Temporal Logic Based Static Analysis for Non-uniform Behaviours

Matthias Colin, Xavier Thirioux, and Marc Pantel

ENSEEIHT-IRIT
2, rue Camichel
31071 Toulouse Cedex
France
{colin,thirioux,pantel}@enseeiht.fr

Abstract. The main purpose of our work is the typing of concurrent, distributed and mobile programs based on the actor programming model, that is non-uniform behaviour concurrent objects communicating by asynchronous message passing. One of the key difficulties is to give a precise definition of "message not understood" errors in this context. In this paper, we investigate temporal logic and model-checking based technologies for an asynchronous message passing process calculi. We focus on non uniform input interfaces for processes, and then define a temporal logic tailored to their description and analyses. This logic deals with infinite-state systems, as mailboxes of actors are unbounded multisets of messages, but yet happens to be decidable. We use our logic to specify possible communication errors in actor-based programs in order to give precise and sound definition of type disciplines.

Introduction

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 the unreliability of networks makes it difficult to validate any distributed functionality using informal approaches, our work is focused on applying formal type systems to improve concurrent object oriented programming.

To obtain widely usable tools, we have chosen to use the actor model proposed by Hewitt in [HBS73] and developed by Agha in [Agh86]. 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

E. Najm, U. Nestmann, and P. Stevens (Eds.): FMOODS 2003, LNCS 2884, pp. 94–108, 2003.

actor may handle, yielding a more accurate programming model. This model, also known as concurrent objects with non uniform behaviour (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). Due to behaviour change, it may happen that a message cannot be handled by its target in some execution path (a sequence of behaviours the actor can assume) and could be handled in some other path. Such messages are called orphan messages.

Type systems for concurrent objects and actors, with uniform or non-uniform behaviours, have been the subject of active research in the last years ([RV00], [Nie95], [Kob97], [Pun96], [NNS99], [FLMR97], [Bou97] and their more recent works). Two opposite approaches have been followed: explicit and implicit typing. Explicit types (see [RR01,CRR02]) 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). We advocate the use of implicit typing, i.e. type inference, as it is simpler to use.

In a first approach, our type systems were defined using Cap, an actor calculus derived from asynchronous π-calculus and Cardelli's Calculus of Primitive Objects (see [CPS96]). Two type systems were developed: the former (see [CPS97]) is based on usual object type abstractions and catches all functional and communication errors but only trivial orphans, the latter detects a large set of orphans but requires a much more complex type abstraction (see [CPDS99]).

Despite several unsuccessful attempts to provide formal definitions for the various kinds of orphan messages we are interested in ([CPDS99] and [DPCS00]), most of the time, the best definition we could give was: *orphan messages are the messages detected by our analyses* which was neither satisfying nor useful. The purpose of this paper is to present a decidable temporal logic dedicated to the specification of orphan messages which we use to compute various notions of input capacities for actors.

We initially give a short description of Cap, a primitive actor calculus used to define our static analyses.

1 Cap: A Primitive Actor Calculus

Various encodings of concurrent objects in the π-calculus or similar formalisms have been proposed [Wal95,PT97]. Message labels and actors mail addresses are usually both expressed using names. Therefore, the typing of encoded programs generally leads to type information which does not reflect the structure of the original program. The authors defined in [CPS96] the Cap calculus in order to overcome these constraints.

As in the π-calculus [Mil91] and in the ν-calculus [HT91] the basis of the calculus is the *name* representing the actors mail addresses. Following Abadi and Cardelli's calculus of Primitive Objects [AC94], actor's different behaviours 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 behaviours and addresses as primitives that allow to express very easily actor programs. We will assume some restrictions on Cap programs in order to define our analyses in a strict actor framework.

The remaining part of this section is devoted to a quick introduction to Cap syntax (a more precise presentation of the calculus and its semantic are available in [CPS96,CPS97]).

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 actor names a and b using the ν operator. An actor is built (via $\rhd$) by the association of an address (a) and a behaviour. In the previous example, the behaviour of a has two states recursively defined (via ζ). The first state (the *empty* buffer) only understands one *put* message and then switches to the second state (the *full* buffer) where it can understand 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 behaviour* to s (called *self*). This operator is inspired by the ς defined by Abadi and Cardelli [AC94] to formalize self-substitution in objects. In our context, the capture of *self* and *ego* is used to formalize behaviour changes without introducing a fixpoint operator.

To define Cap syntax the following sets are used: Var an infinite set of variable symbols (x,x_i,e_i,$s_i \in Var$) and $Label$ a finite set of feature labels ($m_i \in Label$). Sequences of symbols are represented by a tilde ($\tilde{}$).

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

$$\begin{array}{rcl} C & ::= & \emptyset \mid C \parallel C \mid \nu x.C \mid x \rhd T \mid x \lhd m(\tilde{T}) \mid (C) \\ T & ::= & [m_i(\tilde{x_i}) = \zeta(e_i, s_i)C_i]^{i \in I} \mid x \mid (T) \end{array}$$

The configurations C represent respectively: an empty process, processes in parallel, creation of actor's name, an actor named x waiting for a message (input transition) with T as its current behaviour and finally a message labelled m with its parameters $\tilde{T}$ sent to an actor x (output transition).

The terms T are either behaviours (between brackets) or variables. A behaviour is a set of reactions C_i to the labelled messages m_i (with their parameters $\tilde{x_i}$) that an actor can handle at a given time.

Syntactically, the sharing of the same address by several different actors is not forbidden. So, we will assume that Cap programs respect a linearity hypothesis: actors are not allowed to change the behaviour of other actors and only one

behaviour is associated to an actor at each time (changing behaviour is only expressed using the *ego* variable: we will write $a \rhd [m_i() = \zeta(e_i, s_i)C \parallel e_i \rhd b_i]$ and not $a \rhd [m_i() = \zeta(e_i, s_i)C \parallel a \rhd b_i]$). Moreover, in this paper, values in parameters are only names of actors and not behaviours; this point will be discussed in conclusion.

2 Computation of the Interface of an Actor

The interface or behavioural type of an actor shall merely denote a finite-state transition system, where each state (respectively each transition) corresponds to an input-waiting state of the actor (respectively a label of an understood message at this current state). Thus we abstract away output events, as well as dynamic creation of actors, and even messages contents. Of course, as such, our analysis here is not intended to deal with properties of a complete actor (or process-calculi) term, but to focus on the specific input part of such a term. We insist on the fact that we don't provide in this paper any type discipline for a full actor language (as it is the main subject of some of our past papers [CPS97,CPDS99]).

A first part describes the translation of an actor term into a behavioural type. Then a finite-state transition system for these types is defined, as an operational semantics. We are indeed interested in the product system $Mailbox \times Control$, where $Mailbox$ represents the set of possible mailbox contents, i.e. multisets of labels and $Control$ is the first transition system. Then we give the operational semantics of this product system, which we shall further call an asynchronous transition system. Due to the unbounded nature of the mailbox, this system is an infinite-state system, but yet from any given initial state, only a finite number of steps can occur, until the mailbox is empty or we are stuck in a state where input capabilities don't match the mailbox content.

2.1 Behavioural Types

We adopt the TyCO syntax for behavioural types [RV00], i.e. types that denote the possible sequences of method invocations for a given actor. We recall here the grammar of behavioural types (without the "‖" operator and message parameters):

Definition 1 (Behavioural Types).

$$\alpha \ ::= \ \Sigma_{i \in I} \mathtt{m}_i.\alpha_i \mid \Sigma_{i \in I} \tau.\alpha_i \mid \mu t.\alpha \mid t \mid \mathtt{0}$$

The labelled sum $\Sigma_{i \in I} \mathtt{m}_i.\alpha_i$ gathers together several method types to form the type of an actor that offers the corresponding set of methods. $\mathtt{0}$ denotes the sum with the empty indexing set.

The silent sum $\Sigma_{i \in I} \tau.\alpha_i$ is used to represent the internal non-determinism of an actor (a choice for instance). As soon as this choice is carried out, the actor behaves according to one of the types α_i.

2.2 Extraction of Behavioural Types

In this section, we present how to extract behavioural types from a CAP program. The principle is to keep only information related to the actors' input capacities. The type of each actor is produced and then analysed (represented by the function $\mathcal{O}$). We ignore the sending of messages and the parameters (as they are not behaviours).

As the creation of names and the installation of behaviours are separated, we have to carry a list of associations (variable of actor $\mapsto$ behavioural type) produced by the $x \triangleright T$ configurations and then to extract the right behavioural type in the creation configurations $\nu x.C$. The list of associations is represented by ML lists (the operator "::" adds an element to a list and "@" merges two lists).

The configurations and the terms are typed in an environment Δ containing the *self* variables of each behaviour. The analysis of a term only returns one behavioural type, considering the linearity hypothesis we have made.

The following rules describe the process of extracting behavioural types and are explained thereafter:

$$\frac{}{\Delta \vdash \emptyset : \mathtt{0}} \qquad \frac{}{\Delta \vdash x \lhd m(\tilde{T}) : \mathtt{0}} \qquad \frac{\Delta \vdash C_1 : l_1 \quad \Delta \vdash C_1 : l_2}{\Delta \vdash C_1 \parallel C_2 : l_1 @ l2}$$

$$\frac{\Delta \vdash C : l \quad \mathcal{E}(x, l) = (\alpha, l') \quad \mathcal{O}(\alpha)}{\Delta \vdash \nu x.C : l'} \qquad \frac{\Delta \vdash T : \alpha}{\Delta \vdash x \triangleright T : [x \mapsto \alpha]}$$

$$\frac{\Delta, s_i : \alpha \vdash C_i : l_i \quad \mathcal{E}(e_i, l_i) = (\alpha_i, [])}{[m_i(\tilde{x_i}) = \zeta(e_i, s_i) C_i]^{i \in I} : \mu\alpha.\Sigma_{i \in I} \mathtt{m}_i.\alpha_i} \qquad \frac{}{\Delta \vdash x : \Delta(x)}$$

$$\frac{\mathcal{E}(x, l) = (\alpha', l')}{\mathcal{E}(x, y \mapsto \alpha :: l) = (\alpha', y \mapsto \alpha :: l')} \qquad \frac{}{\mathcal{E}(x, x \mapsto \alpha :: l) = (\alpha, l)} \qquad \frac{}{\mathcal{E}(x, \emptyset) = (\mathtt{0}, \emptyset)}$$

The empty configuration as well as the sending of message have the null type $\mathtt{0}$. The parallel configuration gathers the associations from the two configurations.

The typing of the name creation consists in typing the configuration and then in extracting the behavioural type of this name, which is then analysed by the system described in the next sections (function $\mathcal{O}$). The resulting configuration returns the trailing associations. The installation of a behaviour produces a list with one element: the association of the variable x and the behavioural type resulting from the analysis of the term T (considering our hypothesis).

A reaction to a message is typed in an environment containing the *self* associated with the final type of the behaviour. The type of the next behaviour is extracted from the list of associations l_i which contains at least one element (the next behaviour is precised or not and an actor can't install a behaviour on another actor). This point is expressed by enforcing the outcome of the function $\mathcal{E}$ to be the empty list. Finally, the behavioural type is expressed by a fixpoint and gathers all the branches of the behaviour expression.

The type of a variable is given by the environment Δ if it represents effectively a behaviour (as we consider correct programs according to the first type system of [CPS97]).

The extraction function $\mathcal{E}(x, l)$ gives the behavioural type associated with x in the list l, if it exists, or the null type $\mathtt{0}$ instead. It also returns the tail of the list (without the association concerning x).

Example. According to this system, the one-slot buffer has the following simple type: $\mu\alpha.get.\mu\alpha'.put.\alpha$.

We can remark that we have not used the sum type with anonymous transitions. This type is required for languages allowing a kind of choice (for instance the conditional statement *if*).

2.3 Operational Semantics of Behavioural Types

We give a semantics of the types via a labelled transition relation where l denotes an element of $Label \cup \{\tau\}$. A label $\mathtt{m}_i$ - a basic transition - corresponds to the invocation of a method with name $\mathtt{m}_i$ whereas the label τ denotes a silent transition that corresponds to an internal hidden computation.

Our transition rules for the control part take the form $\Gamma \vdash p \xrightarrow{l} q$, where Γ is an environment of (recursive) behaviour definitions, p and q are behaviours (α-terms), and l is the label of the transition, $\mathtt{m}_i$ or τ. The term $\Gamma[t]$ denotes the definition p associated to the variable t in Γ, this association being denoted $t \mapsto p$.

Definition 2 (Transition rules for processes control part).

$$\frac{j \in I}{\Gamma \vdash \Sigma_{i\in I} l_i.\alpha_i \xrightarrow{l_j} \alpha_j} \qquad \frac{\Gamma \vdash \Gamma[t] \xrightarrow{l} p}{\Gamma \vdash t \xrightarrow{l} p} \qquad \frac{\Gamma \cup \{t \mapsto p\} \vdash t \xrightarrow{l} q}{\Gamma \vdash \mu t.p \xrightarrow{l} q}$$

2.4 Operational Semantics of Asynchronous Systems

We can now build the product system of control transitions and data transitions. the transition rules take the form $\Gamma \vdash (p, \omega) \xrightarrow{\mathtt{m}} (p', \omega')$ where ω (respectively ω') is the mailbox (i.e. a set of messages) relative to p (respectively to p'). Therefore τ-transitions have been discarded.

Definition 3 (Transition rules for asynchronous processes).

$$\frac{\Gamma \vdash p \xrightarrow{\mathtt{m}} q}{\Gamma \vdash (p, \omega \cup \{\mathtt{m}\}) \xrightarrow{\mathtt{m}} (q, \omega)} \qquad \frac{\Gamma \vdash p \xrightarrow{\tau} q \qquad \Gamma \vdash (q, \omega) \xrightarrow{\mathtt{m}} (q', \omega')}{\Gamma \vdash (p, \omega) \xrightarrow{\mathtt{m}} (q', \omega')}$$

When not necessary, we shall merely ignore the environment part (Γ) of transition systems and consider only systems where Γ is abstracted away.

2.5 About Mailboxes

A mailbox, i.e. a multiset of labels, may be seen as a strictly positive integer vector (or equivalently as a function) in $\mathbb{N}^{Label}$, and we feel free to omit this coercion and to test membership of mailbox in constraint solutions when clear from the context. In the remaining, the main issue about these multisets will be to determine whether it is possible or not to define their possible values within Presburger arithmetics (additive integer arithmetics), i.e. as Presburger formulas about their components as integer vectors. The set *Label* always depends upon a given actor term under analysis, in which the number of different labels is obviously finite and known.

3 Expressing Input Capabilities of Asynchronous Systems

The type we extract from an actor is not the end of the road, as we must now work out the possible meanings of *message-not-understood* like errors, with in mind the ability to devise type systems from these definitions.

The dedicated logic we propose to explore these issues is a very expressive mix of CTL-like temporal features (see [CES86]) to specify future behaviours and Presburger arithmetics (see [Pug91]) to specify mailbox contents, well suited to expressing pervasive and important properties relative to existence of reduction paths and emptiness of mailbox. We first define the grammar $\mathcal{F}$ of such properties.

3.1 Logic for Asynchronous Systems (LAS)

Definition 4 (The temporal logic LAS).

$$\begin{array}{rcll} \mathcal{F} & ::= & \exists\Diamond\ \mathcal{F} \mid \exists\bigcirc\ \mathcal{F} & \textit{(temporal operators)} \\ & & \mid \neg\mathcal{F} \mid \mathcal{F} \vee \mathcal{F} \mid \textit{true} & \textit{(boolean operators)} \\ & & \mid \exists\Uparrow\ \mathcal{F} \mid \exists x.\mathcal{F} \mid \mathcal{E} \geq \mathcal{E} & \textit{(arithmetical operators)} \end{array}$$

where $\mathcal{E}$ denotes an expression belonging in Presburger arithmetics. As usual, the other standard relational operators ($=, \neq, >, \leq$ and $<$) can easily be retrieved from boolean operations and $\geq$.

Variables occurring in arithmetical expressions are either fresh quantified integer variables (introduced with $\exists x \ldots$) or label variables denoting the number of corresponding messages (with the same label) in the mailbox. For a constraint $e_1 \geq e_2$, the term $[\![e_1 \geq e_2]\!]$ shall denote its set of integer solutions.

Then, we give the satisfaction relation of our logic with respect to the underlying transition system.

Definition 5 (Satisfaction relation for LAS). *The satisfaction relation between a configuration (p, ω) and a formula f, denoted $(p, \omega) \vDash f$, is the smallest fixpoint of the following structural inductive rules on f:*

$$\frac{\exists \omega' :\ (p, \omega \cup \omega') \vDash f}{(p,\omega) \vDash \exists \Uparrow f} \qquad \frac{\exists (p',\omega') :\ \vdash (p,\omega) \longrightarrow^* (p',\omega') : (p',\omega') \vDash f}{(p,\omega) \vDash \exists \Diamond\ f}$$

$$\frac{(p,\omega) \nvDash f}{(p,\omega) \vDash \neg f} \qquad \frac{\exists (p',\omega') :\ \vdash (p,\omega) \longrightarrow (p',\omega') : (p',\omega') \vDash f}{(p,\omega) \vDash \exists \bigcirc\ f}$$

$$\frac{(p,\omega) \vDash f[K/x]}{(p,\omega) \vDash \exists x.f} \qquad \frac{(p,\omega) \vDash f_1}{(p,\omega) \vDash f_1 \vee f_2} \qquad \frac{(p,\omega) \vDash f_2}{(p,\omega) \vDash f_1 \vee f_2}$$

$$\frac{\omega \in [\![e_1 \geq e_2]\!]}{(p,\omega) \vDash e_1 \geq e_2} \qquad \frac{}{(p,\omega) \vDash true}$$

As usual, we shall sometimes identify a formula f with the set of configurations where f holds, i.e. its denotation $[\![f]\!] \triangleq \lambda p.\{\omega \mid (p,\omega) \vDash f\}$

From these operators, we can further define the following abbreviations for practical purposes:

Definition 6 (Derived operators of LAS).

`blocked`	$\triangleq \neg\exists \bigcirc\ true$	`empty_mb`	$\triangleq \bigwedge_{\mathtt{m} \in Label} (\mathtt{m} = 0)$
$f_1 \wedge f_2$	$\triangleq \neg(\neg f_1 \vee \neg f_2)$	$f_1 \Rightarrow f_2$	$\triangleq \neg f_1 \vee f_2$
$\forall \Uparrow\ f$	$\triangleq \neg\exists \Uparrow \neg f$	$\forall x.f$	$\triangleq \neg\exists x.\neg f$
$\forall \Box\ f$	$\triangleq \neg\exists \Diamond\ \neg f$		

Moreover, we shall say that a formula f is "downward-closed", whenever formula $\exists \Uparrow f \Rightarrow f$ holds.

4 Decidability of Model-Checking in LAS

We state here very succinctly that for any formula $f \in$ LAS and any behavioural type p, $[\![f]\!](p)$ is Presburger definable, i.e. the set of label multisets (or mailbox contents) satisfying $[\![f]\!](p)$ is definable as a formula in Presburger arithmetics. It follows that model-checking LAS formulas with respect to behavioural types is decidable because model-checking in Presburger arithmetics is decidable too.

This section is organised as follows. First, we recall Presburger definability of flattened regular languages. Then, we handle the LAS operators.

4.1 Flattening Regular Languages

Definition 7 (Multiset interpretation of words). *For a given word w (i.e. a finite string of letters), we note $w^\flat$ its "flat" interpretation as the multiset of its letters, whatever their original order in w. Also, we define its pointwise extension to sets of words (i.e. languages).*

Now we state that for any regular language W, $W^\flat$ is a semilinear set, and as such is Presburger definable.

Theorem 1 ($W^\flat$ belongs in Presburger arithmetics). *For any regular language W, $W^\flat$ is a semilinear set, i.e.*

$$W^\flat \triangleq \bigcup_{i\in I}\{v_i + M_i.\kappa \mid \quad v_i \in \mathbb{N}^{Label} \wedge M_i \in \mathbb{N}^{Label\times d_i} \wedge \kappa \in \mathbb{N}^{d_i}\}$$

(where d_i, v_i and M_i are non-trivially derived from W) and is therefore Presburger definable.

Proof (reference). This result is known since the original works of Parikh, see [Esp95] for a complete insight.

4.2 Example

To get the reader used to flattening regular languages, we illustrate this operation on a small artificial example. Starting from:

$$W = \mathtt{a}.(\mathtt{b}.(\mathtt{c}+\mathtt{d}))^* + \mathtt{e}$$

we get:

$$W^\flat = (\mathtt{a}.(\mathtt{b}.(\mathtt{c}+\mathtt{d}))^* + \mathtt{e})^\flat = (\mathtt{a}.(\mathtt{b}.\mathtt{c})^*.(\mathtt{b}.\mathtt{d})^* + \mathtt{e})^\flat$$

from which, applying theorem 1, we finally get $I = \{1, 2\}$, $d_1 = 2$, $d_2 = 0$ and:

$$\begin{array}{l|l} v_1 = (1,0,0,0,0) & v_2 = (0,0,0,0,1) \\ M_1 = \begin{pmatrix} 0,1,1,0,0 \\ 0,1,0,1,0 \end{pmatrix} & M_2 = () \end{array}$$

or equivalently, using Presburger arithmetics:

$$(\mathtt{a} = 1 \wedge \mathtt{b} = \mathtt{c} + \mathtt{d} \geq 0 \wedge \mathtt{e} = 0) \ \vee \ (\mathtt{a} = \mathtt{b} = \mathtt{c} = \mathtt{d} = 0 \wedge \mathtt{e} = 1)$$

4.3 Computing Denotation for Full LAS

Theorem 2 (LAS belongs in Presburger arithmetics). *Given a LAS formula f and a behavioural type p, $[\![f]\!](p)$ is Presburger definable.*

Proof (reference). A proof for similar concerns (namely CTL logic for Basic Parallel Processes) can be found in [Esp97]. In particular, the proof for the $\exists\Diamond$ operator uses theorem 1. Adapting the proof to our systems and logic is straightforward.

5 Expressivity of LAS: About Orphan Messages

Here, with the help of our expressive logic LAS, we explore different properties whose common goal is to avoid communication errors.

A pervasive feature will be the downward-closedness of these properties, which ensures that we can use them in type systems. Indeed, the same way

that a behavioural type is merely an abstraction of real input behaviours, the set of messages that will effectively be received by any actor cannot be computed very precisely in the general case. In order to devise a type system, we need to compute an upper approximation of incoming messages, which we somehow match against input capabilities (see [CPS97,CPDS99]). So, in the end, as an actor will receive no more messages than computed, and usually strictly less, a simple solution is to use a downward-closed input language, so that we avoid many spurious typing errors.

The different properties exposed in this section will be illustrated by the one-slot buffer example and a more complex one $\mu q_0.\mathtt{a}.(\mathtt{d} + \mathtt{b}.\mu q_2.(\mathtt{a}.q_2 + \mathtt{c}).q_0)$ pictured as a finite state transition system in the following figure 1.

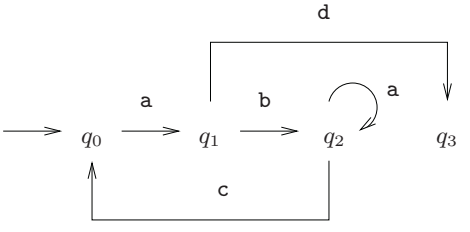

Fig. 1. A small example.

This example could represent a classic application with a main control loop with a little inner loop (with message a) and an exit branch (with message d).

We give results, i.e. $[\![f]\!]$ where f is the property under scrutiny, only for the initial state of our example systems, and as flattened regular expressions (except for the first example).

5.1 Strongly Safe Input Language

A very natural idea that comes first to mind is to rule out any configuration that may lead to a blocked situation, where mailbox is not empty and yet no further reduction is ever possible. A simple formula suffices to express this strong requirement:

$$\mathit{Strong} \triangleq \forall\Box\ (\mathtt{blocked} \Rightarrow \mathtt{empty\_mb}) \equiv \forall\Diamond\mathtt{empty\_mb}$$

This property entails a very fine-grained analysis of input capabilities, and therefore is very valuable for an end-user. We may slightly decrease its acuteness, by taking its downward closure $\exists\Uparrow \mathit{Strong}$.

Examples. For the one-slot buffer, we obtain for $[\![\mathit{Strong}]\!](\alpha)$:

$$\mathtt{get} \geq 0 \wedge \mathtt{get} \geq \mathtt{put} - 1 \wedge \mathtt{put} \geq \mathtt{get}$$

which can be represented by the flattened regular expression:

$$((\mathtt{put}.\mathtt{get})^* + \mathtt{put}.(\mathtt{put}.\mathtt{get})^*)^\flat$$

For example of figure 1, we obtain for $[\![Strong]\!](q_0)$:

$$(\Lambda + \mathtt{a} + \mathtt{a.b} + \mathtt{a}^2\mathtt{.b.a}^* + \mathtt{a.b.c} + \mathtt{a}^2\mathtt{.b.c} + \mathtt{a.d})^\flat$$

First, whether the system will take the exit branch or stay within the main loop depends on the presence of a message `d`.

Second, assuming we are stuck in the main loop, the inner loop adds a major constraint because we can decide to handle message `a` either in state q_0 or in state q_2. So we can send at most two messages `a` with `b` and `c`, because otherwise we could get stuck in state q_0 with `b` and/or `c` orphan messages.

Without the inner loop, the result becomes:

$$(\mathtt{a} + \mathtt{a.d} + \mathtt{a.}(\mathtt{a.b.c})^+ + (\mathtt{a.b.c})^* + \mathtt{a.b.}(\mathtt{a.b.c})^*)^\flat$$

expressing that the actor may loop the main loop any number of times (except if message `d` is sent).

Discussion. As a conclusion, this property is indeed too strong as for instance it doesn't allow interferences between nested loops, as illustrated by our example 1. This stems from the fact that our formula requires that every message should be handled in every computation path from the initial state of an actor, whereas in real life, input/output communication patterns may ensure a strict sequence of messages, that will be received one at a time. So, as a conclusion, the *Strong* property is not really suited to build a loose enough type system, one that would allow real life applications to be well-typed.

Moreover, a type system would enjoy better properties if well-typedness of a given system would imply well-typedness of each of its sub-process taken in isolation. For instance, the correctness of a process should not rely upon its environment sending a message that unblocks the treatment of otherwise orphan messages already present in an actor's mailbox.

The following alternative definitions of input languages we shall retain for designing type systems does reflect this simulation of an angelic environment that provides unblocking messages when needed. They are also downward-closed sets of messages, i.e. every subset of a solution is a solution too.

5.2 First Orphan-Free Input Language

This input language is only a rephrasing in our logic of earlier works presented for instance in [CPDS99]. As this language suffers from some important weaknesses, it is mentioned here mainly for historical reasons, and in witness of our claim for the expressivity of our logic.

$$First(\mathtt{m}) \triangleq \forall m_0.m_0 = \mathtt{m} \Rightarrow \forall \Uparrow \forall d.d = \mathtt{m} - m_0 \Rightarrow \forall \Box \exists \Uparrow \exists \Diamond\ \mathtt{m} \leq d$$

Sketchily, this property ensures that an actor will have at any time the ability to treat any message `m` in its mailbox, provided that a kind environment may send unblocking messages during the execution. For finite branches, this property boils down to computing multiset intersection of such paths. The full input language is then the intersection of $First(\mathtt{m})$ for every label `m`.

Examples. The result for the one-slot buffer is not a surprise, because of its linear behaviour: $(\texttt{get}^*.\texttt{put}^*)^\flat$. For example of figure 1, we find: $(\Lambda + \texttt{a} + \texttt{d} + \texttt{a}.\texttt{d})^\flat$ that doesn't allow to loop the main loop even if no message `d` is ever sent.

Discussion. Our present formulation doesn't do justice to the simplicity of the original framework, whose computation only involved a simple greatest fixpoint, instead of a rather complicated formula. Yet, our formula may be computed with respect to every label independently, which makes it worthwhile for complexity and efficiency issues. This property has already been used in a type system for the full CAP actor's language, despite the fact that finite branches dominate the computation, no matter how many loops may exist. In practice, this drawback totally prevents programmers from using exit branches, so that only actors with a finite or a forever cycling lifetime may get well typed. Moreover, finite branches with no common message (for instance, a choice between two different messages) are systematically ill-typed, because the intersection of all finite paths is empty in this case.

5.3 Second Orphan-Free Input Language

This last input language is currently the best answer as it fulfills our requirements. It involves a slight extension of our logic, because we need to compute a greatest fixpoint. This fixpoint obviously needs to be well defined and computed within a finite amount of time.

Definition 8 (Greatest fixpoint in LAS). *For any LAS property scheme $F(X)$ where X is a property variable, then we define $\nu X.F(X)$, provided the following constraints hold:*

1. *X only occurs nested inside an even number of "$\neg$" operators. $F(X)$ is then monotonous, and least as well as greatest fixpoints are then well defined.*
2. *$F(X)$ is downward-closed. Computation of greatest fixpoint (starting from $X_0 \triangleq true$) then terminates, because there is no infinite strictly decreasing chains in $\mathbb{N}^{Label}$.*

Theorem 3 ($\nu X.F(X)$ is Presburger definable). *Any term $\nu X.F(X)$ satisfying the previous conditions is Presburger definable.*

Proof (sketch). During fixpoint computation, the first iterate (*true*) is obviously Presburger definable, and each new iterate is itself Presburger definable from the previous one. Termination is obtained by deciding implication between two successive iterates.

Now, we define our second language. We base our definition on three simple facts. First, an empty mailbox should always be accepted. Second, adding some unblocking messages to an accepted configuration should lead to another accepted configuration, whatever the execution path. Third, added messages should not be orphan, i.e. one execution path that can handle all added messages should exist:

$$Second \triangleq \nu X.\exists\Uparrow (\exists\Diamond\ \texttt{empty\_mb} \wedge \forall\Box\ X)$$

Examples. The one-slot buffer has the same result: $(\texttt{get}^*.\texttt{put}^*)^\flat$. For example of figure 1, we get: $(\texttt{d} + \texttt{a}.\texttt{d} + \texttt{a}^*.\texttt{b}^*.\texttt{c}^*)^\flat$ so, either the exit branch is followed or the actor accepts any number of messages a,b and c. Without the inner loop, the result is still the same as we doesn't take deadlocks into account, indeed in this case our angelic environment can always safely solve such deadlocking configurations, i.e. without sending some other orphan message.

Discussion. This property has a very simple expression, and this time we can cope with finite branches and loops without losing precision, or computing a severe under-approximation. This is the right choice, as shown in the next section, under the assumption that fixpoint computations will never be too much resource demanding to be a part of an intensively used type system.

6 Orphan-Free Input Language and Full Type Discipline

In order to embed our input languages in a full type discipline, we should state some continuity and subject-reduction properties of the type system. Unfortunately, we lack room to put forward our ideas about typing actors and invite the interested reader to consult [CPS97,CPDS99].

Nevertheless, we can state invariance properties which should help the reader to convince himself about the possibility of using our *Second* input language to state subject-reduction properties dedicated to ruling out communication errors in real executions of actor configurations.

The following theorem handles blocked as well as unblocked configurations, expressed at the level of behavioural types. The first case represents one execution step, whereas the second one deals with blocked configurations. As shown in our earlier works, systems that may get stuck in a blocked configuration may still get well-typed, under our assumption of an angelic environment.

Theorem 4 (*Second* is invariant). *For any configuration* (p, ω)*, message label* $\texttt{m}$ *and* p'*, we have:*

$$\frac{p \xrightarrow{\texttt{m}} p' \ (p, \omega \cup \{\texttt{m}\}) \vDash \mathit{Second} \wedge \neg\texttt{blocked}}{(p', \omega) \vDash \mathit{Second}} \qquad \frac{(p, \omega) \vDash \mathit{Second} \wedge \texttt{blocked}}{(p, \omega) \vDash \exists\Uparrow \ (\mathit{Second} \wedge \neg\texttt{blocked})}$$

Proof (omitted).

Our last theorem states the absence of a kind of communication errors called *static orphan* messages, namely messages which, once in an actor's mailbox, would never get a chance to be treated, whatever the (angelic) environment put in parallel.

Theorem 5 (*Second* implies orphan-free). *For any configuration* (p, ω)*, we have:*

$$\frac{(p, \omega) \vDash \mathit{Second}}{(p, \omega) \vDash \exists\Uparrow \ \exists\Diamond \ \texttt{empty\_mb}}$$

Proof (omitted).

Conclusion

As a conclusion, we have achieved to devise an interesting general purpose logic to express behavioural properties, for some restricted kind of infinite-state systems. Our logic is expressive enough to rephrase many past and current studies about behaviours of actor programs. For the time being, we only deal with input capabilities, but it seems that it can be extended to handle transducers, i.e. input/output devices. Moreover, modeling message contents, as long as we only allow them to denote behaviours as well as addresses, is planned for future work. The main issue in this case will be to deal with the possibility for behaviours to send messages to global addresses, leading to rather intricate scope extrusion problems. As far as we know, our logic is quite unique and we haven't found yet any tool dealing with a similar logic of the same behavioural kind. Therefore we decided to implement a prototype model-checker for LAS, on top of a decision procedure for Presburger arithmetics [Pug91]. In a rather user-friendly fashion, our prototype logs every calculation it makes, as a sequence of literally written Presburger equations, so that the user can trace the computational meaning of its specification, if needed. The generated log file can serve as a basis for step-by-step debugging purposes for instance.

From a theoretical viewpoint, our work on LAS can be extended to handle full TyCo types, including the "||" operator, as it is partially shown in [Esp97]. We can therefore apply our present work (including our full type discipline not presented here) to other concurrent programming languages, such as Pict [PT97].

References

[AC94] M. Abadi and L. Cardelli. A theory of primitive objects: untyped and first-order systems. In *Proc. Theor. Aspects of Computer Software*, 1994.

[Agh86] Gul Agha. *Actors: A model of concurrent computation in distributed systems.* MIT Press, Cambridge, Mass., 1986.

[Bou97] Gérard Boudol. Typing the use of resources in a concurrent calculus. In R. K. Shyamasundar and K. Ueda, editors, *Proceedings of ASIAN '97*, volume 1345 of *LNCS*, pages 239–253. Springer-Verlag, 1997.

[CES86] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.

[CPDS99] Jean-Louis Colaço, Marc Pantel, Fabien Dagnat, and Patrick Sallé. Static safety analysis for non-uniform service availability in actors. In *Proc. of FMOODS*, February 1999.

[CPS96] Jean-Louis Colaço, Marc Pantel, and Patrick Sallé. Cap: An actor dedicated process calculus. In *Prof. of Proof Theory of Concurrent Object-Oriented Programming*, May 1996.

[CPS97] Jean-Louis Colaço, Marc Pantel, and Patrick Sallé. A set-constraint-based analysis of actors. In *Proc. of FMOODS*, July 1997.

[CRR02] Sagar Chaki, Sriram K. Rajamani, and Jakob Rehof. Types as models: model checking message-passing programs. In *Symposium on Principles of Programming Languages*, pages 45–57, 2002.

[DPCS00] Fabien Dagnat, Marc Pantel, Matthias Colin, and Patrick Sallé. Typing concurrent objects and actors. *L'Objet – Méthodes formelles pour les objets*, Volume 6(1/2000):pages 83–106, May 2000.

[Esp95] Javier Esparza. Petri nets, commutative context-free grammars, and basic parallel processes, 1995.

[Esp97] Javier Esparza. Decidability of model checking for infinite-state concurrent systems. *Acta Informatica*, 34(2):85–107, 1997.

[FLMR97] Cédric Fournet, Cosimo Laneve, Luc Maranget, and Didier Rémy. Implicit typing à la ml for the join-calculus. In *Proc. of CONCUR*, LNCS 1283, pages 196–212. Springer-Verlag, 1997.

[HBS73] Carl Hewitt, P. Bishop, and R. Steiger. A universal modular actor formalism for artificial intelligence. In *Proc. of Int. Joint Conference on Artificial Intelligence*, 1973.

[HT91] Kohei Honda and Mario Tokoro. An object calculus for asynchronous communication. In P. America, editor, *Proc. ECOOP '91*, July 1991.

[Kob97] Naoki Kobayashi. A partially deadloack-free typed process calculus. In *Proc. of the conf. Logic In Computer Science*, 1997.

[Mil91] Robin Milner. The polyadic π-calculus: a tutorial. Technical Report ECS-LFCS-91-180, LFCS, 1991.

[Nie95] Oscar Nierstrasz. Regular types for active objects. In *In Object-Oriented Software Composition, ACM SIGPLAN Notices*, pages 99–121. Prentice Hall, October 1995.

[NNS99] E. Najm, A. Nimour, and J-B. Stefani. Infinite types for distributed object interfaces. In *Proc. of FMOODS*, February 1999.

[PT97] B. Pierce and D. Turner. Pict: A programming languages based on the π-calculus. Technical Report 476, Indiana Univ., March 1997.

[Pug91] William Pugh. The omega test: a fast and practical integer programming algorithm for dependence analysis. In *Supercomputing*, pages 4–13, 1991.

[Pun96] Franz Puntigam. Type for active objects based on trace semantics. In *Proc. of FMOODS*, pages 5–20, March 1996.

[RR01] Sriram K. Rajamani and Jakob Rehof. A behavioral module system for the pi-calculus. *Lecture Notes in Computer Science*, 2126:375–??, 2001.

[RV00] António Ravara and Vasco T. Vasconcelos. Typing non-uniform concurrent objects. In *CONCUR'00*, volume 1877 of *LNCS*, pages 474–488. Springer-Verlag, 2000.

[Wal95] D. Walker. Objects in the π-calculus. *Information and Computation*, (116), 1995.