Abstract Interpretation-based Static Safety for Actors

Pierre-Loïc Garoche, Marc Pantel, and Xavier Thirioux
Institut de Recherche en Informatique de Toulouse, France
Email: {garoche,pantel,thirioux}@enseeiht.fr

***Abstract*— The actor model eases the definition of concurrent programs with non uniform behaviors. We present here an encoding of a higher-order actor calculus, CAP, into an abstract interpretation-based framework for the static analysis of mobile systems. Then, we prove that a CAP term and its encoding are bisimilar. Non-trivial properties are captured using existing abstract domains, as well as new ones such as our linearity abstract domain. As far as we know, it is one of the first analyzes that deals with behavioral and occurrence counting properties on a higher-order calculus.**

***Index Terms*— abstract interpretation, concurrent calculus, actor model, safety analyzes**

I. Introduction

A. Context – Motivation

It is now common folklore to say that programming in the context of distributed and concurrent applications, as can be found for instance in the telecommunication industry is an error-prone task. To alleviate this task, the paradigm of concurrent objects is currently employed, in particular for the development of distributed and concurrent applications for the Open Distributed Computing framework (ITU X901-X904) and the Object Description Language (TINA-C extension of OMG IDL with multiple interfaces).

To obtain widely usable tools, we have chosen to use an extended model known as non-uniform concurrent objects, which has been adopted by the telecommunication industry. This model, originally called the actor model, has been proposed by Hewitt [22] and developed by Agha [2]. It is based on a network of autonomous and cooperative agents (called actors), which encapsulate data and programs, communicating using an asynchronous point to point protocol. An actor stores each received message in a mailbox and when idle, processes one it can handle. Besides, an actor can dynamically change its interface, i.e. the set of messages it may handle, accounting for the non uniform flavor.

Since non-determinism resulting from network communications makes it difficult to validate any distributed functionality using informal approaches, our work is focused on applying formal methods to improve actor based programming.

Until now, we have designed several type based analyzes for an actor model. Our main objective was, and still is, to detect in a most accurate way typical flaws of distributed applications, like for instance communication deadlock or non linearity (i.e. the fact that several distributed actors have the same address). Due to limitations in our previous attempts, we decided to move to the framework of abstract interpretation, whose tools and ideas have now significantly grown in maturity and are being used in industrial contexts [1,4]. We now investigate these techniques in order to capture our long standing properties of interest as well as new ones, especially dedicated to control of resources' usage.

In the first section, we define our actor calculus. Then, in the second part, we introduce our non standard semantics upon which we define, in the third part, an abstraction. Finally, in the last part, we explain how to use the abstraction to observe properties about an analyzed term.

B. Related Works

Concerning concurrent objects and actors with uniform or non-uniform behaviors, and more generally process calculi, typing systems (usually related to data-flow like analyzes) have been the subject of active research. Two opposite approaches have been followed: type checking and type inference. Explicit typing may provide precise information but types are sometimes very hard to write for the programmer (they might be much more complex than the program itself). Independently, properties are split in two categories depending whether a resource usage is considered constant during the resource lifetime or not, this last case also being called behavioral.

In the paradigm of type systems, many works focus on behavioral properties and use processes of a simple algebra as types, for instance CCS (Calculus of Communicating Systems) processes. This sometimes allows a form of subtyping through simulation relations or language containment. The works by Kobayashi *et al.* [23,25] aim at ensuring deadlock-freedom and race-freedom for the π-calculus and result in type inference systems. But still, the proposed static analyses don't cope well with recursive programs obtained through replication or other similar operators. Yet, the most recent results [26] lower the number of false alarms at the price of requiring that parameters of a recursive program denote different channels at each call. So, for instance, the authors can conclude that a classical encoding in π-calculus of the factorial function is well typed, whereas a simpler tail recursive

This article is an extended version of [20]

version, without any intermediate channel, remains ill typed. Our own analyzes can conclude that both version are deadlock-free.

In the realm of type checking systems, RAVARA *et al.* [32] want to check a weak form of stuck-freedom in an asynchronous calculus resembling our. RAJAMANI *et al.* [31] also follow this line of thought, bringing model-checking issues for those processes-as-types in the scope, yet their system may be not terminating in some cases. Also the works by NAJM *et al.* [6,27] and PUNTIGAM [30] make use of more exotic types (interface types and trace types respectively) to model behavioral patterns and ensure similar properties. We can also handle such properties, while sticking to an annotation-free framework.

Alternatively, we have flow based algorithms, related to behavior and communication patterns reconstruction, advocated by the works of NIELSON *et al.* [3] and PANTEL *et al.* [7,9,10]. These approaches don't require any user-supplied information but may lead to less precise results. Most of them address simple data-flow properties and the possibility of smoothly extending these works to include more properties is questionable. We are interested here in similar behavioral properties, but our generic framework more easily lends itself to extension.

As for uniform non-behavioral analyzes, most works rely upon unification based typing algorithms focusing on resources' uniform usage control witnessed by the type systems of FOURNET *et al.* [17] and BOUDOL *et al.* [5]. HENNESSY *et al.* [21] introduce sophisticated type features such as dependent and existential types for a higher-order π-calculus, devoted to a fine grained analysis of channel usage. This system, while dealing with higher-order features, requires strong annotations from the user and disallows type inference.

Finally, one drawback of type-based analyzes is that they are mainly concerned with data-flow analyzes (as types basically represent sets of possible values for variables). In this context, control flow analyzes can be mimicked with sophisticated encodings [28] but abstract interpretation seems to be more adequate in this respect. It has been recently applied with success to concurrent and distributed programming by the work of VENET [33] and later FERET [15,16]. Furthermore, analyzing higher-order features using type systems is much more difficult if ever possible, whereas our abstract interpretation framework handles these features quite uniformly with respect to other ordinary process calculi constructs and doesn't incur unbearable extra complexity.

II. CAP: A PRIMITIVE ACTOR CALCULUS

In order to ease the definition of static analysis for actor based programming, we proposed, in 96, the CAP primitive actor calculus [8], which merges asynchronous π-calculus and CARDELLI's Primitive Object Calculus. The following example illustrates both replication and behavior passing mechanisms in CAP. To facilitate the reading and describe the evolution of the term, we automatically annotate the term with program points. We will then describe transitions among threads using these program points. The ν operator defines two addresses, a and b, then two actors denoted by program points 1 and 7 are defined on those addresses with their behavior sets respectively denoted by 2 and 4 for a and 8 for b. In each of such behaviors, the ζ operator is a reflexivity operator. It allows to bind in the continuation both the actor's name and its set of behaviors. For example, e and s defined at point 5 are respectively associated with values α and $\{2,4\}$, denoted here by 1, after the first transition between actor 1 and message 6.

At this point the actor 1 can handle messages called m or *send* when b can only handle *beh* messages.

$$\begin{array}{lll} \nu a^{\alpha}, b^{\beta}, & a \rhd^{1} [m^{2}() = \zeta(e,s)(a \rhd^{3} s), & \\ & \qquad send^{4}(x) = \zeta(e,s)(x \lhd^{5} beh(s))] & \\ \quad || & a \lhd^{6} send(b) & \\ \quad || & b \rhd^{7} [beh^{8}(x) = \zeta(e,s)(e \rhd^{9} x)] & \\ \quad || & b \lhd^{10} m() & \end{array}$$

There are also two messages in the initial configuration. One is labeled *send* and is sent to a, the other one is labeled m and is sent to b. In the initial configuration, there is only one possible interaction; the actor a handles the message *send*. The message m is an orphan one: it is in the configuration but cannot be handled at the moment. After one interaction between a and the message *send*, the message *beh* whose argument is the behavior set of a is sent to b. Thus b can handle that message. In its continuation, the actor b assumes the behavior set of a. Thus b can now handle the message m. This example shows how to send a behavior to another actor. Such a mechanism increases the difficulty of statically inferring properties. An source of difficulty in the analysis is the non-determinism of our term. If multiple messages are present in a configuration for the same actor, it can take any one of them when computing a transition. For example, let us add another message $b \lhd^{11} m()$ in the initial configuration[1]. The last reduction can either consume the message labeled 10 or the message labeled 11. The other message will then stay in the configuration. Another source of non-determinism is when multiple behavior description in an actor are able to handle message with the same label are arguments arity.

Stuck-freeness analysis, *i.e.* the detection of the set of permanent orphan messages, or linearity analysis, *i.e.* verifying that at most one actor is associated to a particular address at the same time, are harder to statically infer when we allow behavior passing. This point was one of the constraints which led us to switch from type based analysis to abstract interpretation.

A. Syntax and Semantics

Let $\mathcal{N}$ be an infinite set of actor names, $\mathcal{V}$ be an infinite set of variables. Let $\mathcal{L}_m$ be a finite set of message

[1]Program points allow to disambiguate parts of the term in the latter formalization but are not used in the current use of CAP.

$$\frac{T = [m_i^{l_i}(\widetilde{x_i}) = \zeta(e_i,s_i)C_i^{\,i=1,\ldots,n}] \quad \begin{cases} k \in [1,\ldots,n], \\ m = m_k, \\ \mid \widetilde{T_l} \mid = \mid \widetilde{x_k} \mid \end{cases}}{a \rhd T \;||\; a \lhd^l m(\widetilde{T_l}) \xrightarrow{(l,l_k)} C_k[e_k \leftarrow a, s_k \leftarrow T, \widetilde{x_k} \leftarrow \widetilde{T_l}]}$$

In order to distinguish transitions, we label the interacting parts of terms. Here the message has label l and the matching behavior label l_k.

Figure 1. Transition rule of CAP standard semantics

labels, $\mathscr{L}_p$ be the finite set of program point labels and $\mathscr{L}_n$ be the finite set of name labels. In the following, behaviors will be described by their thread program point. Values of variables will then denote either names or behaviors. Let $\mathscr{L} = \mathscr{L}_p \cup \mathscr{L}_n$ be the set of variable values. The syntax of configurations is described as follows: Let $a \in \mathscr{N}$, x,k, s.t. $k \in \widetilde{Var}, \in \mathscr{V}$, $m, m_i \in \mathscr{L}_m$, $\alpha \in \mathscr{L}_n$ and $l, l_i \in \mathscr{L}_p$.

$$\begin{array}{lcl} C & ::= & 0 \mid \nu a^\alpha\, C \mid C \,||\, C \mid a \rhd^l P \mid a \lhd^l m(\widetilde{P}) \\ P & ::= & x \quad\mid\quad [m_i^{l_i}(\widetilde{Var}) = \zeta(e,s)C_i^{\,i=1\ldots n}] \end{array}$$

Configurations can be: an empty process, a creation of an actor address, parallel execution, installation of an actor on address a with behavior defined by P and, finally, a message sent to an address a with arguments $\widetilde{P}$. Program points define messages, behavior installations or external choices between some actor behaviors. They will be used to build traces of the execution control flow.

Name binders are defined the ν and ζ operators as well as message label in behavior branches. In the configuration $(\nu a^\alpha)C$, the name a is bound in C. In a behavior branch $m_i^{l_i}(\widetilde{Var}) = \zeta(e,s)C$, the operator $\zeta(e,s)C$ binds variables e and s in C and the message label m_i binds all $\widetilde{Var}$ variables in C. The ζ operator is our reflexivity operator, it catches both address and behavior of its actor and allows to re-use them in the behavior. We denote by $\mathcal{FN}(C)$ the set of free names in C and by $\mathcal{FV}(C)$ the set of free variables. We consider only closed term. The standard semantics of CAP was defined using, as usual, a transition rule (*cf.* Fig. 1), a congruence relation (*cf.* Fig. 2) and a set of context rules (*cf.* Fig. 3). Our presentation differs from the standard Milner's semantics for process calculi using LTS. We have a single unlabeled reduction rule. We annotate it with a tuple of program point in order to ease the later formalization.

III. Non Standard Semantics

In order to ease the definition of abstract interpretations, we need to define, in this section, another semantics for CAP and prove it bisimilar to standard CAP semantics. The non standard semantics allows us to label each process with the history of transitions which led to both its creation and the creation of its values. Our work is based on a generic non standard semantics which has been defined by Feret [15,16] to model first order process calculi as π-calculus, spi-calculus, Ambients, Bio-ambients calculus. We also describe in this section how

$$\begin{array}{rcll} C & \equiv & D & \text{C } \alpha\textit{-convertible to D} \\ C||0 & \equiv & C & \\ C||D & \equiv & D||C & \\ (C||D)||E & \equiv & C||(D||E) & \\ (\nu a^\alpha)\varnothing & \equiv & \varnothing & \\ a \rhd T_1 & \equiv & a \rhd T_2 & \text{if } T_1 \equiv T_2 \\ (\nu a^\alpha)(\nu b^\beta)C & \equiv & (\nu b^\beta)(\nu a^\alpha)C & \textit{if } a \neq b \\ (\nu a^\alpha)C||D & \equiv & (\nu a^\alpha)(C||D) & \textit{if } a \notin \mathcal{FN}(D) \\ (\nu a^\alpha)C||D & \equiv & (\nu a^\alpha)(C||D) & \textit{if } a \notin \mathcal{FN}(D) \\ [m_i(\widetilde{x_i}) = \zeta(e_i,s_i)C_i^{\,i=1,\ldots,n}] & \equiv & & \rho \textit{ a permutation} \\ & & [m_{\rho(i)}(\widetilde{x_{\rho(i)}}) = \zeta(e_{\rho(i)},s_{\rho(i)})C_{\rho(i)}^{\,i=1,\ldots,n}] & \end{array}$$

Figure 2. Congruence relation of CAP standard semantics

$$\frac{D \equiv C \quad C \longrightarrow C' \quad C' \equiv D'}{D \longrightarrow D'}\ STRUCT$$

$$\frac{C \longrightarrow C'}{C \;||\; D \longrightarrow C' \;||\; D}\ PAR \qquad \frac{C \longrightarrow C'}{\nu x C \longrightarrow \nu x C'}\ RES$$

Figure 3. Context rules

we adapt this general framework to express the CAP language which has a notion of higher order due to its behavior passing and reflexivity mechanism (ζ operator). We then briefly describe the operational semantics of the generic non standard semantics.

A configuration of a system, in this semantics, is a set of threads. Each thread t is a triple defined as $t = (p,id,E) \in \mathscr{L}_p \times \mathscr{M} \times (\mathscr{V} \mapsto (\mathscr{L} \times \mathscr{M}))$ where p is the program point representing the thread in the CAP term, id is its history marker, also called its identity, and E its environment. This environment is a partial map from a variable to a pair $(value, marker)$. Each marker is a word on program points representing the history of transitions which led to the creation of values or threads. It is required in order to differentiate recursive instances of a value or thread. All threads with the same program point have an environment defined on the same domain, called the program point interface.

We will describe some primitives that allow us to define the non standard semantics, then, briefly, we show how to compute transitions in this semantics. We associate to each program point a set of partial interactions which define how threads related to this program point can interact with others. We also define the set of variables associated to each thread, constituting its environment, according to its program point. An extraction function will then compute for each CAP subterm the appropriate set of partial interaction and of interface variables depending on it syntax.

A. Partial Interactions

Here, in CAP, partial interactions can represent a syntactically defined actor, one of its particular behavior, a dynamic actor (an actor whose behavior is defined by a variable) or a sent message. This encoding, differentiating syntactic actors from dynamic ones, allows us to deal with

behavior passing. Once a behavior has been declared, it is present in the configuration as a behavior thread, while a reference to it is used in messages. Dynamic actors could then use such a behavior if they are associated with their corresponding reference.

We thus define the set of partial interaction names $\mathscr{A} = \{static\_actor_n, behavior_n, message_n \mid n \in \mathbb{N}\} \cup \{dynamic\_actor\}$ and their arities as follows:

$$Ari = \left\{ \begin{array}{rcl} static\_actor_n & \mapsto & (2, n+2), \\ behavior_n & \mapsto & (1, n+2), \\ dynamic\_actor & \mapsto & (2, 0), \\ message_n & \mapsto & (n+2, 0) \end{array} \right\}$$

Partial interaction arities define the number of parameters and the number of bound variables.

Both partial interaction $static\_actor_n$ and $behavior_n$ denote a particular behavior of an actor. The first one is associated to an address when the second one is alone and can be used with a dynamic actor. The second one acts as a definition and stays in the configuration when used, whereas the first one is deleted. They are parametrized by their message labels and bind $n+2$ variables, the variables under the ζ operator expressing reflexivity as well as the parameters of the message it can handle. The first one is also parametrized by its actor name.

The partial interaction $dynamic\_actor$ denotes a thread representing an actor. It is consumed when interacting. It has only two parameters: its name and set of behaviors. It binds no variables.

Finally the partial interaction $message_n$ represents the message that is sent to a particular address (actor). So it has $n+2$ parameters: one for the address, one for the message name and n for the variables of this message. It is consumed when interacting.

We associate to each partial interaction a type denoting whether such a partial interaction is consumed or not when interacting.

B. Abstract Syntax Extraction

We now define the syntax extraction function that takes a CAP term describing the initial state of a system of agents in the standard syntax and extracts its abstract syntax.

We map each program point labeled $l \in \mathscr{L}_p$ to a set of partial interaction and to an interface.

A partial interaction pi is given by a tuple $(s, (parameter_i), (bound_i), constraints, continuation)$ where $s \in \mathscr{A}$ is a partial interaction name, $(m,n) = Ari(s)$ its arity, $(parameter_i) \in \mathscr{V}^m$ its finite sequence of variables (X_i), $(bound_i) \in \mathscr{V}^n$ its finite sequence of distinct variables (Y_i), $constraints \subseteq \{v \diamond v' \mid (v,v') \in \mathscr{V}^2, \diamond \in \{=, \neq\}\}$ its synchronization constraints and finally $continuation \in \wp(\mathscr{L}_p \times (\mathscr{V} \to \mathscr{L}))$ its syntactic continuation. Such a continuation is called syntactic because it has to be updated with value passing to be a valid term of the process calculus. We check constraints about thread environment defined in the set $constrains$ with the use of the sequence $(parameter_i)$, then we use both sequences $(parameter_i)$ and $(bound_i)$ to compute value passing, finally we deal with the set $continuation$ to determine which threads have to be inserted in the system.

- the label of a program point $a \rhd^l [m_i^{l_i}(\tilde{x_i}) = \zeta(e_i, s_i)C_i^{\,1 \leqslant i \leqslant m}]$ is associated to the interface $\{a\}$ and to the following set of partial interactions:

$$\left\{ \begin{array}{l} \left\{(static\_actor_n, [a, m_1], [e_1, s_1, \tilde{x_1}], \beta(C_1, \varnothing))\right\} \\ \left\{(static\_actor_n, [a, m_2], [e_2, s_2, \tilde{x_2}], \beta(C_2, \varnothing))\right\} \\ \ldots \\ \left\{(static\_actor_n, [a, m_m], [e_m, s_m, \widetilde{x_m}], \beta(C_m, \varnothing))\right\} \end{array} \right\}$$

- the label of a program point $a \rhd^l x$ is associated to the interface $\{a, x\}$ and to the following set of partial interactions: $\left\{(dynamic\_actor, [a, x], \varnothing, \varnothing)\right\}$
- the label of a program point $a \lhd^l m(\tilde{P})$ is associated to the interface $\{a\} \cup \mathcal{FV}(\tilde{P})$ and to the following set of partial interactions: $\left\{(message_n, [a; m; \tilde{P}], \varnothing, \varnothing)\right\}$
- the label of a program point l_i corresponding to a particular behavior of an actor *i.e.* $m_i^{l_i}(\tilde{x}) = \zeta(e_i, s_i)C_i$ is associated to the interface $\mathcal{FV}(C_i) \backslash \{e_i, s_i\}$ and to the following set of partial interactions: $\left\{(behavior_n, [m_i], [e_i, s_i, \tilde{x}], \beta(C_i, \varnothing))\right\}$

Finally, the syntax extraction function β is defined inductively over the standard syntax of the syntactic continuation, as follows:

$$\begin{array}{rcl} \beta((\nu a^\alpha)C, E_s) & = & \beta(C, E_s[a \mapsto \alpha]) \\ \beta(\varnothing, E_s) & = & \{\varnothing\} \\ \beta(C_1 || C_2, E_s) & = & \beta(C_1, E_s) \cup \beta(C_2, E_s) \\ \beta(a \rhd^l [m_i^{l_i}(\tilde{x_i}) = \zeta(e_i, s_i)C_i^{\,i=1,\ldots,n}], E_s) & = & \{(l, E_s)\} \cup \bigcup_{i=1,\ldots,n} \{(l_i, E_s)\} \\ \beta(a \rhd^l B, E_s) & = & \{\{(l, E_s)\}\} \\ \beta(a \lhd^l m(\tilde{P}), E_s) & = & \{\{(l, E_s)\}\} \end{array}$$

One could notice that the syntax extraction of a syntactically defined actor generates not only the thread associated with the actor but also all behavior threads defining all behavior branches of the actor. The extraction of a message in which at least one of the variables is a syntactically defined behavior would generate a similar set of behavior threads.

The initial state for a term $\mathscr{S}$ is described by $init_s$, a set of potential continuations in $\wp(\wp(\mathscr{L}_p \times (\mathscr{V} \to \mathscr{L})))$ defined as $\beta(\mathscr{S}, \varnothing)$.

C. Formal Rules

We now define the formal rules that drive the interaction between threads. In the case of CAP, we have two rules that describe an actor handling a message, depending on the kind of actor we have, a static or a dynamic one.

In the following, the i-th parameter, the j-th bound variable, and the identity of the k-th partial interaction are

respectively denoted by X_i^k, Y_j^k and I^k. We define the endomorphism *behavior_set* on the set $\mathscr{L}_p \times \mathscr{M}$ as follows: $(p,m) \mapsto (p',m)$ where p is a behavior program point and p' is the program point where p has been syntactically defined. As an example, in the term $\nu^{\alpha} a, a \rhd^1 [foo^2() = \zeta(e,s)C]$, we have $behavior\_set(2,marker) = (1,marker)$.

a) Communication with a static actor: The first rule needs two threads, the first one must denote a partial interaction *static_actor* when the second one must denote a partial interaction $message_n$. We both check that the actor's address (X_1^1) is equal to the message's receiver (X_1^2) and that the actor behavior name (X_2^1) is equal to the message name (X_2^2).

We then define *v_passing* that describe the value passing due to both the ζ operator and message handling.

$$static\_trans_n = (2, components, compatibility, v\_passing)$$

where

$$1)\ components = \left\{ \begin{array}{lcl} 1 & \mapsto & static\_actor_n, \\ 2 & \mapsto & message_n \end{array} \right.$$

$$2)\ compatibility = \left\{ \begin{array}{l} X_1^1 = X_1^2; \\ X_2^1 = X_2^2; \end{array} \right.$$

$$3)\ v\_passing = \left\{ \begin{array}{l} Y_1^1 \leftarrow X_1^1; \\ Y_2^1 \leftarrow I^1; \\ Y_{i+2}^1 \leftarrow X_{i+2}^2, \forall i \in [\![1;n]\!]; \end{array} \right.$$

b) Communication with a dynamic actor: The second rule needs three threads: the first one must denote a partial interaction $behavior_n$, the second one a partial interaction *dynamic_actor* and the third one a message $message_n$. We check the equality between actor's address (X_1^2) and receiver (X_1^3), behavior name (X_1^1) and message name (X_2^3). With the *behavior_set* function we check the link between the behavior and the actor. The value passing is defined in the same way as in the first rule.

$$dyn\_trans_n = (3, components, compatibility, v\_passing)$$

where

$$1)\ components = \left\{ \begin{array}{lcl} 1 & \mapsto & behavior_n, \\ 2 & \mapsto & dynamic\_actor, \\ 3 & \mapsto & message_n \end{array} \right.$$

$$2)\ compatibility = \left\{ \begin{array}{l} X_1^2 = X_1^3; \\ behavior\_set(I^1) = X_2^2; \\ X_1^1 = X_2^3; \end{array} \right.$$

$$3)\ v\_passing = \left\{ \begin{array}{l} Y_1^1 \leftarrow X_1^2; \\ Y_2^1 \leftarrow X_2^2; \\ Y_{i+2}^1 \leftarrow X_{i+2}^3, \forall i \in [\![1;n]\!]; \end{array} \right.$$

D. Operational Semantics

We now briefly describe how to use the preceding definitions to express in the non standard syntax both an initial term and the computation of a transition according to one of the aforementioned rules. We recall that our calculus is embedded into the generic framework of FERET [16], which provides a single syntax and a single parametrized semantics and can encode various calculi. It allows to explicate the control flow of the system while representing a configuration as a set of threads. Therefore the current operational semantics is not based on a labeled transition relation and a congruence relation but rather on a set of reduction rules, guided by our formal rules that can be applied on any tuple of threads.

Initial configurations are obtained by launching a continuation in $init_s$ with an empty marker and an empty environment. That means inserting in an empty configuration one thread for each pair (p,E_s) in $\beta(init_s)$ where each value in E_s is associated with an empty marker. We focus now on the interaction computation according to one of the two rules. First of all, we have to find some correct interaction. It means that we have to find some threads in the current configuration that can be associated to the right partial interaction according to the matching formal rule. Then, we check that their interfaces satisfy the synchronization constraints. Thus we can compute the interaction. All these steps are performed using the primitives which are mentioned in parentheses:

- we remove interacting threads according to the type of their exhibited partial interactions (exhibits, components);
- we choose a syntactic continuation for each thread;
- we compute dynamic data for each of these continuations (sync):
  - we compute the marker (marker);
  - we take into account name passing (vpassing);
  - we create fresh variables and associate them with the correct values (launch);
  - we restrict the environment according to the interface associated with the program point (launch);
- we remove interacting threads depending on their types (remove).

An explicit definition is given in Fig. 4.

E. Example

To illustrate the use of the non standard semantics, we compute the first transition of the example given in section II.

The initial configuration[2] is:

$$(1,\varepsilon,\left[\begin{array}{lcl} a & \mapsto & \alpha,\varepsilon \end{array}\right]) \quad (2,\varepsilon,\left[\begin{array}{lcl} a & \mapsto & \alpha,\varepsilon \end{array}\right]) \quad (4,\varepsilon,[])$$
$$(6,\varepsilon,\left[\begin{array}{lcl} a & \mapsto & \alpha,\varepsilon \\ b & \mapsto & \beta,\varepsilon \end{array}\right]) \quad (7,\varepsilon,\left[\begin{array}{lcl} b & \mapsto & \beta,\varepsilon \end{array}\right]) \quad (8,\varepsilon,[])$$
$$(10,\varepsilon,\left[\begin{array}{lcl} b & \mapsto & \beta,\varepsilon \end{array}\right])$$

At this point, the only possible transition is labeled by $1,6$ and corresponds to the $static\_trans_n$ rule. Program point 1 is able to exhibit the two following partial interactions:

$$\left\{ \begin{array}{l} \left\{(static\_actor_n, [a,m], [e,s], \beta(a \rhd^3 s, \varnothing))\right\}, \\ \left\{(static\_actor_n, [a,send], [e,s,x], \beta(x \lhd^5 beh(s), \varnothing))\right\} \end{array} \right\}$$

when the program point 6 exhibits the only partial interaction:

$$\left\{(message_n, [a,send,b], \varnothing, \varnothing)\right\}$$

[2]We can notice the absence of threads at program points 3, 5 and 9 which correspond to sub-terms. There are not present in the initial configuration.

Let C be a configuration. Let $\mathscr{R} = (n, components, compatibility, v\_passing)$ be a formal rule. Let us be given a tuple $(t^k)_{1\leqslant k\leqslant n} = (p^k, id^k, E^k)_{1\leqslant k\leqslant n} \in C^n$ of distinct threads and a tuple $(pi^k)_{1\leqslant k\leqslant n} = (s^k, (parameter_l)^k, (bd_l)^k, constraints^k, continuation^k)_{1\leqslant k\leqslant n}$ of partial interactions, such that:

1) $\forall k \in [\![1;n]\!], exhibits(t^k, pi^k)$;
2) $\forall k \in [\![1;n]\!], components(k) = s^k$;
3) $sync((t^k)_k, ((parameter_l)^k)_k, compatibility) \neq \bot$.

Then

$C \xrightarrow{(\alpha^i)^n} (C \setminus removed\_threads) \cup news\_threads$

with:

- $removed\_threads = remove\Big((t^k)_{1\leqslant k\leqslant n}\Big)$;
- $new\_threads = \bigcup_{1\leqslant k\leqslant n} launch\Big(Ct^k, \overline{id}, \overline{E}^k\Big)$, where $\forall k \in [\![1;3]\!]$:
  - $Ct_k \in continuation^k$;
  - $\overline{id} = marker\Big((p^{k'}, id^{k'}, E^{k'})_{1\leqslant k'\leqslant n}\Big)$;
  - $\overline{E}^k = vpassing\Big(k, (t^{k'})_{1\leqslant k'\leqslant n}, ((bd_l)^k)_k, ((parameter_l)^k)_k, communications\Big)$.
- $\forall k \in [\![1;n]\!], \alpha^k = (t^k, pi^k, E^k)$.

Figure 4. Non standard operational semantics

We choose the second partial interaction for 1. We first check synchronization constraints. We need that $X_1^1 = X_1^2$ and $X_2^1 = X_2^2$. So $(\alpha,\varepsilon) = (\alpha,\varepsilon)$ and both threads share the same label *send*. We can now compute value passing, thread launching and removing. We have to remove interacting threads and to add threads in $\beta(x \lhd^5 beh(s), \varnothing)$ with their environments updated by value passing. Value passing gives the value of e, s and x, we have respectively (α,ε), $(1,\varepsilon)$ and (β,ε). Thus the launched thread is $(5,\varepsilon,\begin{bmatrix} x & \mapsto & \beta,\varepsilon \\ s & \mapsto & 1,\varepsilon \end{bmatrix})$.

We obtain the new configuration:

$$(2,\varepsilon,[\ a \ \mapsto \ \alpha,\varepsilon\ ]) \quad (4,\varepsilon,[]) \quad (5,\varepsilon,\begin{bmatrix} x & \mapsto & \beta,\varepsilon \\ s & \mapsto & 1,\varepsilon \end{bmatrix})$$
$$(7,\varepsilon,[\ b \ \mapsto \ \beta,\varepsilon\ ]) \quad (8,\varepsilon,[]) \quad (10,\varepsilon,[\ b \ \mapsto \ \beta,\varepsilon\ ])$$

We recall that when computing a transition using the $dynamic\_trans_n$ rule, new launched threads are associated to a new marker.

F. Correspondence

1) Translation: We need to prove the correspondence between CAP semantics and its expression in the meta language. In the following, pi denotes a partial interaction.

Lemma 1 (non standard term well-formedness): Let C be a non standard term. C is a set of triples $\{(p,id,E)\}$ where $p \in \mathscr{L}_p$ denotes a program point, $id \in \mathscr{M}$ a marker and $E \in \wp(\mathscr{V} \to (\mathscr{L} \times \mathscr{M}))$ its environment. Let *interaction* be the partial map from program points to partial interactions. Let *behavior_set* be the partial map defined upon the term defined as in III-C which maps a value denoting a syntactically defined actor program point to its set of behavior program points.

Every term C is well formed, that is

1) $\forall (p,id,E) \in C$, $interaction(p) \neq \varnothing$ and $\forall (name, var, param, cont) \in interaction(p)$, $\forall v \in var, E(v)$ is defined;
2) $\forall (p,id,E) \in C$, such that $\forall pi \in interaction(p)$, pi is a *static_actor* partial interaction, then $|var_{pi}| = 2$ and there is, in the system, exactly one thread (p_i, id, E_i) for each associated behavior program point p_i. Each of those threads must exhibit a $behavior_n$ pi.
3) $\forall (p,id,E) \in C$, such that $\exists! pi \in interaction(p)$ and pi is a *dynamic_actor* partial interaction, then $|var_{pi}| = 2$ and let s be the 2nd variable of var_{pi}. There is, in the system, exactly one thread $(p_i, snd(E(s)), E_i)$ for each $p_i \in behavior\_set(E(s))$. Each of those partial interactions must exhibit a $behavior_n$ pi.
4) $\forall (p,id,E)$, such that $interaction(p)$ is a $behavior_n$ pi, then $|var_{pi}| = 1$.
5) For each variable denoting a behavior, threads associated to this value must be present in the system and share the same marker as the one of the variable.

Proof: The proof can be made by induction on created terms issued from the initial state $\beta(\mathscr{S}, \varnothing)$. We give here only proof sketches of the different cases:

1) Initial threads as well as every created threads are computed using the β function and correspond to either static actors, dynamic ones, or messages. Therefore they are able to exhibits partial interaction as defined in III-B. We recall that we only consider closed term. Then, the initial threads are defined after a sequence of ν operators that bind their variables. By induction on the number of computed transition, one can show that each transition using one of our two formal rules preserve the property: each variable used in one thread partial interaction is defined and bound either by an internal ν operator, a ζ operator, a message argument or previously in the matching actor.
2) By definition of the abstract syntax extraction in III-B. Each actor on program point l associated with a syntactic behavior $(\{l_i\})$ is mapped, by the β function, to one thread (l,E) and a set of threads (l_i,E). Each of those static threads will then be launched by the *launch* primitive. By definition, all the behavior branches will be associated with threads that exhibit behavior partial interaction.
3) Similar to the previous case. Using the result of 1, the variable s is defined and denotes the program point of the static actor defining the behavior.
4) By definition of the syntax extraction.
5) All threads representing the different branches of a behavior are launched together. Therefore they are all presents and share the same marker.

■

To simplify the translation and allow us to differentiate between recursive instances of the same variable

declaration (*i.e.* name binder), we define an auxiliary function f which maps each pair $(p,m) \in \mathscr{L} \times \mathscr{M}$ to the name p^m iff p is the label corresponding to a term νa^p and the term $[m_i^{l_i}(\tilde{x_i}) = \zeta(e_i,s_i)C_i^{i=1,\ldots,n}]$ iff p is a term $a \rhd^p [m_i^{l_i}(\tilde{x_i}) = \zeta(e_i,s_i)C_i^{i=1,\ldots,n}]$. Such a function also allows us to replace dynamically defined actors by their static equivalent ones. The α-conversion rule of the CAP congruence relation allows us to rearrange the term.

Let $\{c_i \mid i \in [\![1;k]\!]\}$ defined as $\{f(E(x))|(p,id,E) \in C, E(x)$ *defined,and* $fst(E(x))$ *a name*$\}$ be the set of actors' names used in the term.

We define a translation function Π which maps a set of thread denoting a well formed non standard configuration C to the corresponding CAP configuration. The Π function is defined by:

$$\Pi(C) = (\nu c_1)\ldots(\nu c_k)(M_1 \parallel \ldots \parallel M_p \parallel A_1 \parallel \ldots \parallel A_q)$$

We define C as the disjoint union of M, A and B:

$$C = M \cup A_s \cup A_d \cup B$$

where M is the set of threads associated to $message_n$ partial interactions, A_s the set of threads associated to $static\_actor_n$ partial interactions, A_d the set of threads associated to *dynamic_actor* partial interactions, and, finally, B the set of threads associated to $behavior_n$ partial interactions.

The Π function can be recursively applied on the set of threads in C, each iteration computing a message or an actor of the resulting CAP term. We construct $\{M_i\}$ and $\{A_j\}$ as follows:

- $i \in [\![1;Card(M)]\!]$
- M_i is the message $a \lhd^l (\tilde{x})$ corresponding to the translation of the thread $(l,id,E) \in M$.
- $j \in [\![1;Card\,(A_s \cup A_d)]\!]$
- A_j is the actor $a \rhd^l [m_i^{l_i}(\tilde{x_i}) = \zeta(e_i,s_i)C_i^{i=1,\ldots,n}]$ associated to the thread $(l,id,E) \in A_s \cup A_d$.
  - when $(l,id,E) \in A_s$, the actor is obtained from the sub-term associated to the program point l;
  - when $(l,id,E) \in A_d$, the actor is composed of all behaviors represented by its associated threads $(l_i,id,E) \in B$; *i.e.* behavior threads which program points are associated by *behavior_set* function to the second variable value of the actor thread and which markers are equal to the one of this same variable. More formally, the actor's thread second variable is associated to the pair (l',id'), $id = id'$ and all l_i are associated to l' by the *behavior_set* function.

  As the C term is well formed, when a dynamic actor is in the system, all the behaviors of its behavior set must be in the system too.

Finally, in both M_i and A_j, we update E. We replace a and each free variable of $\tilde{x}$, respectively a and each free variable in all C_i, with their images by function f: $x \mapsto f(E(x))$.

The translation system is well defined thanks to the congruence rules: associativity, commutativity and swapping.

2) Correspondence: The following theorem states that CAP standard semantics and its non standard semantics are in strong bisimulation. They share equivalent initial states and each possible set of transitions from the initial state in the non standard semantics (respectively in the standard one) is computable in the standard one (respectively in the non standard one).

Theorem 2: We have $\mathscr{S} \equiv \Pi(\mathcal{C}_0)$, and for each non standard configuration C and for each word $u \in (\mathscr{L}^2 \cup \mathscr{L}^3)^*$ such that $\mathcal{C}_0 \xrightarrow{u}{}^* C$:

1) $\forall \lambda \in (\mathscr{L}^2 \cup \mathscr{L}^3), C \xrightarrow{\lambda} C' \Longrightarrow \Pi(C) \xrightarrow{\lambda} \Pi(C')$;
2) $\forall \lambda \in (\mathscr{L}^2 \cup \mathscr{L}^3), \Pi(C) \xrightarrow{\lambda} P \Longrightarrow \exists D, \begin{cases} C \xrightarrow{\lambda} D \\ \Pi(D) \equiv P \end{cases}$

Proof: Let C be a non standard configuration and let u be a word in $(\mathscr{L}^2 \cup \mathscr{L}^3)^*$ such that $\mathcal{C}_0 \xrightarrow{u}{}^* C$,

1) Let C' be a non standard configuration such that $C \xrightarrow{\lambda} C'$. Suppose that C contains only interacting threads. Our coding contains two transition rules, we check the property for each. In order to simplify the proof, we denote by $\mathcal{E}$ and $\mathcal{S}$ the address and behavior set of the interacting actor and by $\mathcal{M}$ the environment of the interacting message.
   a) **Static-trans rule.** Such a rule defines the interaction between two threads. Necessarily, C must contain:

   $$(p_1,id_1,E_1) \qquad (p_2,id_2,E_2)$$

   where partial interactions associated to program points are:

   - p_1: $(static\_actor_n, [ego, label], [e_i, s_i, \tilde{y_k}], \beta(C_i,\varnothing))$
   - p_2: $(message_n, [dest, label, \tilde{x_l}], \varnothing)$

   with $|\tilde{y_k}| = |\tilde{x_l}| = n$. Then, we have the following relations $E_1(ego) = E_2(dest)$ and $E_1(label) = E_2(label)$. The C' term obtained after transition $\lambda = (p_1,p_2)$ is:

   $$\left\{ \begin{array}{l} (p_j,id_1,E_j) \text{ s.t. } (p_j,Es_j) \in \beta(C_i,\varnothing) \text{ and} \\ E_j = Es_j \left[\begin{array}{l} e_i \mapsto E_2(ego) \\ s_i \mapsto I^1 = (p_1,id_1) \\ \forall k \in [\![1;n]\!], \\ y_k \mapsto E_2(x_k) \end{array} \right] \end{array} \right\}$$

   Standard configuration $\Pi(C')$ is the closed term composed of the set of messages and actors with their appropriate behaviors, as defined by the C' term. Let us study each C_i case by induction on CAP syntax. We have here $\mathcal{E} = E_1(ego)$, $\mathcal{S} = I_1$ and $\mathcal{M} = E_2$.

   - when $C_i = a \rhd^{p_a} [m_l^{p_{m_l}}(\tilde{z_l}) = \zeta(e_l,s_l)C_l]$, then, by definition of β, $\beta(C_i,\varnothing) = (p_a, [self \mapsto p_a]) \cup \bigcup (p_{m_l}, [\,])$.
     The *exhibits* function maps each program point p_{m_l} to a partial interaction:

     $$(behavior_n, [m_l], [e_l, s_l, \tilde{z_l}], \beta(C_l,\varnothing))$$

After value passing, we obtain for C':

$$\left\{\left(p_a, new\_id, E_a\left[\begin{array}{l|l} e_i \mapsto \mathcal{E} & \\ s_i \mapsto \mathcal{S} & \text{such that } E_a(x) \\ \forall k \in [\![1;n]\!], & \text{is defined iff} \\ y_k \mapsto \mathcal{M}(x_k) & x \in interface(p_a) \\ x \mapsto E_1(x) & \end{array}\right]\right)\right\}$$

$$\cup \bigcup_l \left\{\left(p_l, new\_id, E_l\left[\begin{array}{l|l} e_i \mapsto \mathcal{E} & \\ s_i \mapsto \mathcal{S} & \text{such that } E_l(x) \\ \forall k \in [\![1;n]\!], & \text{is defined iff} \\ y_k \mapsto \mathcal{M}(x_k) & x \in interface(p_l) \\ x \mapsto E_1(x) & \end{array}\right]\right)_l\right\}$$

The term translation by Π function gives

$$\nu\tilde{c}, a \rhd^{p_a} [m_l^{p_l}(\tilde{z_l}) = \zeta(e_l, s_l)C_l]$$

where $\tilde{c} = \{x \mid E_a(x)$ where $E_l(x)$ is defined and denotes a ν program point $\}$.

- when $C_i = a \rhd^{p_a} b$, then necessarily, the set of behaviors denoted by the b variables has already been defined and is in the current configuration. We recall that we only consider closed terms. Therefore b is defined and denotes a set of behaviors of the past. The resulting term is similar to the preceding case without introducing behavior threads on program points p_i.
- when $C_i = a \lhd^{p_a} m(\tilde{z})$, then by definition of β, $\beta(C_i, \varnothing) = (p_a, \varnothing)$. The *exhibits* function gives:

$$(message_n, [a, m, \tilde{z}], [\,], \varnothing)$$

The p_a interface is $\{a\} \cup \mathcal{FN}(\tilde{z})\}$. The only thread we obtain, in the resulting term C', once the value passing is computed, is:

$$\left\{\left(p_a, new\_id, E\left[\begin{array}{l|l} e_i \mapsto \mathcal{E} & \\ s_i \mapsto \mathcal{S} & \text{such that } E(x) \\ \forall k \in [\![1;n]\!], & \text{is defined iff} \\ y_k \mapsto \mathcal{M}(x_k) & x \in interface(p_a) \\ & \end{array}\right]\right)\right\}$$

Its translation by Π is:

$$\nu\tilde{c}, a \lhd^{p_a} m(\tilde{z})$$

where $\tilde{c} = \{x \mid E(x)$ is defined and denotes a ν program point $\}$.

b) **Dynamic-trans rule.** Such a rule defines the interaction between three threads. Necessarily C must contain:

$$(p_1, id_1, E_1) \qquad (p_2, id_2, E_2) \qquad (p_3, id_3, E_3)$$

where partial interactions associated to program points are:

- p_1: $(behavior_n, [label], [e_i, s_i, \tilde{y_k}], \beta(C_i, \varnothing))$
- p_2: $(actor, [ego, self], [\,], \varnothing)$
- p_3: $(message_n, [dest, label, \tilde{x_l}], \varnothing)$

with $|\tilde{y_k}| = |\tilde{x_l}| = n$. Then, we have the following relations $E_2(ego) = E_3(dest)$, $E_1(label) = E_3(label)$ and $behavior\_set((p_1, id_1)) = E_2(self)$. The C' term obtained after transition $\lambda = (p_1, p_2, p_3)$ is:

$$\{(p_1, id_1, E_1)\} \cup \left\{(p_j, new\_id, E_j) \;\middle|\; \begin{array}{l} (p_j, Es_j) \in \beta(C_i, \varnothing) \\ E_j = Es_j\left[\begin{array}{l} e_i \mapsto E_2(ego) \\ s_i \mapsto E_2(self) \\ \forall k \in [\![1;n]\!], \\ y_k \mapsto E_3(x_k) \end{array}\right] \end{array}\right\}$$

where $new\_id = id_2.p_2$.

Standard configuration $\Pi(C')$ is, as preceding, the closed term composed of the set of messages and actors with their appropriate behaviors, as defined by the C' term. We have now to study each C_i case by induction on CAP syntax. We obtain the same results as in the *static_rule* case with $\mathcal{E} = E_2(ego)$, $\mathcal{S} = E_2(self)$ and $\mathcal{M} = E_3$.

c) Finally, the last cases are managed by an induction over the shape of C_i terms, independently of the matching rule.

- when $C_i = \nu a^{\alpha}, C_a$, then $\beta(C_i, \varnothing) = \beta(C_a, [a \mapsto \alpha])$. By induction hypothesis, the resulting term of the update of the static environment of program points $\beta(C_a, \varnothing)$ is C_a. The variable a may be free in the C_a term.
  The resulting term of the environment update with the relation $a \mapsto \alpha, new\_id$, is translated by Π to $\nu a^{\alpha} C_a$;
- when $C_i = C_1 \;||\; C_2$ then $\beta(C_i, \varnothing) = \beta(C_1, \varnothing) \cup \beta(C_2, \varnothing)$. By induction, we have $\beta(C_1, \varnothing)$ which is translated in C_1 by Π and $\beta(C_2, \varnothing)$ which is translated in C_2. The term $\beta(C_i, \varnothing)$ translation with updated environment by value passing is the closed term of CAP composed of both actors and messages of C_1 and C_2;
- when $C_i = \varnothing$ then $\beta(C_i, \varnothing) = \varnothing$. It's a trivial implication.

We have shown that the implication is valid if term C contains only matching threads. Compatibility rules as well as congruence relations allow us to compute a transition step when the actor can handle more behaviors and when interacting threads are under a variable restriction or in parallel with other threads. So, the implication is true in the general case for well formed configurations.

We have the implication $\forall \lambda \in (\mathscr{L}^2 \cup \mathscr{L}^3), C \xrightarrow{\lambda} C' \Longrightarrow \Pi(C) \xrightarrow{\lambda} \Pi(C')$;

2) Let P be a standard configuration such that $\Pi(C) \xrightarrow{\lambda} P$. Let $(p_i)_i = \lambda$ the transition label. Configuration $\Pi(C)$ contains at least the interacting threads $\{p_i\}$. By definition of translation function Π, non standard term C must contain at least some threads $(p_1, id_1, E_1), (p_2, id_2, E_2)$ and (p_3, id_3, E_3) or (p_1, id_1, E_1) and (p_2, id_2, E_2), depending on the matching rule, with the appropriate constraints on

markers and environments to allow transition λ. Therefore, there exists a non standard configuration D image of C by transition λ. Let $\Pi(D)$ be its image in the CAP standard syntax by the translation function Π. By reusing preceding property, and the fact that term C' is well formed, we obtain $\Pi(D) \equiv P$ by α-conversion and extrusion.

Therefore we have $\Pi(C) \xrightarrow{\lambda} P \implies \exists D, C \xrightarrow{\lambda} D$ and $\Pi(D) \cong P$.

3) Let us show that $\mathscr{S} \equiv \Pi(\mathcal{C}_0)$. By induction, we show that $\forall C_i$ closed, $\beta(C_i, \varnothing)$ is closed. Furthermore, $\Pi(\beta(C_i, \varnothing)) \equiv C_i$ is closed. Let $C_0 = \mathcal{C}_0$ be the initial configuration. As $\mathcal{C}_0 = \beta(\mathscr{S}, \varnothing)$ by definition, we have $\Pi(C_0) \equiv \mathscr{S}$.

■

IV. Abstract Semantics

In order to ensure properties on all the possible executions of the non standard semantics, we rely on the abstract interpretation approach which combines in a single one all the possible executions.

A. Abstract Interpretation

Abstract interpretation [11] is a theory of discrete approximation of semantics. A fundamental aspect of this theory is that every semantics can be expressed as fixed points of monotonic operators on complete partial orders. A concrete semantics is defined by a tuple $(S, \subseteq, \bot, \cup, \top, \cap)$. Following [12], an abstract semantics is defined by a pre-ordered set $(S^\#, \sqsubseteq)$, an abstract iteration basis $\bot^\#$, a concretization function $\gamma : S^\# \to S$ and an abstract semantics function $\mathbb{F}^\#$.

Abstract Interpretation of Mobile Systems: We approximate here the mobile systems' semantics as described in [16,33]. The collecting semantics of a configuration $\mathscr{C}_0$ is defined as the least fixed point of the complete join morphism $\mathbb{F}$:

$$\mathbb{F}(X) = (\{\varepsilon\} \times \mathscr{C}_0) \cup \left\{ \; (u.\lambda, C') \;\middle|\; \exists C \in \mathscr{S}, (u, C) \in X \text{ and } C \xrightarrow{\lambda} C' \; \right\}$$

An abstraction $\left(\mathscr{C}^\#, \sqsubseteq^\#, \sqcup^\#, \bot^\#, \;\; \gamma^\#, C_0^\#, \rightsquigarrow, \nabla\right)$ in this framework must define as usual a pre-order, a join operator, a bottom element, a widening operator (when abstract domains are infinite) as well as:

- the initial abstract configuration $C_0^\# \in \mathscr{C}^\#$ with $\{\varepsilon\} \times \mathscr{C}_0 \subseteq \gamma(C_0^\#)$
- the abstract transition relation $\rightsquigarrow \in \wp(\mathscr{C}^\# \times \Sigma \times \mathscr{C}^\#)$ such that:
  $\forall C^\# \in \mathscr{C}^\#, \forall (u, C) \in \gamma(C^\#), \forall \lambda \in \Sigma, \forall C' \in \mathscr{C},$
  $$C \xrightarrow{\lambda} C' \implies \exists C'^\# \in \mathscr{C}^\#, (C^\# \overset{\lambda}{\rightsquigarrow} C'^\#) \text{ and } (u.\lambda, C') \in \gamma(C'^\#)$$
  Such an abstract transition computes all the concrete transitions labeled λ from all possible C represented by $C^\#$.

The abstract counterpart of the $\mathbb{F}$ function is the abstract function $\mathbb{F}^\#$ defined as:

$$\mathbb{F}^\#(C^\#) = \bigsqcup^\# \left(\left\{ C'^\# \mid \exists \lambda \in \Sigma, C^\# \rightsquigarrow^\lambda C'^\# \right\} \sqcup \{C_0^\#; C^\#\} \right)$$

B. Abstract Domains

An element of an abstract domain expresses the set of invariant properties of a set of terms. We project the initial term into an abstract element to describe its properties. Then, we use an abstract counterpart of the transition rules to obtain the set of valid properties when applying the transition rules to all elements of the initial set. Then, we compute the union of both abstract elements, to only keep the set of properties which are valid before and after the transition. We repeat these steps until a fixed point is reached. The use of the union and the widening functions guarantees the monotony of the transition and thus the existence of the fixed point. Finally, we obtain an abstract element describing the set of valid properties in all possible evolutions of the initial term. It is the abstraction of a post fixed point of the collecting semantics' least fixed point. Our abstractions are sound counterparts of the non standard semantics.

In order to avoid a too coarse approximation of the collecting semantics, we need, at least, to use a good abstraction of the control flow. We associate to each program point an abstract element describing its set of values and markers. Then, a second abstract domain relates the information of another set of properties satisfied by the collecting semantics of the analyzed term. Therefore, we use, as a main abstract domain, the cartesian product of an abstract domain to approximate non uniform control flow information in conjunction with one or more domains to approximate specific properties. Such properties can often be expressed in terms of occurrence counting of threads in configurations. These domains can describe or approximate a configuration globally by giving a property verified by all configurations or more locally like in the control flow abstraction, providing an approximation for each program point.

Generic Abstractions: In this section, we will briefly describe two abstract domains defined by FERET respectively in [14] and [13], that are used to approximate the non standard semantics of CAP. Their operational semantics is then given in Figs. 5(a) and 5(b).

a) Control Flow Abstract Domain: This abstract domain approximates variable values of thread environments as well as their markers for a given configuration. It is parametrized by an abstract domain called an Atom Domain. We associate to each program point an atom which describes the values of both variables and markers of the threads that can be associated with this program point. When computing an interaction, we merge the interacting atoms associated to the interacting threads (primitive *reagents*$^\#$) and add synchronization constraints (primitive *sync*$^\#$). If they are satisfiable, the interaction is possible. We then compute the value passing and the marker computation (function *marker_value*). Finally, we launch new threads (primitive *launch*$^\#$) and update the atom of each program point by computing its union with the appropriate resulting atom.

In this domain, we only focus on values, so we completely abstract away occurrences of threads and thus

deletion of interacting threads.

The Atom Domain we use is a reduced product of four domains. The first two represent equality and disequality among values and markers using graphs, the third one approximates the shape of markers and values with an automaton and the fourth one approximates the relationship between occurrences of letters in Parikh's vectors [29] associated to each value and marker.

b) Occurrence Counting Abstract Domain: In this domain, we count both threads associated to a particular program point and transition label, the set of which is denoted by $\mathscr{V}_c$. We first approximate the non standard semantics by the domain $\mathbb{N}^{\mathscr{V}_c}$, associating to each program point its thread occurrence in the configuration and to each transition label its occurrence in the trace that leads to the configuration. At the level of the collecting semantics, we obtain an element in $\wp(\mathbb{N}^{\mathscr{V}_c})$. We then abstract such a domain by a domain $\mathscr{N}_{\mathscr{V}_c}$ which is a reduced product between the domain of intervals indexed by $\mathscr{V}_c$ and the domain of affine equalities [24] built over $\mathscr{V}_c$. When computing a transition, we check that the occurrences of interacting threads are sufficient to allow it (primitive $SYNC_{\mathscr{N}_{\mathscr{V}_c}}$). If we do not obtain the bottom element of our abstract domain, *i.e.* the synchronization constraint is satisfiable, we add (primitive $+^{\#}$) the new transition label, the launched threads (primitives $\beta^{\#}$ and $\Sigma^{\#}$) and remove (primitive $-^{\#}$) consumed threads.

A Forward and Backward Control Flow Abstract Domain: We present here an abstract domain defined in [19]. It allows to compute a *future* usage mode associated to each variable in threads' environments. Such usage mode in $\{\bot, \circ, \bullet, \top\}$ counts whether this variable will be used to bind zero, one or more actors. If different usage modes are inferred during the analysis for the same variable, computing their union yields the value $\top = \circ \cup \bullet$ which represents here a failure in determining such usage modes. The value $\top$ is also associated to the meaning "binds more than one actor", for example when two actors are installed to a name bound by the same ν. This abstract domain and its associated semantics is similar to the control flow abstract domain but also presents major differences. Instead of computing only a forward flow and updating the original abstract element with the value obtained for the new launched threads, *i.e.* the one in the continuation, the current abstract semantics considers both a forward and a backward flow. A first flow builds these new abstract representation of the new launched threads with their associated usage modes. Then, the knowledge of such continuations and their future uses allows to constraint the interacting thread that computes this transition. For example, a variable associated to $\bot$ (uninitialized value) in the received message, can be used in the continuation to install exactly one actor and is not used with the value $\bullet$ in any other part of such a continuation. Therefore the variable x must have the value $\bullet$ in the interacting message. The resulting element of a transition in this domain is therefore the union of the initial element with the resulting updated threads as well as the interacting

Let $C^{\#}$ be an abstract configuration, let $(p_k)_{1\leqslant k\leqslant n} \in \mathscr{L}_p$ be a tuple of program points label and $(pi_k)_{1\leqslant k\leqslant n} = (s_k, (par_k), (bd_k), cons_k, cont_k)$ be a tuple of partial interactions.

We define mol by
$reagents^{\#}((p_k), (par_{k,l}), (cons_k), C^{\#})$.
When
$\forall k \in [\![1;n]\!], pi_k \in interaction(p_k)$ *; and* $mol \neq \bot_{(I(p_k))_k}$
Then
$C \xrightarrow{(p_k)_k}_{\#} \bigsqcup\{C; mol; new\_threads\}$
Where
1. $mol' = marker\_value((p_k)_k, mol, (bd_k)_k, (par_k)_k, v\_passing)$
2. $new\_threads = launch^{\#}((p_k, cont_k)_k, mol')$.

(a) Abstract semantics for control flow approximation.

We define the tuple $t \in \mathbb{N}^{\mathscr{V}_c}$ so that t_v be the occurrence of v in $(p_k)_{1\leqslant k\leqslant n}$.
When
$\forall k \in [\![1;n]\!], pi_k \in interaction(p_k)$;
$SYNC_{\mathscr{N}_{\mathscr{V}_c}}(t, C^{\#}) \neq \bot_{\mathscr{N}_{\mathscr{V}_c}}$
Then
$C \xrightarrow{(p_k)_k}_{\#} SYNC_{\mathscr{N}_{\mathscr{V}_c}}(t, C^{\#}) +^{\#} Transition +^{\#} Launched -^{\#} Consumed$
Where
1. $Transition = 1_{\mathscr{N}_{\mathscr{V}_c}}(p_1)$;
2. $Launched = \Sigma^{\#}\left((\beta^{\#}(cont^k))_k\right)$;
3. $Consumed = \Sigma^{\#}(1_{\mathscr{N}_{\mathscr{V}_c}}(p_k))_{k\in[\![1;n]\!] \wedge type(s_{k'}) \neq replication\}}$

(b) Abstract semantics for occurrence counting.

Let $local_k$ be the pair $((par_k), (bd_k))$.
$C \xrightarrow{(p_k)_k}_{\#} \bigsqcup\{C; new\_threads; interacting\_threads\}$
Where
1. $modes \triangleq count\_mode^{\mathrm{Lin}}((p_k), (local_k), v\_passing, (cont_k), C^{\#})$;
2. $new\_threads = launch^{\mathrm{Lin}}((p_k), (cont_k), modes)$;
3. $interacting\_threads = update^{\mathrm{Lin}}((p_k), new\_threads, (local_k), v\_passing)$

(c) Abstract semantics for linearity.

Figure 5. Abstract operational semantics

updated threads. The semantics of such a domain is given in Fig. 5(c).

V. Properties

The abstract semantics computes an approximation of all the executions in the non standard one. Its result can then be used in order to check many different properties. In this section, we describe interesting properties and how to observe them in the post fixed point of the analysis.

A. Linearity

Linearity is a property that expresses the fact that each actor in each possible configuration is bound to a different address. It can be expressed as in π-calculus when each process listens to at most one channel. It is a useful property to map addresses to resources.

We here use the forward and backward control flow abstract domain to verify this linearity property over CAP terms. If each variable at each program point has been associated to a unique usage mode when computing an over-approximation of all the possible transitions, then the term is linear. In contrary, if at least one of the variable at one program point is associated to the value $\top$ then we cannot say that the analyzed term is linear.

We give here two examples. The first one is linear and the second one is non linear. We decorate the term with the usage mode automatically inferred by the analysis.

$$\begin{array}{ll} \nu a^{\alpha}, b^{\beta}, & a^{\bullet} \rhd^{1} [m()^{2} = \zeta(e,s)(e^{\bullet} \rhd^{3} s), \\ & \qquad send^{4}(x^{\circ}) = \zeta(e,s)(x^{\circ} \lhd^{5} beh(s))] \\ \| & b^{\bullet} \rhd^{6} [beh^{7}(x) = \zeta(e,s)(e^{\bullet} \rhd^{8} x)] \\ \| & a^{\circ} \lhd^{9} send(b^{\circ}) \| b^{\circ} \lhd^{10} m() \end{array}$$

$$\begin{array}{ll} \nu a^{\alpha}, b^{\beta}, c^{\gamma}, & a^{\bullet} \rhd^{1} [m^{2}(x^{\top}) = \zeta(e,s)(x^{\top} \rhd^{3} s \| e^{\bullet} \rhd^{4} s)] \\ \| & b \rhd^{5} [p^{6}(x^{\top}) = \zeta(e,s)(a^{\circ} \lhd^{7} m(x^{\top}) \| e^{\bullet} \rhd^{8} s)] \\ \| & b^{\circ} \lhd^{9} p(c^{\top}) \| b^{\circ} \lhd^{10} p(c^{\top}) \end{array}$$

In this last non linear example, the analysis detects that the variable c which is transitively transferred to the program point 3 will be used to install an actor twice. Therefore it gives the value $\top$ to all variable that could receive this value c.

B. Bounded Resources

As CAP is an asynchronous calculus, when a message is sent we cannot ensure that it will be handled. With this property, we want to determine if the system grows infinitely; if the system creates more messages than it can handle. Our analysis is able to infer such a property. We first check which message can have an unbounded number of occurrences. Then we check in the global numerical invariants of the system a constraint between the number of occurrences of this message and the number of occurrences of a transition labeled with the same message label. When such a constraint can be found, we can say that this message will be in the system an unbounded number of times, but it will be handled the same number of times. The system size is constant, it does not diverge.

In the following example, our analysis is able to find that at most one message is present in the system: program points 3, 7 and 9 associated with interval $[\![0;1]\!]$. The system described by this term is bounded. Furthermore, we have the constraint $p_1 + p_4 + p_8 = 1$.

$$\nu a^{\alpha}, \nu b^{\beta}, a \lhd^{1:[\![0;1]\!]} ping() \| a \rhd^{2:[\![0;1]\!]} [ping^{3:[\![1;1]\!]}() = \zeta(e,s)(b \lhd^{4:[\![0;1]\!]} pong() \| e \rhd^{5:[\![0;1]\!]} s)] \| b \rhd^{6:[\![0;1]\!]} [pong^{7:[\![1;1]\!]}() = \zeta(e,s)(a \lhd^{8:[\![0;1]\!]} ping() \| e \rhd^{9:[\![0;1]\!]} s)]$$

In addition, we can also detect whether a system does not generate an unbounded number of actors present at the same time in a given configuration.

$$\nu a^{\alpha} a \rhd^{1:[\![0;1]\!]} [m^{2:[\![1;1]\!]}() = \zeta(e,s)(\nu b^{\beta} b \rhd^{3:[\![0;1]\!]} s \| b \lhd^{4:[\![0;1]\!]} m())] \| a \lhd^{5:[\![0;1]\!]} m()$$

In the preceding example, we automatically detect that the number of threads associated to program point 3 lies in $[\![0;1]\!]$.

C. Unreachable Behaviors

We are interested in determining the subset of behaviors that are really used for each set of behaviors. Due to its higher-order capability, CAP allows to send the set of behaviors syntactically associated to an actor to other actors. Therefore the use of the behavior set highly depends on the exchanged messages.

In the following example, all the branches of the behavior syntactically defined at program point 1 are used. We check such a property by checking that each label of transition is present at least once or its continuation has been launched. *I.e.* $\forall t \in \mathscr{V}_c, Inter(t) \neq [\![0;0]\!]$ where *Inter* is the function that maps each element of $\mathscr{V}_c$ to its image in interval part of the analysis post fixed point.

$$\begin{array}{ll} \nu a^{\alpha}, b^{\beta}, c^{\gamma}, & a \rhd^{1} [m_0^2() = \zeta(e,s)(b \lhd^{3} n_1(s) \| \\ & \qquad b \lhd^{4} m_1(c)), \ m_1^5(dest) = \zeta(e,s)(dest \lhd^{6} m_2()), \\ & \qquad m_2^7() = \zeta(e,s)(\varnothing)] \\ \| & b \rhd^{8} [n_1^9(self) = \zeta(e,s)(e \rhd^{10} self \| c \lhd^{11} n_2(self))] \\ \| & c \rhd^{12} [n_2^{13}(self) = \zeta(e,s)(e \rhd^{14} self)] \\ \| & a \lhd^{15} m_0() \end{array}$$

We can use such an analysis to clean the term with garbage collecting-like mechanisms.

VI. Conclusion

Our previous type-based analyzes were unable to deal efficiently with higher-order features, such as communication of behaviors within messages. To enable these features, we have adapted the framework of FERET [16] to deal with full CAP terms. Moreover, in contrary to our aforementioned analyzes about actor calculus, we are now able to count occurrences of both actors and messages. Therefore, we can easily cope with properties related to occurrence counting. We can detect whether the number of actors and messages is finite, whether there is dead code and whether the message queues are bounded. Interestingly, we can also check the linearity property, provided we compute a more involved forward and backward information flow.

Nevertheless, one of the most interesting property with an asynchronous process calculus with non uniform behavior, is the absence of orphan messages, *i.e.* stuck-freeness. An orphan is a message which may not be handled by its target in some execution path. We are currently devising a new abstract domain that can detect such orphan messages. More generally, we also want to define a generic abstract domain dedicated to the data-flow like analyzes provided by type systems. Such an abstract domain can be useful to automatically build domains to observe properties for which we already have a type system. As a matter of fact, there seems to be a general correspondence between covariance and contravariance in type systems on the one side, and forward and backward information flows in the abstract domains on the other side.

ACKNOWLEDGMENT

We deeply thank Jérome Feret for fruitful discussions and careful proof reading of the first author's Master's thesis [18].

REFERENCES

[1] *Polyspace technologies, www.polyspace.org*.

[2] G. Agha. *Actors: A model of concurrent computation in distributed systems*. MIT Press, Cambridge, Mass., 1986.

[3] T. Amtoft, F. Nielson, and H. R. Nielson. Type and behaviour reconstruction for higher-order concurrent programs. *Journal of Functional Programming*, 7(3):321–347, 1997.

[4] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software, invited chapter. In *The Essence of Computation: Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones*, LNCS 2566, pages 85–108. Springer, October 2002.

[5] G. Boudol. Typing the use of resources in a concurrent calculus. In *Proc. of ASIAN'97*, volume 1345 of *LNCS*, 1997.

[6] C. Carrez, A. Fantechi, and E. Najm. Behavioural contracts for a sound composition of components. In *Proc. of FORTE 2003*, volume 2767 of *LNCS*. Springer, 2003.

[7] J.-L. Colaco, M. Pantel, F. Dagnat, and P. Sallé. Static safety analysis for non-uniform service availability in Actors . In *Proc. of FMOODS'99*, volume 139, pages 371–386. Kluwer, B.V., 1999.

[8] J.-L. Colaco, M. Pantel, and P. Sallé. An actor dedicated process calculus. In *Proc. of the ECOOP'96 Workshop on Proof Theory of Concurrent Object-Oriented Programming*, 1996.

[9] J.-L. Colaco, M. Pantel, and P. Sallé. Static analysis of behavior changes in Actor languages. In *Object-Oriented Parallel and Distributed Programming*, pages 53–72. Hermès Science, 8, quai du Marché-Neuf, 75004 Paris, France, 2000.

[10] M. Colin, X. Thirioux, and M. Pantel. Temporal logic based static analysis for non uniform behaviors. In *Proc. of FMOODS'03*. Springer, 2003.

[11] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. of POPL'77*, pages 238–252. ACM Press, 1977.

[12] P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, 1992.

[13] J. Feret. Occurrence counting analysis for the pi-calculus. In *Proc. of the 1st Workshop on GEometry and Topology in COncurrency Theory*, volume 39.2 of *ENTCS*. Elsevier, 2001.

[14] J. Feret. Dependency analysis of mobile systems. In *Proc. of ESOP'02*, number 2305 in LNCS. Springer, 2002.

[15] J. Feret. Abstract interpretation of mobile systems. *Journal of Logic and Algebraic Programming*, 63.1, 2005. special issue on pi-calculus, 2005.

[16] J. Feret. *Analysis of Mobile Systems by Abstract Interpretation*. PhD thesis, École polytechnique, Paris, France, february 2005.

[17] C. Fournet, C. Lavene, L. Maranget, and D. Rémy. Implicit typing à la ml for the join-calculus. In *Proc. of CONCUR'97*, volume 1283 of *LNCS*. Springer, 1997.

[18] P.-L. Garoche. Static analysis of actors by abstract interpretation. Master's thesis, École Normale Supérieure de Cachan, 2005.

[19] P.-L. Garoche, M. Pantel, and X. Thirioux. Static Analysis of Actors: From Type Systems to Abstract Interpretation. In *Proc. of EAAI'06*, 2006.

[20] P.-L. Garoche, M. Pantel, and X. Thirioux. Static Safety for an Actor Dedicated Process Calculus by Abstract Interpretation. In *Proc. of FMOODS'06*, pages 78–92. Springer Verlag LNCS, 2006.

[21] M. Hennessy, J. Rathke, and N. Yoshida. Safedpi: a language for controlling mobile code. In *Proc. of FoSSaCS'04*, LNCS, pages 241–256. Springer, 2004.

[22] C. Hewitt, P. Bishop, and R. Steiger. A universal modular actor formalism for artificial intelligence. In *Proc. of IJCAI'73*, 1973.

[23] A. Igarashi and N. Kobayashi. A generic type system for the pi-calculus. *Theoretical Computer Science*, 311(1-3):121–163, January 2004.

[24] M. Karr. Affine relationships among variables of a program. *Acta Informatica*, 6:133 – 151, 1976.

[25] N. Kobayashi. A type system for lock-free processes. *Information and Computation*, 177(2):122–159, 2002.

[26] Naoki Kobayashi. A new type system for deadlock-free processes. In *Proc. of CONCUR'06*, volume 4137 of *LNCS*, pages 233–247. Springer, 2006.

[27] E. Najm, A. Nimour, and J.-B. Stefani. Infinite types for distributed object interfaces. In *Proc. of FMOODS'99*, volume 139. Kluwer, B.V., 1999.

[28] J. Palsberg and P. O'Keefe. A type system equivalent to flow analysis. In *Proc. of POPL'95*, pages 367–378, 1995.

[29] R. Parikh. On context-free languages. *Journal of the ACM*, 13(4):570–581, 1966.

[30] F. Puntigam. Types for Active Objects based on Trace Semantics. In Elie Najm et al., editor, *Proc. of FMOODS'96*, Paris, France, 1996. Chapman & Hall.

[31] S. Rajamani and J. Rehof. A behavioral module system for the pi-calculus. In *Proc. of SAS'01*, volume 2126 of *LNCS*, pages 375–394. Springer, 2001.

[32] A. Ravara and V. Vasconcelos. Typing non-uniform concurrent objects. In *Proc. of CONCUR'00*, volume 1877 of *LNCS*. Springer, 2000.

[33] A. Venet. *Static Analysis of Dynamic Graph Strutures in Untyped Languages*. PhD thesis, École polytechnique, Paris, France, december 1998.