta_f \vdash \langle \Gamma \cup \{(s, \sigma)\}, \delta_c \rangle \rightarrow_g \langle \Gamma \cup \{(s', \sigma), (s_f, [x_1 \mapsto v_1 \ldots x_m \mapsto v_m])\}, \delta_c \rangle}$$

$$\mathtt{GRU}\ \dfrac{\langle s, \sigma, \delta_c \rangle \xhookrightarrow{\mathtt{r}(ch,v)}_g \langle s', \sigma, \delta_c \rangle;\ \delta_c(ch) = (0, [\,], 1)}{\delta_f \vdash \langle \Gamma \cup \{(s, \sigma)\}, \delta_c \rangle \xrightarrow{\mathtt{r}(ch,v)}_g \langle \Gamma \cup \{(s', \sigma)\}, \delta_c \rangle}$$

$$\mathtt{GRB}\ \dfrac{\delta_c(ch) = (n, l, 1);\ n > 0;\ |l| < n}{\delta_f \vdash \langle \Gamma, \delta_c \rangle \xrightarrow{\mathtt{r}(ch,v)}_g \langle \Gamma, \delta_c[ch \mapsto (n, l :: [v], 1)] \rangle}$$

$$\mathtt{GSU1}\ \dfrac{\langle s, \sigma, \delta_c \rangle \xhookrightarrow{\mathtt{s}(ch,v)}_g \langle s', \sigma, \delta_c \rangle;\ \delta_c(ch) = (0, [\,], 1);\ v \notin \mathtt{dom}(\delta_c)}{\delta_f \vdash \langle \Gamma \cup \{(s, \sigma)\}, \delta_c \rangle \xrightarrow{\mathtt{s}(ch,v)}_g \langle \Gamma \cup \{(s', \sigma)\}, \delta_c \rangle}$$

$$\mathtt{GSU2}\ \dfrac{\langle s, \sigma, \delta_c \rangle \xhookrightarrow{\mathtt{s}(ch,ch')}_g \langle s', \sigma, \delta_c \rangle;\ \delta_c(ch) = (0, [\,], 1);\ \delta_c(ch') = (n', l', g')}{\delta_f \vdash \langle \Gamma \cup \{(s, \sigma)\}, \delta_c \rangle \xrightarrow{\mathtt{s}(ch,\nu ch')}_g \langle \Gamma \cup \{(s', \sigma)\}, \delta_c[ch' \mapsto (n', l', 1)] \rangle}$$

$$\mathtt{GSB1}\ \dfrac{\delta_c(ch) = (n, [v] :: l, 1);\ n > 0;\ v \notin \mathtt{dom}(\delta_c)}{\delta_f \vdash \langle \Gamma, \delta_c \rangle \xrightarrow{\mathtt{s}(ch,v)}_g \langle \Gamma, \delta_c[ch \mapsto (n, l, 1)] \rangle}$$

$$\mathtt{GSB2}\ \dfrac{\delta_c(ch) = (n, [ch'] :: l, 1);\ n > 0;\ \delta_c(ch') = (n', l', g')}{\delta_f \vdash \langle \Gamma, \delta_c \rangle \xrightarrow{\mathtt{s}(ch,\nu v)}_g \langle \Gamma, \delta_c[ch \mapsto (n, l, 1), ch' \mapsto (n', l', 1)] \rangle}$$

Table 2. Global Transition Rules

`Loc` specifies the independent transition of a single Go-routine. Asynchronous communication will also take this transition since `RvB` and `SdB` are both silent transitions. `LGo` creates a new Go-routine. `Com` defines the synchronous communication between two Go-routines over unbuffered channels. The rules `Loc`, `LGo` and `Com` all specify internal actions of a running program.

A Go program can communicate with the environment via global channels. `GRU`, `GSU1` and `GSU2` describe how a Go program interact with the environment via unbuffered channels, and `GRB`, `GSB1` and `GSB2` describe interactions via buffered channels. Because communication over buffered channels are asynchronous, the labels in `GRB`, `GSB1` and `GSB2` indicate how a global channel interacts with the environment. For instance, in `GRB` the label $\texttt{r}(ch, v)$ means that the channel (buffer) ch receives a value v from the environment. The two rules `GSU2` and `GSB2` also describe how a local channel is exposed to the environment and becomes a global channel, by communication upon global channels. The ν in the label is required only when the value is a local channel ($g' = 0$).

Let $t = \alpha_1 \dots \alpha_n$ where each α_i is a global action, we write $\hat{t}$ for the action sequence obtained by eliminating all the occurrences of τ in t. We write $P, \mathcal{B} \xrightarrow{t}_g P', \mathcal{B}'$ if $P, \mathcal{B} \xrightarrow{\alpha_1}_g \cdots \xrightarrow{\alpha_n}_g P', \mathcal{B}'$, and $P, \mathcal{B} \overset{t}{\Rightarrow}_g P', \mathcal{B}'$ if $P, \mathcal{B} \Rightarrow_g \xrightarrow{\alpha_1}_g \Rightarrow_g \cdots \Rightarrow_g \xrightarrow{\alpha_n}_g \Rightarrow_g P', \mathcal{B}'$, where $\Rightarrow_g$ is the reflexive and transitive closure of $\xrightarrow{\tau}_g$.

Definition 5. *A symmetric binary relation $\mathcal{R}$ over global configurations is a* (weak) bisimulation *if $\Lambda_1 \mathcal{R} \Lambda_2$ and $\Lambda_1 \xrightarrow{\alpha}_g \Lambda_1'$ implies $\exists \Lambda_2' \,.\, \Lambda_2 \overset{\hat{\alpha}}{\Rightarrow}_g \Lambda_2' \wedge \Lambda_1' \mathcal{R} \Lambda_2'$. Two global configurations are bisimilar, written as $\Lambda_1 \approx_g \Lambda_2$, if they are related by some bisimulation.*

Two Go programs gp_1, gp_2 are bisimilar, if their initial global configurations (with the same δ_c) are bisimilar.

3.2 Encoding

The encoding of Go in the π_b-calculus is achieved by the translation function $[\![\cdot]\!]_g(r)$, which maps Go expressions and statements to π_b processes. The parameter r is the name along which the result of an expression is returned or the termination of a statement is signaled. Some of the encodings are as follows.

Make $[\![\texttt{make(chan}\ \tau, 0)]\!]_g(r) = \underline{\tau}.(\nu a)\overline{r}\langle a\rangle \quad [\![\texttt{make(chan}\ \tau, n)]\!]_g(r) = \underline{(\nu b : n)}\overline{r}\langle b\rangle$

Recv $[\![\texttt{<-}e]\!]_g(r) = (\nu r')([\![e]\!]_g(r') | r'(y).\underline{y(z)}.\overline{r}\langle z\rangle)$

Send $[\![e_1\texttt{<-}e_2]\!]_g(r) = (\nu r')(LR(e_1, e_2, r') | r'(y, z).\underline{\overline{y}\langle z\rangle}.\overline{r})$

Go $[\![\texttt{go}\ f(e_1 \dots e_n)]\!]_g(r) = (\nu r')(LR(e_1 \dots e_n, r') | r'(y_1 \dots y_n).\underline{\overline{f}\langle y_1 \dots y_n\rangle}.\overline{r})$

In the encoding, we use synchronous communication via local names to arrange the evolution order of π_b processes. For instance, in Recv, the right hand side of the composition will not proceed unless the left hand side outputs along local name r'.

MAKE returns the local name denoting the newly created channel. A receive operation corresponds to an input prefix in RECV, while a send operation corresponds to an output prefix in SEND. Auxiliary process LR captures the left-to-right evaluation of a sequence of expressions. For the go statement, after evaluating the argument expressions, these arguments are sent to the function to which f refers. The statement does not wait for the function, rather it outputs the termination signal along r immediately.

In the encoding, some prefixes and extended new operators are underlined. They are the most significant part and will be discussed later. The translation function can be extended to a mapping from global configurations (with δ_f) to π_b processes. We write $[\![\Lambda]\!]_g$ for the pair $(P, \mathcal{B})$, where P is the encoding of Λ and δ_f, while $\mathcal{B}$ is a valid buffer store inferred from channel store δ_c.

3.3 Correctness

The correctness of the encoding is demonstrated by a full abstraction theorem with respect to (weak) bisimulation. The following lemma says that a global transition may be simulated by a nontrivial sequence of transitions of its encoding. Usually, the encoding will perform some internal adjustments before and after the real simulation.

Lemma 6. *If $\Lambda \xrightarrow{\alpha}_g \Lambda'$, then $[\![\Lambda]\!]_g \Rightarrow\xrightarrow{M(\alpha)}\Rightarrow [\![\Lambda']\!]_g$, where M is an bijection.*

The lemma is proved by induction on the depth of inference of the premise in the local transition system. Conversely, a sequence of transitions of $[\![\Lambda]\!]_g$ should reflect certain global transitions of Λ. However it is not always possible, since the simulation may not yet complete, even worse the transition sequence simulating one global transition may interleave with transition sequences simulating others. Fortunately, by observing the proof of the previous lemma, we find that actually only one transition in the sequence plays the crucial role, as this transition uniquely identifies a global transition. Other τ transitions, whether preceding or following this special transition, are internal adjustments which prepare for the special transition immediately after them. We call the special transition a *simulating transition*, and the other non-special τ transitions *preparing transitions.*

Definition 7. *A transition $P, B \xrightarrow{\alpha} P', B'$ is a simulating transition if the action α is induced by the underlined prefixes and extended new operators specified in the encoding in Section 3.2. Otherwise, it is a preparing transition.*

Definition 8. *Let Λ be a global configuration, the set $\mathcal{T}_\Lambda$ is defined as follows:*

1. *$[\![\Lambda]\!]_g \in \mathcal{T}_\Lambda$.*
2. *$(P, \mathcal{B}) \in \mathcal{T}_\Lambda$ and $(P, \mathcal{B}) \to (P', \mathcal{B})$ is a preparing transition, then $(P', \mathcal{B}) \in \mathcal{T}_\Lambda$.*
3. *$(P, \mathcal{B}) \in \mathcal{T}_\Lambda$ and $(P', \mathcal{B}) \to (P, \mathcal{B})$ is a preparing transition, then $(P', \mathcal{B}) \in \mathcal{T}_\Lambda$.*

Any of the processes in $\mathcal{T}_\Lambda$ can be seen as the encoding of Λ.

Lemma 9. *If $(P, B) \in \mathcal{T}_\Lambda$ and $(Q, B) \in \mathcal{T}_\Lambda$, then we have $(P, B) \approx (Q, B)$.*

As a consequence, bisimulation is preserved by the encoding.

Theorem 10. *$\Lambda_1 \approx_g \Lambda_2$ if and only if $[\![\Lambda_1]\!]_g \approx [\![\Lambda_2]\!]_g$.*

4 Conclusion

We have presented the π_b-calculus which extends the π-calculus by buffered names. Native asynchronous communication is achieved via buffered names. We have provided a fully abstract encoding of an imperative concurrent language in the π_b-calculus with respect to weak bisimulation. A fully abstract translation from Core Erlang to the π_b-calculus can also be obtained. Since Erlang processes are communicated via Erlang mailboxes, the main difference of the encoding of Erlang from that of Go is the explicit modelling of Erlang mailboxes by sequences of buffered names. The details are relegated to the technical report [5].

Acknowledgement. The authors would like to thank Hao Huang for interesting discussion on Erlang.

References

1. Armstrong, J.L.: The Development of Erlang. In: ICFP. pp. 196–203 (1997)
2. Beauxis, R., Palamidessi, C., Valencia, F.D.: On the Asynchronous Nature of the Asynchronous π-Calculus. In: Concurrency, Graphs and Models. pp. 473–492 (2008)
3. Boudol, G.: Asynchrony and the π-calculus. Rapport de recherche RR-1702, INRIA (1992), `http://hal.inria.fr/inria-00076939`
4. Carlsson, R.: An introduction to Core Erlang. In: PLI'01 Erlang Workshop (2001)
5. Deng, X., Zhang, Y., Deng, Y., Zhong, F.: The Buffered π-Calculus: A Model for Concurrent Languages (Full version) (2012), available at `http://arxiv.org/abs/1212.6183`
6. Google Inc.: The Go Programming Language Specification (2012), `http://golang.org/ref/spec`
7. Hoare, C.A.R.: Communicating Sequential Processes. Communications of the ACM 21(8), 666–677 (1978)
8. Honda, K., Tokoro, M.: An Object Calculus for Asynchronous Communication. In: ECOOP. pp. 133–147 (1991)
9. Milner, R.: Communication and Concurrency. PHI Series in computer science, Prentice Hall (1989)
10. Milner, R.: The Polyadic π-Calculus: a tutorial. Tech. Rep. ECS-LFCS-91-180, Computer Science Dept., University of Edinburgh (1991)
11. Milner, R., Parrow, J., Walker, D.: A Calculus of Mobile Processes, Parts I and II. Information and Computation 100(1), 1–77 (1992)
12. Noll, T., Roy, C.K.: Modeling Erlang in the π-calculus. In: Erlang Workshop. pp. 72–77 (2005)
13. Roy, C.K., Noll, T., Roy, B., Cordy, J.R.: Towards Automatic Verification of Erlang Programs by π-Calculus Translation. In: Erlang Workshop. pp. 38–50 (2006)
14. Sangiorgi, D., Walker, D.: The π-Calculus: A Theory of Mobile Processes. Cambridge University Press (2001)
15. Walker, D.: Objects in the π-Calculus. Information and Computation 116(2), 253–271 (1995)