STATIC SAFETY ANALYSIS FOR NON-UNIFORM SERVICE AVAILABILITY IN ACTORS

Jean-Louis Colaco, Marc Pantel, Fabien Dagnat, Patrick Sallé

LIMA/ENSEEIHT/INPT/IRIT,
2, rue C. Camichel,
31071 TOULOUSE CEDEX
FRANCE

{colaco,pantel,dagnat,salle}@enseeiht.fr

Abstract: The main purpose of this work is the static detection of orphan messages in actor based languages. An orphan is a message which may not be handled by its target in some execution paths. Two kinds of orphan messages may be encountered, i.e., safety and liveness ones. Safety orphans occur when all target behaviors on a given execution path do not know how to handle the message. Liveness orphans occur when one of the target behaviors in each execution path knows how to handle the message but the target is deadlocked and will never assume the corresponding behavior. This paper presents a safe static analysis which detects all safety orphan messages in actor-based programs. This result extends previous work derived from sequential object-oriented languages type systems to non-uniform behaviors.

Keywords:. Concurrent Objects, Actors, Process calculi, Non-uniform service availability, Safety orphans, Static analysis, Type inference, Subtyping

Introduction

Most of the type systems designed for concurrent objects rely on the uniform behavior assumption : an object is a) always able to handle requests for the same set of methods, and b) always accessible (each method can be handled any number of times). This hypothesis allows the use of type systems designed for sequential object-oriented languages (either kind-based ones as proposed by Vasconcelos and Tokoro [24] and by Kobayashi and Yonezawa [15], or subtype-based ones as advocated by the authors in [6]). In the case of objects with non-uniform behavior (i.e., short lifetime objects

P. Ciancarini et al. (eds.), *Formal Methods for Open Object-Based Distributed Systems*
© Springer Science+Business Media New York 1999

or behavior changing actors), an object may be able to handle a request to one of its method at a given time and not be able to handle it at some other time. If the request cannot be handled, the associated message is called a "safety orphan". The previous type systems could only detect rather trivial safety orphan messages. The system described in this paper extends our subtype-based previous work in order to catch all potential safety orphans in actor-based programs.

In this purpose, a new safe type-based abstraction of all possible behaviors for a given actor is proposed. We extend a Primitive Actor Calculus introduced in the first section and defined in a previous paper [4, 6] in order to give a simple semantic characterization of safety orphan messages.

A sound type system based on the type abstraction is then presented. It rejects all the programs which may produce safety orphan messages. In conclusion, related works and possible extensions are discussed.

1 CAP : A PRIMITIVE ACTOR CALCULUS

Type systems for concurrent calculus have been the subject of many recent studies [22, 14, 12, 10]. Most of these investigations address the problem of typing variants of the π-calculus. Various encodings of concurrent objects in the π-calculus or similar formalisms have been proposed [20, 9, 25, 21]. Message labels and actors mail addresses are usually both expressed using names. Therefore, the typing of encoded programs generally lead to type information which do not reflect the structure of the original program. In a previous work on typing objects and actors, Vasconcelos et al. advocated the use of an extension of the asynchronous π-calculus with record-like objects (see [24]). Their calculus relies on replication in order to express the recursive structure of objects. As actors can change their behavior when they handle a message, their behaviors are defined as mutually recursive object structures. The replication based encoding of mutually recursive structures produces type information which do not reflect this recursive structure. Therefore, an extension of Vasconcelos and Tokoro calculus of concurrent objects allowing mutually recursive behaviors was required. A dedicated process calculus CAP which expresses syntactically the key features of the Actor model was then defined.

As in the π-calculus [16] and in the ν-calculus [11] the basis of the calculus is the *name* representing the actors mail addresses. Following Abadi and Cardelli's calculus of Primitive Objects [1], actor's different behaviors are represented by mutually recursive records of methods only accessible by communication.

CAP does not follow all the principles of Agha's actors, but provides behaviors and addresses as primitives that allow to express very easily actor programs. Syntactically, the sharing of the same address by several different actors is not forbidden. Following Kobayashi et al. in [14], linearity could be enforced for the subset of CAP used in this paper by restricting the use of weakening and contraction on the typing environment. However, typing full CAP calculus, which provides a restricted form of reflexivity by allowing actors to change the behavior of other actors, requires the use of a more sophisticated analysis described in [5]. In order to reduce the already complex presentation of our analysis, the linearity analysis will not be recalled in this paper. So, we will assume that CAP programs respect the linearity hypothesis.

The remaining part of this section is devoted to a quick introduction of CAP (a more precise presentation of the calculus and its semantic are available in [4, 6].

1.1 CAP syntax

As a first example of a CAP expression, we construct a term corresponding to the "one-slot buffer" beginning with an empty state which is sent a *put* message.

$$\nu a, b(a \rhd [\, put(v) = \zeta(e, s_e)(e \rhd [get(c) = \zeta(e', s_f)(c \lhd rep(v) \parallel e' \rhd s_e)])] \\ \parallel a \lhd put(b))$$

First, we "create" the two actors names a and b using the ν operator. An actor is built (via $\rhd$) by the association of an address (a) and a behavior. In the previous example, the behavior of a has two states defined recursively (via ζ). The first state *empty* only understands one *put* message and then switches to the second state *full* where it can treat only one *get* message. Before switching back to *empty*, it sends (via $\lhd$) the value coming from the corresponding *put* request to the argument of the *get* message.

When an actor accepts a message, $\zeta(e, s)$ binds the actor's *address* to e (called *ego*) and its *current behavior* to s (called *self*). This operator is inspired by the ς defined by Abadi and Cardelli [1] to formalize self-substitution in objects. In our context, the capture of *self* and *ego* is used to formalize behavior changes without introducing a fixpoint operator.

To define CAP syntax the following sets are used: N an infinite set of name symbols (a, b, $e_i \in N$), V an infinite set of variable symbols (s, $s_i \in V$) and L a finite set of feature labels ($m_i \in L$). Sequences of symbols are represented by a tilde ($\tilde{}$).

A configuration is a concurrent combination of actors and messages sent to actors. Their set C is built by the following grammar :

C	::=	ϕ	– empty configuration
	|	$\nu a\, C$	– address creation
	|	$C \parallel C$	– concurrent combination
	|	$a \rhd s$	– a behavior installation
	|	$a \rhd [m_i(\tilde{c_i}) = \zeta(e_i, s_i)C_i^{i \in I}]$	– a behavior installation
	|	$a \lhd m(\tilde{b})$	– a message sending
	|	$(\, C \,)$	– associativity and priority

1.2 CAP semantics

CAP semantics is defined in Milner's fashion (see [16]) using a term congruence to simplify the reduction rules.

The syntax of CAP contains three binders : ν which binds a name, $\zeta(e, s)$ which binds the name e and the variable s and lastly $m_i(\tilde{c_i})$ which bind the names c_i. The notions of free names $\mathcal{FN}()$, free variables $\mathcal{FV}()$ and substitution $\sigma = \{C/x\}$ are naturally defined. Note that substitution is only defined if C and x are "coherent" (whether C and x are names or C is a behavior and x a variable).

We then build "$\equiv$" which is the smallest congruence relation on CAP expressions defined by the following principles :

1. if C_1 and C_2 are α-convertible then $C_1 \equiv C_2$
2. $\mathcal{C}$, $\|$, ϕ is commutative monoid
3. the order of method definition in behavior formation is not significant
4. if $a \notin \mathcal{FN}(D)$ then $\nu a(C) \parallel D \equiv \nu a(C \parallel D)$

The reduction of CAP expressions can then be defined as the smallest relation "$\longrightarrow$" generated by the following set of inference rules:

$$\text{STRUCT}: \frac{D \equiv C \quad C \longrightarrow C' \quad C' \equiv D'}{D \longrightarrow D'} \qquad \text{PAR}: \frac{C \longrightarrow C'}{C \parallel D \longrightarrow C' \parallel D} \qquad \text{RES}: \frac{C \longrightarrow C'}{\nu a C \longrightarrow \nu a C'}$$

$$\text{COMM}: \frac{k \in I \qquad len(\tilde{b}) = len(\tilde{c_k})}{\begin{array}{l} a \lhd m_k(\tilde{b}) \parallel a \rhd [m_i(\tilde{c_i}) = \zeta(e_i, s_i) C_i{}^{i \in I}] \\ \quad \longrightarrow C_k\{a/e_k\}\{[m_i(\tilde{c_i}) = \zeta(e_i, s_i) C_i{}^{i \in I}]/s_k\}\{\tilde{b}/\tilde{c_k}\} \end{array}}$$

A specific configuration called *Error* is introduced in order to model the dynamic occurrence of an error. Two kinds of dynamic errors may occur during the reduction of a CAP term.

The former corresponds to an arity mismatch between a message and the corresponding definition in the behavior of the target actor :

$$\text{COMM-ERR}: \frac{k \in I \qquad len(\tilde{b}) \neq len(\tilde{c_k})}{a \lhd m_k(\tilde{b}) \parallel a \rhd [m_i(\tilde{c_i}) = \zeta(e_i, s_i) C_i{}^{i \in I}] \longrightarrow Error}$$

This kind of error has been thoroughly studied in the literature, including work based on kinded types by [24, 15] and based on subtyping [6].

The main contribution of this paper is the detection of the latter kind of error, i.e., safety orphans (which will not be handled by any of their target future behaviors).

1.3 Safety orphan messages

A decorated (or instrumented) version of CAP is used to present a formal characterization of safety orphan messages. Each actor (i.e. the binding of behaviors to addresses) will be decorated with σ, the set of messages that it will be able to handle in its future. The only change in CAP syntax is that we replace ($\rhd$) by ($\overset{\sigma}{\rhd}$). The annotation (σ) can be seen as a type which will be computed by our inference algorithm.

Safety orphan messages can then be characterized by the semantic value *Error* and its introduction rule :

$$\text{ORPH-ERR}: \text{if } m \notin \sigma \text{ then } a \lhd m(\tilde{v}) \parallel a \overset{\sigma}{\rhd} [m_i(\tilde{x_i}) \cdots] \longrightarrow Error$$

As in any explicitly typed calculus, we need to ensure that an expression is well-decorated.

Definition 1.1 (Well decorated CAP expression) *A closed decorated* CAP *expression C is well decorated :*

1. *if there exists a derivation* $\emptyset \vdash C : \mathcal{W}$ *in the following deduction system,*
2. *and if the decorations are the least fixpoints of the system of equations generated by the rule* (Wd-Behavior).

$E \vdash C : \mathcal{W}$ *means that the configuration* C *is well-decorated in the environment* E *(which contains associations both between names and decorations and between variables and decorations).* $E \vdash B : \sigma$ *means that the behavior* B *(either a variable or a behavior) can only be used in an actor decorated with* σ *in the environment* E.

$$\text{(Wd-Empty)}\frac{}{E \vdash \phi : \mathcal{W}} \qquad \text{(Wd-Rest)}\frac{E, a : \sigma_a \vdash C : \mathcal{W}}{E \vdash \nu a\, C : \mathcal{W}} \qquad \text{(Wd-Parallel)}\frac{E \vdash C : \mathcal{W} \quad E \vdash D : \mathcal{W}}{E \vdash C \parallel D : \mathcal{W}}$$

$$\text{(Wd-Behavior)}\frac{\forall i \in I \qquad E, e_i : \sigma_i, s_i : \sigma \vdash C_i : \mathcal{W}}{E \vdash [m_i(x_i) = \zeta(e_i, s_i)C_i{}^{i \in I}] : \sigma} \left(\sigma = \bigcup_{i \in I}(\sigma_i \cup \{m_i\})\right)$$

$$\text{(Wd-Actor)}\frac{E \vdash a : \sigma \qquad E \vdash B : \sigma}{E \vdash a \overset{\sigma}{\triangleright} B : \mathcal{W}} \qquad \text{(Wd-Message)}\frac{}{E \vdash a \lhd m(b) : \mathcal{W}}$$

We do not give here the usual introduction and extraction rules for the environment.

The following actor described in CAP instrumented syntax is well decorated, the proof tree is given below.

$a \overset{\{m,p\}}{\triangleright} [m() = \zeta(e,s)(e \overset{\{p\}}{\triangleright} [p() = \zeta(e',s')(e' \overset{\{p\}}{\triangleright} s')]$

$$\dfrac{\{a : \sigma_a\} \vdash a : \sigma_a \qquad \dfrac{\{a : \sigma_a, e : \sigma_e, s : \sigma_a\} \vdash e : \sigma_e \qquad \dfrac{\dfrac{E \vdash e' : \sigma_e \qquad E \vdash s' : \sigma_e}{E \vdash e' \overset{\sigma_e}{\triangleright} s' : \mathcal{W}}}{\{a : \sigma_a, e : \sigma_e, s : \sigma_a\} \vdash [p() = \cdots] : \sigma_e}}{\{a : \sigma_a, e : \sigma_e, s : \sigma_a\} \vdash e \overset{\sigma_e}{\triangleright} [p() = \cdots] : \mathcal{W}}}{\{a : \sigma_a\} \vdash [m() = \cdots] : \sigma_a}}{\{a : \sigma_a\} \vdash a \overset{\sigma_a}{\triangleright} [m() = \cdots] : \mathcal{W}}$$

Avec $E = \{a : \sigma_a, e : \sigma_e, s : \sigma_a, e' : \sigma_e, s' : \sigma_e\}$.

This proof tree generates the following system : $\sigma_a = \sigma_e \cup \{m\}$ and $\sigma_e = \{p\} \cup \sigma_e$ which least solution is : $\sigma_e = \{p\}$ and $\sigma_a = \{m, p\}$.

2 TYPE SYNTAX AND SEMANTICS

To begin with, a crude and informal but intuitive description of the type abstraction is proposed.

Interfaces represent both inputs (i.e., the set of messages which can be handled by an object) and outputs (i.e., the set of requests (or methods) which will be sent to the object). The main difference between objects and actors is that actors can change dynamically their behavior. Therefore, if an interface is associated to each behavior, then a given program execution for an actor can be described by a sequence of interfaces. In order to take into account every possible execution, an actor can be described by a graph whose nodes are the interfaces of the actor's possible behaviors and whose edges correspond to behavior changes. This graph is a regular tree whose root is the initial behavior of the actor. An execution path is then a branch of this tree.

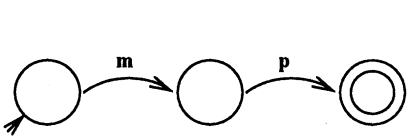
A safe abstraction of all the branches is required in order to be sure that messages sent to the actor will not become safety orphans. A simple way to obtain such an abstraction is to compute the intersection of all the target possible behavior interfaces and then only allow to send messages which are present in this intersection. This solution is yet too restrictive because it forbids the use of message labels which are not present in all behaviors. For example, in the case of the one-place buffer behavior previously defined, no message can be sent to it. Following the work of Kobayashi and Yonezawa [15], union was used instead of intersection in a preliminary work leading to a type system which could only detects crude safety orphans (see [6]).

The analysis proposed in this paper combines the two previous approaches : the multi-union of all the behavior interfaces along a given branch of the tree and the multi-intersection of all the multi-unions. Multi-operators take into account the number of times (ω representing unknown or infinite) a message may be sent or handled. Indeed, the same message can be handled by several behaviors in the same path and each occurrence must be taken into account. In order to deal safely with the message parameter types, a multi-flattening operator which is a natural multi-set extension of the flattening operator described in our previous paper [6] must be defined. This operator combines the possible behaviors of a given actor in a safe way.

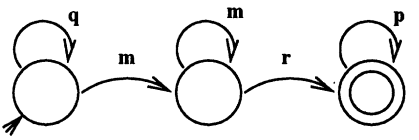
In this new system, interfaces are still used as types but a *multiplicity* (finite or ω) is associated to each method label in the interface (producing label multi-sets instead of sets). When interfaces are used to describe inputs, *multiplicity* represents the number of messages (with this label) that can be handled by the behavior. When interfaces are used for outputs, it collects the quantity of messages that may be sent to a given target.

In the case of outputs, the type represents an upper-bound of what may happen during the execution. To give a better insight on input types, let us consider behaviors as finite state deterministic automata in which transitions are accepted messages without their parameters (a kind of trace semantic). Our aim is to abstract this input by the multi-set of transitions that are common to all paths in the automaton, including infinite ones.

Let us use the two following automata :



Automaton 1



Automaton 2

- the former accepts m then p and is abstracted by $\{m, p\}$ or $\{p, m\}$. As message sending is asynchronous, the abstraction does not take into account the order of transitions (in fact, this information is only required for liveness orphan detection)

- the latter accepts some q, an m, some other m, an r and some p (as described by the regular expression "$q^*mm^*rp^*$") and is abstracted by $\{m, p^\omega, r\}$ because all the recognized sequences contain at least one m, one r and an arbitrary number of p (q^ω and m^ω are omitted because none of these messages can be handled after handling an m and an r whereas p can always be handled).

The second example shows that if there exists a transition that does not appear in all the branches, then, the considered type will not contain this transition.

The main principle of the analysis consists in computing for each name an input type and an output type and in comparing them in a "*multi-set sense*".

2.1 Expressiveness of the type abstraction

In order to get a decidable type system, the safe approximation of actors behaviors rejects some programs which could not lead to safety orphan messages. This abstraction introduces the following constraint in the definition of behaviors : at a given execution time, a message can be sent to an actor if there exists, in each execution path, a future behavior knowing how to treat this message.

Consequently, a message that does not appear in the intersection of all the branches is hidden and should not be sent to the actor. At a given point in the behavior tree, an actor can only be sent safely the messages which are common (and in the same number) to all the branches (however, at the next point, he may be sent more messages if all the branches can handle more messages). For example, if a message m is sent to an actor a, one of the behaviors in each of the branches should be able to handle m.

An *ifthenelse* actor (which has only one behavior which can handle either one message *true*, or one message *false*) has an empty type $\langle\rangle$ and cannot be sent safely any message according to the type system.

In order to override this restriction, an *ifthenelse* actor must accept a *true* (respectively *false*) message and then a *false* (respectively *true*) message.

Practically, this slight burden did not often occur in our actor-based extension of ML where conditionals are treated in the functional part of the language.

However, this restriction is still a significant improvement from the previous object-based abstraction which required that all possible behaviors of an actor must know how to handle every message which may be sent to the actor.

2.2 Type syntax and semantics

$$
\begin{array}{lcll}
\tau & ::= & \alpha \mid \beta \mid \wp & \\
\alpha & ::= & \langle m_i{}^{\mu_i}(\tilde{\alpha}_i)^{i\in I}\rangle \mid \top & \text{– interface type} \\
\beta & ::= & (\alpha, \alpha_o) & \text{– behavior type} \\
\mu & \in & \mathbb{N}^* \cup \{\omega\} & \text{– multiplicity}
\end{array}
$$

A type is built from the grammar given above where $\wp$ corresponds to a well-typed configuration. An actor is approximated by an interface type: the multi-set of the messages it can handle. The lattice of interface types has a top which is denoted : $\top$. The behavior types are composed of two parts α and α_o. The first multi-set represents the messages which can be handled by this behavior (its input type), whereas the second part accumulates the messages sent by a behavior to itself (actors can send messages to their ego).

Multiplicity arithmetic ($\mathbb{N}^* \cup \{\omega\}$) only uses "+" and "−1" defined as usually for naturals and verifying $\omega + \mu = \mu + \omega = \omega$ and $\omega - 1 = \omega$.

Given a set of message names L, the set of all possible types is the usual Herbrand universe $\mathcal{H}$ which is the limit of the following equations:

$$\begin{array}{lcl} \mathcal{H}^0 & = & \{\top\} \\ \mathcal{H}^{n+1} & = & \mathcal{H}^n \cup \{\langle m_i{}^{\mu_i}(t_1^i \ldots t_{k_i}^i)^{i\in I}\rangle \ / \ t_j^l \in \mathcal{H}^n,\ \mu_i \in \mathbb{N}^* \cup \{\omega\},\ \{m_i{}^{i\in I}\} \subseteq L\} \end{array}$$

2.3 Unlimited types and environments

The type system will be defined using two environments. The former will hold the behavior and actor types associated to the definition of behaviors; it will also hold the bindings of names to behaviors. The latter will hold the multi-set of messages which can be sent to actors.

When a message is sent to a free name inside a behavior, this message may be sent several times to the same actor, as the behavior may be assumed several times without any change to the external binding of this variable. Therefore, the multiplicity associated to the sent messages must be set to ω as the number of behavior reductions is unknown. Following the work of Kobayashi et al. in [14], this constraint is introduced in the type system using unlimited types ($\langle m_i{}^{\omega}(\tilde{\alpha}_i)^{i\in I}\rangle$) and unlimited environments (in which all types are unlimited).

2.4 Operations on interface types

The handling of a message in CAP is equivalent to a β-reduction in the λ-calculus. Therefore, an interface type must be contravariant on the arguments of each message. Figure 1 contains the definition of the complete lattice of types ($\mathcal{H}(\Subset, \langle\rangle, \top, \Cup, \Cap)$) and of some other operations used in the typing rules. Intuitive descriptions of these operations are given below.

The *max-union* computes the least upper-bound of two interfaces. It is used to model nondeterminism in the choice of the message which will be handled by a given behavior. Its dual operation, the *min-intersection*, is involved in the contravariant definition.

The *multi-union* is used to combine the effects of several CAP expressions evolving concurrently by adding the various multiplicities of a given message. Yet, as shown in our previous work [6], the *multi-union* is not adequate to represent the safe merging of the current and future behaviors of an actor. Indeed, it cannot produce a type which is compatible with the subtyping relation. We must therefore define *multi-flattening* which safely combines the types corresponding to all possible behaviors of an actor.

Finally, the typing rules will need two more technical operators. One is devoted to express the handling of a message, by decrementing the multiplicity of a given label : *minus*. The other is necessary to generate the unlimited type which is the least upper bound of a given type; i.e. *saturation*.

The comments of the typing rules will give more details about the type operators.

2.5 Relation between term decoration and interface type

Safety orphan messages are defined in subsection 1.3 using a decorated calculus. The decoration defines a complete lattice $\mathcal{D}(\subseteq, \emptyset, L_p, \cup, \cap)$ on the set of messages appear-

max-union $\tilde{\bigcup}$ and min-intersection $\tilde{\bigcap}$:

$$\langle m_i{}^{\mu_i}(\tilde{\alpha}_i)^{i\in I}\rangle \,\tilde{\bigcup}\, \langle m_j{}^{\mu'_j}(\tilde{\alpha}'_j)^{j\in J}\rangle = \left\langle \begin{array}{l} m_i{}^{\mu_i}(\tilde{\alpha}_i)^{i\in I\setminus J}, \\ m_k{}^{Max(\mu_k,\mu'_k)}(\tilde{\alpha}_k \tilde{\sqcap} \tilde{\alpha}'_k)^{k\in(I\cap J)}, \\ m_j{}^{\mu'_j}(\tilde{\alpha}'_j)^{j\in J\setminus I} \end{array} \right\rangle$$

$$\top \,\tilde{\bigcup}\, \_ = \top$$

$$\langle m_i{}^{\mu_i}(\tilde{\alpha}_i)^{i\in I}\rangle \,\tilde{\bigcap}\, \langle m_j{}^{\mu'_j}(\tilde{\alpha}'_j)^{j\in J}\rangle = \langle m_k{}^{min(\mu_k,\mu'_k)}(\tilde{\alpha}_k \tilde{\bigcup} \tilde{\alpha}'_k)^{k\in(I\cap J)}\rangle$$

$$\top \,\tilde{\bigcap}\, \alpha = \alpha$$

multi-inclusion $\tilde{\subseteq}$:

$$\langle m_i{}^{\mu_i}(\tilde{\alpha}_i)^{i\in I}\rangle \,\tilde{\subseteq}\, \langle m_j{}^{\mu'_j}(\tilde{\alpha}'_j)^{j\in J}\rangle \iff I \subseteq J \;\wedge\; (\forall k \in I)(\tilde{\alpha}_k \tilde{\exists} \tilde{\alpha}'_k \;\wedge\; \mu_k \leq \mu'_k)$$

multi-union $\tilde{\uplus}$:

$$\langle m_i{}^{\mu_i}(\tilde{\alpha}_i)^{i\in I}\rangle \,\tilde{\uplus}\, \langle m_j{}^{\mu'_j}(\tilde{\alpha}'_j)^{j\in J}\rangle = \left\langle \begin{array}{l} m_i{}^{\mu_i}(\tilde{\alpha}_i)^{i\in I\setminus J}, \\ m_k{}^{\mu_k+\mu'_k}(\tilde{\alpha}_k \tilde{\sqcap} \tilde{\alpha}'_k)^{k\in(I\cap J)}, \\ m_j{}^{\mu'_j}(\tilde{\alpha}'_j)^{j\in J\setminus I} \end{array} \right\rangle$$

$$\top \,\tilde{\uplus}\, \_ = \top$$

multi-flattening $\tilde{\biguplus}^{\flat}$:

$$\langle m_i{}^{\mu_i}(\tilde{\alpha}_i)^{i\in I}\rangle \,\tilde{\biguplus}^{\flat}\, \langle m_j{}^{\mu'_j}(\tilde{\alpha}'_j)^{j\in J}\rangle = \left\langle \begin{array}{l} m_i{}^{\mu_i}(\tilde{\alpha}_i)^{i\in I\setminus J}, \\ m_k{}^{\mu_k+\mu'_k}(\tilde{\alpha}_k \tilde{\bigcup} \tilde{\alpha}'_k)^{k\in(I\cap J)}, \\ m_j{}^{\mu'_j}(\tilde{\alpha}'_j)^{j\in J\setminus I} \end{array} \right\rangle$$

minus $\alpha \setminus^{+} m_i$:

$$\langle m_i{}^{\mu_i}(\tilde{\alpha}_i)^{i\in I}\rangle \setminus^{+} m_k = \begin{cases} \langle m_i{}^{\mu_i}(\tilde{\alpha}_i)^{i\in I\setminus\{k\}}, m_k^{\mu_k-1}(\tilde{\alpha}_k)\rangle & \text{if } k \in I \text{ and } \mu_k > 1; \\ \langle m_i{}^{\mu_i}(\tilde{\alpha}_i)^{i\in I\setminus\{k\}}\rangle & \text{otherwise.} \end{cases}$$

$$\top \setminus^{+} m = \top$$

saturation:

$$sat^{\omega}(\langle m_i{}^{\mu_i}(\tilde{\alpha}_i)^{i\in I}\rangle) = \langle m_i{}^{\omega}(\tilde{\alpha}_i)^{i\in I}\rangle$$

Figure 1 The lattice of interface types and other operations on types

ing in a given CAP term. An interface type can be abstracted by the corresponding decoration, and give the greatest type associated to a decoration :

$$A_b : \mathcal{H} \longmapsto \mathcal{D} \qquad C_o : \mathcal{D} \longmapsto \mathcal{H}$$
$$A_b(\langle m_i{}^{\mu_i}(\alpha_i)^{i\in I}\rangle) = \{m_i \mid i \in I\} \qquad C_o(\{m_i \mid i \in I\}) = \langle m_i{}^{\omega}(\langle\rangle)^{i\in I}\rangle$$

Proposition 2.1 *The pair* (A_b, C_o) *is a Galois connection between* $\mathcal{D}$ *and* $\mathcal{H}$ *:*

$$\forall \alpha \in \mathcal{H}, \forall \sigma \in \mathcal{D} : (A_b(\alpha) \subseteq \sigma) \Longleftrightarrow (\alpha \nsubseteq C_o(\sigma))$$

This pair will be used to express the type system on decorated terms and to show that our inference algorithm can be applied on a non-decorated CAP expression.

3 TYPE SYSTEM

The following type system checks that there is no potential safety orphan message in a CAP expression. In fact, this system will be defined for the decorated CAP calculus in order to present the relation between the interface used in the dynamic detection of safety orphan messages and the abstraction of an actor type. However, the resolution algorithm has been devised in order to be applied on pure CAP terms. The decoration will be computed and implicitly taken into account by the inference algorithm.

A typing judgment for a CAP expression is : $E; O \vdash Exp : \tau$. E is a standard typing environment for names and behaviors. O is a message output typing environment which holds the messages sent to the free names in the expression. The operations on interfaces are easily extended to environments.

$$\text{(Empty)}\frac{}{E; \emptyset \vdash \phi : \wp} \qquad \text{(Parallel)}\frac{E; O_C \vdash C : \wp \quad E; O_D \vdash D : \wp}{E; O_C \uplus O_D \vdash C \parallel D : \wp}$$

Following Kobayashi et al. in [14], in the (Parallel) rule, the composition must be typed in the $O_C \uplus O_D$ environment, to take into account the messages in each sub-configuration.

$$\text{(Restriction)}\frac{E, a : \alpha\,;\, O, a : \alpha_o \vdash C : \wp}{E; O \vdash \nu a\, C : \wp}(\alpha \ni \alpha_o)$$

Checking for safety orphan messages is done upon escaping the scope of the name "a".

$$\text{(Behavior)}\frac{\begin{array}{l}\forall i \in I \;\; (s_i \in \mathcal{FV}(C_i) \Rightarrow O_i \text{ Unlimited}) \\ E, e_i : \alpha_{e_i}, s_i : (\alpha, \alpha_o), x_i : \langle\rangle\,; \\ O_i, e_i : \alpha_{o_{e_i}}, \tilde{x_i} : \tilde{\alpha_i}\end{array} \vdash C_i : \wp}{E, \biguplus_{i\in I} O_i \vdash [m_i(\tilde{x_i}) = \zeta(e_i, s_i)C_i{}^{i\in I}] : (\alpha, \alpha_o)} \left(\begin{array}{ll} \alpha_{e_i} \ni \alpha_{o_{e_i}} \uplus (\alpha_o \setminus^{+} m_i) & (1) \\ \alpha = \bigsqcap_{i\in I} (\langle {m_i}^1(\tilde{\alpha_i})\rangle \uplus^{\flat} \alpha_{e_i}) & (2)\end{array}\right)$$

The (Behavior) rule introduces the constraints on the various variables which are substituted during a communication (self, ego, message parameters). As we are escaping the scope of ego (e_i), we must check (as in the (Restriction) rule) that the messages sent to it can be handled ($\alpha_{e_i} \ni \alpha_{o_{e_i}}$). Furthermore, after handling m_i it must

also be able to treat all the other messages sent to the actor before the communication took place ($\alpha_o \stackrel{+}{\backslash} m_i$). These two constraints produce (1).

The behavior associated type is the *min-intersection* of the flattening of all possible sequences of behavior changes starting from one of the m_i messages. It is computed by the equation (2) to build the multi-set of all the messages which may be handled whatever execution path is chosen. To take in account all possible execution paths, the environment is the least upper bound of all the messages sent in all the configurations C_i ($\biguplus_{i \in I} O_i$).

As the self-application introduces recursion in the definition of behaviors, it is not possible to determine statically how many times a given behavior will be duplicated. Therefore, the message sent to the free variables in a behavior should be taken as unlimited (the messages are labelled with ω) via the constraint introduced by "O_i Unlimited". But when s_i is unused in C_i, the behavior is not recursive and O_i is not required to be unlimited.

$$\text{(Actor)}\frac{E, a : \alpha; O, a : \alpha_o \vdash \mathit{Behavior} : (\alpha, \alpha'_o)}{E; O \vdash a \overset{\sigma}{\triangleright} \mathit{Behavior} : \wp} \quad \left(\begin{array}{c} \alpha'_o \supseteq \alpha_o \\ A_b(\alpha) \subseteq \sigma \end{array} \right)$$

The binding of a behavior to a name requires that both have the same interface and that the behavior can handle the messages sent on the name ($\alpha'_o \supseteq \alpha_o$). We also check that all the messages that the actor can handle are declared in the decoration. This last constraint will be implicitly ensured by the resolution algorithm used in the type inference system.

$$\text{(Message)}\frac{}{E; \{x : \langle m^1(\alpha_{o_y})\rangle\} \uplus \{y : \alpha_{o_y}\} \vdash x \lhd m(y) : \wp}$$

In a message sending, the message itself ($\{x : \langle m^1(\alpha_{o_y})\rangle\}$) and the messages ($\{y : \alpha_{o_y}\}$) which may be sent to its arguments are introduced in the output environment.

3.1 Properties of the type system

Proposition 3.1 (Structural preserving of type assignment) *If $A \equiv B$ and $E; O \vdash A : \alpha$ then $E; O \vdash B : \alpha$.*

Proposition 3.2 (Subject reduction) *If A is a* CAP *term such that $E; O \vdash A : \tau$ with $\forall a \in Dom(O), E(a) \supseteq O(a)$ and $A \longrightarrow A'$ then there exists E', O' and τ' verifying $E'; O' \vdash A' : \tau'$ and $\forall a \in Dom(O'), E'(a) \supseteq O'(a)$.*

More precise information can be given for the new environments E', O' and τ'. If the reduction step is a communication (rule COMM) taking place on name a, then $\forall x \neq a$, $E'(x) = E(x) \wedge O'(x) = O(x)$ and $\tau' = \wp$. For the other reduction rules, the environments and the type are unchanged.

The subject reduction property takes a much simpler form if the typed expression is closed (a program).

Proposition 3.3 (Subject reduction on closed terms) *If C is a closed* CAP *configuration such that $\emptyset, \emptyset \vdash C : \wp$ and $C \longrightarrow C'$ then $\emptyset, \emptyset \vdash C' : \wp$.*

The complete proofs of these properties are detailed in [3].

Proposition 3.4 (Type system soundness) *If the typing of a closed* CAP *expression A succeeds then its reduction can not produce any safety orphan message.*

PROOF: The semantic value *Error* from the COMM-ERR and ORPH-ERR rules cannot be typed. Then the subject reduction property ensures that any typable expression will not be reduced to an untypable one. □

3.2 Typing examples

Linear cell (changing interfaces actors). The following storage cell behavior is initially empty, it can be written once and then can be read forever :

$$Lin\_cel \equiv [\, put(v) = \zeta(e_e, \_)(e_e \rhd [get(c) = \zeta(e_f, s_f)(c \lhd rep(v) \;\|\; e_f \rhd s_f)]) \,]$$

It has the type: (α, α_o) with:

- $\alpha = \langle get^{\omega}(\langle rep^1(t_v)\rangle) \;,\; put^1(t_v)\rangle$ where t_v is the type of the argument v,
- α_o represents the messages sent to the behavior in its context. Without any specific context, it takes the empty value $\alpha_o = \langle\rangle$.

α gives a rather informative abstraction of this behavior. It specifies all the messages that this behavior will be able to handle, that is, exactly one *put* message and any number of *get* messages. It also specifies that the parameter of a *get* message must be able to handle one *rep* message. However, it does not specify that the *put* message must be handled before any *get* message can be treated.

Let us use this behavior in the following configuration :

$$C_1 \equiv a \rhd Lin\_cel \;\|\; d \rhd [m() = \zeta(e, s)\phi] \;\|\; a \lhd put(d)$$

C_1 is well typed, so we have $E_1, O_1 \vdash C_1 : \wp$. E_1 is such that the input type of the actor d is $E_1(d) = \langle m^1()\rangle$; which means that "$d$ can only handle the message m and that it will handle it only once".

Let us consider another configuration :

$$C_2 \equiv \nu a, b, d(C_1 \;\|\; a \lhd get(b) \;\|\; b \rhd [rep(r) = \zeta(e, s)(r \lhd m() \;\|\; e \rhd s)] \,)$$

This configuration cannot be typed. The input type of d is $\langle m^1()\rangle$ and its output type is $\langle m^{\omega}()\rangle$. The constraint in the (Restriction) rule for d ($\langle m^1()\rangle \trianglerighteq \langle m^{\omega}()\rangle$) is not satisfied. This failure results from the following interpretation : d is held in the linear cell and its value is required by b. As other actors could require the value from the cell, the type system must take into account that other messages m may be sent to the value stored in the cell. Therefore, it is necessary to consider the message sent to the value returned by the cell as unlimited.

Replicated computation (short lifetime objects). In order to replicate computation carried out in a client/server model, computation requests should be sent to an actor *rep* which will duplicate and forward the request to both servers *hal* and *sal*. *rep* will also create for each *eval* request a new, short-lifetime, actor *jn* whose purpose is to receive the results from both servers and to forward one of the result to the initial client. In the following example, the *join* actor *jn* forwards the first result and eliminates the second one.

$$
\begin{array}{l}
C_3 \equiv \quad rep \rhd [\ eval(\ c) = \zeta(e,s)(\quad e \rhd s \\
\qquad\qquad\qquad\qquad\qquad\qquad \| \ \nu jn(hal \lhd eval(\ jn) \\
\qquad\qquad\qquad\qquad\qquad\qquad \| \ sal \lhd eval(\ jn) \\
\qquad\qquad\qquad\qquad\qquad\qquad \| \ jn \rhd [\ result(\ v) = \zeta(e,s)(\quad c \lhd result(\ v) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \| \ e \rhd [\ result(\ \_) = \zeta(e,s)(\phi)]) \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad])) \\
\qquad] \\
\qquad \| \ hal \rhd [\ eval(\ c) = \zeta(e,s)(\cdots \ \| \ c \lhd result(\ vv) \ \| \ e \rhd s)\ ] \\
\qquad \| \ sal \rhd [\ eval(\ c) = \zeta(e,s)(\cdots \ \| \ c \lhd result(\ ww) \ \| \ e \rhd s)\ ]
\end{array}
$$

This configuration is well-typed ($E_3, O_3 \vdash C_3 : \wp$) with:

$$
E_3 = \left\{ \begin{array}{l} rep : \langle eval^{\omega}(\langle result^1(t_v)\rangle)\rangle; \\ hal : \langle eval^{\omega}(\langle result^1(t_{vv})\rangle)\rangle; \\ sal : \langle eval^{\omega}(\langle result^1(t_{ww})\rangle)\rangle \end{array} \right\}
$$

$$
O_3 = \{rep : \langle\rangle; hal : \langle eval^{\omega}(\langle result^1(t_{vv})\rangle)\rangle; sal : \langle eval^{\omega}(\langle result^1(t_{ww})\rangle)\rangle\}
$$

The environments E and O before applying the (Restriction) rule for the name jn are such that $E(jn) = \langle result^2(t_v)\rangle$ and $O(jn) = \langle result^2(t_{vv} \fatsemi t_{ww})\rangle$. This example presents the use of short lifetime objects such as jn. The static analysis ensures that jn does not introduce any safety orphan messages (it can handle two messages and will be sent less than two).

4 THE INFERENCE ALGORITHM

The previous type system is defined for decorated CAP expressions; the inference algorithm decides if a non-decorated expression is typable or not. The application of the previous typing rules is completely syntax-directed, so these rules can be used to extract constraints on type variables, leading to a system that will be solved by the algorithm. The constraints are the side-conditions appearing on the previous rules, except $A_b(\alpha) \subseteq \sigma$ appearing on the (Actor) rule that will be automatically verified by the strategy of the algorithm.

There are two other kinds of constraints: equalities, which define the input type associated to a name, and multi-inclusions for checking communication safety. In summary, the algorithm is based on three steps:

1. constraints extraction from the non-decorated expression (not detailed here);
2. computation of input types (equalities resolution);
3. inclusion constraints solving.

The resulting algorithm (see [3, 7]) is complete in the sense that it always terminates and its answer is equivalent to the typability problem.

5 RELATED WORK AND CONCLUSION

Many different studies are currently related to the static analysis of non-uniform service availability.

Following lines proposed by Nierstarz [18], Ravara and Vasconcelos [23] on one hand, and Najm and Nimour [17] on the other hand, propose the use of more sophisticated type abstraction (in fact, the structure of their types is a process calculus). Their work is very promising as their abstractions preserve more causal relations. However, the feasibility of type inference is still a conjecture. In this purpose, a joint work between the first author and Ravara has recently begun using a more sophisticated constraint resolution algorithm than the one presented in this paper.

Our approach for the abstraction of message sendings was derived from the effect system proposed by Kobayashi et al. [13]. The main difference is that they must consider finite sets of possible values determinated by an integer M $(0, \ldots, M-1, \omega)$ which represent the accuracy of the analysis in order to have a terminating algorithm. Our algorithm terminates without choosing a maximal value.

The type system presented in this paper rejects statically all CAP terms which may lead to safety orphan messages. This type system has been implemented using CaML-light. This prototype will be integrated in the type system of ML-ACT, an actor based extension of ML (see [8]).

A first restriction for the use of this type system results from the safe type abstraction which rejects some correct programs as developed in 2.1. This restriction follows from the (Behavior) rule which computes a safe approximation of the messages sent in all the configurations C_i $(\biguplus_{i \in I} O_i)$. The same kind of limitation is found in Kobayashi et al. [13]. In order to overcome this restriction, the introduction of a conditional type operator as advocated by Aiken et al. [2] and the authors [19] is currently under investigation.

A second restriction follows from deadlocks in programs which can still produce messages which will not be handled even if they are not safety orphans in the sense described in this paper (they can be handled in the future but never will be because of the deadlocks, for example, sending *get* messages to a linear buffer without ever sending a *put* message).

Therefore, the second part of our future work will be the introduction of causality information to detect some deadlocks along the line proposed by Kobayashi in [12]. Each binding of a behavior to an address and each sending of a message to an actor will be decorated with a time-stamp, ordered by the imbrication of the term structure. Cyclic time-stamp chains can then be interpreted as potential deadlocks.

Apart from improving the type abstraction and the multiset constraints solver, the precise information (maximum number of messages which may be handled and which will be sent) synthesized by the type system will be used in order to improve the garbage collection strategy and to optimize the code generated by the ML-Act compiler (mainly by using arrays as mailboxes instead of lists).

References

[1] M. Abadi and L. Cardelli. A theory of primitive objects: untyped and first-order systems. In *Proc. Theor. Aspects of Computer Software*, 1994.

[2] A. Aiken, E. Wimmers, and T. Lakshman. Soft typing with conditional types. In *Proc. of the 21st. ACM Symp. on PoPL*, January 1994.

[3] J-L. Colaço. *Analyses statiques par typage de langages d'Acteurs.* PhD thesis, Institut National Polytechnique de Toulouse, October 1997.

[4] J-L. Colaço, M. Pantel, and P. Sallé. CAP: An actor dedicated process calculus. In *ECOOP'96 Workshop on Proof Theory of Concurrent Object-Oriented Programming*, May 1996.

[5] J-L. Colaço, M. Pantel, and P. Sallé. Analyse de linéarité par typage dans un calcul d'acteurs. In *Actes des Journées Francophones des Langages Applicatifs*, January 1997.

[6] J-L. Colaço, M. Pantel, and P. Sallé. A set-constraint-based analysis of actors. In *Proc. of the 1997 IFIP Intl. conf. on Formal Methods for Open Object-based Distributed Systems*, July 1997.

[7] J-L. Colaço, M. Pantel, and P. Sallé. From set based to multiset based analysis: a practical approach. October 1998. Proc. of the the 1998 Workshop on Set constraints and Constraint based program analysis.

[8] F. Dagnat, M. Pantel, and P. Sallé. Ml-act : Un langage fonctionnel d'acteurs. In *Actes des Journées Francophones des Langages Applicatifs*, February 1998.

[9] J. Darlington and Y.K. Guo. Formalising actors in linear logic. In *Proc. of the Intl. Conf. on Object-Oriented Information Systems*, 1994.

[10] C. Fournet, C. Laneve, L. Maranget, and D. Rémy. Implicit typing à la ML for the join-calculus.

[11] Kohei Honda and Mario Tokoro. An object calculus for asynchronous communication. In P. America, editor, *Proc. ECOOP '91*, July 1991.

[12] N. Kobayashi. A partially deadloack-free typed process calculus. In *Proc. of the conf. Logic In Computer Science*, 1997.

[13] Naoki Kobayashi, Motoki Nakade, and Akinori Yonezawa. Static analysis of communication for asynchronous concurrent programming languages. In *Proc. of Intl. Static Analysis Symposium*, 1995.

[14] Naoki Kobayashi, Benjamin C. Pierce, and David N. Turner. Linearity and the pi-calculus. In *Proc. of the ACM Symposium on Principles of Programming Languages*, 1996.

[15] Naoki Kobayashi and Akinori Yonezawa. Type-theoretic foundations for concurrent object-oriented programming. In *Proc. of ACM SIGPLAN Conf. on Object-Oriented Programming Systems, Languages, and Applications*, 1994.

[16] Robin Milner. The polyadic π-calculus: a tutorial. Technical Report ECS-LFCS-91-180, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, UK, October 1991. Also in *Logic and Algebra of Specification*, ed. F. L. Bauer, W. Brauer and H. Schwichtenberg, Springer-Verlag, 1993.

[17] E. Najm and A. Nimour. A calculus of object binding. In *Proc. of the 1997 IFIP intl. conf. on Formal Method s for Open Object-based Distributed Systems*, July 1997.

[18] Oscar Nierstrasz. Regular types for active objects. In *Proc. OOPSLA '93, ACM SIGPLAN Notices*, October 1993.

[19] M. Pantel and P. Sallé. Typage souple pour le langage FOL. In *Actes des Journées Francophones des Langages Applicatifs*, January 1994.

[20] B. Pierce and D. Turner. Concurrent objects in a process calculus. In *Proc. of Theory and Practice of Parallel Programming*, November 1994.

[21] B. Pierce and D. Turner. Pict: A programming languages based on the π-calculus. Technical Report 476, Indiana Univ., March 1997.

[22] Benjamin Pierce and Davide Sangiorgi. Typing and subtyping for mobile processes. *Mathematical Structures in Computer Science*, 1995.

[23] António Ravara and Vasco T. Vasconcelos. Behavioural types for a calculus of concurrent objects. In *Euro-Par'97*, 1997.

[24] Vasco T. Vasconcelos and Mario Tokoro. A typing system for a calculus of objects. In *Proc. of the Intl. Symposium on Object Technologies for Advanced Software*, November 1993.

[25] D. Walker. Objects in the π-calculus. *Information and Computation*, (116), 1995.