Static Safety for an Actor Dedicated Process Calculus by Abstract Interpretation

Pierre-Loïc Garoche, Marc Pantel, and Xavier Thirioux

IRIT, Toulouse
{garoche,pantel,thirioux}@enseeiht.fr

Abstract. The actor model eases the definition of concurrent programs with non uniform behaviors. Static analysis of such a model was previously done in a data-flow oriented way, with type systems. This approach was based on constraint set resolution and was not able to deal with precise properties for communications of behaviors. We present here a new approach, control-flow oriented, based on the abstract interpretation framework, able to deal with communication of behaviors. Within our new analyses, we are able to verify most of the previous properties we observed as well as new ones, principally based on occurrence counting.

1 Introduction

1.1 Context – Motivation

The development of the telecommunication industry and the generalization of network use bring concurrent and distributed programming in the limelight. In that context, programming is a hard task and, generally, the resulting applications contain much more *bugs* than usual centralized software. As sequential object oriented programming is commonly accepted as a *good* way to build software, concurrent object oriented programming seems to be well-suited for programming distributed systems. 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 concurrent object oriented programming.

To obtain widely usable tools, we have chosen to use the actor model proposed by HEWITT [19] and developed by AGHA [1]. This model 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 queue and when idle, processes the first message it can handle in this queue. Besides those conventions (which are also true for concurrent objects), an actor can dynamically change its interface. This property allows to increase or decrease the set of messages an actor may handle, yielding a more accurate programming model. This model, also known as concurrent objects with non uniform behavior (or interface), has been adopted by the telecommunication industry 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). Until now, we have been designing several analyses for an actor model, all of which based on typing systems. 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 of our previous attempts, which we could somehow overcome but at the price of a much greater complexity unmatched with only a small gain in precision, we decided to move to the framework of abstract interpretation, whose tools and ideas have now significantly grown in maturity and are being widely used in industrial contexts, or are on the verge of being so. We now investigate these techniques in order to capture our long standing properties of interest (detection of orphan messages, that is messages sent to an actor which will not handled them) as well as new ones, especially dedicated to control of resources' usage.

In a 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.

1.2 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 analysis) have been the subject of active research. Two opposite approaches have been followed: type declaration and type inference. In the first case, most proposals make use of types as processes of a simple algebra, for instance CCS (Calculus of Communicating Systems) processes. This allows a form of subtyping through simulation relations or language containment. The works of KOBAYASHI *et al.* [20,22], RAVARA *et al.* [28], NAJM *et al.* [4,23], PUNTIGAM [26], and HENNESSY *et al.* [18] follow this line of thought, to which we can add the works of RAJAMANI *et al.* [5,27], bringing model-checking issues for those processes-as-types in the scope. The second case is again twofold: on one side we have unification based typing algorithms focusing on resources' usage control witnessed by the works of FOURNET *et al.* [16] and BOUDOL *et al.* [3], whereas on the other side we have flow based algorithms, related to behavior and communication patterns reconstruction, advocated by the works of NIELSON *et al.* [2] and PANTEL *et al.* [6,8,9]. Explicit typing may provide more precise information but are sometimes very hard to write for the programmer (they might be much more complex than the program itself). Implicit typing requires less user supplied information but lead to less precise results.

One drawback of type-based analyses is that they are mainly concerned with data-flow analyses (as types basically represent sets of possible values for variables). In this context, control flow analyses can be mimicked with sophisticated encodings [24] 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 [29] and later FERET [14,15].

2 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 [7], which merge asynchronous π-calculus

and CARDELLI's Primitive Object Calculus. The following example illustrates both replication and behavior passing mechanisms of CAP. 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 the behavior set respectively denoted by 2 and 4 for a and 8 for b.

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

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

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, in which the actor a handles the message *send*. The message m is an orphan one: it is in the configuration but cannot be handled for the moment. After one interaction between a and the message *send*, the message *beh* which argument is the behavior's set of a is sent to b. Thus b can handle that message. In its continuation, the actor b assumes the behavior's 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. Stuck-freeness, *i.e.* the detection of the set of permanent orphans messages, or linearity, *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.

2.1 Syntax and semantics

Let $\mathscr{N}$ be an infinite set of actor names, $\mathscr{V}$ be an infinite set of variables. Let $\mathscr{L}_m$ be a set of message labels, $\mathscr{L}_p$ be the set of program point labels and $\mathscr{L}_n$ be the set of name labels. In the following, we denote $\mathscr{L}_p \cup \mathscr{L}_n$ by $\mathscr{L}$. The syntax of configurations is described as follows:

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

Configurations can be an empty process, a creation of actor's address, parallel execution, 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, behaviors' installation or external choices between some actors' behaviors. They will be used to build traces of the execution control flow. Name restriction, in the configuration $(\nu a^{\alpha})C$, acts as a name binder, so does the ζ operator and the message label for variables in the behavior description of an actor, *i.e.* in the behavior $[m_i^{l_i}(\widetilde{x_i}) = \zeta(e_i,s_i)C_i^{\,i=1\ldots n}]$, therefore the occurrences of a in C, $\widetilde{x_i}$ in $\zeta(e_i,s_i)C_i$ and e_i and s_i in C_i are bound. 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. The standard semantics of CAP was defined, à la Milner, by both the usual transition rule (*cf.* Fig. 1) and the congruence relation (*cf.* Fig. 2).

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

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.

Fig. 1. Transition rule of CAP standard semantics

3 Non standard semantics

In order to ease the definition of abstract interpretations, we need to define 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 [14,15] to model first order process calculi as π-calculus, spi-calculus, Ambients, Bio-ambients calculus. We also describe in this section how 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 the 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.

3.1 Partial interactions

We associate to each program point a partial interaction which defines 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. Here, in CAP, partial interactions can represent a syntactically defined actor, a dynamic

$$
\begin{array}{rcll}
C & \equiv & D & \text{C } \alpha\text{-convertible to D} \quad (\alpha - \mathit{conversion}) \\
C||0 & \equiv & C & (\mathit{inaction}) \\
C||D & \equiv & D||C & (\mathit{commutativity}) \\
(C||D)||E & \equiv & C||(D||E) & (\mathit{associativity}) \\
(\nu a)\varnothing & \equiv & \varnothing & (\mathit{garbage\ collecting}) \\
T \rhd T_1 & \equiv & T \rhd T_2 & \text{if } T_1 \equiv T_2 \quad (\mathit{behavior\ equivalence}) \\
(\nu a)(\nu b)C & \equiv & (\nu b)(\nu a)C & \text{if } a \neq b \quad (\mathit{swapping}) \\
(\nu a)C||D & \equiv & (\nu a)(C||D) & \text{if } a \notin \mathcal{fn}(D) \quad (\mathit{extrusion})
\end{array}
$$

Fig. 2. Congruence relation of CAP standard semantics

one (an actor whose behavior is defined by a variable) and a particular behavior of an actor or a sent message.

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

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

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

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.

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 binds $n+2$ 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's name.

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.

3.2 Abstract syntax extraction

We now define the syntax extraction function that takes a CAP term describing the initial state of an agents' system 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. We will check constraints defined in the set *constrains* about thread environment with the use of the sequence $(parameter_i)$, then we will use both sequences $(parameter_i)$ and $(bound_i)$ to compute value passing, finally we will deal with the set *continuation* to determine which threads have to be inserted in the system.

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

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

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

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

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

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 $\beta(\mathscr{S}, \varnothing)$.

3.3 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 bounded 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 $\nu^{\alpha}a, a \rhd^1 [m^2() = \zeta(e,s)C]$, we have $behavior\_set(2,m) = (1,m)$.

Communication with a syntactic defined 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 label (X_2^1) is equal to the message label (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

$$1.\ components = \begin{cases} 1 \mapsto static\_actor_n, \\ 2 \mapsto message_n \end{cases} \qquad 2.\ compatibility = \begin{cases} X_1^1 = X_1^2; \\ X_2^1 = X_2^2; \end{cases}$$

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

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 label (X_1^1) and message label (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.

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

where

$$1.\ components = \begin{cases} 1 \mapsto behavior_n, \\ 2 \mapsto dynamic\_actor, \\ 3 \mapsto message_n \end{cases} \quad 2.\ compatibility = \begin{cases} X_1^2 = X_1^3; \\ behavior\_set(I^1) = X_2^2; \\ X_1^1 = X_2^3; \end{cases}$$

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

3.4 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 a formal rule.

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 interface satisfies the synchronization constraints. Thus we can compute the interaction:

- we remove interacting threads according to the type of their exhibited partial interaction;
- we choose a syntactic continuation for each thread;
- we compute dynamic data for each of these continuations:
  - we compute the marker;
  - we take into account name passing;
  - we create fresh variables and associate them with the correct values;
  - we restrict the environment according to the interface associated with the program point.

3.5 Correspondence

Theorem 1 (correspondence) *CAP standard semantics and its non standard semantics are in strong bisimulation*

Proof. The proof can be found at the first author's web page, www.enseeiht.fr/ garoche.

3.6 Example

To illustrate the use of the non standard semantics, we will compute the first transition of the example given in section 2.

The initial configuration[1] is:

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

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: $\left\{ \begin{array}{l} \{(static\_actor_n,[a,m],[e,s],\beta(a \rhd^3 s,\varnothing))\}, \\ \{(static\_actor_n,[a,send],[e,s,x],\beta(x \lhd^5 beh(s),\varnothing))\} \end{array} \right\}$ when the program point 6 exhibits the only partial interaction:

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

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

We choose the first 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 message 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 environment 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:

$$(2,\varepsilon,\left[a \mapsto \alpha,\varepsilon\right])\ (4,\varepsilon,[])\ (5,\varepsilon,\begin{bmatrix} x \mapsto \beta,\varepsilon \\ s \mapsto 1,\varepsilon \end{bmatrix})$$

$$(7,\varepsilon,\left[b \mapsto \beta,\varepsilon\right])\ (8,\varepsilon,[])\ (10,\varepsilon,\left[b \mapsto \beta,\varepsilon\right])$$

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

4 Abstract semantics

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

4.1 Abstract Interpretation

Abstract interpretation [10] 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 [11], 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 [15,29]. 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:

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

4.2 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 rule 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 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. But, most of our properties can be expressed in terms of occurrence counting. We also need to approximate configurations globally. Therefore, we use, as an abstract domain, the cartesian product of an abstract domain to approximate non uniform control flow information in conjunction with a domain to approximate the occurrence of threads in configurations.

Generic abstractions. In this section, we will briefly describe the two abstract domains defined, by FERET, respectively in [13] and [12] that are used to approximate the non standard semantics of CAP. Their operational semantics is then given in Figs. 3(a) and 3(b).

Control Flow Abstract Domain. This abstract domain approximates variable values of thread environments as well as their marker 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 marker 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 [25] associated to each value and marker.

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 threads occurrence in the configuration and to each transition label, its occurrence in the word 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 [21] constructed 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.

5 Properties

The abstract semantics computes an approximation of all the execution 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 fixed point of the analysis.

5.1 Linearity

Linearity is a property that expresses the fact that all actors in each possible configuration are bound to different addresses. 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.

Our analysis is able to prove that a term, without recursive name definitions, *i.e.* without a ν operator inside a behavior continuation, will be linear in all the possible configurations it will take. We can observe such a property with both the control flow domain and the occurrence counting domain. We first determine with the control flow the upper set of program points representing actors that can be associated with each address. Then we check in the occurrence counting domain that each of those program points is mapped to at most one thread in each configuration (within the interval domain) and, moreover, that program points that can be associated with the same address are in mutual exclusion (with the global numerical domain). The mutual exclusion property is observed by exhibiting a constraint from the global numerical domain. Such a constraint must be a linear combination $\Sigma x_i + \Sigma k_j * y_j = 1$ with $\{x_i\}$ the set of program points in mutual exclusion and $\{k_j\}$ a set of positive or null coefficients. Whether such a constraint can be generated by the set of constraints describing the affine space of the

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 $(pi_k)_{1\leqslant k\leqslant n} = (s_k, (parameter_k), (bd_k), constraints_k, continuation_k)$ be a tuple of partial interactions.

We define *mol* by $reagents^\#((p_k), (parameter_{k,l}), (constraints_k), C^\#)$.
When
$\forall k \in [\![1;n]\!], pi_k \in interaction(p_k)$;
$mol \neq \bot_{(I(p_k))_k}$
Then

$$C \xrightarrow{(p_k)_k}_\# \bigsqcup\{C; mol; new\_threads\}$$

Where
1. $mol' = marker\_value((p_k)_k, mol, (bd_{k,l})_{k,l}, (parameter_{k,l})_{k,l}, v\_passing)$
2. $new\_threads = launch^\#((p_k, continuations_k)_k, mol')$.

(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. $Launched = \Sigma^\#\left((\beta^\#(continuation^k))_k\right)$;
3. $Consumed = \Sigma^\#(1_{\mathscr{N}_{\mathscr{V}_c}}(p_k))_{k\in\{k'|1\leqslant k'\leqslant n, type(s_{k'})\neq replication\}}$

(b) Abstract semantics for occurrence counting.

Fig. 3. Abstract operational semantics

global numerical domain then the $\{x_i\}$ program points are in mutual exclusion but they do not have to be present in every configuration of the system.

In the following example, we can automatically determine that the following term satisfies the linearity property.

$$\begin{array}{ll} \nu a^\alpha, b^\beta,\ a \rhd^{1:[\![0;1]\!]} [m()^{2:[\![0;1]\!]}() = \zeta(e,s)(e \rhd^{3:[\![0;1]\!]} s), & \\ \qquad\qquad\qquad\qquad send^{4:[\![0;1]\!]}(x) = \zeta(e,s)(x \lhd^{5:[\![0;1]\!]} beh(s))] & \\ \quad || \qquad b \rhd^{6:[\![0;1]\!]} [beh^{7:[\![0;1]\!]}(x) = \zeta(e,s)(e \rhd^{8:[\![0;1]\!]} x)] & \\ \quad || \qquad a \lhd^{9:[\![0;1]\!]} send(b) \ || \ b \lhd^{10:[\![0;1]\!]} m() & \end{array}$$

All the actors are associated with the interval $[\![0;1]\!]$. The only actor that can be associated to address a is 1 and others (3, 6 and 8) can be associated with address b. Then the constraint $p_3 + p_6 + p_8 = 1$ can be observed in the global numerical part of the post fixed point of the analysis. We can notice that we have a stronger property: there is exactly one actor on the address b in every configuration of this term.

5.2 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 $p_3 + p_7 + p_9 = 1$.

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

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

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

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

5.3 Unreachable behaviors

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

In the following example, all the behavior 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.

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

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

6 Conclusion

We have adapted the framework of FERET [15] to deal with a higher order process calculus modeling actor languages. With such a framework, we are able to analyze CAP terms without any restriction about the kind of values sent within messages: we can now handle behavior passing, which was not able with our previous type based analysis. In contrary to our aforementioned analyses about actor's calculus, we are able to easily count occurrences of both actors and messages. Therefore, most of the properties we obtain are 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. We also have the linearity property under certain restrictions.

To go further, we need another abstraction which will split thread's information into computation units representing the recursive instances of the same thread. Such an abstract domain will allow us to deal with linearity in the general case as well as handling more properties. In fact the most interesting property with an asynchronous process calculus with non uniform behavior, is the detection 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 distinguish two kinds of orphan: safety ones and liveness ones. Safety orphans occur when all future behaviors of the target on a given execution path cannot handle such a message. On the contrary, liveness orphans occur when one of the target behaviors in each execution paths knows how to handle such a message but the target is deadlocked and will never assume the corresponding behavior. We advocate that with this new abstract domain we will be able to detect both kinds of orphans. We also want to define a generic abstract domain dedicated to the data-flow like analyses 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.

Acknowledgement.

We deeply thank Jérome Feret for fruitful discussions and careful proof reading of the first author's Master's thesis [17].

References

1. G. Agha. *Actors: A model of concurrent computation in distributed systems*. MIT Press, Cambridge, Mass., 1986.
2. 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.
3. G. Boudol. Typing the use of resources in a concurrent calculus. In *Proc. of ASIAN'97*, volume 1345 of *LNCS*, 1997.
4. 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.
5. S. Chaki, S. Rajamani, and J. Rehof. Types as models: model checking message-passing programs. In *Proc. of POPL'02*. ACM Press, 2002.
6. J.-L. Cola co, 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.

7. 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.
8. 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.
9. M. Colin, X. Thirioux, and M. Pantel. Temporal logic based static analysis for non uniform behaviors. In *Proc. of FMOODS'03*. Springer, 2003.
10. 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.
11. P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, 1992.
12. 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.
13. J. Feret. Dependency analysis of mobile systems. In *Proc. of ESOP'02*, number 2305 in LNCS. Springer, 2002.
14. J. Feret. Abstract interpretation of mobile systems. *Journal of Logic and Algebraic Programming*, 63.1, 2005. special issue on pi-calculus, 2005.
15. J. Feret. *Analysis of Mobile Systems by Abstract Interpretation*. PhD thesis, École polytechnique, Paris, France, february 2005.
16. 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.
17. P.-L. Garoche. Static analysis of actors by abstract interpretation. Master's thesis, École Normale Supérieure de Cachan, 2005.
18. 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.
19. C. Hewitt, P. Bishop, and R. Steiger. A universal modular actor formalism for artificial intelligence. In *Proc. of IJCAI'73*, 1973.
20. A. Igarashi and N. Kobayashi. A generic type system for the pi-calculus. *Theoretical Computer Science*, 311(1-3):121–163, January 2004.
21. M. Karr. Affine relationships among variables of a program. *Acta Informatica*, 6:133 – 151, 1976.
22. N. Kobayashi. A type system for lock-free processes. *Information and Computation*, 177(2):122–159, 2002.
23. 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.
24. J. Palsberg and P. O'Keefe. A type system equivalent to flow analysis. In *Proc. of POPL'95*, pages 367–378, 1995.
25. R. Parikh. On context-free languages. *Journal of the ACM*, 13(4):570–581, 1966.
26. 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.
27. 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.
28. A. Ravara and V. Vasconcelos. Typing non-uniform concurrent objects. In *Proc. of CONCUR'00*, volume 1877 of *LNCS*. Springer, 2000.
29. A. Venet. *Static Analysis of Dynamic Graph Strutures in Untyped Languages*. PhD thesis, École polytechnique, Paris, France, december 1998.