An interpretation of Erlang into value-passing Calculus

Citation for published version (APA):

DOI:
10.4304/jnw.8.7.1504-151

Document status and date:
Published: 01/01/2013

Document Version:
Publisher’s PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:

• A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher’s website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the “Taverne” license above, please follow below link for the End User Agreement:
www.tue.nl/taverne

Take down policy
If you believe that this document breaches copyright please contact us at:
openaccess@tue.nl
providing details and we will investigate your claim.

Download date: 26. Jul. 2019
An Interpretation of Erlang into Value-passing Calculus

Hao Huang\(^1\) and Fei Yang\(^1\)

\(^1\) Department of Computer Science
Shanghai Jiaotong University

MOE-MS Key Laboratory for Intelligent Computing and Intelligent Systems
Email: mengxi@sjtu.edu.cn

Abstract—In this paper we present a new process calculus \( \Sigma \) and an interpretation for core Erlang. \( \Sigma \) is at least as expressive as \( \text{VPC} \) and it is more effective for verification with some built-in functions. The well-established symbolic bisimulation in value-passing calculus is now helpful for solving the infinite-state space problem of data values in equivalence checking and verification. Compared to the previous formalization work of modeling core Erlang in \( \pi \)-calculus, the new interpretation has the following main merits: some essential features for Erlang are implemented faithfully; and above all the soundness of the interpretation is proved with respect to late symbolic bisimulation.

Index Terms—verification, value-passing calculus, symbolic bisimulation

I. INTRODUCTION

The relation between process calculi and programming languages has been discussed extensively. Some important conceptions in programming language have been explained in \( \pi \)-calculus, such as: function [1], object [2], typed \( \lambda \)-calculus [3] etc. They all depend on the name exchange ability of \( \pi \)-calculus [4] to characterize concurrent programming languages. However, it is difficult to use the semantics and equivalence relations [5] in \( \pi \)-calculus to guide the implementation of call-by-value languages. One of the main reasons is that they pay little attention to the infinite values, which are the actual data exchanged between processes [6].

The practical applications of process calculi are mainly in the design (e.g. [7]) and verification (e.g. [8], [9]) of concurrent programming languages. Semantics of programming languages into process calculi [8], [2] are mostly focused in \( \pi \)-calculus (not in CCS [10]). In our opinion, the infinite values should be considered firstly no matter for the design of a programming language or for the formalization of a model-checking method. In \( \pi \)-calculus, the main approaches to handle infinite values contain the following:

- Make use of a fixed name [8] or infinite names to characterize natural numbers [3]. This is not practical and cannot ensure the correctness.

- All constant values are encoded as processes [11], e.g. the integer “10000” would be encoded as \( \overline{10000} \). Clearly this is inefficient and impractical.

- The language Pict adds some built-in types, such as boolean, character, string, integer, to the syntax of \( \pi \)-calculus [7]. It does not allow the communication of channel constants except channel variables. The syntax of Pict is more like a combination of \( \pi \)-calculus and \( \text{VPC} \).

If we want to improve the state-of-the-art of programs verification by using extensions of the \( \pi \)-calculus with integers, tuples and others, the existing equivalence relations in \( \pi \)-calculus cannot be used, like strong, weak bisimulation [10] and so on. An equivalence relation that is a criteria to judge the correctness of formalization needs to be redefined. This equivalence relation may involve type restrictions and infinite-state spaces problem [5] for verification. Comparatively, many proof tools are for basic calculus with primitive actions and do not consider data and types, for example the Concurrency Workbench [12] and the Mobility Workbench for \( \pi \)-calculus [13]. However the symbolic bisimulation [5] in value-passing calculus can be utilized to handle the infinite-state spaces problem in equivalence checking. And we need not consider types because the exchange of channel names between processes is not allowed in values-passing calculus. A verification tool VPAM for value-passing processes based on symbolic bisimulation has been presented in [14].

Erlang is a concurrent functional programming language with high performance and always used to implement some complicated real-time concurrent systems, for example communication switches or routers and mobile base stations. The popular usage of Erlang in telecommunication systems requires the development of the verification and analysis for concurrent communication [15]. To improve the Erlang formalization work in [8], [9] and handle the infinite-state spaces problem by symbolic bisimulation, we propose a \( \text{VPC}_\Sigma \) as a variant of value-passing calculus in Section III. Some built-in functions that make \( \text{VPC}_\Sigma \) more effective for practical verification are added. In our paper the value-space of \( \text{VPC} \) presented by Milner in [10] is \( \mathbb{N} \), the set of natural numbers. The relation between Turing complete process calculi [16] is:
The high-order and single assignment feature [17] are for functions calls, for example a call of function: call \( m : f l (c o n s t _ { 1 } , . . . , c o n s t _ { n } ) \) can be evaluated only when there is an export function \( f l \) with arity \( n \) defined in module \( m \); pattern matching are used for case branching; asynchronous communication between processes are by receive expressions in concurrent environment; try expressions are for error handling.

More detailed specifications can be found in [19], [18].

\[
\text{module} ::= \text{module } A t o m \left[ \begin{array}{l} \text{fname}_1, \ldots, \text{fname}_n \end{array} \right] \\
\text{attributes} ::= \{ \begin{array}{l} A t o m = const_1, \ldots \end{array} \}
\]

\[
\text{fname} ::= \text{Atom} | \text{Integer} | \text{Float} | \text{Atom} | \text{Char} | \text{String} | \}
\]

\[
\text{const} ::= \text{lit} | \{ \text{const}_1, \text{const}_2 \} | \{ \text{const}_1, \ldots, \text{const}_n \}
\]

\[
\text{lit} ::= \text{Integer} | \text{Float} | \text{Atom} | \text{Char} | \text{String} | \}
\]

\[
\text{fun} ::= \text{var}_1, \ldots, \text{var}_n \rightarrow \text{exprs}
\]

\[
\text{var} ::= \text{VariableName}
\]

\[
\text{vars} ::= \text{var} | < \text{var}_1, \ldots, \text{var}_n > 
\]

\[
\text{expr} ::= \text{var} | \text{fname} | \text{lit} | \text{fun} | \left[ \begin{array}{l} \text{exprs}_1 \left| \text{exprs}_2 \right. \end{array} \right] \\
\text{let} \text{exprs} ::= \text{fun}_1, \ldots, \text{in} \text{exprs}_2
\]

\[
\text{apply} \text{exprs} ::= \text{expr}_1, \ldots, \text{expr}_n \\}
\]

\[
\text{primop} \text{exprs}_1 ::= \text{exprs}_2, \ldots, \text{exprs}_n \}
\]

\[
\text{try} \text{exprs} ::= \text{catch} \left[ \begin{array}{l} \text{var}_1, \ldots, \text{var}_2 \rightarrow \text{exprs} \end{array} \right] \\
\text{case} \text{exprs} ::= \text{clause}_1, \ldots, \text{clause}_n \\}
\]

\[
\text{receive} \text{clause}_i \ldots, \text{clause}_n \rightarrow \text{exprs} \}
\]

\[
\text{exprs} ::= \text{expr} | < \text{expr}_1, \ldots, \text{expr}_n > \\
\text{clause} ::= \text{pat} \rightarrow \text{exprs} \\
\text{pat} ::= \text{pat} \rightarrow \text{pat}_1, \ldots, \text{pat}_n
\]

\[
\text{pat} ::= \text{var} | \text{lit} | \left[ \begin{array}{l} \text{pat}_1 | \text{pat}_2 \end{array} \right] | \left[ \begin{array}{l} \text{pat}_1, \ldots, \text{pat}_n \end{array} \right] \\
\text{pat} ::= \text{var} = \text{pat}
\]

\[
\text{Figure 2. Syntax of Core Erlang}
\]

II. SYNTAX OF CORE ERLANG

Core Erlang is an intermediate representation of Erlang. It has more regular structures and simpler semantics than Erlang. Core Erlang is designed to make program analysis [18] easier. The syntax of Core Erlang is given in Fig 2. The entities of of Core Erlang mainly include:

1. Modules: all functions must be in a module with some attributes. A complex algorithm can be easily implemented for its modularity.

2. Expressions: variables, functions (name or abstraction), literals, tuple and list are some basic expressions in Core Erlang. In addition, let expressions binds variables to values; do expressions are for sequencing executions; apply, call, primop expressions are for functions calls, for example a call of function: call \( m : f l (c o n s t _ { 1 } , . . . , c o n s t _ { n } ) \) can be evaluated only when there is an export function \( f l \) with arity \( n \) defined in module \( m \); pattern matching are used for case branching; asynchronous communication between processes are by receive expressions in concurrent environment; try expressions are for error handling.

More detailed specifications can be found in [19], [18].
$\mathcal{N}$ is a set of names ranged over by $a, b, c$. The set $\overline{\mathcal{N}}$ of co-names is $\{ \pi \mid a \in \mathcal{N} \}$. The union $\mathcal{N} \cup \overline{\mathcal{N}}$ is ranged over by $\zeta, \eta$. The set of all actions $\mathcal{A} = \{a(\bar{x}), \pi([\bar{x}]) \mid a \in \mathcal{N} \} \cup \{ \tau \}$ is ranged over by $\lambda, \Lambda$. A name cannot be passed around and is not a $\Sigma$-term. $\bar{x}$ is a sequence of distinct variables $x_1 \ldots x_n$ and $I$ is a sequence of $\Sigma$-terms $\ell_1 \ldots \ell_m$. $m, n$ are their length.

$\text{VPC}_\Sigma$-terms, ranged over by $S, T, \ldots$, are defined as follows.

$$T := 0 | \sum_{i \in I} [\varphi_i]a(\bar{x}).T_i | \sum_{i \in I} [\varphi_i][\pi(\bar{I})].T_i | T | T' | (c)T | [\varphi]T | D(\bar{x}_1, \ldots, \bar{x}_k)$$

where $\varphi$ is a conditional expression and $I$ is a finite indexing set. The input $a(\bar{x})$ binds variables $x_1 \ldots x_n$ in $T$. Otherwise a variable is free in $T$. A $\text{VPC}_\Sigma$-term is closed if it does not contain any free variables. The set of $\lambda, \Lambda$-variable $T$ by $\text{free}(T)$. The abbreviation of $[\varphi]\lambda \diamond 0$ is $[\varphi]\Lambda$. A parametric definition is given by the equation $D(\bar{x}_1, \ldots, \bar{x}_k) = T$. The $\bar{x}_1 \ldots \bar{x}_k$ are parameter variables. If neither $\bar{x}$ nor $I$ occurs in $T$ of $a(\bar{x}), T$ and $\pi(\bar{I})$, we take the abbreviation $a.T$ and $\pi.T$.

In this paper we require that $\text{free}(T) \subseteq \{ \bar{x}_1, \ldots, \bar{x}_k \}$. Conditional expressions, ranged over by $\varphi, \psi, \ldots$, are defined as follows.

$$\varphi ::= T | \bot | \neg \psi | m \neq n | \varphi \lor \psi | \varphi \lor \psi$$

where $m = n$ and $m \neq n$ stand for match and mismatch respectively. $\bot$ and $\top$ are respectively logical true and false.

The relation $< \mathrel{\triangleq}$ natural numbers can be defined by a finite set of matches between numerals $m < n \iff (0 = m) \lor (1 = m) \lor \ldots \lor (n - 1 = m)$.

The symmetric semantics of $\text{VPC}_\Sigma$ is defined in Fig. 3. The parallel operator is associative and commutative. The symmetric rules are omitted. The preconditions set $\text{BExp}$ includes some conditional expressions $\varphi, \psi, \ldots$. A symbolic action is a tuple $(\varphi, \lambda)$ ranged over by $\lambda^*, \epsilon^*$, and the $\lambda$ is executed if $\text{BExp} \vdash \varphi$. We do not take the asynchronous version [6], for a complete axiomatic system with an additional asynchronous theory can be used for asynchronous $\text{VPC}_\Sigma$ similarly in [20]. The sub-bisimilarity $\sqsubseteq$ [16] is used to compare the expressiveness between process calculi.

**Lemma 1**: $\text{VPC}_\Sigma \sqsubseteq \text{VPC}$ and $\text{VPC} \sqsubseteq \text{VPC}_\Sigma \Rightarrow \text{VPC}_\Sigma \equiv \text{VPC}$

**Proof**: The sub-bisimilarity $\sqsubseteq$ and similar proof could be found in [16].

**IV. A $\text{VPC}_\Sigma$-INTERPRETATION FOR CORE ERLANG**

A challenge is that we cannot use the capability of name passing in $\pi$-calculus to formalize Core Erlang. The choice operator $\sum$ is excluded [7], [21] because its implementation is unnecessary and expensive [22].

A program called shell with prompt $>$ is used for evaluating expressions in Erlang. The interpretation denotes this by $[\cdot]_{pen} : \Sigma^* \rightarrow T$. We will use some infrastructure services that can actually be obtained by a process. $S = \{ S_1, S_2, \ldots \}$ is the set of infrastructure services. Each $S_i$ in $S$ is a process that maintains some string lists $\ell_1, \ell_2, \ldots$ as above. The common elementary operations of the services in $S$ are: querying, allocation and release. A service $S_i$ can have some other specific operations. For example the computing services can do a set of basic arithmetic and shift operations for evaluation. The encoding uses following basic operations for list $\ell = [1, 2, \ldots, n]$.

$$\text{SEARCH}(\ell, 0) = j \quad \text{INSCR}(\ell, m, i) = [1, 1, \ldots, 1; \underbrace{m, i, \ldots, m}_m]$$

$$\text{ADD}(\ell, m) = [1, 1, \ldots, 1, \ldots, m]$$

$$\text{DEL}(\ell, i) = [1, 1, \ldots, 1, \ldots, 1]$$

where $j$ is the first position which I occur in $\ell$ and $0 \leq i \leq \text{size}(\ell) = C(\ell, i) + 1$.

The computing and storage services are denoted by $S_c(\ell, i)$ and $S_e(\ell, i)$ respectively where $\ell_c = [\bar{k}_1, \ldots, k_m]$. Each $\ell_c$ is a triple $[\text{index}, \text{addr}, \text{corr}]$. $\ell_c$ is similar.

The index is the keyword for searching. We assume that $\text{index}_{x_i} \neq \text{index}_{x_j}$ where $1 \leq i \leq m, 1 \leq j \leq m, i \neq j$ in the list. The $\text{addr}$ is the position in the list. The $\text{corr}$ are relevant terms, for example assigned values and function bodies.

$$S_c(\ell, i) = qu_{c}(x) \{ [-\varphi_1]_{\text{de}_{ald}(x, \text{unallocated})}.S_c(\ell, i) \mid [\varphi_1]_{\text{de}_{ald}(x, \text{corr}_{x_i}).S_c(\ell, i)} \}$$

$$\text{in}_{c}(x, y) \{ [-\varphi_1]_{\text{de}_{ald}(x, \text{unallocated})}.S_c(\ell, i) \mid [\varphi_1]_{\text{in}_{c}(x, \text{updated})} \}$$

$$\text{ADD}(\ell, m) \{ [\varphi_1]_{\text{de}_{ald}(x, \text{released})}.S_c(\ell, i) \mid [\varphi_1]_{\text{ADD}(\ell, m)} \}$$

$$\text{DEL}(\ell, i) \{ [\varphi_1]_{\text{de}_{ald}(x, \text{released})}.S_c(\ell, i) \mid [\varphi_1]_{\text{DEL}(\ell, i)} \}$$

$$\text{AL}(\ell, k) \{ [\varphi_1]_{\text{size}(\ell) < \text{MAX}_e} \}$$

$$\text{ADD}(\ell, m) \{ [\varphi_1]_{\text{size}(\ell) < \text{MAX}_e} \}$$

$$\text{DEL}(\ell, i) \{ [\varphi_1]_{\text{size}(\ell) < \text{MAX}_e} \}$$

$$\text{qu}_{c}(x) \{ \text{new} \}.$$
some position information in distributed environments. The input \( \mathbf{x}^c \) for the evaluation of computing services should be discoverable without any function calls or applications and the \( \forall \mathcal{ACL} \forall \mathcal{E} \mathbf{x}^c \) is the evaluation result of \( \mathbf{x}^c \). Based on the set of infrastructure services, additional services such as the definitions for modules and functions, the encoding is defined as follows:

\[
\mathcal{S} \mathcal{M} \mathcal{I}(\mathcal{P} \mathcal{O}) = [\Rightarrow \mathcal{P} \mathcal{O}] \cup \mathcal{S} \mathcal{L}(\mathcal{P} \mathcal{O}) \cup \mathcal{S} \mathcal{M}(\mathcal{P} \mathcal{O})
\]

An exported function declaration \( \mathbf{f} = \mathbf{fun} (\mathbf{var}_1, \ldots, \mathbf{var}_n) \Rightarrow \mathbf{es} \) within \( \mathcal{S} \mathcal{M} \mathcal{I} \) can be an additional service. We let it range over \( \mathcal{S} \mathcal{M} \mathcal{D} : \mathcal{f} \).

A module declaration is a set of additional services defined by exported functions and local functions. We let it range over \( \mathcal{S} \mathcal{M} \mathcal{D} = \{ \mathcal{S} \mathcal{M} \mathcal{D}_1, \ldots, \mathcal{S} \mathcal{M} \mathcal{D}_n \} \), \( \{ \mathcal{f}_1, \ldots, \mathcal{f}_n \} \), \( \{ \mathcal{g}_1, \ldots, \mathcal{g}_m \} \), \( \{ \mathcal{h}_1, \ldots, \mathcal{h}_l \} \). An element in the list for a module records some attributes and local functions.

The evaluation would directly output itself through the link \( \mathcal{d} \mathcal{e} \mathcal{v}_\mathcal{C} \). An abstraction is a value and a normal form [23] in call-by-value lambda calculus. The following is a formation rule for \( \mathcal{S} \mathcal{M} \mathcal{D} \) in [23].

A term is bound if all free variables have been assigned to a closed value. Variables in Erlang have the property of single assignment: an assignment to a variable can only be made once [17]. Destructive assignment may happen in the implementation given by Thomas Noll [8].

The higher-order property can be achieved through evaluating a \( \text{fname} \) or \( \text{fun} \) to an abstraction. To simplify some notations, an evaluation macro \( \text{em}(T, S) \) is as follows.

We mainly use \( S \) to record the evaluation result of \( T \).

The evaluation of the list or the tuple can terminate only if all elements are convergent.

The subsequent evaluation can be done only after the first evaluation in computing services has an output value.

The replacement of \( \text{fname} \) for a recursive function can be the \( \text{fname} \) itself. A divergence may be caused by infinite self replacements.

Function \( \mathcal{I} \mathcal{A} \mathcal{B} \mathcal{S}() \) that judges whether a string is an abstraction, \( \mathcal{V} \mathcal{A} \mathcal{R} \mathcal{B}() \) that returns parameters of an abstraction and \( \mathcal{B} \mathcal{D} \mathcal{Y} \mathcal{O} \mathcal{B}() \) that returns the body of an abstraction can be implemented by built-in functions in section 3. We omit the details of implementation.

Function \( \mathcal{A} \mathcal{R} \mathcal{B}() \) has been defined explicitly.

The case expressions use pattern matching in Erlang. The value of \( \text{es} \) is sequentially matched against \( \mathcal{p}_1, \ldots, \mathcal{p}_n \). Function \( \mathcal{SP} \mathcal{F}(\mathbf{y}_1, \mathcal{p}_1 \mathcal{p}_2 \mathcal{p}_3 \mathcal{p}_4) \) return 1 when \( \mathbf{y}_1 \) and \( \mathcal{p}_1 \mathcal{p}_2 \mathcal{p}_3 \mathcal{p}_4 \) have the same syntactic structure that means the elements at each position of their parser trees are both \( \text{constants} \) or a value assignment to variable. It can also be constructed by built-in functions in section 3.
\[ \langle Y_1, constp_j \rangle, \ldots, \langle Y_n, constp_{j_n} \rangle \] is the set of comparisons between constants, and \[ \{ \langle Y_1, varp_j \rangle, \ldots, \langle Y_n, varp_{j_n} \rangle \} \] is the set of variable bindings.

\[
\begin{align*}
\text{letrec} & \quad \text{\texttt{call('erlang')}} \quad \langle \text{\texttt{'spawn'(mod, f, [es_{i1}, \ldots, es_{in}] )}} \rangle \\
& \quad \text{\texttt{def}} \quad (\text{\texttt{(v_{i1}, \ldots, v_{in})}}) \\
& \quad \text{\texttt{let}} \quad \langle \text{\texttt{es_{i1}}}, \ldots, \text{\texttt{es_{in}}} \rangle \quad \text{\texttt{=} \quad \text{\texttt{es(fun'_i/fname_i, \ldots)}}} \\
& \quad \text{\texttt{where}} \quad f_i \quad \text{\texttt{=} \quad \text{\texttt{fun(var_{i1}, \ldots, var_{in}) \Rightarrow es_{i}, and for i \in \{ 1, \ldots, n \}, fun'_i \quad \text{\texttt{=} \quad \text{\texttt{fun(var_{i1}, \ldots, var_{in}) \rightarrow letrec fname_i = f_i \ldots es_{i}}}}}}
\end{align*}
\]

The local function \( f_i \) is maintained by the modified \texttt{fun} abstraction in the evaluation of letrec expressions.

\[
\begin{align*}
\text{[try \ es_1 \ catch \ (var_{i1}, varp_{i1}) \Rightarrow es_2 \def\quad (v_1)} \\
\text{\texttt{let}} \quad \langle \text{\texttt{es(fur'_1/fname_1, \ldots)}} \rangle \\
& \quad \text{\texttt{where}} \quad f_i \quad \text{\texttt{=} \quad \text{\texttt{fun(var_{i1}, \ldots, var_{in}) \rightarrow es_{i}, and for i \in \{ 1, \ldots, n \}, fun'_i \quad \text{\texttt{=} \quad \text{\texttt{fun(var_{i1}, \ldots, var_{in}) \rightarrow letrec fname_i = f_i \ldots es_{i}}}}}}
\end{align*}
\]

A call to \texttt{spawn} returns a new \texttt{pid} rather than the evaluation result of the function call. The process releases its \texttt{pid} and mailbox after its evaluation. The restricted name \texttt{self} is for BIF \texttt{self()} that returns the process identifier of the current calling process.

\[
\begin{align*}
\text{[call 'erlang' \quad 'spawn'(mod, f, [es_{i1}, \ldots, es_{in}] ) \quad \text{\texttt{def}} \quad (v_{i1}, \ldots, v_{in})} \\
& \quad \text{\texttt{let}} \quad \langle \text{\texttt{es(fur'_1/fname_1, \ldots)}} \rangle \\
& \quad \text{\texttt{where}} \quad f_i \quad \text{\texttt{=} \quad \text{\texttt{fun(var_{i1}, \ldots, var_{in}) \rightarrow es_{i}, and for i \in \{ 1, \ldots, n \}, fun'_i \quad \text{\texttt{=} \quad \text{\texttt{fun(var_{i1}, \ldots, var_{in}) \rightarrow letrec fname_i = f_i \ldots es_{i}}}}}}
\end{align*}
\]

Sending a message is an asynchronous operation.

\[
\begin{align*}
\text{[call 'erlang' \quad 'spawn'(mod, f, [es_{i1}, \ldots, es_{in}] ) \quad \text{\texttt{def}} \quad (v_{i1}, v_{i2})} \\
& \quad \text{\texttt{let}} \quad \langle \text{\texttt{es(fur'_1/fname_1, \ldots)}} \rangle \\
& \quad \text{\texttt{where}} \quad f_i \quad \text{\texttt{=} \quad \text{\texttt{fun(var_{i1}, \ldots, var_{in}) \rightarrow es_{i}, and for i \in \{ 1, \ldots, n \}, fun'_i \quad \text{\texttt{=} \quad \text{\texttt{fun(var_{i1}, \ldots, var_{in}) \rightarrow letrec fname_i = f_i \ldots es_{i}}}}}}
\end{align*}
\]
Definition 5.2: A transition system of Core Erlang \( \text{TS}_{ce} = (\text{Rs, EN, A, } X^c \rightarrow_{CE} R) \) with Rs a readiness, EN an environment, A a set of actions, \( X^c \rightarrow_{CE} \subseteq (\text{Rs, EN}) \times (\text{Rs, EN}) \) the transition relation under \( X^c \in A \) and R some rules of the form:

\[
\{ (i, i+1), EN \} \rightarrow_{CE} (\text{Rs}', EN') \quad i \in I_1
\]

where \( X^c, X^c_i \in A \), \( i \in I_1 \) and \( I_1 \) is an index set for transition relations. An action \( X^c \) is either an empty action \( \tau \) or a visible action. The elements of \( \{ (\text{Rs}, EN) \rightarrow_{CE} (\text{Rs}', EN') \mid i \in I_1 \} \) are called premises and \( (\text{Rs, EN}) \rightarrow_{CE} (\text{Rs}', EN') \) is the conclusion. If the premises set of a rule \( r \in R \) is \( \emptyset \), then \( r \) is an axiom. \( \Phi \) is a set of preconditions.

A. Internal semantics

An expression substitution is denoted by \( e' \{ e_1, e, \ldots , e_n \} \). We denote a substitution according to an assignment \( A \{ e \} \). A transformation of \( P \) or \( EN = (M \text{Defs, As, P}) \) caused only by expression changing from \( e \) to \( e' \) in a process \( P \) is denoted by \( P = (P - e) \cup \{ e', primop, pid, mb \} \) or \( EN = (M \text{Defs, As, P}) \).

The internal semantic rules are given in Fig. 4:

**Mod:**

\[
\text{Mod:} \quad (\text{module } \text{mod, end, pid, EN}) \rightarrow_{CE} (\text{en_mod})
\]

\[
(\text{ok, null, (MDefs, As, P - e}) \rightarrow_{CE} (\text{en_mod})
\]

\( m \) is a fresh atom of module name

**VarEval:**

\[
(\text{var, pid, EN}) \rightarrow_{CE} (\text{en_var})
\]

\[
(V^0, \text{null}, (\text{MDefs, As, P - e})) \rightarrow_{CE} (\text{en_var})
\]

**FnameEval:**

\[
(\text{fname, var, \ldots, EN}) \rightarrow_{CE} (\text{en fName})
\]

\[
(\text{func, var, \ldots, EN}) \rightarrow_{CE} (\text{en_func})
\]

**Seq:**

\[
(\text{do } e_1, e_2, \text{pid, EN}) \rightarrow_{CE} (\text{en_sequencing})
\]

\[
(\text{do } V^0, e_2, \text{pid, EN}) \rightarrow_{CE} (\text{en_sequencing})
\]

**Rec:**

\[
(\text{letrec } \text{fun}, \ldots, e, \text{pid, EN}) \rightarrow_{CE} (\text{en_rec})
\]

\[
(\text{letrec } \text{fun}, \ldots, e, \text{pid, EN}) \rightarrow_{CE} (\text{en_rec})
\]

**Call:**

\[
(\text{call } \text{fun}, \ldots, e, \text{pid, EN}) \rightarrow_{CE} (\text{en_call})
\]

**App:**

\[
(\text{apply fun, \ldots, e, EN}) \rightarrow_{CE} (\text{en_app})
\]

The sequential evaluations in list(or tuple) is similar to Sequencing. The method in [24] is not effective since the assignment to the pattern \( p \) could be infinite. The Match() based on a structural comparison \( SPF() \) is:

\[
\text{Match}(V^0, \text{pid}) = \text{As, if } SPF(V^0, \text{pid}) = 1 \land \ldots \land V^m \rightarrow_{CE} \text{const, otherwise}
\]

B. Communication semantics

We give the communication semantics of Core Erlang language in Fig. 5:

**Spawn:**

\[
(\text{call } \text{erlang}'! \text{'(pid, EN), V^0, \ldots, e, \text{pid, EN})} \rightarrow_{CE} (\text{en_spawn})
\]

**Send:**

\[
(\text{call } \text{erlang}'! \text{'(pid, EN), V^0, \ldots, e, \text{pid, EN})} \rightarrow_{CE} (\text{en_send})
\]

**MatchRec:**

\[
(\text{receive } \text{e}_1, \ldots, e_n \rightarrow V^0 \rightarrow e_2, \text{pid, EN}) \rightarrow_{CE} (\text{en_matchRec})
\]

**MismatchRec:**

\[
(\text{receive } \text{e}_1, \ldots, e_n \rightarrow V^0 \rightarrow e_2, \text{pid, EN}) \rightarrow_{CE} (\text{en_mismatchRec})
\]
VI. CORRECTNESS

We modify the definition of late symbolic bisimulations [5] slightly and prove the correctness of our interpretation. The $\lambda_1 \rightarrow_s ^\dagger \lambda_2 \rightarrow_s T'$ and $\lambda_1 \rightarrow_s T'$ is that $T_1$ cannot reach $T_2$, the same as $\lambda_1 \rightarrow_s \lambda_2 \rightarrow_s ^\dagger$.

Definition 6.1: A symmetric relation $\rho$ on $VPC_S$ is a late symbolic bisimulation and $S\rightarrow^\dagger T$ if whenever $S \rightarrow^\dagger T'$ and there is a collection of branching conditional expressions $Br = \{ \phi_i^\dagger | i \in I \}$ such that $\phi \land \phi_1 \Rightarrow \bigvee_{i \in I} \phi^\dagger_i$, $\phi \land \phi_1 \Rightarrow \bot$ (for all $i, j \in I, i \neq j$) and the following properties:

- If $\lambda_1 = \tau$ then $\lambda_2 = \tau$ and $S' \rightarrow^\dagger T'$.
- If $\lambda_1 = \sigma(l)$ then $\lambda_2 = \sigma(l')$ and $S' \rightarrow^\dagger T'$.
- If $\lambda_1 = \alpha(x)$ then $\lambda_2 = \alpha(y)$, and $S' \rightarrow^\dagger T'$ when $\vec{x} \neq \vec{y}$ and $S' \rightarrow^\dagger T'$.

The late symbolic bisimulation $\simeq^\dagger_{LS}$ is the largest late symbolic bisimulation for $VPC_S \Rightarrow$ means the logic implication relation.

Lemma 2 (strict lemma): For all closed programs $p_{ce}$ in Core Erlang, one of the following holds:

1) if $p_{ce}$ is a module definition $(\langle p_{ce}, pid_i \rangle, En)$ $\Rightarrow_{CE} (\langle ok, null \rangle, EN')$ then for some $t_0', t'_0$ such that:

$$\Rightarrow_{CE} (\langle ok, null \rangle, EN')$$

2) if $p_{ce}$ is an expression $c_{ce}$ such that $(\langle c_{ce}, pid_i \rangle, En)$ $\Rightarrow_{CE} (\langle V_0, null \rangle, EN')$ then for some $t_0', t'_0$ such that:

$$\Rightarrow_{CE} (\langle V_0, null \rangle, EN')$$

3) if $p_{ce}$ is an expression $c_{ce}$ with divergent evaluation such that $(\langle c_{ce}, pid_i \rangle, En) \Rightarrow_{CE} (\langle V_0, null \rangle, EN')$ then for any $t_0', t'_0$ such that:

$$\Rightarrow_{CE} (\langle V_0, null \rangle, EN')$$

Proof: The proof of case 1 is obtained directly from the interpretation. The proof of cases 2 and 3 are by induction on programs which reduce to a value. The infinite replacements of recursive function body would cause divergence.

Definition 6.2: A specific running system is $RS = \{RD_1, \ldots, RD_n, EN \}$ where $RD_i = (p_{ce}, pid_i)$ and $pid_i \neq null$. RS could evolve to $RS'$ by the following two rules:

Parallel:

$$\Rightarrow_{RS} (\langle p_{ce}, pid_i \rangle, En') \Rightarrow_{RS} (\langle p_{ce}', pid_i \rangle, En')$$

Strict:

$$\Rightarrow_{RS} (\langle p_{ce}, pid_i \rangle, En) \Rightarrow_{RS} (\langle p_{ce}', pid_i \rangle, En')$$

VII. CONCLUSION AND FUTURE WORK

In this paper we proposed an intermediate calculus $VPC_S$ to improve the state-of-the-art of practical languages verification by process calculi. And we give a new interpretation of Core Erlang. The community of the Erlang language can have the following benefits:

- thereafter model checking tools such as $\mu$-calculus [25] can be used to automatically verify some specifications of Erlang system that be described by a logic formulae and guide the implementation of specifications [26]. We need to embed the $VPC_S$ in $\mu$-calculus;
- the proven correspondence result can establish correctness properties of complex systems implemented in Erlang for industry’s testing. Some previous work in $\pi$-calculus [8] have not this property partly because the equivalence relations of $\pi$-calculus pay little attention to infinite values.

In the future we would like to extend our work further. Practically we should embed the $VPC_S$ in $\mu$-calculus that is a model checking tool of value-passing processes [25]. Theoretically, the interpretive ability of $\pi$-calculus and $VPC_S$ for programming languages could be examined.

ACKNOWLEDGMENTS

We appreciate Prof. Yuxi Fu for his advice and suggestions. Sandy Harris has helped in improving the paper’s quality. This work is supported by NSFC (Grant No. 61033002) and STCSM (Grant No. 11XD1402800).
REFERENCES


A. An Example

We give a simple program of Core Erlang to describe the interpretation. The call of function format in the module io provides standard input/output very simplified.

\[ \text{fun} \langle N, \text{Pong} \rangle \rightarrow \text{case} \ N \ of \]
\[0 \ \text{when} \ \text{true} \rightarrow \]
\[ \text{call} \ \text{Erlang} \ : \ \langle '!'(Pong, 'finished') \rangle \]
\[ \text{case} \ N \ of \]
\[X \ \text{when} \ \text{true} \rightarrow \]
\[ \text{receive} \ \text{pong} \ \text{when} \ \text{true} \rightarrow \]
\[ \text{call} \ \text{io} : \ \langle \text{format}(\langle \text{Ping} \ received \ pong \sim n \rangle, []) \rangle \]
\[ \text{apply} \ \langle \text{ping} \rangle /2 \langle X-1, \text{Pong} \rangle \]
\[ \langle \text{ping} \rangle /0 = \]
\[ \text{fun} () \rightarrow \]
\[ \text{receive} \ \langle \text{finished} \rangle \ \text{when} \ \text{true} \rightarrow \]
\[ \text{call} \ \text{io} : \ \langle \text{format}(\langle \text{Pong} \ received \ ping \sim n \rangle, []) \rangle \]
\[ \langle \text{ping}, \ \text{Ping} \rangle \ \text{when} \ \text{true} \rightarrow \]
\[ \text{do} \]
\[ \text{call} \ \text{Erlang} : \ \langle '!'(Pong, 'pong') \rangle \]
\[ \text{apply} \ \langle \text{pong} \rangle /0 \rangle \]
\[ \text{after} \ \langle \text{infinity} \rangle \rightarrow \text{true} \]

\[ \langle \text{Pong} \rangle /0 = \]
\[ \text{fun} () \rightarrow \]
\[ \text{receive} \ \langle \text{finished} \rangle \ \text{when} \ \text{true} \rightarrow \]
\[ \text{call} \ \text{io} : \ \langle \text{format}(\langle \text{Pong} \ received \ ping \sim n \rangle, []) \rangle \]
\[ \langle \text{ping}, \ \text{Ping} \rangle \ \text{when} \ \text{true} \rightarrow \]
\[ \text{do} \]
\[ \text{call} \ \text{Erlang} : \ \langle '!'(Pong, 'pong') \rangle \]
\[ \text{apply} \ \langle \text{pong} \rangle /0 \rangle \]
\[ \text{after} \ \langle \text{infinity} \rangle \rightarrow \text{true} \]


VIII. APPENDIX

A. An Example

We give a simple program of Core Erlang to describe the interpretation. The call of function format in the module io provides standard input/output very simplified.

\[ \langle \text{ping} \rangle /2 = \]
\[ \text{fun} \langle N, \text{Pong} \rangle \rightarrow \text{case} \ N \ of \]
\[0 \ \text{when} \ \text{true} \rightarrow \]
\[ \text{do} \]
\[ \text{call} \ \text{Erlang} : \ \langle '!'(Pong, 'finished') \rangle \]
\[ \text{call} \ \text{io} : \ \langle \text{format}(\langle \text{ping} \sim n \rangle, []) \rangle \]
\[ \text{receive} \ \text{pong} \ \text{when} \ \text{true} \rightarrow \]
\[ \text{call} \ \text{io} : \ \langle \text{format}(\langle \text{Ping} \ received \ pong \sim n \rangle, []) \rangle \]
\[ \text{after} \ \langle \text{infinity} \rangle \rightarrow \text{true} \]

\[ \langle \text{ping} \rangle /0 = \]
\[ \text{fun} () \rightarrow \]
\[ \text{receive} \ \langle \text{finished} \rangle \ \text{when} \ \text{true} \rightarrow \]
\[ \text{call} \ \text{io} : \ \langle \text{format}(\langle \text{Pong} \ received \ ping \sim n \rangle, []) \rangle \]
\[ \langle \text{ping}, \ \text{Ping} \rangle \ \text{when} \ \text{true} \rightarrow \]
\[ \text{do} \]
\[ \text{call} \ \text{Erlang} : \ \langle '!'(Pong, 'pong') \rangle \]
\[ \text{apply} \ \langle \text{pong} \rangle /0 \rangle \]
\[ \text{after} \ \langle \text{infinity} \rangle \rightarrow \text{true} \]
Execution

\[
\text{Transition } \rightarrow (\text{pingpong}) \quad \overset{\triangleq}{=} \quad [\text{\textit{> module 'pingpong' ... end}}] \left( S_0(e_0) \mid S_0(e_0) \right)
\]

\[
\text{\textit{ok.pingpong}} \quad \overset{\triangleq}{=} \quad [\text{\textit{> call 'pingpong' : 'start'().}}] \left( S_0(e_0) \mid S_0(e_0) \right)
\]

1) If \( \omega \) is \( \lambda_c \); \( \tilde{\mathcal{T}} \) is \( \text{init}() \).

2) \( \text{[400]} \) \( \tilde{\mathcal{T}} \) is \( \text{spawn}() \).

3) \( \text{[400]} \) \( \tilde{\mathcal{T}} \) is \( \text{spawn}() \).

\[
\begin{aligned}
\text{Ping received pong} & \text{ \quad \overset{\triangleq}{=} \quad } S_0(e_0) \left( \left( \text{Ping received pong} \right) \right) \\
\text{Ping received pong} & \text{ \quad \overset{\triangleq}{=} \quad } S_0(e_0) \left( \left( \text{Ping received pong} \right) \right) \\
\text{Ping received pong} & \text{ \quad \overset{\triangleq}{=} \quad } S_0(e_0) \left( \left( \text{Ping received pong} \right) \right) \\
\text{Ping finished} & \text{ \quad \overset{\triangleq}{=} \quad } S_0(e_0) \left( \left( \text{Ping finished} \right) \right) \\
\text{Ping finished} & \text{ \quad \overset{\triangleq}{=} \quad } S_0(e_0) \left( \left( \text{Ping finished} \right) \right)
\end{aligned}
\]

where \( E_1 \) is the body expression of the function \text{`ping pong'}. Hence \( E_2 \) is the body expression of the function \text{`pong'}. 0).

B. Proofs of Theorem 1 and Theorem 2

Theorem 1 (Corollary 1) For a running system \( \text{RS} \) of Core Erlang, if \( \text{RS} \vdash_c \text{RS}' \) then \( \text{RS} \vdash_c \text{RS}' \).

<table>
<thead>
<tr>
<th>Proof:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1) If ( \omega = \tau; \text{tm}(l) ), then we prove it by 1 of Lemma 2.</td>
</tr>
<tr>
<td>2) If ( \omega = \lambda_b; \text{tm}(l) ), and ( \text{[100]} ) ( \lambda_b ) we proof it by 2 of Lemma 2.</td>
</tr>
<tr>
<td>3) If ( \omega = \lambda_b; \text{tm}(l) ), and ( \text{[100]} ) ( \lambda_b ) we proof it by 3 of Lemma 2.</td>
</tr>
</tbody>
</table>

Theorem 2 (Corollary 2) For a running system \( \text{RS} \) of Core Erlang, if \( \text{RS} \vdash_c \text{RS}' \) then there exists an \( \text{RS}'' \) such that \( \text{RS} \vdash_c \text{RS}' \text{ and } \text{RS}' \vdash_c \text{RS}'' \text{ where } \tilde{\omega} \text{ is an action sequence } \omega_1, \ldots, \omega_k. |

<table>
<thead>
<tr>
<th>Proof:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1) If ( n_t = 0 ) then ( T \equiv \text{RS} ) and ( T' \equiv \text{RS}' ).</td>
</tr>
<tr>
<td>2) If ( n_t &gt; 0 ) then we assume that ( \text{RS} \vdash_c \text{RS}' \text{ and } \text{RS}' \vdash_c T'' \text{ where } \tilde{\omega} \text{ is an action sequence } \omega_1, \ldots, \omega_k.</td>
</tr>
</tbody>
</table>

\[
\begin{aligned}
\text{[VarAssign]} & \text{ \quad \overset{\triangleq}{=} \quad } \left( \text{\textit{let \textit{vars = e1 \cdots en}}} \mid \text{\textit{in \textit{e2 \cdots en}}} \right) \cell^c\text{RS} \\
\text{[LocRec]} & \text{ \quad \overset{\triangleq}{=} \quad } \left( \text{\textit{let \textit{kink = k}}} \mid \text{\textit{in \textit{kink = k}}} \right) \cell^c\text{RS} \\
\text{[Call]} & \text{ \quad \overset{\triangleq}{=} \quad } \left( \text{\textit{call \textit{f(e1 \cdots en) \cdots}}} \right) \cell^c\text{RS} \\
\text{[Match]} & \text{ \quad \overset{\triangleq}{=} \quad } \left( \text{\textit{match \textit{f(e1 \cdots en)}}} \mid \text{\textit{in \textit{e2 \cdots en}}} \right) \cell^c\text{RS} \\
\text{[MisMatch]} & \text{ \quad \overset{\triangleq}{=} \quad } \left( \text{\textit{match \textit{f(e1 \cdots en)}}} \mid \text{\textit{in \textit{e2 \cdots en}}} \right) \cell^c\text{RS} \\
\text{[MatchRec]} & \text{ \quad \overset{\triangleq}{=} \quad } \left( \text{\textit{match \textit{f(e1 \cdots en)}}} \mid \text{\textit{in \textit{e2 \cdots en}}} \right) \cell^c\text{RS} \\
\end{aligned}
\]
(\tilde{s_{n}})(\ldots \mid \triangleright p'_{ce,i-1}\triangleright \text{RE} \mid \triangleright p'_{ce,i+1}\triangleright \text{RE} \mid \ldots \mid S_{s}(\ell'_{i}) \mid S_{c}(\ell'_{i}')) \simeq_{\text{LS}}^{\triangleright \omega'} [\text{RS}'] \text{ and RS}' = \{\ldots, p_{d_{i-1}}, p_{d_{i+1}}, \ldots, \text{EN'}\}, \tilde{\omega} = \omega'_{1}, \ldots, \omega'_{l}, \tau .

The remaining cases are similar.

b) If $m_{\tau} \geq 1$ and $[\text{RS}] \Rightarrow_{s} T_{1} \xrightarrow{\varphi_{s,\tau}} T$, then there is a fixed finite path $[\text{RS}] \xrightarrow{\varphi_{s,\tau}} \ldots \xrightarrow{\varphi_{s,\tau}} T_{1}$. By the induction hypothesis, we also have $\text{RS} \xrightarrow{\omega'_{1}, \ldots, \omega'_{l}, \tau_{c,E}} \text{RS}_{1}$.

The intermediate states of running system are $\text{RS}'_{1}, \ldots, \text{RS}'_{l}$. There should exist a maximum pair $i, j$ such that $T_{i} \equiv [\text{RS}'_{j}]$ for $i \in \{0, \ldots, m_{\tau}\}, j \in \{0, \ldots, l\}$, $T_{0} \equiv [\text{RS}] \equiv [\text{RS}'_{0}]$. The latest intermediate state $\text{RS}'_{j} = \{p_{d_{j}}', \ldots, p_{d_{j}}', \text{EN'}\}$. We discuss the possibility of $\tau$ actions as follows:

i) It is an internal $\tau$ action of a certain $\triangleright p'_{ce,i}\triangleright \text{RE}$ in $p_{d_{j}}$ of $\text{RS}'_{j}$ to record a temporary result of evaluation. By induction on the structure of $p'_{ce,i}$, the $\text{RS}'_{j}$ should change to a certain $\text{RS}''$ after the execution of $p'_{ce,i}$ in the process with pid$_{i}$ and $T \Rightarrow_{s} \simeq_{\text{LS}}^{\triangleright \omega'} [\text{RS}'']$.

ii) It is an interaction between $S_{s}(\ell'_{i})$ and $S_{c}(\ell'_{i})$ caused by a call of spawn in $p_{d_{j}}$ of $\text{RS}$. The last result for the call of spawn would change the $\text{RS}$ to $\text{RS}'$ that includes a new process with pid$_{\text{new}}$ and $T \Rightarrow_{s} \simeq_{\text{LS}}^{\triangleright \omega'} [\text{RS}'']$.

iii) It is an interaction between $\triangleright p'_{ce,i}\triangleright \text{RE}$ and $S_{s}(\ell'_{i})$ (or $S_{c}(\ell'_{i})$). By induction on the structure of $p'_{ce,i}$ similarly, the $\text{RS}'_{j}$ should change to a $\text{RS}''$ after the execution of $p'_{ce,i}$ and $T \Rightarrow_{s} \simeq_{\text{LS}}^{\triangleright \omega'} [\text{RS}'']$.

\hao Huang is currently a Ph.D. candidate in department of Computer Science, Shanghai Jiaotong University.

His research interests mostly lie in functional languages and process calculi, including the theory of functional programming, the verification of functional languages in process calculi, formal analysis of program analysis.

\fei Yang is currently a M.S. student in department of computer science, Shanghai Jiaotong University.

His research interests lie in concurrency theory, equivalence checking problems in process calculi.