A Synergy Between Model-Checking and Type Inference for the Verification of Value-Passing Higher-Order Processes*

M. Debbabi, A. Benzakour, and B. Ktari

Computer Science Department,
Laval University,
Quebec, Canada
{debabi,benzako,ktari}@@ift.ulaval.ca

Abstract. In this paper, we present a formal verification framework for higher-order value-passing process algebra. This framework stems from an established synergy between type inference and model-checking. The language considered here is based on a sugared version of an implicitly typed λ-calculus extended with higher-order synchronous concurrency primitives. First, we endow such a syntax with a semantic theory made of a static semantics together with a dynamic semantics. The static semantics consists of an annotated type system. The dynamic semantics is operational and comes as a two-layered labeled transition system. The dynamic semantics is abstracted into a transitional semantics so as to make finite some infinite-state processes. We describe the syntax and the semantics of a verification logic that allows one to specify properties. The logic is an extension of the modal μ-calculus for handling higher-order processes, value-passing and return of results.

1 Motivation and Background

Concurrent, functional and imperative programming languages emerged as a multi-paradigmatic alternative appropriate for the development of concurrent and distributed applications. Such languages harmoniously combine syntactic compactness together with higher semantic expressiveness. Furthermore, they support functional abstraction (latent computations) and process abstraction (latent communications). Their expressivity is significantly increased by the higher-order aspect i.e. functions, pointers, channels and processes are first-class computable values (mobile values). Consequently, they cover both data and control aspects.

Concurrent and distributed systems are very often subjected to safety requirements. Accordingly, it is mandatory to have analysis and validation tools whereby one can formally guarantee the correctness of their behaviors with respect to the expected requirements. Model-checking refers to a formal, automatic and exhaustive verification technique. It consists of the extraction of a model from a formal description of the system to be verified. That model is afterwards checked against a logical or a behavioral

* This research has been funded by a grant from FCAR (Fonds pour la Formation de Chercheurs et l'Aide la Recherche), Quebec, Canada.

specification. Obviously, from the decidability standpoint, the infiniteness of the model is a limiting factor to the feasibility of model-checking. One solution, is the application of abstraction techniques, which aim to abstract an infinite model to a finite one, in such way that if some property holds for the abstracted model, it also holds for the original model.

The main contribution of this paper is a new approach for the verification of higher-order value-passing processes. This approach rests on an established synergy between model-checking and type inference. Such a synergy is achieved thanks to three major results. First, we present an abstraction technique that aims to derive finite models (transition systems) from concurrent and functional programs. The models extracted are rich enough to cope with the verification of data and control aspects of concurrent and distributed applications. Indeed, starting from a concrete dynamic semantics of a core-syntax, we derive an abstract dynamic semantics. The computable values that may be a source of infiniteness are abstracted into finite representations that are types. By doing so, a large class of infinite models will likely be reduced to finite verifiable models. Second, we present a temporal logic that is used to express data and control properties of concurrent and functional programs. Such a logic is defined as an extension of the propositional modal μ-calculus of Kozen [14] to handle communication, value-passing and higher-order objects. The logic is semantically interpreted over the abstract dynamic semantics. Third, we present a verification algorithm based on model-checking techniques. In fact, since the model is finite, the usual algorithms may be easily accommodated to the model-checking of our logic. As an example, we present an accommodation of the Emerson's algorithm.

Here is the way the rest of this paper is organized. Section 2 is devoted to the presentation of the related work. Section 3 is dedicated to the presentation of the language core-syntax considered in this work. In Section 4, we present the static semantics of our core-syntax. In Section 5, we present the dynamic operational semantics. The latter is abstracted in Section 6. The syntax and semantics of the verification logic is given in Section 7. A detailed discussion of the model-checking algorithm is presented in Section 8. Finally, a few concluding remarks and a discussion about further research are ultimately sketched as a conclusion in Section 9.

2 Related Work

The first attempt in the design of concurrent and distributed languages mainly consisted in extending some imperative languages with concurrency and distribution primitives. Accordingly, this gave rise to languages such as Ada, Chill, Modula 2 and Occam. Lately, a great deal of interest has been expressed in concurrent and functional programming. This interest is motivated by the fact that functional programming demonstrated an extensive support of abstraction through the use of abstract data types and the composition of higher-order functions. Accordingly, plenty of languages (Concurrent ML [19, 20], Facile [11], LCS [2], etc.), calculi (CHOCS [23, 24], π-calculus [15, 16]) and semantic theories [3, 4, 8, 10, 13, 17, 18] has been advanced.

Verification techniques could be structured in two major approaches: deductive techniques and semantic-based techniques. Deductive techniques consist of the use of

a logic together with the associated theorem prover. Verification is performed by deduction and is usually semi-automatic. Semantics-based verification techniques, also known as model-checking techniques, consist of the automatic extraction of a model from the program to be verified. This model approximates the dynamic behaviors of the program. Afterwards, the model is checked against another model (the specification) or against a logical specification. Logical specifications are usually expressed as formulae in modal temporal logics. In [21, 22], the author addresses the verification by proposing a methodology for generating semantically safe abstract, regular trees for programs that do not possess obvious, finite, state-transition diagram depictions. One primary result of this research is that one can, from infinite data sets, generate finite structures for model-checking. Furthermore, the methodology proposed can deal with various model infinity sources like computable values, infinite process and channel creation. In [5], the author addresses the verification by model-checking of a shared-memory concurrent imperative programming language. The author uses abstract interpretation on a true-concurrent operational semantics based on higher-dimensional transition systems. In [6], the author addresses the verification of CML programs. He presents an operational semantics for CML based on infinite domains of higher-dimensional automata. The author uses dual abstract interpretation to derive finite automata that represent sound but imprecise semantics of programs.

Recently, a surge of interest has been devoted to the verification of higher-order processes in the presence of value-passing. In [1] the authors address the specification and verification problem for process calculi such as CHOCS, CML and Facile where processes or functions are transmissible values. Their work takes place in the context of a static treatment of restriction and of a bisimulation-based semantics. They put the emphasis on (Plain) CHOCS. They show that CHOCS bisimulation can be characterized by an extension of Hennessy-Milner logic including a constructive implication, or function space constructor. Towards a proof system for the verification of process specifications, they present an infinitary sound and complete proof system for the fragment of the calculus not handling restriction. In [7], the author introduces a temporal logic for the polyadic π-calculus based on fixed point extensions of Hennessy-Milner logic. A proof system and a decision procedure are developed based on Stirling and Walker's approach to model-checking the μ-calculus using constants. A proof system and a decision procedure are obtained for arbitrary π-calculus processes with finite control.

3 Language

In this section, we present the Concurrent ML core-syntax considered in this work. We have kept the number of constructs to a bare minimum so as to facilitate a more compact and complete description of our verification framework. The BNF syntax of the core language is presented in Table 1.

Along this paper, we will write $m_{x_1,x_2,\ldots}$, the map m excluding the associations of the form $x_i \mapsto \_$. Given two maps m and m', we will write $m \dagger m'$ the overwriting of the map m by the associations of the map m' i.e. the domain of $m \dagger m'$ is $dom(m) \cup dom(m')$ and we have $(m \dagger m')(a) = m'(a)$ if $a \in dom(m')$ and $m(a)$ otherwise.

Table 1. The core syntax

Exp $\ni e$ ::=	$x \mid v \mid e\ e' \mid \texttt{rec}\ f(x) => e$	(Expressions)
	$\mid\ (e, e') \mid e\ ;\ e' \mid \texttt{let}\ x = e\ \texttt{in}\ e'\ \texttt{end}$	
	$\mid\ \texttt{if}\ e\ \texttt{then}\ e'\ \texttt{else}\ e''\ \texttt{end}$	
	$\mid\ \texttt{spawn}(e) \mid \texttt{sync}(e) \mid \texttt{receive}(e) \mid \texttt{transmit}(e, e')$	
	$\mid\ \texttt{choose}(e, e') \mid \texttt{channel}()$	
Val $\ni v$ ::=	$c \mid \texttt{fn}\ x => e$	(Values)
Cst $\ni c$ ::=	$() \mid \texttt{true} \mid \texttt{false} \mid \texttt{num}\ n$	(Constants)

4 Static Semantics

Our intention here is to endow our core-syntax with a static semantics. The latter is a standard annotated effect type system. We introduce the following static domains:

1. The domain of *regions*: regions are intended to abstract channels. Their domain consists in the disjoint union of a countable set of constants ranged over by r and variables ranged over by ϱ. We will use $\rho, \rho', ...$ to represent values drawn from this domain.
2. The domain of *side* and *communication effects* is inductively defined by:

   $$\sigma ::= \emptyset \mid \varsigma \mid \sigma \cup \sigma' \mid create(\rho, \tau) \mid in(\rho, \tau) \mid out(\rho, \tau)$$

   We use $\emptyset$ to denote an empty effect and ς to denote an effect variable. The communication effect $create(\rho, \tau)$ represents the creation, in the region ρ, of a channel that is a medium for values of type τ. The term $in(\rho, \tau)$ denotes the communication effect resulting from receiving a value of type τ on a channel in the region ρ and $out(\rho, \tau)$ denotes the communication effect resulting from sending a value of type τ on a channel in the region ρ. The effect $\sigma \cup \sigma'$ stands for an effect that represent an upper approximation of σ and σ' (effect cumulation). Actually, only one of the two effects, σ or σ', will emerge at the dynamic evaluation. We write $\sigma \sqsupseteq \sigma' \Leftrightarrow \exists \sigma''.\sigma = \sigma' \cup \sigma''$. Equality on effects is modulo ACUI (Associativity, Commutativity and Idempotence) with $\emptyset$ as the neutral element.
3. The domain of *types* is inductively defined by:

   $$\tau ::= unit \mid int \mid bool \mid \alpha \mid \tau \times \tau' \mid chan_\rho(\tau) \mid event_\sigma(\tau) \mid \tau \xrightarrow{\sigma} \tau'$$

   The term $chan_\rho(\tau)$ is the type of channels in the region ρ that are intended to be media for values of type τ. The term $\tau \xrightarrow{\sigma} \tau'$ is the type of functions that take parameters of type τ to values of type τ' with a latent effect σ. By latent effect, we refer to the effect generated when the corresponding function expression is evaluated. The type $event_\sigma(\tau)$ denotes inactive processes having potential effect (latent effect σ) that are expected to return a value of type τ once their execution terminated.

Table 2. The typing rules

(cte)	$\dfrac{\tau \lhd \mathit{TypeOf}(c)}{\mathcal{E} \vdash c : \tau, \emptyset}$
(var)	$\dfrac{\tau \lhd \mathcal{E}(x)}{\mathcal{E} \vdash x : \tau, \emptyset}$
(abs)	$\dfrac{\mathcal{E}_x \dagger [x \mapsto \tau] \vdash e : \tau', \sigma}{\mathcal{E} \vdash \ \mathtt{fn}\, x => e : \tau \xrightarrow{\sigma} \tau', \emptyset}$
(app)	$\dfrac{\mathcal{E} \vdash e : \tau \xrightarrow{\sigma} \tau', \sigma' \ \ \mathcal{E} \vdash e' : \tau, \sigma''}{\mathcal{E} \vdash (e\ e') : \tau', ((\sigma'; \sigma''); \sigma)}$
(let)	$\dfrac{\mathcal{E} \vdash e : \tau, \sigma \ \ \mathcal{E}_x \dagger [x \mapsto \mathit{Gen}(\mathcal{E}, \tau, \sigma)] \vdash e' : \tau', \sigma'}{\mathcal{E} \vdash \mathtt{let}\ x = e\ \mathtt{in}\ e'\ \mathtt{end} : \tau', (\sigma; \sigma')}$
(pair)	$\dfrac{\mathcal{E} \vdash e : \tau, \sigma \ \ \mathcal{E} \vdash e' : \tau', \sigma'}{\mathcal{E} \vdash (e, e') : \tau \times \tau', (\sigma; \sigma')}$
(seq)	$\dfrac{\mathcal{E} \vdash e : \tau, \sigma \ \ \mathcal{E} \vdash e' : \tau', \sigma'}{\mathcal{E} \vdash e\ ;\ e' : \tau', (\sigma; \sigma')}$
(if)	$\dfrac{\mathcal{E} \vdash e : \mathit{bool}, \sigma \ \ \mathcal{E} \vdash e' : \tau, \sigma' \ \ \mathcal{E} \vdash e'' : \tau, \sigma''}{\mathcal{E} \vdash \mathtt{if}\ e\ \mathtt{then}\ e'\ \mathtt{else}\ e''\ \mathtt{end} : \tau, \sigma \cup \sigma' \cup \sigma''}$
(rec)	$\dfrac{\mathcal{E}_{x,f} \dagger [x \mapsto \tau, f \mapsto \tau \xrightarrow{\sigma} \tau'] \vdash e : \tau', \sigma}{\mathcal{E} \vdash \mathtt{rec}\ f(x) => e : \tau \xrightarrow{\sigma} \tau', \emptyset}$
(obs)	$\dfrac{\mathcal{E} \vdash e : \tau, \sigma \ \ \mathit{Observe}(\mathcal{E}, \tau, \sigma) \sqsubseteq \sigma'}{\mathcal{E} \vdash e : \tau, \sigma'}$

Table 2 presents the static semantics of our core language.

The static semantics manipulates sequents of the form $\mathcal{E} \vdash e : \tau, \sigma$, which state that under some typing environment $\mathcal{E}$ the expression e has type τ and effect σ. We also define type schemes of the form $\forall v_1, \ldots, v_n.\tau$, where v_i can be type, region or effect variable. A type τ' is an instance of $\forall v_1, \ldots, v_n.\tau$, noted $\tau' \lhd \forall v_1, \ldots, v_n.\tau$, if there exists a substitution θ defined over $v_1, \ldots, v_n$ such that $\tau' = \theta\tau$.

Type generalization in this type system states that a variable cannot be generalized if it is free in the type environment $\mathcal{E}$ or if it is present in the inferred effect:

$$\mathit{Gen}(\mathcal{E}, \tau, \sigma) = \mathtt{let}\ v_{1..n} = fv(\tau) \backslash (fv(\mathcal{E}) \cup fv(\sigma))\ \mathtt{in}\ \forall v_{1..n}.\tau\ \mathtt{end}$$

where $fv(\_)$ denotes the set of free variables. The observation criterion was introduced in order to report only effects that can affect the context of an expression.

$$\begin{aligned}\mathit{Observe}(\mathcal{E}, \tau, \sigma) = &\{\varsigma \in \sigma \mid \varsigma \in fv(\mathcal{E}) \cup fv(\tau)\} \\ &\cup \{\mathit{create}(\rho, \tau') \in \sigma \mid \rho \in fr(\mathcal{E}) \cup fr(\tau) \wedge \tau' \in \text{S\_T}\} \\ &\cup \{\mathit{in}(\rho, \tau') \in \sigma \mid \rho \in fr(\mathcal{E}) \cup fr(\tau) \wedge \tau' \in \text{S\_T}\} \\ &\cup \{\mathit{out}(\rho, \tau') \in \sigma \mid \rho \in fr(\mathcal{E}) \cup fr(\tau) \wedge \tau' \in \text{S\_T}\}\end{aligned}$$

where S_T is the domain of types, $fr(\mathcal{E})$ stands for the set of free channel regions in the static environment $\mathcal{E}$. The function *TypeOf* allows the typing of built-in primitives as defined in the Table 3.

Table 3. The initial static basis

$$
\begin{array}{lll}
\textit{TypeOf} = & [\ () & \mapsto unit, \\
& \texttt{true} & \mapsto bool, \\
& \texttt{false} & \mapsto bool, \\
& \texttt{num}\ n & \mapsto int, \\
& \texttt{channel} & \mapsto \forall\alpha,\varrho,\varsigma.unit \xrightarrow{\varsigma\cup create(\varrho,\alpha)} chan_{\varrho}(\alpha), \\
& \texttt{receive} & \mapsto \forall\alpha,\varrho,\varsigma,\varsigma'.chan_{\varrho}(\alpha) \xrightarrow{\varsigma} event_{\varsigma'\cup in(\varrho,\alpha)}(\alpha), \\
& \texttt{transmit} & \mapsto \forall\alpha,\varrho,\varsigma,\varsigma'.chan_{\varrho}(\alpha)\times\alpha \xrightarrow{\varsigma} event_{\varsigma'\cup out(\varrho,\alpha)}(unit), \\
& \texttt{choose} & \mapsto \forall\alpha,\varsigma,\varsigma',\varsigma'',\varsigma'''.event_{\varsigma}(\alpha)\times event_{\varsigma'}(\alpha) \xrightarrow{\varsigma''} event_{\varsigma'''\cup\varsigma\cup\varsigma'}(\alpha), \\
& \texttt{spawn} & \mapsto \forall\varsigma,\varsigma'.(unit \xrightarrow{\varsigma} unit) \xrightarrow{\varsigma'} unit, \\
& \texttt{sync} & \mapsto \forall\alpha,\varsigma,\varsigma'.event_{\varsigma}(\alpha) \xrightarrow{\varsigma'} \alpha \\
&] &
\end{array}
$$

5 Concrete Dynamic Semantics

In this section, we endow our core-syntax with a dynamic operational semantics. The latter is now standard and will be defined here as a two-layered labeled transition system following [12]. First of all, we need to introduce some semantic domains and to extend the expression syntax to intermediate expressions (expressions that may occur during the dynamic evaluation). As illustrated in Table 4, we introduce six semantic categories.

Table 4. The semantic categories

CVal	$\ni cv$	$::= c \mid \texttt{fn}\ x => i \mid k \mid ev \mid (cv, cv)$	(Computable Values)
Evt	$\ni ev$	$::= \langle ec, cv\rangle$	(Events)
IExp	$\ni i$	$::= e \mid cv \mid cv\ e \mid \texttt{let}\ x = cv\ \texttt{in}\ e\ \texttt{end}$	
		$\mid\ (cv, e) \mid cv; e$	(Intermediate Expressions)
ECon	$\ni ec$	$::= \texttt{receive} \mid \texttt{transmit} \mid \texttt{choose}$	(Event Constructors)
Com	$\ni com$	$::= k?cv \mid k!cv$	(Communications)
Act	$\ni a$	$::= com \mid \epsilon \mid \lambda(k) \mid \phi(cv)$	(Actions)

The semantic category CVal is ranged over by cv and corresponds to the domain of computable values. The semantic category Evt of events is ranged over by ev. An event is a pair consisting of the event constructor ec and its argument cv. An event constructor ec is a member of the syntactic domain ECon. The semantic category IExp is ranged over by i and corresponds to the domain of intermediate expressions. The semantic category Com is ranged over by com and corresponds to the domain of communications.

Input communications are of the form $k?cv$ where k is a channel computable value and cv is another computable value that will be received on the channel k. Output communications are of the form $k!cv$ where k is a channel computable value and cv is another computable value that will be sent along the channel k. The semantic category Act is ranged over by a and corresponds to the domain of actions. The silent action ϵ denotes internal moves. A creation of a channel k is considered as an action and is written $\lambda(k)$. A process spawning of a value cv is considered as an action and is written $\phi(cv)$.

The operational semantics is structured in two layers, one for expressions in isolation and one multiset of expressions running in parallel. These two layers involve three transition relations whose definitions are given hereafter.

5.1 Expression Semantics

The first relation, written $\_ \stackrel{\_}{\Longrightarrow} \_ \subseteq \text{Evt} \times \text{Com} \times \text{IExp}$, is a transition relation that is meant to define the communication potential of events. The rules that define this relation are presented in Table 5.

Table 5. The semantic rules of the relation $\_ \stackrel{\_}{\Longrightarrow} \_$

(transmit) $\langle \texttt{transmit}, (k, cv)\rangle \stackrel{k!cv}{\Longrightarrow} ()$	(receive) $\langle \texttt{receive}, k\rangle \stackrel{k?cv}{\Longrightarrow} cv$
(choose$_1$) $\dfrac{ev_1 \stackrel{com}{\Longrightarrow} i}{\langle \texttt{choose}, (ev_1, ev_2)\rangle \stackrel{com}{\Longrightarrow} i}$	(choose$_2$) $\dfrac{ev_2 \stackrel{com}{\Longrightarrow} i}{\langle \texttt{choose}, (ev_1, ev_2)\rangle \stackrel{com}{\Longrightarrow} i}$

A transition of the form $ev \stackrel{com}{\Longrightarrow} i$ intuitively means that the event ev has the potential of performing the communication com (when `sync` is applied to the event) and then it will behave as the intermediate expression i.

The second relation, written $\_ \stackrel{\_}{\hookrightarrow} \_ \subseteq \text{IExp} \times \text{Act} \times \text{IExp}$, is the one that defines the operational semantics of processes. A transition of the form $i \stackrel{a}{\hookrightarrow} i'$ intuitively means that by performing the action a, the intermediate expression i will behave as i'. The rules that define this relation are presented in Table 6.

5.2 Program Semantics

Now, in order to define the third transition relation we need to introduce the following semantic functions and domains. We denote by $CV[\![i]\!]$ the set of channels k occurring in an intermediate expression i.

We view a program as a multiset of intermediate expressions. We let Prog be IExp-MultiSet i.e. the set of program multisets. The set of channels that occur in a multiset P is obtained by including the channels in each intermediate expression. The operational semantics of programs is based on the evolution of the so-called configurations. We define a K-configuration, and we write $K :: P$, to be a pair where the first component K is the set of all channels allocated up to a certain point, and the second component

Table 6. The concrete operational semantics of processes

$$\text{(app}_1\text{)}\quad \frac{i_1 \overset{a}{\hookrightarrow} i_1'}{i_1\ i_2 \overset{a}{\hookrightarrow} i_1'\ i_2} \qquad \text{(app}_2\text{)}\quad \frac{i \overset{a}{\hookrightarrow} i'}{cv\ i \overset{a}{\hookrightarrow} cv\ i'}$$

$$\text{(beta}_1\text{)}\quad (\texttt{fn}\ x \texttt{=>}\ i)\ cv \overset{\epsilon}{\hookrightarrow} i[cv/x] \qquad \text{(beta}_2\text{)}\quad ec\ cv \overset{\epsilon}{\hookrightarrow} \langle ec, cv\rangle$$

$$\text{(pair}_1\text{)}\quad \frac{i_1 \overset{a}{\hookrightarrow} i_1'}{(i_1, i_2) \overset{a}{\hookrightarrow} (i_1', i_2)} \qquad \text{(pair}_2\text{)}\quad \frac{i \overset{a}{\hookrightarrow} i'}{(cv, i) \overset{a}{\hookrightarrow} (cv, i')}$$

$$\text{(seq}_1\text{)}\quad \frac{i_1 \overset{a}{\hookrightarrow} i_1'}{i_1; i_2 \overset{a}{\hookrightarrow} i_1'; i_2} \qquad \text{(seq}_2\text{)}\quad \frac{i \overset{a}{\hookrightarrow} i'}{cv; i \overset{a}{\hookrightarrow} cv; i'}$$

$$\text{(chan)}\quad \texttt{channel}() \overset{\lambda(k)}{\hookrightarrow} k \qquad \text{(spawn}_1\text{)}\quad \texttt{spawn}\ cv \overset{\phi(cv)}{\hookrightarrow} ()$$

$$\text{(rec)}\quad \texttt{rec}\ f(x) \texttt{=>}\ i \overset{\epsilon}{\hookrightarrow} \texttt{fn}\ x \texttt{=>}\ i[(\texttt{rec}\ f(x) \texttt{=>}\ i)/f]$$

$$\text{(if}_1\text{)}\quad \frac{i_1 \overset{a}{\hookrightarrow} i_1'}{\texttt{if}\ i_1\ \texttt{then}\ i_2\ \texttt{else}\ i_3\ \texttt{end} \overset{a}{\hookrightarrow} \texttt{if}\ i_1'\ \texttt{then}\ i_2\ \texttt{else}\ i_3\ \texttt{end}}$$

$$\text{(if}_2\text{)}\quad \texttt{if true then}\ i_1\ \texttt{else}\ i_2\ \texttt{end} \overset{\epsilon}{\hookrightarrow} i_1$$

$$\text{(if}_3\text{)}\quad \texttt{if false then}\ i_1\ \texttt{else}\ i_2\ \texttt{end} \overset{\epsilon}{\hookrightarrow} i_2$$

$$\text{(let}_1\text{)}\quad \frac{i_1 \overset{a}{\hookrightarrow} i_1'}{\texttt{let}\ x = i_1\ \texttt{in}\ i_2\ \texttt{end} \overset{a}{\hookrightarrow} \texttt{let}\ x = i_1'\ \texttt{in}\ i_2\ \texttt{end}}$$

$$\text{(let}_2\text{)}\quad \texttt{let}\ x = cv\ \texttt{in}\ i\ \texttt{end} \overset{\epsilon}{\hookrightarrow} i[cv/x]$$

$$\text{(sync)}\quad \frac{ev \overset{com}{\Longrightarrow} i}{\texttt{sync}\ ev \overset{com}{\hookrightarrow} i}$$

P is a program (a multiset of intermediate expressions). Let Chan be the set of channel computable values. The domain of K-configuration Conf_K is defined as follows:

$$\text{Conf}_K = \{K :: P \mid K \in \text{Chan} \ \wedge\ P \in \text{Prog} \ \wedge\ CV[P] \subseteq K\}$$

The semantics of K-configurations is given in terms of the labeled transition system $(\text{Conf}_K, \text{Com} \cup \{\epsilon\}, \longrightarrow)$. The transition relation $\longrightarrow$ is defined as the smallest subset of $\text{Conf}_K \times \text{Com} \cup \{\epsilon\} \times \text{Conf}_K$ closed under the rules presented in Table 7.

The function Msg extracts the set of channels that are transmitted in a communication.

6 Abstract Dynamic Semantics

In this section, we describe an abstract dynamic semantics derived from the concrete dynamic semantics viewed in the previous section. The motivation is to abstract computable values that could be a source of infiniteness. These values are abstracted into finite representations that are types. By doing so, we ensure that a large class of infinite models will likely be reduced to finite verifiable models.

The abstract semantic categories are illustrated in Table 8. The abstract semantic category AVal is ranged over by cv and corresponds to the domain of abstract values. The semantic category AFExp is ranged over by *afe* and corresponds to the abstract functional expressions. The abstract semantic category ACst is ranged over by ac and

Table 7. The operational semantics of programs

(action)	$\dfrac{i \overset{a}{\hookrightarrow} i'}{K :: \{\!\lvert i \rvert\!\} \overset{a}{\longrightarrow} K \cup \mathit{Msg}(a) :: \{\!\lvert i' \rvert\!\}}$
(channel)	$\dfrac{i \overset{\lambda(k)}{\hookrightarrow} i'}{K :: \{\!\lvert i \rvert\!\} \overset{\epsilon}{\longrightarrow} K \cup \{k\} :: \{\!\lvert i' \rvert\!\}}\ k \notin K$
(spawn$_2$)	$\dfrac{i \overset{\phi(cv)}{\hookrightarrow} i'}{K :: \{\!\lvert i \rvert\!\} \overset{\epsilon}{\longrightarrow} K :: \{\!\lvert i', cv() \rvert\!\}}$
(communication)	$\dfrac{i_1 \overset{k?cv}{\hookrightarrow} i_1', \quad i_2 \overset{k!cv}{\hookrightarrow} i_2'}{K :: \{\!\lvert i_1, i_2 \rvert\!\} \overset{\epsilon}{\longrightarrow} K :: \{\!\lvert i_1', i_2' \rvert\!\}}$
(isolation)	$\dfrac{K :: P_1 \overset{a}{\longrightarrow} K' :: P_1'}{K :: P_1 \cup P_2 \overset{a}{\longrightarrow} K' :: P_1' \cup P_2}$

Table 8. The abstract semantic categories

AVal	$\ni cv$	$::= ac \mid k_{chan_\rho(\tau)} \mid ev \mid \mathit{afe} \mid (cv, cv)$	(Abstract Values)
AFExp	$\ni \mathit{afe}$	$::= \texttt{fn}\ x \texttt{=>}\ i \mid \tau \overset{\sigma}{\longrightarrow} \tau'$	(Abstract Functional Exp.)
ACst	$\ni ac$	$::= c \mid \mathit{int} \mid \mathit{bool} \mid \mathit{unit}$	(Abstract Constants)
AChan	$\ni k_{chan_\rho(\tau)}$	$::= k \mid \mathit{chan}_\rho(\tau)$	(Abstract Channels)
AEvt	$\ni ev$	$::= \langle ec, cv \rangle \mid \mathit{event}_\sigma(\tau)$	(Abstract Events)
ECon	$\ni ec$	$::= \texttt{receive} \mid \texttt{transmit} \mid \texttt{choose}$	(Event Constructors)
AIExp	$\ni i$	$::= e \mid cv \mid cv\ e \mid (cv, e) \mid cv; e$ $\mid\ \texttt{let}\ x = cv\ \texttt{in}\ e\ \texttt{end}$	(Abstract Intermediate Exp.)
ACom	$\ni com$	$::= k_{chan_\rho(\tau)}?\tau \mid k_{chan_\rho(\tau)}!cv$	(Abstract Communications)
AAct	$\ni a$	$::= com \mid \epsilon \mid \lambda(k_{chan_\rho(\tau)}) \mid \phi(cv)$	(Abstract Actions)
Loc	$\ni l$	$::= n \mid n.l \quad \text{where} \quad n \in \mathbb{N}$	(Locations)

corresponds to the domain of abstract constants. The abstract semantic category AChan is ranged over by $k_{chan_\rho(\tau)}$ and corresponds to the domain of abstract channels. This category includes channel computable values together with the type of channels. The abstract semantic category AEvt is ranged over by ev and corresponds to the domain of abstract events. This category includes events together with the type of events. The abstract semantic category AIExp is ranged over by i and corresponds to the domain of abstract intermediate expressions. The abstract semantic category ACom is ranged over by com and corresponds to the domain of abstract communications. Abstract input communications are of the form $k_{chan_\rho(\tau)}?\tau$ and stand for the action of receiving values, abstracted by their type, on an abstract channel. Abstract output communications are of the form $k_{chan_\rho(\tau)}!cv$ and stand for the transmission of an abstract value along an abstract channel. The abstract semantic category AAct is ranged over by a and corresponds to the domain of actions. The abstract semantic category Loc is ranged over by l and corresponds to the domain of locations.

6.1 Expression Semantics

The rules that define the relation $\_ \stackrel{\cdot}{\Longrightarrow} \_ \subseteq$ AEvt $\times$ ACom $\times$ AIExp are presented in Table 9.

Table 9. The abstract semantic rules of the relation $\_ \stackrel{\cdot}{\Longrightarrow} \_$

(transmit)	$\langle \texttt{transmit}, (k_{chan_\rho(\tau)}, cv)\rangle \overset{k_{chan_\rho(\tau)}!cv}{\Longrightarrow} ()$
(receive)	$\langle \texttt{receive}, k_{chan_\rho(\tau)}\rangle \overset{k_{chan_\rho(\tau)}?\tau}{\Longrightarrow} \tau$
(choose$_1$)	$\dfrac{ev_1 \overset{com}{\Longrightarrow} i}{\langle \texttt{choose}, (ev_1, ev_2)\rangle \overset{com}{\Longrightarrow} i}$
(choose$_2$)	$\dfrac{ev_2 \overset{com}{\Longrightarrow} i}{\langle \texttt{choose}, (ev_1, ev_2)\rangle \overset{com}{\Longrightarrow} i}$

The (transmit) and (receive) rules are changed to reflect the use of abstract values. For example, the second one means that the event $\langle \texttt{receive}, k_{chan_\rho(\tau)}\rangle$ has the potential of performing the communication $k_{chan_\rho(\tau)}?\tau$, and then behaves as the abstract value τ. This means that the possible values received in the channel $k_{chan_\rho(\tau)}$ are abstracted as their type. This abstraction ensures that large class of infinite models will likely be reduced to finite models. The rules that define the relation $\_ \stackrel{\cdot}{\hookrightarrow}_{\cdot} \_ \subseteq$ AIExp $\times$ AAct $\times$ Loc $\times$ AIExp are presented in Table 10.

For instance, the rule (beta$_3$) is defined to evaluate the application of a class of functions that have the same type ($\tau \overset{\sigma}{\longrightarrow} \tau'$) to an abstract value cv.

6.2 Program Abstract Semantics

As the concrete semantics, the program abstract semantics is based on the evolution of K-configurations. Definitions of the domain of K-configuration and the semantic

Table 10. The abstract operational semantics of processes

$$\text{(app}_1\text{)}\quad \frac{i_1 \overset{a}{\underset{l}{\hookrightarrow}} i_1'}{i_1\ i_2 \overset{a}{\underset{1.l}{\hookrightarrow}} i_1'\ i_2} \qquad \text{(app}_2\text{)}\quad \frac{i \overset{a}{\underset{l}{\hookrightarrow}} i'}{cv\ i \overset{a}{\underset{2.l}{\hookrightarrow}} cv\ i'}$$

$$\text{(beta}_1\text{)}\quad (\texttt{fn}\ x \texttt{=>}\ i)\ cv \overset{\epsilon}{\underset{0}{\hookrightarrow}} i[cv/x] \qquad \text{(beta}_2\text{)}\quad ec\ cv \overset{\epsilon}{\underset{0}{\hookrightarrow}} < ec, cv >$$

$$\text{(beta}_3\text{)}\quad (\tau \xrightarrow{\sigma} \tau')(cv) \overset{\epsilon}{\underset{0}{\hookrightarrow}} \tau' \qquad \text{(chan)}\quad \texttt{channel}() \overset{\lambda(k_{chan_\rho(\tau)})}{\underset{0}{\hookrightarrow}} k_{chan_\rho(\tau)}$$

$$\text{(pair}_1\text{)}\quad \frac{i_1 \overset{a}{\underset{l}{\hookrightarrow}} i_1'}{(i_1, i_2) \overset{a}{\underset{1.l}{\hookrightarrow}} (i_1', i_2)} \qquad \text{(pair}_2\text{)}\quad \frac{i \overset{a}{\underset{l}{\hookrightarrow}} i'}{(cv, i) \overset{a}{\underset{2.l}{\hookrightarrow}} (cv, i')}$$

$$\text{(seq}_1\text{)}\quad \frac{i_1 \overset{a}{\underset{l}{\hookrightarrow}} i_1'}{i_1; i_2 \overset{a}{\underset{1.l}{\hookrightarrow}} i_1'; i_2} \qquad \text{(seq}_2\text{)}\quad \frac{i \overset{a}{\underset{l}{\hookrightarrow}} i'}{cv; i \overset{a}{\underset{2.l}{\hookrightarrow}} cv; i'}$$

$$\text{(spawn}_1\text{)}\quad \texttt{spawn}\ cv \overset{\phi(cv)}{\underset{0}{\hookrightarrow}} ()$$

$$\text{(rec)}\quad \texttt{rec}\ f(x) \texttt{=>}\ i \overset{\epsilon}{\underset{0}{\hookrightarrow}} \texttt{fn}\ x \texttt{=>}\ i[(\texttt{rec}\ f(x) \texttt{=>}\ i)/f]$$

$$\text{(if}_1\text{)}\quad \frac{i_1 \overset{a}{\underset{l}{\hookrightarrow}} i_1'}{\texttt{if}\ i_1\ \texttt{then}\ i_2\ \texttt{else}\ i_3\ \texttt{end} \overset{a}{\underset{1.l}{\hookrightarrow}} \texttt{if}\ i_1'\ \texttt{then}\ i_2\ \texttt{else}\ i_3\ \texttt{end}}$$

$$\text{(if}_2\text{)}\quad \texttt{if true then}\ i_1\ \texttt{else}\ i_2\ \texttt{end} \overset{\epsilon}{\underset{0}{\hookrightarrow}} i_1$$

$$\text{(if}_3\text{)}\quad \texttt{if false then}\ i_1\ \texttt{else}\ i_2\ \texttt{end} \overset{\epsilon}{\underset{0}{\hookrightarrow}} i_2$$

$$\text{(let}_1\text{)}\quad \frac{i_1 \overset{a}{\underset{l}{\hookrightarrow}} i_1'}{\texttt{let}\ x = i_1\ \texttt{in}\ i_2\ \texttt{end} \overset{a}{\underset{1.l}{\hookrightarrow}} \texttt{let}\ x = i_1'\ \texttt{in}\ i_2\ \texttt{end}}$$

$$\text{(let}_2\text{)}\quad \texttt{let}\ x = cv\ \texttt{in}\ i\ \texttt{end} \overset{\epsilon}{\underset{0}{\hookrightarrow}} i[cv/x]$$

$$\text{(sync)}\quad \frac{ev \overset{com}{\Longrightarrow} i}{\texttt{sync}\ ev \overset{com}{\underset{0}{\hookrightarrow}} i}$$

function *Msg* remain the same except that the latter is defined over abstract values. The semantics of K-configuration is given in terms of the labeled transition system $(\text{Conf}_K, \text{ACom} \cup \{\epsilon\}, \longrightarrow)$. The transition relation $\longrightarrow$ is defined as the smallest subset of $\text{Conf}_K \times \text{ACom} \cup \{\epsilon\} \times \text{Conf}_K$ closed under the rules presented in Table 11.

Table 11. The abstract operational semantics of programs

(action)	$\dfrac{i \overset{a}{\underset{l}{\hookrightarrow}} i'}{K :: \{\!\|i\|\!\} \xrightarrow{a} K \cup Msg(a) :: \{\!\|i'\|\!\}}$
(channel)	$\dfrac{i \overset{\lambda(k_{chan_{\rho(\tau)}})}{\underset{l}{\hookrightarrow}} i'}{K :: \{\!\|i\|\!\} \xrightarrow{\epsilon} K \cup \{k_{chan_{\rho(\tau)}}\} :: \{\!\|i'\|\!\}}\ k_{chan_{\rho(\tau)}} \notin K$
(spawn$_2$)	$\dfrac{i \overset{\phi(cv)}{\underset{l}{\hookrightarrow}} i'}{K :: \{\!\|i\|\!\} \xrightarrow{\epsilon} K :: \{\!\|i', cv()\|\!\}}$
(Communication)	$\dfrac{i_1 \overset{k_{chan_{\rho(\tau)}}?\tau}{\underset{l}{\hookrightarrow}} i'_1, \quad i_2 \overset{k_{chan_{\rho(\tau)}}!cv}{\underset{l'}{\hookrightarrow}} i'_2}{K :: \{\!\|i_1, i_2\|\!\} \xrightarrow{\epsilon} K :: \{\!\|i'_1[cv]_l, i'_2\|\!\}}$
(Isolation)	$\dfrac{K :: P_1 \xrightarrow{a} K' :: P'_1}{K :: P_1 \cup P_2 \xrightarrow{a} K' :: P'_1 \cup P_2}$

where $i'_1[cv]_l$ stands for the term i'_1 in which the subterm at location l will be replaced by cv.

6.3 Correctness of the abstraction

The correctness of the abstraction is assured since there is an equivalence between the abstract transition graph and the concrete one. In fact, by unfolding in the abstract graph each transition containing an abstract term by the equivalent set of transitions composed by concrete values, we transform an abstract transition graph into a concrete one.

7 A Modal logic for concurrent ML

In this section we introduce a logic that allows one to specify properties of expressions. The logic we consider may be viewed as a variant of the modal μ-calculus [14], or the Hennessey-Milner Logic with recursion. In the proposed logic, modal formulae can also be used to express communication, value-passing and result returns. This logic is semantically interpreted over the abstract dynamic semantics.

7.1 Syntax

The syntax of formulae is presented in Table 12. We refer to this logic as L_μ.

Table 12. The logic

ψ	::= $\texttt{tt} \mid \texttt{ff} \mid X \mid \neg\psi \mid \psi \vee \psi' \mid \psi \wedge \psi' \mid$	(Boolean Expressions)
	$\mid$ <a> $\psi \mid$ <$return(cv)$>	(Diamond Formulae)
	$\mid$ [a] $\psi \mid$ [$return(cv)$]	(Box Formulae)
	$\mid \mu X.\psi$	(Greatest Fixpoint Formulae)
	$\mid \nu X.\psi$	(Least Fixpoint Formulae)
a	::= $k\ dir\ cv \mid \epsilon$	(Actions)
dir	::= $! \mid ?$	(Directions)

The symbols $\neg$, $\vee$ and $\wedge$ respectively represent negation, disjunction and conjunction. The symbol <a> (resp. <$return(cv)$>) is a modal operator indexed by a (resp. by $return(cv)$) known as the diamond. The meaning of modalized formulae appeal to transition behavior of a program. For instance, a program satisfies the formula <a> ψ if it can evolve to some K-configuration obeying ψ by performing an action a. The actions can either be the silent action ϵ or the communication actions $k!cv$ or $k?cv$. Furthermore, a program satisfies the formula <$return(cv)$> if it can return the value cv. In the same way, the symbol [a] (resp. [$return(cv)$]) is a modal operator known as box. A program satisfies the formula [a] ψ if after every performance of an action a, each result K-configuration satisfies ψ. Furthermore, a program satisfies the formula [$return(cv)$] if it returns necessarily the value cv. Variables are ranged over by X. The formulae $\mu X.\psi$ (resp. $\nu X.\psi$) is a recursive formula where the least fixpoint operator μ (resp. greatest fixpoint operator ν) binds all free occurrences of X in ψ. An occurrence of X is free if it is not within the scope of a binder μX or νX. Note that like the μ-calculus, all occurrences of X in ψ must appear inside the scope of an even number of negations. This is to ensure the existence of fixpoints.

7.2 Semantics

Formulae are interpreted over models of the form $M = \langle\mathcal{ST}, L\rangle$, where $\mathcal{ST} = (\text{Conf}_K, \text{ACom} \cup \{\epsilon\}, \longrightarrow)$, and environment of the form $e = [X_i \mapsto P_i]$ which maps variables X_i to sets of K-configurations. Semantically, formulae of the logic correspond to sets of K-configurations for which they are true. The meaning function $[\![.]\!]_e^M : L_\mu \rightarrow 2^{\mathcal{C}}$ is described in Table 13. The set $\mathcal{C}$ refers to the set of K-configurations.

Intuitively, all K-configurations satisfy the formula tt while there are no K-configurations that satisfy ff. The meaning of a variable X is simply the K-configurations that are bound to X in the environment e. Negation, disjunction and conjunction are interpreted in a classical way. The meaning of formulae <a> ψ are K-configurations c that can evolve, by performing an action a, to some K-configuration c' such that c' is part of the meaning of ψ. More accurately, if the action a is an output communication action involving a value cv, then we must ensure the existence of a constant cv' such that $cv \preceq cv'$. The preorder relation $\preceq$ is defined on abstract values as below:

$$cv \preceq cv' \Longleftrightarrow \exists\theta.\theta(cv') = cv$$

Table 13. The semantic

$[\![\mathtt{tt}]\!]_e^M = \mathcal{C}$
$[\![\mathtt{ff}]\!]_e^M = \emptyset$
$[\![X]\!]_e^M = e(X)$
$[\![\neg\psi]\!]_e^M = \mathcal{C} \setminus [\![\psi]\!]_e^M$
$[\![\psi_1 \vee \psi_2]\!]_e^M = [\![\psi_1]\!]_e^M \cup [\![\psi_2]\!]_e^M$
$[\![\psi_1 \wedge \psi_2]\!]_e^M = [\![\psi_1]\!]_e^M \cap [\![\psi_2]\!]_e^M$
$[\![\texttt{<}\epsilon\texttt{>}\ \psi]\!]_e^M = \{c \in \mathcal{C} \mid \exists c'.c \xrightarrow{\epsilon} c' \wedge c' \in [\![\psi]\!]_e^M\}$
$[\![\texttt{[}\epsilon\texttt{]}\ \psi]\!]_e^M = \{c \in \mathcal{C} \mid \forall c'.c \xrightarrow{\epsilon} c' \Rightarrow (c' \in [\![\psi]\!]_e^M)\}$
$[\![\texttt{<}k!cv\texttt{>}\ \psi]\!]_e^M = \{c \in \mathcal{C} \mid \exists c', cv'.c \xrightarrow{k!cv'} c' \wedge c' \in [\![\psi[cv'/cv]]\!]_e^M \wedge cv \preceq cv'\}$
$[\![\texttt{[}k!cv\texttt{]}\ \psi]\!]_e^M = \{c \in \mathcal{C} \mid \forall c', \exists cv'.c \xrightarrow{k!cv'} c' \Rightarrow (c' \in [\![\psi[cv'/cv]]\!]_e^M \wedge cv \preceq cv')\}$
$[\![\texttt{<}k?cv\texttt{>}\ \psi]\!]_e^M = \{c \in \mathcal{C} \mid \exists c', \tau \text{ where } \mathit{TypeOf}(cv) = \tau.\ c \xrightarrow{k?\tau} c' \wedge c' \in [\![\psi[\tau/cv]]\!]_e^M\}$
$[\![\texttt{[}k?cv\texttt{]}\ \psi]\!]_e^M = \{c \in \mathcal{C} \mid \forall c', \exists \tau \text{ where } \mathit{TypeOf}(cv) = \tau.\ c \xrightarrow{k?\tau} c' \Rightarrow c' \in [\![\psi[\tau/cv]]\!]_e^M\}$
$[\![\texttt{<}return(cv)\texttt{>}]\!]_e^M = \{c \in \mathcal{C} \mid \exists n \in \mathbb{N}.c \xrightarrow{a_1} c_1 \xrightarrow{a_2} c_2 \ldots \xrightarrow{a_n} cv\}$
$[\![\texttt{[}return(cv)\texttt{]}]\!]_e^M = \{c \in \mathcal{C} \mid \forall cv', \exists n \in \mathbb{N}.c \xrightarrow{a_1} c_1 \xrightarrow{a_2} c_2 \ldots \xrightarrow{a_n} cv' \Rightarrow (cv = cv')\}$
$[\![\mu X.\psi]\!]_e^M = \bigcap\{C \subseteq \mathcal{C} \mid [\![\psi]\!]_{e[X \mapsto C]}^M \subseteq C\}$
$[\![\nu X.\psi]\!]_e^M = \bigcup\{C \subseteq \mathcal{C} \mid C \subseteq [\![\psi]\!]_{e[X \mapsto C]}^M\}$

where θ is a substitution. Moreover, the K-configuration c' must be part of the meaning of the formula ψ in which each occurrence of cv is replaced by cv'. If the action a is an input communication action involving a value cv, then we must ensure the existence of a type τ such that *TypeOf*$(cv) = \tau$. And the K-configuration c' must be part of the meaning of the formula ψ in which each occurrence of cv is replaced by the type τ.

The meaning of formulae $\texttt{[}a\texttt{]}\ \psi$ are K-configurations c such that after every action a, each result K-configuration c' is part of the meaning of ψ. The meaning of formulae $\texttt{<}return(cv)\texttt{>}$ are K-configurations c that can evolve through n transitions such that the resulting K-configuration is the value cv. In the same way, the meaning of formula $\texttt{[}return(cv)\texttt{]}$ are K-configurations that when they evolve through n transitions, the resulting K-configurations must be the value cv. The meaning of the fixpoint formulae is the same as defined in the μ-calculus. Hence the greatest fixpoint is given as the union of all post-fixpoints whereas the least fixpoint is the intersection of all pre-fixpoints.

8 A Model-checking algorithm

In this section, we present an adaptation of the model checking algorithm proposed by Emerson and Lei [9]. Table 13 contains an algorithm that determines whether or not a structure $M = \texttt{<}\mathcal{ST}, L\texttt{>}$ is a model for a formula ψ_0.

The algorithm follows these three steps:

1. Convert the formula ψ_0 to its equivalent PNF ψ_0'.
2. Compute the set C' of K-configurations in which ψ_0' holds.
3. if $C' \neq \emptyset$ then M is a model for ψ_0 else it's not a model for ψ_0.

Table 14. Symbolic model-checking algorithm

Function $MC(\psi'_0, M)$
  var C', C_i;
begin
  case ψ'_0 of
    tt: $C' = \mathcal{C}$;
    ff: $C' = \emptyset$;
    X: $C' = C_i$;
    $\neg\psi$: $C' = \mathcal{C} \setminus MC(\psi, M)$;
    $\psi_1 \vee \psi_2$: $C' = MC(\psi_1, M) \cup MC(\psi_2, M)$;
    $\psi_1 \wedge \psi_2$: $C' = MC(\psi_1, M) \cap MC(\psi_2, M)$;
    <ϵ> ψ: $C' = \{c \in \mathcal{C} \mid \exists c'.c \xrightarrow{\epsilon} c' \wedge c' \in MC(\psi, M)\}$;
    [ϵ] ψ: $C' = \{c \in \mathcal{C} \mid \forall c'.c \xrightarrow{\epsilon} c' \Rightarrow (c' \in MC(\psi, M))\}$;
    <$k!cv$> ψ: $C' = \{c \in \mathcal{C} \mid \exists c', cv'.c \xrightarrow{k!cv'} c' \wedge c' \in MC(\psi[cv'/cv], M) \wedge cv \preceq cv'\}$;
    [$k!cv$] ψ: $C' = \{c \in \mathcal{C} \mid \forall c', \exists cv'.c \xrightarrow{k!cv'} c' \Rightarrow c' \in MC(\psi[cv'/cv], M) \wedge cv \preceq cv'\}$;
    <$k?cv$> ψ: $C' = \{c \in \mathcal{C} \mid \exists c', \tau$ where $TypeOf(cv) = \tau.$
                $c \xrightarrow{k?\tau} c' \wedge c' \in MC(\psi[\tau/cv], M)\}$;
    [$k?cv$] ψ: $C' = \{c \in \mathcal{C} \mid \forall c', \exists \tau$ where $TypeOf(cv) = \tau.$
                $c \xrightarrow{k!cv'} c' \Rightarrow c' \in MC(\psi[\tau/cv], M)\}$;
    <$return(cv)$>: $C' = \{c \in \mathcal{C} \mid \exists n \in \mathbb{N}.c \xrightarrow{a_1} c_1 \xrightarrow{a_2} c_2 \ldots \xrightarrow{a_n} cv\}$;
    [$return(cv)$]: $C' = \{c \in \mathcal{C} \mid \forall cv', \exists n \in \mathbb{N}.c \xrightarrow{a_1} c_1 \xrightarrow{a_2} c_2 \ldots \xrightarrow{a_n} cv' \Rightarrow (cv = cv')\}$;
    $\mu X.\psi$: $C_i = \emptyset$; repeat $C' = C_i$; $C_i = MC(\psi, M)$; until $C' = C_i$;
    $\nu X.\psi$: $C_i = \mathcal{C}$; repeat $C' = C_i$; $C_i = MC(\psi, M)$; until $C' = C_i$;
  end;
  return(C');
end.

---

9 Conclusion

In this paper, we have considered the problem of formal and automatic verification of data and control aspects for higher-order value-passing process algebra. Our contribution is a new approach that rests on an established synergy between model-checking and type inference. Such a synergy is achieved thanks to three results: First, starting from a concrete dynamic semantics we derive an abstract dynamic semantics. By doing so, we ensure that infinite models will likely be reduced to finite verifiable models. The source of infiniteness are the computable values. The solution is to abstract these values into finite representation that are types. Second, starting from the propositional modal μ-calculus, we define a logic that handles communication, value-passing , result returns, and higher-order objects. The logic is semantically interpreted over the abstract dynamic semantics. Finally, we propose a verification algorithm based on model-checking techniques. Since the model is finite and thanks to the soundness of abstract dynamic semantics, the usual algorithms may be easily accommodated to the model-checking of our logic. We present an accommodation of the Emerson's algorithm.

As future work, we plan to investigate abstraction techniques for dealing with other model infinity sources such as infinite process and channel creation. To that end, we will take advantage of the pioneering work done by D. Schmidt in [22] on the abstract interpretation of small step semantics. Furthermore, we are interested in tracking infinities that may arise from arithmetic manipulation. For that, we will explore the emerging application of Presburger arithmetic to handle this problem. Finally, as a downstream result of this research, we hope to come up with practical tools that address the verification of higher-order concurrent systems.

References

[1] R. M. Amadio and M. Dam. Reasoning about higher-order processes. *In Proc. of CAAP'95, Aarhus, Lecture Notes in Computer Science*, 915, 1995.

[2] B. Berthomieu. Implementing CCS, the LCS experiment. Technical Report 89425, LAAS CNRS, 1989.

[3] D. Bolignano and M. Debbabi. A coherent type inference system for a concurrent, functional and imperative programming language. In *Proceedings of the AMAST'93 Conference*. Springer Verlag, June 1993.

[4] D. Bolignano and M. Debbabi. A semantic theory for CML. In *Proceedings of the TACS'94 Conference*. Springer Verlag, April 1994.

[5] R. Cridlig. Semantic analysis of shared-memory concurrent languages using abstract model-checking. In *Symposium on Partial Evaluation and Program Manipulation*, 1995.

[6] R. Cridlig. Semantic analysis of concurrent ML by abstract model-checking. In *International Workshop on Verification of Infinite State Systems*, 1996.

[7] Mads Dam. Model checking mobile processes. *Information and Computation*, 129(1):35–51, 25 August 1996.

[8] M. Debbabi. *Intégration des paradigmes de programmation parallèle, fonctionnelle et impérative : fondements sémantiques.* PhD thesis, Université Paris Sud, Centre d'Orsay, July 1994.

[9] E. Allen Emerson and Chin-Laung Lei. Efficient model checking in fragments of the propositional mu-calculus (extended abstract). In *Proceedings, Symposium on Logic in Computer*

Science, pages 267–278, Cambridge, Massachusetts, 16–18 June 1986. IEEE Computer Society.

[10] William Ferreira, Matthew Hennessy, and Alan Jeffrey. A theory of weak bisimulation for core CML. *ACM SIGPLAN Notices*, 31(6):201–212, June 1996.

[11] A. Giacalone, P. Mishra, and S. Prasad. Facile: A symmetric integration of concurrent and functional programming. *International Journal of Parallel Programming*, 18(2):121–160, April 1989.

[12] K. Havelund and K. G. Larsen. The fork calculus. In A. Lingas, R. Karlsson, and S. Carlsson, editors, *Proceedings* 20^{th} *ICALP*, volume 700 of *Lecture Notes in Computer Science*. Springer Verlag, 1993.

[13] M. Hennessy and A. Ingólfsdóttir. A theory of communicating processes with value passing. In *Proc. 17th ICALP, LNCS*. Springer Verlag, 1990.

[14] D. Kozen. Results on the propositional mu-calculus. *Theoretical Computer Science*, 23, 1983.

[15] R. Milner. The polyadic π-calculus: A tutorial. Technical report, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, 1991.

[16] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes. Technical report, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, 1989.

[17] Flemming Nielson and Hanne Riis Nielsen. From CML to process algebra. In E. Best, editor, *Proceedings of CONCUR'93*, LNCS 715, pages 493–508. Springer-Verlag, 1993.

[18] Hanne Riis Nielson and Flemming Nielson. Higher-order concurrent programs with finite communication topology. In *Conference Record of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'94)*, pages 84–97, Portland, Oregon, January 17–21, 1994. ACM Press. Extended abstract.

[19] J.H. Reppy. Concurrent programming with events - the Concurrent ML manual. Technical report, Department of Computer Science, Cornell University, November 1990.

[20] J.H. Reppy. CML: A higher-order concurrent language. In *Proceedings of the ACM SIGPLAN '91 PLDI*, pages 294–305. SIGPLAN Notices 26(6), 1991.

[21] D.A. Schmidt. Natural-semantics-based abstract interpretation. In *Proc. 2d Static Analysis Symposium, Glasgow, Sept. 1995*, Lecture Notes in Computer Science 983, pages 1–18. Springer-Verlag, Berlin, 1995.

[22] D.A. Schmidt. Abstract interpretation of small-step semantics. In *Proc. 5th LOMAPS Workshop on Analysis and Verification of Multiple-Agent Languages, Stockholm, June 1996*, Lecture Notes in Computer Science 1192, pages 76–99. Springer-Verlag, Berlin, 1997.

[23] Bent Thomsen. A calculus of higher order communicating systems. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 143–154, Austin, Texas, January 1989.

[24] Bent Thomsen. Plain CHOCS. A second generation calculus for higher order processes. *Acta Informatica*, 30, 1993.