DECIDABILITY PROBLEMS FOR ACTOR SYSTEMS∗

FRANK S. DE BOER a, MOHAMMAD MAHDI JAGHOORI b, COSIMO LANEVE c, AND GIANLUIGI ZAVATTARO d

a CWI/LIACS, The Netherlands
e-mail address: f.s.de.boer@cwi.nl

b Leiden University, The Netherlands
e-mail address: jaghoori@cwi.nl

c,d Department of Computer Science and Engineering, University of Bologna, INRIA Focus, Italy
e-mail address: {cosimo.laneve,gianluigi.zavattaro}@unibo.it

ABSTRACT. We introduce a nominal actor-based language and study its expressive power. We have identified the presence/absence of fields as a crucial feature: the dynamic creation of names in combination with fields gives rise to Turing completeness. On the other hand, restricting to stateless actors gives rise to systems for which properties such as termination are decidable. This decidability result still holds for actors with states when the number of actors is bounded and the state is read-only.

1. INTRODUCTION

Since their introduction in [20], actor languages have evolved as a powerful computational model for defining distributed and concurrent systems [2, 3]. Languages based on actors have been also designed for modelling embedded systems [23, 24], wireless sensor networks [11, 31], multi-core programming [22], and web services [9, 10]. The underlying concurrent model of actor languages also forms the basis of the programming languages Erlang [3] and Scala [19] that have recently gained in popularity, in part due to their support for scalable concurrency.


Key words and phrases: Actors, RESTful services, decidability problems, 2-Counter Machines, well-structured transition systems, embedding relation.

∗ This paper is a full version of an extended abstract that appears in [13]. With respect to the conference paper, this one contains examples and discussions, see Section 2.1, and complete proofs of statements.

a−d This work has been supported by the HATS Project No. FP7-231620 (Highly Adaptable and Trustworthy Software using Formal Models) and by the ENVISAGE Project No. FP7-610582 (Engineering Virtualized Services) of the EC.
In actor languages [2, 20, 32], actors use a queue for storing the invocations to their methods in a FIFO manner. The queued invocations are processed sequentially by executing the corresponding method bodies. The encapsulated memory of an actor is represented by a finite number of fields that can be read and set by its methods and as such exist throughout its life time.

In this paper we introduce a nominal actor-based language and study its expressive power. This language, besides dynamic creation of actors, also supports the dynamic creation of variable names that can be stored in fields and communicated in method calls. As such our nominal actor-based language gives rise to unboundedness in (1) internal queues of the actors, (2) dynamic actor creation/activation and (3) dynamic creation of variable names.

Statelessness has recently been adopted as a basic principle of service oriented computing, in particular by RESTful services. Such services are designed to be stateless, and contextual information should be added to messages, so a service can customize replies simply by looking at the received request messages. In service oriented computing read-only fields (which are initialized upon activation) are used to provide configuration/deployment information that distinguishes the distinct instances of the same service. We have identified the presence/absence of fields as a crucial feature of our language: (1) and (3) in combination with fields gives rise to a Turing complete calculus. On the other hand, restricting to stateless actors gives rise to systems for which properties such as termination and process reachability are decidable. In order to preserve this decidability result to actors with states we have to restrict the number of actors to be finite and the state to be read-only.

More specifically, we model a system consisting of finitely many actors with read-only fields as a well-structured transition system [16] – henceforth termination and process reachability are decidable. Further, we show that an abstraction of systems of unboundedly many stateless actors (i.e., actors without fields) which preserves termination and process reachability is also an instance of well-structured transition system. It turns out that, in the context of unbounded actor creation, this restriction to stateless actors is necessary by a reduction to the halting problem for 2 Counter Machines.

To the best of our knowledge, the technique we use to establish the decidability results for the above languages is original since (i) these systems respectively admit the creation of unboundedly many variables and actor names; (ii) actors in general are sensitive to the identity of names because of the presence of a name-match operator. In particular, in the case of finitely many actors with read-only fields, we define an equivalence on process instances in terms of renamings of the variables that generate the same partition. This equivalence allows us to compute an upper bound to the instances of method bodies, which is the basic argument for the model being a well-structured transition system. In case of systems with unboundedly many stateless actors, the reasonable extensions of this equivalence on process instances have been unsuccessful because of the required abstraction of the identity of actor names. Therefore we decided to apply our arguments to an abstract operational model where messages may be enqueued in every actor of the same class. The above equivalence can be successfully used in this model, thus yielding again the upper bounds for the number of method body instances. Further, the abstract model still provides enough information to derive decidable properties of the language.

Related Works. There exists a vast body of related work on decidability of infinite-state systems (see [1]). However, to the best of our knowledge, the specific characteristics of the
In actor-based languages, the pure asynchronous mechanism of queued and dequeued method calls has not been addressed. It is interesting to observe that the most expressive known fragment of the pi-calculus for which interesting verification problems are still decidable is the depth-bounded fragment [26]. In [33] the theory of well-structured transition systems is applied to prove the decidability of coverability problems for bounded depth pi-calculus. Our nominal actor language also features the creation and communication of new names. In our decidable fragments however, differently from the depth-bounded pi-calculus fragment, we do not restrict the creation and communication of names. For instance, in the queue of an actor we might have unboundedly many messages (representing process continuations) where each message shares one name with the previous message in the queue. Decidability of asynchronous communication via a shared data space, read operation and non-blocking test operations on the shared space of the coordination language Linda has been studied in [8]. In [5] the authors introduce an algorithm for reachability analysis of multithreaded object-oriented programs. This algorithm requires both a bound on the number of context switches (between the threads) and the visible heap. Further their approach is specific to object-oriented programs and is not applicable to actor-based languages, which are based on a different concurrency model. As another final example, a system composed of finitely many actors can be easily modeled as a communicating finite state machine [6], a formalism that is known to be Turing complete [6, 25]. However, this modeling does not scale to infinite actors and does not display the execution model of actors. Recent work on actor-based language focusses on deadlock analysis: In [18], a technique for the deadlock analysis has been introduced for a version of Featherweight Java which features asynchronous method invocations and a synchronization mechanism based on futures variables. The approach followed in [15] for detecting deadlock in an actor-like subset of Creol [7] is based on suitable over-approximations.

2. The language Actor

Four disjoint infinite sets of names are used: actor classes, ranged over by \( C, D, \ldots \), method names, ranged over by \( m, m', n, n', \ldots \), field names, ranged over by \( f, g, \ldots \), and variables, ranged over by \( x, y, z, \ldots \). For notational convenience, we use \( \bar{x} \) when we refer to a list of variables \( x_1, \ldots, x_n \) (and similarly for other kinds of terms).

The syntax of the language Actor uses expressions \( E \) and processes \( P \) defined by the rules

\[
E ::= f \mid x \mid \text{new } C(\bar{E}) \\
P ::= 0 \mid (f \leftarrow E).P \mid \text{let } x = E \text{ in } P \mid x!m(\bar{E}).P \mid [E = E']P; P \mid P + P
\]

An expression \( E \) either denotes a value stored in a field \( f \), or a variable \( x \), or a new actor of class \( C \) with fields initialized to the values of \( \bar{E} \). A process may be either the terminated one, denoted by \( 0 \), or a field update \( (f \leftarrow E).P \), or the assignment \( \text{let } x = E \text{ in } P \) of a value to a variable, or an invocation \( x!m(\bar{E}).P \) of a method \( m \) of the actor \( x \) with arguments \( \bar{E} \), or a check \( [E = E']P; P' \) of the identity of expressions with positive and negative continuations, or, finally a nondeterministic process \( P + P' \). We never write the tailing \( 0 \) in processes; for example \( (f \leftarrow x).0 \) will be always shortened into \( (f \leftarrow x) \). We will also shorten \([E = E']P; 0\) into \([E = E']P\). Following the tradition of process calculi like
The method \textit{init} manifests a basic feature of our actor language: the creation of new variables. In particular, \texttt{tt} and \texttt{ff} are free variables in the method definition of \textit{init}. When \textit{init} will be invoked, they will be replaced by two different fresh variables (said in pi-calculus jargon \cite{28}, we are assuming an implicit \((\nu \texttt{tt}, \texttt{ff})\)- at the beginning of the method body).
These variables are stored in the fields t and f, respectively and they will be used to update the fields fst and snd appropriately. The merger forwards two messages to the server, which is stored in the field srv. These messages have been received through invocations of first and second. No forward occurs if one of the two methods has not been invoked. In particular, first (the method second behaves in a similar way) performs the forward if second has been evaluated and its message has not been already forwarded (field snd equal to #); in this case the parameter of the invocation of second has been stored into g. Otherwise, if there is no previous invocation to first, still to be forwarded, (field fst equal to ff) then fst is set to tt and the parameter stored into g. There is a possibility that first is evaluated and a previous evaluation of first has still to be forwarded (field fst equal to #). In this case, the invocation is bounced back (it is enqueued in the actor queue – see the operational semantics).

Example 2.2. As a second example we model the OpenID authentication protocol [30]. Three actors are considered: a client, a server, and an authProvider. The client sends a login request to the server. The server generates two secure tokens clientToken and authToken, used for secure communication with the client and the authProvider, respectively, and then sends the two tokens to the corresponding actors. Subsequently, the client sends an authentication message to the authProvider that (after checking the correctness of the username and password) communicates to the server whether the authentication succeeds or fails. In the following, we abstract from the management of username and password and we simply assume that they are stored in the fields u and p of the client when it sends the authenticate message to the authServer.

The class C of the client includes the following two definitions:

\[
\begin{align*}
C.\text{init}(server) &= \text{server}!\text{login}(this) \\
C.\text{token}(authServer, clientToken) &= \text{authServer}!\text{authenticate}(u, p, clientToken)
\end{align*}
\]

The class S of the server includes the following four definitions:

\[
\begin{align*}
S.\text{init}(authServer) &= (\text{auth} \leftarrow \text{authServer}) \\
S.\text{login}(client) &= \text{auth}!\text{open}(this, clientToken, authToken). \\
&\quad \text{client}!\text{token}(\text{auth}, \text{authToken}) \\
S.\text{succeeds}(authToken) &= \ldots \\
S.\text{fails}(authToken) &= \ldots
\end{align*}
\]

The class A of the authServer includes the following two definitions (we leave unspecified the check of the correctness of username and password):

\[
\begin{align*}
A.\text{open}(server, cToken, aToken) &= (server \leftarrow server).(ctoken \leftarrow cToken). \\
&\quad (atoken \leftarrow aToken) \\
A.\text{authenticate}(u, p, cToken) &= [ctoken = cToken]server!\text{succeeds}(atoken); \\
&\quad server!\text{fails}(atoken)
\end{align*}
\]

The main program that instantiates the client, the server and the authProvider is as follows:

\[
\text{let client = new C in} \\
\text{let server = new S in} \\
\text{let authProvider = new A in} \\
\text{client!init(server), server!init(authProvider)}
\]
Example 2.3. We illustrate the modeling of a register that stores natural numbers and supports the operations of increment and decrement, when its value is positive. In the absence of data-types in our nominal actor language, we model a register by an actor and implement the value of the register by the number of invocations of the method item stored in its queue. Below we describe the corresponding class R, assuming a class Ctrl which encodes the control of the register machine. When an operation is performed, the register replies with an invocation run(pc, tt, ff), where pc is a suitable continuation. The continuation is unique when the increment is invoked; the decrement has two possible continuations: a positive one, in case of success, and a negative one (the register was 0, no decrement is performed).

```plaintext
// R has a field called dec, Ctrl is an actor in the context
R.inc(pc, tt, ff) = R!item(tt, ff), Ctrl!run(pc, tt, ff)
R.dec(pc, pc', tt, ff) = (dec ← tt). R!checkzero(pc, pc', tt, ff)
R.checkzero(pc, pc', tt, ff) = [dec = tt]. Ctrl!run(pc, tt, ff). (dec ← ff); Ctrl!run(pc, tt, ff)
R.init(tt, ff) = (dec ← ff)
R.item(tt, ff) = [dec = ff]. R!item(tt, ff); (dec ← ff)
```

Method inc simply gives rise to the storage of a new message item and a trigger of the continuation. Executing a message item, amounts to bouncing the invocation back if the value of the field dec is false. In this case the invocation is enqueued again. Otherwise the invocations is “consumed” because there was a pending decrement to perform (see below) and dec is set to true (indicating that there is no pending decrement anymore). Method dec is the tricky one: it sets dec to true and add an invocation of checkzero (we are assuming that the register queue always contains at most one invocation of dec). That is, we are postponing the triggering of the continuation to the evaluation of checkzero that will occur after the evaluation of any other method in the queue of the register. When checkzero will be evaluated either (i) the dec field is true, this means that the register did not contained any item message – its value was 0 – and the continuation pc' is triggered; or (ii) the dec field is false, that is the register has been decremented and the continuation pc is triggered.

A refinement of the above register is used in Theorem 3.2 to demonstrate the Turing completeness of a sublanguage of Actor.

Example 2.4. In Sections 4 and 5 we will respectively prove decidability results for two fragments of our actor language: in the first fragment only the main program can create new actors (so boundedly many actors can be created) and fields are read-only, while in the second fragment actors have an empty state. To illustrate these two fragments we show how they can be used to model a system in which n workers are used to execute n distinct tasks indicated by a client.

In the fragment with bounded actors and read-only fields the model can be described by considering a class C for the client with an init method responsible for invoking the task manager, passing him the description of the n tasks to be executed:

```plaintext
// C has no fields
C.init(TManager) = TManager!tasksExec(t_1, \cdots, t_n)
```

The task manager is an instance of a class providing a method called tasksExec able to pass to the workers the description of their relative tasks:
Finally, a worker is an instance of a class \( W \) with one method \( \text{exec} \) able to execute the passed task (we leave this method unspecified):

\[
// \text{W has no fields and one method exec(p)}
\]

The main program first instantiates the workers, then the task manager by storing in its state the reference to the workers, and finally the client; after the \( \text{init} \) method of the client is invoked passing him a reference to the task manager:

\[
\text{let } w_1 = \text{new } W \text{ in } \cdots \text{ let } w_n = \text{new } W \text{ in} \\
\text{let } \text{taskManager} = \text{new} \text{ TMs}(w_1, \cdots, w_n) \text{ in} \\
\text{let } \text{client} = \text{new } C \text{ in } \text{client!init(taskManager)}
\]

In the fragment with stateless actors we present an alternative specification of the system in which the classes for the client and the workers are as above, while the task manager is defined as follows:

\[
// \text{TM has no fields}
\]

\[
\text{TM.tasksExec}(p_1, \cdots, p_n) = \text{this!nextTask}(p_1, \cdots, p_n, x_1, x_2, \cdots, x_n, x_1)
\]

\[
\text{TM.nextTask}(p_1, \cdots, p_n, v_1, \cdots, v_{n+1}) = [v_1 = v_2]0; \text{let } w = \text{new } W \text{ in } w!\text{exec}(p_1).
\]

\[
\text{this!nextTask}(p_2, \cdots, p_n, p_1, v_2, \cdots, v_{n+1}, v_1)
\]

Upon reception of the \( n \) tasks to be executed, the task manager passes the task descriptions to the method \( \text{nextTask} \). This method instantiates one new worker, passes to it the corresponding task, and then recursively invokes itself to continue with worker instantiations. In order to execute exactly \( n \) recursive calls, \( n+1 \) additional parameters are considered: in the first call all such parameters are distinct excluding the first and the last one, at every call the parameters are circularly shifted, and when the first two parameters coincide the chain of calls is terminated.

2.2. The operational semantics. The operational semantics of the language \textit{Actor} will use an infinite set of \textit{actor names}, ranged over \( A, B, \cdots \). This set is partitioned by the actor classes in such a way that every partition retains infinitely many actor names. We write \( A \in \mathcal{C} \) to say that \( A \) belongs to the partition of \( \mathcal{C} \). In the following, the (\textit{run-time}) expressions will also include actor names and, with an abuse of notation, this extended set of expressions will be ranged over by \( E \). The set of terms that are variables or actor names, called \textit{values}, will be addressed by \( U, V, \cdots \). Similarly for processes that, at run-time, may also have actor names. The extended set will be addressed by \( P \). We notice that \( \text{free}(P) \), when \( P \) belongs to this extended set, may returns variables and actor names. We will also apply \( \text{free}(\cdot) \) to tuples of (extended) expressions: \( \text{free}(\vec{E}) \) returns the set of variables and actor names in \( \vec{E} \).

The semantics is defined in terms of a transition relation \( S \rightarrow S' \), where \( S, S' \), called \textit{configurations}, are sets of terms \( A 
\rightarrow (P, \varphi, q) \) with \( A \) being an actor name, \( \varphi \), the state of \( A \), being a map from \textit{fields}(\( \mathcal{C} \)) to values, where \( A \in \mathcal{C} \), and \( q \) being a queue of terms \( m(U) \). The empty queue will be denoted with \( \varepsilon \). Configurations contain at most one \( A 
\rightarrow (P, \varphi, q) \) for each actor name \( A \). As usual, let \( \rightarrow^* \) be the transitive closure of \( \rightarrow \) and \( \rightarrow^+ \) be \( \rightarrow \rightarrow^+ \).
The operational semantics of `Actor` is defined in Table 2 where the evaluation function \( E \xrightarrow{\varphi} U ; S \) is used (defined in Table 1). This function takes an expression \( E \) and a store \( \varphi \) and returns a value \( U \) and a possibly empty configuration \( S \) of terms \( A \triangleright (0, [\tilde{f} \mapsto \tilde{U}], \varepsilon) \). These terms represent actors created during the evaluation – the names \( A \) are fresh – and \( \varphi \) records the initial values of the fields of \( A \). The auxiliary function \( fresh(\cdot) \) used in the evaluation function takes a class actor and returns an actor name of that class that is fresh. In order to have a finitely branching transition system (see the remark at the end of this section), we assume actor names in classes are totally ordered and \( fresh(C) \) always returns the first unused name of the class \( C \). In this way, fresh names are selected in a deterministic manner.

The initial configuration of a program with main process \( P \) is \( \aleph \triangleright (P, \emptyset, \varepsilon) \), where \( \aleph \) is a name of the root, an actor of a class without fields and methods. We assume that the class of \( \aleph \) does not belong to the classes of the program. Note that the root actor is guaranteed to terminate because its queue remains empty (no method invocation may be enqueued) and the main process (as any other one) terminates.

We finally remark that transition systems of the language `Actor` are finitely branching (every state has a finite number of successor states) because the choices of fresh actor names (in the evaluation of \( new C \)) and of fresh variables (in the instantiation of the bodies of methods) are deterministic.

2.3. Relevant sublanguages. We will consider the following fragments of `Actor` whose relevance has been already discussed in the Introduction:

- **Actor\( sub \)** is the sublanguage where the `new` expression only occurs in the main process (the number of actor names that it is possible to create is bounded).
- **Actor\( ro \)** is the sublanguage without the field update operation \( f \leftarrow E \) (fields are read-only as they cannot be modified after the initialization).
- **Actor\( ba \)** is the intersection of `Actor\( sub \)` and `Actor\( ro \)`.
- **Actor\( sl \)** is the sublanguage with classes without fields (objects are stateless).

Table 1: The evaluation relation \( E \xrightarrow{\varphi} U ; S \)
\begin{itemize}
  \item \textbf{(UPD)} \quad \frac{E \xrightarrow{\varphi} U \; ; \; S}{A \triangleright ((f \leftarrow E).P,\varphi,q) \rightarrow A \triangleright (P,\varphi[f \leftarrow U],q),S}
  \item \textbf{(LET)} \quad \frac{E \xrightarrow{\varphi} U \; ; \; S}{A \triangleright (\text{let } x = E \text{ in } P,\varphi,q) \rightarrow A \triangleright (P[U/x],\varphi,q),S}
  \item \textbf{(INVK-s)} \quad \frac{}{A \triangleright (A!m(\bar{E}).P,\varphi,q) \rightarrow A \triangleright (P,\varphi,q \cdot m(\bar{U})),S}
  \item \textbf{(INVK)} \quad \frac{}{A \triangleright (A!m(\bar{E}).P,\varphi,q),A' \triangleright (P',\varphi',q') \rightarrow A \triangleright (P,\varphi,q),A' \triangleright (P',\varphi',q' \cdot m(\bar{U})),S}
  \item \textbf{(INST)} \quad \frac{A \in C \quad c \cdot m(\bar{x}) = P \quad \bar{y} = \text{free}(P) \setminus \bar{x} \quad \bar{y}' = \text{fresh}(\bar{y})}{A \triangleright (0,\varphi,m(\bar{U}) \cdot q) \rightarrow A \triangleright (P[A/\text{this}][\bar{y}'/\bar{y}][\bar{U}/\bar{x}],\varphi,q)}
  \item \textbf{(MATCH)} \quad \frac{}{E,E' \xrightarrow{\varphi} U,U \; ; \; S}
  \item \textbf{(MMATCH)} \quad \frac{E,E' \xrightarrow{\varphi} U,V \; ; \; S \quad U \neq V}{A \triangleright ([E = E']P;Q,\varphi,q) \rightarrow A \triangleright (Q,\varphi,q),S}
  \item \textbf{(PLUS-L)} \quad \frac{}{A \triangleright (P + Q,\varphi,q) \rightarrow A \triangleright (P,\varphi,q)}
  \item \textbf{(PLUS-R)} \quad \frac{}{A \triangleright (P + Q,\varphi,q) \rightarrow A \triangleright (Q,\varphi,q)}
  \item \textbf{(CONTEXT)} \quad \frac{}{S \rightarrow S'}
\end{itemize}

\begin{table}[h]
\centering
\begin{tabular}{ll}
\hline
$\mathcal{S}$ & $\rightarrow$ & $\mathcal{S}'$ \\
\hline
$\mathcal{S},\mathcal{S}'''$ & $\rightarrow$ & $\mathcal{S}',\mathcal{S}''$ \\
\hline
\end{tabular}
\caption{The transition relation $\mathcal{S} \rightarrow \mathcal{S}'$}
\end{table}
3. Undecidability Results for Actor\textsubscript{ba} and Actor\textsuperscript{∞}

In this section we establish the main undecidability results for the actor language in Section\textsuperscript{2}. In particular, we will prove the undecidability of termination and process reachability.

\textbf{Definition 3.1.} Consider an actor program. The \textit{termination problem} is to decide whether it has no infinite computation; the \textit{process reachability problem} is to decide, given a process $P$, whether there exists a computation of the program traversing a configuration having a term $A \triangleright (P', \varphi, q)$ with $P'$ being equal to $P$ up-to renaming of variables and actor names.

We will use a reduction technique from a Turing-complete model to our actor model. In particular, the Turing-complete models we consider are the 2 Counter Machines (2CMs) \textsuperscript{29} and a faulty variation of them. A 2CM is a machine with \textit{two registers} $R_1$ and $R_2$ holding arbitrary large natural numbers and a \textit{program} $P$ consisting of a finite sequence of numbered instructions of the following type:

- $j : \textbf{Inc}(R_i)$: increments $R_i$ and goes to the instruction $j + 1$;
- $j : \textbf{DecJump}(R_i, l)$: if the content of $R_i$ is not zero, then decreases it by 1 and goes to the instruction $j + 1$, otherwise jumps to the instruction $l$;
- $j : \textbf{Halt}$: stops the computation and returns the value in the register $R_1$.

A state of the machine is given by a tuple $(i, v_1, v_2)$ where $i$ indicates the next instruction to execute (the program counter) and $v_1$ and $v_2$ are the contents of the two registers. The user has to provide the initial state of the machine. The transition relation of the 2CM will be denoted by $\Rightarrow$.

The faulty variation of the 2CM we use are the “two Faulty Towers Machine” (2FTM, for short) \textsuperscript{12}. These machines have \textit{two faulty registers} $R_1$ and $R_2$ holding either arbitrary large natural numbers or the \textit{faulty value} $\bot$. The program of a 2FTM is a finite sequence of numbered instructions that are the same of those in 2CMs. However, in contrast with 2CMs, 2FTMs from a state $(i, v_1, v_2)$ may \textit{nondeterministically} evolve into a faulty state $(i, \bot, v_2)$ or $(i, v_1, \bot)$ or $(i, \bot, \bot)$. If an instruction $i$ is an $\textbf{Inc}/\textbf{DecJump}$ that refers to the register $R_1$ (respectively, $R_2$) then $(i, \bot, v)$ (respectively, $(i, v, \bot)$) evolves to $(0, \bot, v)$ (respectively, $(0, v, \bot)$). In 2FTMs, the instruction numbered 0 is always assumed to be $\textbf{Halt}$. Let $\Rightarrow_F$ be the transition relation of the 2FTMs.

By definition, every 2CM program with a 0-numbered instruction $\textbf{Halt}$ is a 2FTM program and conversely. If we restrict to 2CM with a 0-numbered instruction $\textbf{Halt}$, it is easy to verify that every such machine has an infinite computation with a 2CM-semantics if and only if it has an infinite computation with a 2FTM-semantics. Similarly for instruction reachability, if we consider any non 0-numbered instruction.

In the sequel, we consider 2FTMs and 2CMs whose initial state is $(1, 0, 0)$.

\subsection{The language Actor\textsubscript{ba}}

We encode the value $n$ stored in a register as $n$ messages (of the same type) that are enqueued in an actor – see Figure\textsuperscript{1}. Namely, let $R_1$ and $R_2$ be two actors of class $R$ and let the number of messages $\textbf{item}$ in $R_1$ and $R_2$ be their value.

The instruction $\textbf{Inc}$ is implemented by inserting one $\textbf{item}$ message in the queue of the corresponding register. In our formalism, this is done by invoking the method $\textbf{item}$ whose execution has two possible outcomes: (i) the invocation is enqueued again; (ii) the invocation is discarded because we are in the presence of a residual of a $\textbf{DecJump}$ operation, as described next.
R

// R has fields dec, ctr, loop and stop
R.item(\&, ff) = ![stop = ff](\dec = \ff \text{ this! item}(\&; ff); (dec \leftarrow \ff))
R.inc(pc, \&; ff) = ![stop = ff](loop \leftarrow \ff). this! item(\&; ff). ctr! run(pc, \&; ff)
R.decjump(pc, pc', \&; ff) = ![stop = ff](loop \leftarrow \ff). (dec \leftarrow \&). this! checkzero(pc, pc', \&; ff)
R.checkzero(pc, pc', \&; ff) = ![stop = ff](loop \leftarrow \ff).
\[(\dec = \&)] ctr! run(pc', \&; ff). (dec \leftarrow \ff); ctr! run(pc, \&; ff)\]
R.init(\&; ff, Ctrl) = (dec \leftarrow \ff). (ctr \leftarrow Ctrl). (loop \leftarrow \ff). (stop \leftarrow \ff).
\text{ this! bottom(\&; ff) }
R.bottom(\&; ff) = ![loop = ff](loop \leftarrow \&). this! bottom(\&; ff); (stop \leftarrow \&)

Ctrl

// Ctrl has fields stm\_1, \ldots, stm\_n and r\_1 and r\_2
Ctrl.run(pc, \&; ff) = ![pc = stm\_1][Instruction\_1]\_i.e.f
\ldots
![pc = stm\_n][Instruction\_n]\_i.e.f
Ctrl.init() = r\_1! init(\&; ff, this). r\_2! init(\&; ff, this). this! run(stm\_1, \&; ff)
Ctrl.halt() = 0

where \([Instruction\_i]\_i.e.f\) is equal to
- \(r\_j! inc(stm\_{i+1}, \&; ff)\) if \(Instruction\_i = inc(R_j)\);
- \(r\_j! decjump(stm\_{i+1}, stm\_k, \&; ff)\) if \(Instruction\_i = DecJump(R_j, k)\);
- \(this! halt\) if \(Instruction\_i = Halt\).

The main process is

\text{let } x = \text{new Ctrl}(x_1, \ldots, x_n, \text{new R( } \underline{\underline{ } } \underline{\underline{ } } ), \text{new R( } \underline{\underline{ } } \underline{\underline{ } } )) \text{ in } x! \text{init}.\)

Figure 1: Encoding a 2FTM in Actor ba ("_" denotes an irrelevant initialization parameter)

In case (i), to avoid an infinite sequence of item dequeues and enqueues, we introduce fields stop and loop which are initialized to false and set to true by the bottom method in case the queue contains only item messages. This has as effect that the stored item messages are subsequently purged from the queue of the register. Note that differently from the example in Section 2, we have encoded the boolean values as fields (so that we do not need to pass them around).

In case (ii), registers have a field dec that is set to \& by a decjump method execution. This field means that the actual decrement of the register is delayed to the next execution of checkzero. Since in (ii) item is not enqueued, then the register is actually decremented and the field dec is set to \ff. When checkzero will be executed, since dec = \ff then the next instruction of the 2CM is simulated. On the contrary, when checkzero is executed with dec = \& then the decrement has not been performed (the register is 0) and the simulation jumps.
The _Halt_ instruction is simulated by invoking a method _halt_ that does nothing.

As in the examples of Section 2 booleans are implemented by two variables – see the method _Ctrl_.init – that are distributed during the invocations. With a similar machinery, in the actor class _Ctrl_, the labels of the instructions are represented by the variables \(x_1, \ldots, x_n\), which are stored in the fields _stm_1, \ldots, _stm_n_ of _Ctrl_.

**Theorem 3.2.** Termination and process reachability are undecidable in _Actor_\_ba_.

The undecidability of termination in _Actor_\_ba_ follows by the property that a 2FTM diverges if and only if the corresponding actor program has an infinite computation. As regards process reachability, we need a smooth refinement of the encoding in Figure 1 where the _Halt_ instruction is simulated by a specific process.

**Proof.** Let us assume to have a fixed 2FTM and let \([Instruction_{i,v}]_{i,v}^{0,2}\) be defined in Figure 1. Let also

\[
\begin{align*}
[(i, v_1, v_2)]_{i,v} & \overset{def}{=} \mathbb{N} \triangleright (0, \varnothing, \varepsilon), \\
C \triangleright ([Instruction_{i,v}]_{i,v}, \varphi_{ctrl}, \varepsilon), \\
R_1 \triangleright (0, \varphi_{reg}, bottom(\#), \#) (item(\#))^{v_1}, \\
R_2 \triangleright (0, \varphi_{reg}, bottom(\#), \#) (item(\#))^{v_2}
\end{align*}
\]

\[
\begin{align*}
[(i, \bot, v)]_{i,v} & \overset{def}{=} \mathbb{N} \triangleright (0, \varnothing, \varepsilon), \\
C \triangleright ([Instruction_{i,v}]_{i,v}, \varphi_{ctrl}, \varepsilon), \\
R_1 \triangleright (0, \varphi_{reg}, \varepsilon), \\
R_2 \triangleright (0, \varphi_{reg}, bottom(\#), \#) (item(\#))^{v}
\end{align*}
\]

\[
\begin{align*}
[(0, \bot, v)]_{i,v} & \overset{def}{=} \mathbb{N} \triangleright (0, \varnothing, \varepsilon), \\
C \triangleright (0, \varphi_{ctrl}, \varepsilon), \\
R_1 \triangleright (0, \varphi_{reg}, \varepsilon), \\
R_2 \triangleright (0, \varphi_{reg}, bottom(\#), \#) (item(\#))^{v}
\end{align*}
\]

where

- \(\varphi_{ctrl} \overset{def}{=} [stm_i \mapsto x_i, r_1 \mapsto R_1, r_2 \mapsto R_2];\)
- \(\varphi_{reg} \overset{def}{=} [stop \mapsto f, loop \mapsto f, dec \mapsto f, ctr \mapsto C];\)
- \(\varphi_{reg} \overset{def}{=} [stop \mapsto t, loop \mapsto t, dec \mapsto f, ctr \mapsto C];\)
- \(item(\#))^{v} \overset{def}{=} \underbrace{item(\#) \cdots item(\#)}_{v \text{ times}}.\)

(the definitions of \([i,v,\bot])_{i,v}^{0,2}\) and \([(0,v,\bot)]_{i,v}^{0,2}\) and \([(0,\bot,v)]_{i,v}^{0,2}\) follow the same patterns).

We first observe that \(\mathbb{N} \triangleright (P, \varnothing, \varepsilon) \rightarrow^{*} ([1,0,0])_{i,v}\), where _P_ is the main process in Figure 1. Then we demonstrate the following properties:

(1) if \((i,v_1,v_2) \rightarrow^{*} (j,v'_1,v'_2)\) then \([i,v_1,v_2])_{i,v}^{0,2} \rightarrow^{*} [(j,v'_1,v'_2)]_{i,v}^{0,2};\)

(2) if \([(i,v_1,v_2)]_{i,v}^{0,2}\) has an infinite computation then the computation has infinitely many configurations like

\[
\begin{align*}
\mathbb{N} \triangleright (0, \varnothing, \varepsilon), C \triangleright ([Instruction_{i,v}]_{i,v}, \varphi_{ctrl}, \varepsilon), R_1 \triangleright (P_1, \varphi_1, q_1), R_2 \triangleright (P_2, \varphi_2, q_2) \quad (3.1)
\end{align*}
\]
The undecidability result can be easily extended to the process reachability problem. It suffices to modify the process modeling an Halt (not numbered with 0) by replacing this !halt() with a process \( Q \) different from all the other processes in Figure 1. We have that \( (i, v_1) \mapsto_F (j, v_1) \).

The proof of (1) is a straightforward case analysis on the type of instruction \( i \).

The proof of (2) uses an argument by contradiction. Assume that there are finitely many configurations like \((\mathcal{R}, \mathcal{C})\). Then, there is an infinite suffix of this computation in which the actor \( C \) enters a faulty state \( \bot \). In the second case the instruction has been correctly executed. This means that at least one actor \( R_i \) performs infinitely many actions by executing the methods \( \text{item} \) and \( \text{bottom} \). But this is not possible because \( \text{bottom} \) blocks the actor if it is invoked twice without executing update actions on the register in between.

The proof of (3) simply considers two possible cases: either some fields \( \text{stop} \) is set to \( \top \) during the computation or not. In the first case the corresponding register of the 2FTM enters a faulty state \( \bot \). In the second case the instruction has been correctly executed.

The property (1) guarantees that if a 2FTM has an infinite computation then also the corresponding encoding has an infinite computation. The opposite follows from (1) and (2): if the encoding has an infinite computation, it traverses infinitely many configurations representing configurations of the corresponding 2FTM, thus it also has an infinite computation. The undecidability result can be easily extended to the process reachability problem. It suffices to modify the process modeling an Halt (not numbered with 0) by replacing this !halt() with a process \( Q \) different from all the other processes in Figure 1. We have that \( Q \) is reachable if and only if the 2CM with the same program of the given 2FTM terminates. \( \square \)

3.2. The language \texttt{Actor}. We show that \texttt{Actor} is Turing-complete by means of an encoding of a 2CM – see Figure 2. In this encoding the two registers are represented by two disjoint stacks of actors linked by the \texttt{next} field. The top elements of the two stacks are passed as parameters \( r_1 \) and \( r_2 \) of the \texttt{run} method of the controller. As before, this actor encodes the control of the 2CM.

The instruction \texttt{Inc} is implemented by pushing an element on top of the corresponding stack. This element is an actor of class \texttt{R} storing in its field the old pointer of the stack. The new pointer, \textit{i.e.} the new actor name, is passed to the next invocation of the \texttt{run} method.

The instruction \texttt{DecJump} is implemented by popping the corresponding stack. In particular, the method \texttt{run} of the controller is invoked with the field \texttt{next} of the register being decreased. This pop operation is performed provided the register that is argument of \texttt{run} is different from \texttt{nil}. Otherwise a jump is performed. Note that the other top of
the stack \( r_j \) (\( i \neq j \)) and the next instruction to be executed are simply passed around and therefore they do not need to be stored in updatable fields.

**Theorem 3.3.** Termination and process reachability are undecidable in \( \text{Actor}^{\infty} \).

**Proof.** It is easy to verify that if a 2CM has a computation
\[
(1, 0, 0) \implies (i_1, v_1, v'_1) \implies \ldots \implies (i_n, v_n, v'_n)
\]
then there is a computation
\[
\mathcal{N} \triangleright (P, \emptyset, \varepsilon) \rightarrow^* \mathcal{N} \triangleright (0, \emptyset, \varepsilon), C \triangleright ([\text{Instruction}_1\rightarrow \text{nil}, \text{nil}, \varphi, \varepsilon])
\]
\[
\rightarrow^* \mathcal{N} \triangleright (0, \emptyset, \varepsilon), C \triangleright ([\text{Instruction}_{r_1} \rightarrow r_1', \varphi, \varepsilon], \mathcal{R}_{r_1}, \mathcal{R}_{r_1'}, \mathcal{G}_1)
\]
\[
\rightarrow^* \mathcal{N} \triangleright (0, \emptyset, \varepsilon), C \triangleright ([\text{Instruction}_{r_n} \rightarrow r_n', \varphi, \varepsilon], \mathcal{R}_n, \mathcal{R}_{r_n'}, \mathcal{G}_n)
\]
where \( P \) is the main process of Figure 2, \([\text{Instruction}_{i_j} \rightarrow \varphi]_{r_j, r'_j} \) are defined in Figure 2, \( \varphi = \text{stm}_1 \rightarrow x_1, \ldots, \text{stm}_n \rightarrow x_n, \text{nil} \rightarrow \text{nil} \), \( \mathcal{R}_{r_1} \) and \( \mathcal{R}_{r_j} \) are stacks of register actors whose length is \( v_i \) and \( v'_i \), respectively. For instance, \( \mathcal{R}_{r_1} \) of length \( k \) is
\[
r_1 \triangleright (0, [\text{next} \rightarrow r_2], \varepsilon), r_2 \triangleright (0, [\text{next} \rightarrow r_3], \varepsilon), \ldots, r_k \triangleright (0, [\text{next} \rightarrow \text{nil}], \varepsilon).
\]
The configurations \( \mathcal{G}_i \) only contain register terms \( r\triangleright (0, [\text{next} \rightarrow r'), \varepsilon) \) and represent garbage (they are inactive).

In contrast with Theorem 3.2, the converse implication (every computation of the \( \text{Actor}^{\infty} \) program in Figure 2 may be split in subcomputations of finite lengths that correspond to 2CM transitions) is not difficult because the program of Figure 2 is deterministic.

The above correspondence guarantees that the computation of the actor system terminates if and only if a \( \text{Halt} \) instruction is reached. The undecidability of process reachability is proved by using the same arguments of Theorem 3.2.

4. **Decidability results for \( \text{Actor}^{\infty} \)**

We demonstrate that programs in \( \text{Actor}^{\infty} \) are well-structured transition systems [1, 16]. This will allow us to decide a number of properties, such as termination. We begin with some background on well-structured transition systems.

A reflective and transitive relation is called quasi-ordering. A well-quasi-ordering is a quasi-ordering \((X, \leq)\) such that, for every infinite sequence \( x_1, x_2, x_3, \ldots \), there exist \( i, j \) with \( i \neq j \) and \( x_i \leq x_j \).

**Definition 4.1.** A well-structured transition system is a finitely branching transition system \((\mathcal{S}, \rightarrow, \preceq)\) where \( \preceq \) is a quasi-ordering relation on states such that
\begin{itemize}
\item[(1)] \( \preceq \) is a well-quasi-ordering,
\item[(2)] \( \preceq \) is upward compatible with \( \rightarrow \), i.e., for every \( S_1 \preceq S'_1 \) and \( S_1 \rightarrow S_2 \), there exists \( S'_1 \rightarrow^* S'_2 \) such that \( S_2 \preceq S'_2 \).
\end{itemize}

Given a state \( s \) of a well-structured transition system, \( \text{Pred}(s) \) denotes the set of immediate predecessors of \( s \) (i.e., \( \text{Pred}(s) = \{ s' \mid s' \rightarrow s \} \)) while \( \uparrow s \) denotes the set of states greater than \( s \) (i.e., \( \uparrow s = \{ s' \mid s \preceq s' \} \)). With abuse of notation we will denote with \( \text{Pred}(\downarrow) \) also its natural extension to sets of states.

According to the theory of well-structured transition systems [1, 16], we have that several properties are decidable for such transition systems (under some conditions discussed below). We will consider the following properties.
R // R has a field next
R.dec₁(ctrl, r, stm) = ctrl! run(next, r, stm)
R.dec₂(ctrl, r, stm) = ctrl! run(r, next, stm)

Ctrl // Ctrl has fields \(stm_1, \ldots, stm_n\) and nil

\[
\text{Ctrl.run}(r_1, r_2, pc) = [pc = stm_1][\text{Instruction}_1]_{r_1,r_2}; \\
\ldots \\
[pc = stm_n][\text{Instruction}_n]_{r_1,r_2}
\]

where \([\text{Instruction}_i]_{r_1,r_2}\) is equal to

- \(\text{this! run(new R(r_1), r_2, stm_{i+1})}\) if \(\text{Instruction}_i = \text{Inc}(R_1)\);
- \(\text{this! run(r_1, new R(r_2), stm_{i+1})}\) if \(\text{Instruction}_i = \text{Inc}(R_2)\);
- \([r_1 = \text{nil}] \text{this! run(r_1, r_2, stm_k)}; r_1! \text{dec}_1(\text{this}, r_2, stm_{i+1})\) if \(\text{Instruction}_i = \text{DecJump}(R_1, k)\);
- \([r_2 = \text{nil}] \text{this! run(r_1, r_2, stm_k)}; r_2! \text{dec}_2(\text{this}, r_1, stm_{i+1})\) if \(\text{Instruction}_i = \text{DecJump}(R_2, k)\);
- \(0\) if \(\text{Instruction}_i = \text{Halt}\).

The main process is \(\text{let } x = \text{new Ctrl}(x_1, \ldots, x_n, \text{nil}) \text{ in } x! \text{run}(\text{nil}, \text{nil}, x_1)\).

Figure 2: Encoding a 2CM in Actor²

Definition 4.2. Consider a well-structured transition system \((\mathcal{S}, \rightarrow, \preceq)\). Given \(S \in \mathcal{S}\) the **termination problem** is to decide whether \(S\) has an infinite computation; the **control-state reachability** problem is to decide, given \(T \in \mathcal{S}\), whether there is \(T' \succeq T\) such that \(S \rightarrow^* T'\).

In well-structured transition systems termination is decidable when the transition relation \(\rightarrow\) and the ordering \(\preceq\) are effectively computable. When it is also possible to effectively compute a finite-basis for the set of states \(\text{Pred}(\uparrow s)\) we have that control-state reachability is decidable as well.

In the following we assume given an actor program with its main process and its set of actor class definitions. The first relation we convey is \(=\) that relates renamings, ranged over by \(\rho, \rho', \ldots\) that are functions mapping variables that are not free in the main process into either actor names or variables. Let

\[
\rho = \rho' \overset{\text{def}}{=} \text{for every } x, y:\ (i) \rho(x) = \rho(y) \quad \text{if and only if} \quad \rho'(x) = \rho'(y)
\]

\(\quad (ii) \rho(x) = \rho'(x) \quad \text{if } \rho(x) \text{ or } \rho'(x) \text{ is an actor name or a free variable of the main process}\)

Namely, two renamings are in the relation \(=\) if they identify the same variables, regardless the value they associate when such a value is a variable. For example, \([x \mapsto y, y \mapsto z] = [x \mapsto x, y \mapsto y, z \mapsto z]\) and \([x \mapsto y, y \mapsto y, z \mapsto A] = [x \mapsto x', y \mapsto y', z \mapsto A]\). However \([x \mapsto y, y \mapsto z] \neq [x \mapsto x, y \mapsto x]\) and \([x \mapsto A] \neq [x \mapsto B]\). In general, if \(\rho\) and \(\rho'\) are injective renamings that always return variables then \(\rho = \rho'\). The requirements of \(=\) are stronger for actor names: in this case the two renamings should be identical. By definition,
renamings in relation according to \( \cong \) never apply to free variables of the main process. This because these variables are possibly stored in fields of actors and their renaming might change the behaviours of actors in a way that breaks the upward compatibility of the following relation \( \preceq \) and \( \rightarrow \) (c.f. proof of Theorem 4.4, part (2)). From this also follows that the above renamings do not change the main process (because they do not apply to its free variables).

Let \( \text{dom}(\rho) \) be the domain of the renaming \( \rho \). We denote by \( P\rho \) the result of \( P[\rho(\bar{x})/\bar{z}] \), where \( \bar{x} = x_1, \ldots, x_n \) is a tuple containing the variables in \( \text{dom}(\rho) \) (without repetitions) and \( \rho(\bar{x}) = \rho(x_1), \ldots, \rho(x_n) \).

Next, let \( \cong \) be the least relation on terms \( m(U_1, \ldots, U_n) \) and on processes such that

\[
\begin{align*}
\rho = \rho' & \quad & \rho = \rho' \\
\rho = \rho' & \quad & \rho = \rho'
\end{align*}
\]

- \( m(\rho(x_1), \ldots, \rho(x_k)) \cong m(\rho'(x_1), \ldots, \rho'(x_k)) \)
- \( P\rho \cong P\rho' \)

For example, it is easy to verify that \( m(x, y) \cong m(x', y') \) and that \( [x = A]y! m(x, A, y) \cong [z = A]y! m(z, A, y') \). On the contrary \( [x = A]B! m(x, A, B) \not\cong [z = A]y! m(z, A, y') \). The rationale behind \( \cong \) is that it identifies processes that “behave in similar ways”, namely they enqueue “similar invocations” in the same actor queue. Method invocations \( m(U_1, \ldots, U_n) \) of a given actor are identified if the processes they trigger “behave in similar ways”.

**Lemma 4.3.** Let \( T \) be either a method invocation \( m(U_1, \ldots, U_n) \) or a process of a program in \( \text{Actor}_{ba} \) (and therefore in \( \text{Actor}_{ro} \)). Let \( T = \{ T_{\rho_1}, T_{\rho_2}, T_{\rho_3}, \ldots \} \) be such that \( i \not= j \) implies \( T_{\rho_i} \not\equiv T_{\rho_j} \). Then \( T \) is finite.

**Proof.** We demonstrate the lemma for processes, the argument is similar for method invocations. So, let \( P \) be a process. It is possible to count the number of renamings \( \rho \) on \( \text{free}(P) \) that are different according to \( \cong \). In fact, the values of renamings on variables that are different from \( \text{free}(P) \) do not play any role in the definition of \( T \).

The basic remark is that a renaming \( \rho \) generates a partition of the set \( \text{free}(P) \): two variables \( x \) and \( y \) are in the same partition if and only if \( \rho(x) = \rho(y) \). If we restrict to renamings that map variables to variables (and not actor names), then they are different according to \( \cong \) if they yield different partitions. The number of such renaming is the Bell number \( \text{Bell}(\kappa) \) of the cardinality of \( \text{free}(P) \), let it be \( \text{Bell}(\kappa) \), where \( \kappa \) is the cardinality of \( \text{free}(P) \).

In addition, in our case, renamings may map a variable to an actor name into a finite set \( \{ A_1, \ldots, A_\ell \} \). In this case the identity of the actor name is relevant. If \( \kappa \geq \ell \) then \((\ell ^\kappa \times \ell ! + 1) \times \text{Bell}(\kappa) \) is an upper bound to the different renamings according \( \cong \). If \( \kappa < \ell \) then the upper bound is \((\ell !/\kappa ! + 1) \times \text{Bell}(\kappa) \). In any case the number of different renamings according to \( \cong \) is finite.

Henceforth the set \( T \) is finite as well. \( \square \)

The well-quasi-ordering relation on configurations relies on an (almost standard) embedding relation \( \preceq \) on queues (except the part about \( \cong \), it is the one in [16]):

\[
\text{there exist } i_1 < i_2 < \cdots < i_h \text{ such that, for } j \in 1..k, \ m_j(\hat{U}_j) \cong n_{i_j}(\hat{V}_{i_j})
\]

\[
m_1(\hat{U}_1) \cdots m_k(\hat{U}_k) \preceq m_1(\hat{V}_1) \cdots m_k(\hat{V}_k)
\]
Theorem 4.4. Let \((S, \longrightarrow)\) be a transition system of a program of \(\text{Actor}_{ba}^0\). Then \((S, \longrightarrow, \preceq)\) is a well-structured transition system.

Proof. (1) \(\preceq\) is a well-quasi-ordering. To prove that \(\preceq\) is a well-quasi-ordering, we reason by contradiction. Let \(S_1, S_2, S_3, \cdots\) be an infinite sequence of states in \(S\) such that, for every \(i < j\), \(S_i \not\preceq S_j\). Let \(y_1, \cdots, y_m\) be a sequence of variables not free in the main process. Consider:

\[
\text{subterms}(C) = \{ P \mid \text{there exists a method } m \text{ s.t. } P \text{ is a subterm of } C.m(\overline{x})\} \cup \\
\{m(y_1, \cdots, y_g) \mid \text{there exists a method } m \text{ s.t. } C.m(\overline{x}) \text{ with } |\overline{x}| = g\}
\]

The set \(\text{subterms}(C)\) is finite, but all the \(\preceq\)-possibly infinitely many\(\)-processes that can be executed (or the messages that can be received) by an actor of class \(C\) are renaings \(P \rho\) (or \(m(\overline{y})\rho\)) of these terms. Notice that by Lemma 4.3, the number of terms \(P \rho\) (and \(m(\overline{y})\rho\)) which are different according to \(\simeq\) is finite as well. It is thus possible to extract a subsequence \(S_{i_1}, S_{i_2}, S_{i_3}, \cdots\) from \(S_1, S_2, S_3, \cdots\) such that, for every \(A\), in the elements \(A \triangleright (P_{i_j}^{A_1}, \varphi_{i_j}^{A_1}, q_{i_j}^{A_1})\) and \(A \triangleright (P_{i_k}^{A_2}, \varphi_{i_k}^{A_2}, q_{i_k}^{A_2})\) of \(S_{i_j}\) and \(S_{i_k}\), respectively, we have that \(P_{i_j}^{A_1} \rho_{i_j} \simeq P_{i_k}^{A_2} \rho_{i_k}\). Moreover, as we are considering \(\text{Actor}_{ba}^0\), the actor state cannot be modified, hence \(\varphi_{i_j}^{A_1} = \varphi_{i_k}^{A_2}\).

As we are considering \(\text{Actor}_{ba}^0\), the set of actor is bound. Let \(A_1, \cdots, A_\ell\) be such actor names. Due to the above arguments, the sequence \(S_{i_1}, S_{i_2}, S_{i_3}, \cdots\) may be represented as a sequence of tuples of queues:

\[
(q_{i_1}^{A_1}, \cdots, q_{i_1}^{A_\ell}), (q_{i_2}^{A_1}, \cdots, q_{i_2}^{A_\ell}), (q_{i_3}^{A_1}, \cdots, q_{i_3}^{A_\ell}), \cdots
\]

such that \(S_{i_j} \succeq S_{i_k}\) if and only if \((q_{i_j}^{A_1}, \cdots, q_{i_j}^{A_\ell}) \sqsubseteq (q_{i_k}^{A_1}, \cdots, q_{i_k}^{A_\ell})\) and \((q_{i_j}^{A_1}, \cdots, q_{i_j}^{A_\ell}) \sqsubseteq (q_{i_k}^{A_1}, \cdots, q_{i_k}^{A_\ell})\), where \(\sqsubseteq\) is the coordinate-wise order defined by

\[
(q_1, \cdots, q_\ell) \sqsubseteq (q_1', \cdots, q_\ell') \quad \text{def} \quad \text{for every } h : q_h \leq q_h'
\]

(\(\leq\) is the above embedding relation).

We are finally reduced to an infinite sequence of tuples of queues such that every tuple cannot be in relation according to \(\sqsubseteq\) with any of the subsequent ones. This fact contradicts the

Higman’s Lemma \([21]\): if \((X, \leq)\) is a well-quasi-ordering and \((X^*, \leq^*)\) is the set of finite \(X\)-sequences ordered by the embedding relation \(\leq^*\) defined using \(\leq\) as pointwise ordering, then \((X^*, \leq^*)\) is a well-quasi-ordering.

More precisely, the contradictions follows from the following consequence of the Higman’s Lemma:

- if \(X\) is a finite set and \((X^*, \leq)\) is the set of finite \(X\)-sequences ordered by the embedding relation, then \((X^*, \leq)\) is a well-quasi-ordering.

and from the following statement
• if \((X, \preceq)\) is a well-quasi-ordering then \((X^\ell, \preceq^\ell)\) is a well-quasi-ordering.

(2) \(\preceq\) is upward compatible with \(\rightarrow\). A state \(\varphi\) is normed, if, for every field \(f\), \(\varphi(f)\) is either a free variable in the main process or an actor name. A configuration is normed if the states of the actors are normed. We observe that the initial configuration is normed. We also let \((\tilde{E})\rho \simeq (\tilde{E})\rho'\) whenever \(\rho \ast \rho'\).

We first demonstrate that, if \(\tilde{E} \xrightarrow{\varphi} U ; \emptyset\) with \(\varphi\) normed, then

(exp-i) if \(\tilde{E} \simeq \tilde{E}'\) and \(S = \emptyset\) then \(\tilde{E}' \xrightarrow{\varphi} U ; \emptyset\) and \(\tilde{U} \simeq \tilde{U}';\)

(exp-ii) if \(\tilde{E}\) (respectively \(P\)) only contain free variables in the main process and actor names then \(\tilde{E} \simeq \tilde{E}'\) (respectively \(P \simeq P'\)) implies \(\tilde{E} = \tilde{E}'\) (respectively \(P = P'\)) and \(\tilde{E} \xrightarrow{\varphi} U ; S\) implies that \(U\) contain free variables in the main process and actor names and \(S\) is normed.

(exp-i) is proved by induction on the right of the proof-tree of \(\tilde{E} \xrightarrow{\varphi} U ; \emptyset\). There are two basic cases: (1) \(E = U\) and (2) \(E = f\). As regards (1), \(E' = U'\) and the property is immediate by the hypothesis that \(E \simeq E'\). As regards (2), \(E' = f\) because \(E \simeq E'\); henceforth the property (because \(E'\) is evaluated in the state \(\varphi\) as well). There is one inductive case (because the case of \texttt{new} is not possible, otherwise \(S\) cannot be empty), which is immediate.

(exp-ii) is an immediate consequence of the definition of \(\ast\) and \(\simeq\).

Let \(S_1 \rightarrow S_2\). We demonstrate that

(i) if \(S_1\) is normed then \(S_2\) is normed as well (this means that the transition system \((S, \rightarrow)\) of a program of \texttt{Actor} has normed configurations because the initial state is normed);

(ii) if \(S_1 \preceq S'_1\) then there exists \(S'_1 \rightarrow^* S'_2\) such that \(S_2 \preceq S'_2\).

As regards (i), it follows by remarking that in programs of \texttt{Actor}, there is no field update and the unique process that may create states is the one of \(\emptyset\) (the main process). Then (i) derives from the property (exp-ii).

As regards (ii), its proof is a case analysis on the proof-tree of \(S_1 \rightarrow S_2\) where the cases correspond to the unique rule appearing in the tree that is not an instance of (context). Let \(S_1 = A_1 \triangleright (P_1 \rho_1, \varphi_1, q_1), \ldots, A_\ell \triangleright (P_\ell \rho_\ell, \varphi_\ell, q_\ell)\). Since \(S_1 \preceq S'_1\) then \(S'_1 = A_1 \triangleright (P_1 \rho_1', \varphi_1, q'_1), \ldots, A_\ell \triangleright (P_\ell \rho_\ell', \varphi_\ell, q'_\ell)\) such that, for every \(i\), \(P_i \rho_i \ast P_i \rho_i'\) and \(q_i \preceq q'_i\). The cases are discussed in order.

(1) \(S_1 \rightarrow S_2\) contains an instance of (let), namely

\[ A \triangleright (\text{let } x = E \text{ in } P, \varphi, q), \rightarrow A \triangleright (P[U/x], \varphi, q), S_3. \]

where \(E \xrightarrow{\varphi} U ; S\). By \(S_1 \preceq S'_1\), \(S'_1\) must contain \(A \triangleright (\text{let } x = E' \text{ in } P', \varphi, q)\) such that \(\text{let } x = E \text{ in } P \simeq \text{let } x = E' \text{ in } P'\) (without loss of generality, we are assuming the two bound variables are the same) and \(q \preceq q'\). There are two subcases: (1.1) \(A = \emptyset\) and (1.2) \(A \neq \emptyset\). In (1.1), By (exp-ii), this is possible provided \(E = E'\) and \(P = P'\). It is easy to verify that \(S'_1 \rightarrow S'_2\) and \(S_2 \preceq S'_2\) because their unique difference with \(S_1\) and \(S'_1\) is due to the two processes \(P\) and \(P'\). In (1.2), \(S_3 = \emptyset\) because no \texttt{new} can occur in \(E\). Additionally, by definition of \(\simeq\), \(E \simeq E'\) and \(P \simeq P'\). Let \(E \xrightarrow{\varphi} U', \emptyset\). By (exp-i) we have \(U \simeq U'\) and it is easy to verify that \(P[U/x] \simeq P'[U'/x]\). Henceforth \(S'_1 \rightarrow S'_2\) and \(S_2 \preceq S'_2\) because their unique difference with \(S_1\) and \(S'_1\) is due to the two processes \(P[U/x]\) and \(P'[U'/x]\).
(2) $S_1 \rightarrow S_2$ contains an instance of ($\text{INVK-s}$), namely

$$A \triangleright (A!\cdot m(\bar{E}), P, \varphi, q), \rightarrow A \triangleright (P, \varphi, q \cdot m(\bar{U})).$$

Since $S_1 \preceq S'_1$ then $S'_1$ contains $A \triangleright (A!\cdot m(\bar{E}'), P', \varphi', q')$ with $\bar{E} \simeq \bar{E}'$, $P \simeq P'$, and $q \leq q'$. We observe that $A \neq \emptyset$ and if $\bar{E} \not\simeq \bar{U}$, $\varphi$ and $\bar{E}' \not\simeq \bar{U}'$, $\varphi$ then $\bar{U} \not\simeq \bar{U}'$ by (exp-i). Therefore $q \cdot m(\bar{U}) \leq q' \cdot m(\bar{U}')$ and $S'_1 \rightarrow S'_2$ with $S_2 \preceq S'_2$ because their unique difference with $S_1$ and $S'_1$ is due to the two terms $A \triangleright (P, \varphi, q \cdot m(\bar{U}))$ and $A \triangleright (P', \varphi, q' \cdot m(\bar{U}'))$.

(3) $S_1 \rightarrow S_2$ contains an instance of ($\text{INVK}$), namely

$$A \triangleright (B!\cdot m(\bar{E}) \cdot P, \varphi, q), B \triangleright (Q, \psi, p) \rightarrow A \triangleright (P, \varphi, q), B \triangleright (Q, \psi, p \cdot m(\bar{U})), S_3.$$

There are two subcases: either $A = \emptyset$ or $A \neq \emptyset$. When $A = \emptyset$ the proof is similar to the above case (1.1); when $A \neq \emptyset$ the proof is similar to case (2).

(4) $S_1 \rightarrow S_2$ contains an instance of ($\text{INST}$), namely

$$A \triangleright (0, \varphi, m(\bar{U}) \cdot q) \rightarrow A \triangleright (P[A/\text{this}][\bar{y}/\bar{y}][\bar{U}/\bar{z}], \varphi, q),$$

where $C \cdot m(\bar{z}) = P$, $C$ being the class of $A$, $\bar{y} = \text{free}(P) \setminus \bar{z}$ and $\bar{y}' = \text{fresh}(\bar{y})$. Therefore $S_1 = A \triangleright (0, m(\bar{U}) \cdot q), T_1$ and $S_2 = A \triangleright (P[A/\text{this}][\bar{y}/\bar{y}][\bar{U}/\bar{z}], q), T_1$. Since $S_1 \preceq S'_1$ then $S'_1 = A \triangleright (0, \varphi, n_1(\bar{V}) \cdots n_h(\bar{V}_h) \cdot m(\bar{V}) \cdot q'), T'_1$ and $m(\bar{U}) \simeq m(\bar{V})$ and $q \leq q'$ and $T_1 \preceq T'_1$. By the operational semantics rules, we get $S'_1 \rightarrow^* A \triangleright (0, \varphi, m(\bar{V}) \cdot q' \cdot q''), T'_1$ by performing transitions of the actor $A$, with $T'_1 \preceq T'_2$ and, by definition, $q \leq q' \cdot q''$. At this stage, we notice that $A \triangleright (0, \varphi, m(\bar{V}) \cdot q' \cdot q''), T'_1 \rightarrow A \triangleright (P[A/\text{this}][\bar{z}/\bar{y}][\bar{V}/\bar{z}], \varphi, q' \cdot q''), T'_2$. We notice that $P[A/\text{this}][\bar{y}/\bar{y}][\bar{U}/\bar{z}] = P[A/\text{this}][\bar{y} \mapsto \bar{y}, \bar{z} \mapsto \bar{V}]$ and $P[A/\text{this}][\bar{y}/\bar{y}][\bar{V}/\bar{z}] = P[A/\text{this}][\bar{y} \mapsto \bar{z}, \bar{x} \mapsto \bar{V}]$ and $[\bar{y} \mapsto \bar{y}', \bar{x} \mapsto \bar{U}] = [\bar{y} \mapsto \bar{z}, \bar{x} \mapsto \bar{V}]$. Therefore

$$P[A/\text{this}][\bar{z}/\bar{y}][\bar{V}/\bar{z}] \simeq P[A/\text{this}][\bar{z}/\bar{y}][\bar{V}/\bar{z}]$$

which implies that $S'_1 \rightarrow^* S'_2$ and $S_2 \preceq S'_2$ because their unique difference with $S_1$ and $S'_1$ is due to the two above processes.

(5) $S_1 \rightarrow S_2$ contains an instance of ($\text{MATCH}$), namely

$$A \triangleright (E = E') P; Q, \varphi, q) \rightarrow A \triangleright (P, \varphi, q), S_3.$$

We discuss the case $A \neq \emptyset$ because the other one is similar to (1.1). There are three subcases (5.1) both $E$ and $E'$ are variables; (5.2) $E$ is a variable and $E'$ is a field; (5.3) $E$ and $E'$ are both fields. In case (5.1), let $E = x = E'$. Since $S_1 \preceq S'_1$, then $S'_1$ must contain $A \triangleright ([x = z] P'; Q', \varphi, q')$ with $[x = x] P; Q \simeq [z = z] P'; Q'$ and $q \leq q'$. Therefore we may use (MATCH) to derive $S'_1 \rightarrow S'_2$ with $S_2 \preceq S'_2$. In case (5.2), let $E = U$ and $E' = \emptyset$. There are two subcases: (5.2.1) $U$ is a variable or (5.2.2) $U$ is an actor name. In (5.2.1), $U$ has to be a free variable in the main process because we are using (MATCH) ($\emptyset$ may contain either such variables or actor names, additionally, renamings never return free variables in the main process). Therefore, by $S_1 \preceq S'_1$, we have that $S'_1$ contains $A \triangleright ([U = \emptyset] P'; Q', \varphi, q')$ with $[U = \emptyset] P; Q \simeq [U = \emptyset] P'; Q'$ and $q \leq q'$. The consequence is that $S'_1 \rightarrow S'_2$ with $S_2 \preceq S'_2$ because their unique difference with $S_1$ and $S'_1$ is due to the two either the pair of processes $P$, $P'$ or $Q$, $Q'$. Similarly for (5.2.2). The case (5.3) is obvious.

(6) $S_1 \rightarrow S_2$ contains an instance of ($\text{M MATCH}$). Similar to (5).

(7) $S_1 \rightarrow S_2$ contains an instance of ($\text{PLUS-L}$) or of ($\text{PLUS-R}$). Straightforward. $\square$
We notice that the well-structured transition system \((S, \rightarrow, \preceq)\) has decidable algorithms for computing \(\preceq\) and for computing the next states. Then decidability of termination directly follows from the above mentioned results of the theory of well-structured transition systems that we have previously recalled.

**Theorem 4.5.** In programs of \(\text{Actor}_{ba}^{ro}\) the termination problem is decidable.

We now move to the definition of an appropriate algorithm for the computation of a finite basis for the predecessors of a given configuration, so to conclude also the decidability of control-state reachability.

**Lemma 4.6.** Let \((S, \rightarrow, \preceq)\) be a well-structured transition system of a program in \(\text{Actor}_{ba}^{ro}\), and let \(S \in S\). Then there is a finite set \(X \subseteq \text{Pred}(S)\) such that, for every \(S' \in \text{Pred}(S)\), there is \(T \in X\) with \(T \preceq S'\). \(X\) can be effectively computed.

**Proof.** We show how to compute \(X\). Let \(S = A \triangleright (P, \varphi, q), S'\). The predecessor processes of \(P\) are the following ones: (i) \(\text{let } x = E \text{ in } P'\), with \(P = P'[\overline{U}/x]\), for some \(\overline{U}\) and some \(\overline{x}\); (ii) \(x \triangleright \text{tm}(E_1, \ldots, E_n), P\); (iii) \(|U = U| P; Q\); (iv) \(|U = V| Q; P\); (v) \(P + Q\); (vi) \(Q + P\); (vii) \(P\) is an instance of a method body of the actor class of \(A\). If \(A\) is of actor class \(C\) then we take all the method bodies of \(C\) with a suffix matching one of the cases (i)-(vi) above (in this case, the expressions in (ii) are either variables or actor names). If \(A = \emptyset\) then we look for a matching suffix of the main process. The above six cases are demonstrated in the presence of such suffixes.

We only discuss case (i), the other ones are similar. In case (i), if \(A\) is of actor class \(C\), then \(E = y\), for some \(y\). If \(x \in \text{free}(P')\) then \(X\) contains the configuration \(A \triangleright (\text{let } x = y \text{ in } P', \varphi, q), S'\) with \(P = P'[y/x]\). Otherwise \(X\) contains the configuration \(A \triangleright (\text{let } x = z \text{ in } P', \varphi, q), S'\), for \(z \in \text{free}(P')\) and for a unique \(z \notin \text{free}(P')\). When \(A = \emptyset\) then \(E\) may be new \(C\) (otherwise the argument is as before). If \(x \in \text{free}(P')\) and \(S' = A' \triangleright (0, \varphi, \varepsilon)\), \(S''\) with \(A' \in C\) then \(X\) contains the configuration \(A \triangleright (\text{let } x = \text{new } C \text{ in } P', q), S''\) (and this for every possible \(A' \in C\) such that \(A' \triangleright (0, \varepsilon)\) is in \(S'\)).

Lemma 4.6 and the above mentioned results on well-structured transition systems allow us to decide the control-state reachability problem.

**Theorem 4.7.** In programs of \(\text{Actor}_{ba}^{ro}\) the control-state reachability problem is decidable.

**Proof.** Let \(\uparrow S = \{S' \in S | S \preceq S'\}\). Let also \(\text{Pred}(\uparrow S) = \{T | T \rightarrow S' \text{ and } S' \preceq S\}\). By definition of \(\preceq\), \(\text{Pred}(\uparrow S) \subseteq \text{Pred}(S)\). Therefore \(\uparrow \text{Pred}(\uparrow S) \subseteq \uparrow \text{Pred}(S) \subseteq \uparrow X\), where \(X\) is the finite set of Lemma 4.6 that is effectively computable.

Next we discuss the process reachability problem – see Definition 3.1 – in \(\text{Actor}_{ba}^{ro}\). To this aim, we use a simpler version of the (classical) diamond property.

**Proposition 4.8.** Let \((S, \rightarrow)\) be a transition system of a program of \(\text{Actor}_{ba}^{ro}\) and let \(\emptyset \triangleright (P, \emptyset, \varepsilon), S \rightarrow \emptyset \triangleright (P, \emptyset, \varepsilon), S'\) (\(\emptyset\) does not move) and \(\emptyset \triangleright (P, \emptyset, \varepsilon), S' \rightarrow \emptyset \triangleright (P', \emptyset, \varepsilon), S''\) with \(P' \neq P\) (\(\emptyset\) moves). Then there exists \(S'''\) such that \(\emptyset \triangleright (P, \emptyset, \varepsilon), S \rightarrow \emptyset \triangleright (P', \emptyset, \varepsilon), S''' \rightarrow \emptyset \triangleright (P', \emptyset, \varepsilon), S''\).

It is worth noticing that the language \(\text{Actor}\) also owns the more classical diamond property: if in a configuration there are two transitions inferred by two distinct actors, then it is possible to perform them in any order reaching the same configurations up-to bijective renaming. We omit the formalization of this property since it is not needed in the rest of the paper.
We prove that in Actor where the delivery of messages is inexact: it may be enqueued in every actor of the same. Therefore, we decided to apply our arguments to an abstraction of the operational model. Therefore, in order to compute an upper bound to the instance of method bodies, which is the decorated syntax adds a sequence of natural numbers in front of the process of an actor, namely, we use $A \triangleright (\sigma : P, \emptyset, q)$ where $\sigma$ has the following meaning: if $\sigma = \sigma' \cdot n$, then $\sigma'$

Corollary 4.9. The process reachability problem is decidable in Actor$^{ba}$.

Proof. In order to verify whether a configuration $A \triangleright (P', \varphi, q), S$ is reachable with $P'$ equal to $P$, up-to renaming of variables and actor names, we proceed as follows.

First, by Proposition 4.8, it is not restrictive to consider the set of configurations $T$ reachable by completely executing the actor $N$ only. The cardinality of $T$ is bounded by $2^k$, where $k$ is the maximal nesting of $+$ in the main process. If one of the processes in the configurations reached by executing $N$ is equal to $P$, up-to renaming of variables and actor names, then we are done. Otherwise, let $\bar{u}$ be the free variables in the main process. For each of $T = N \triangleright (0, \emptyset, \varepsilon), A_1 \triangleright (0, \varphi_1, q_1), \ldots, A_\ell \triangleright (0, \varphi_\ell, q_\ell)$ in $T$, we check control-state reachability from $T$ to at least one of the states in the following finite set:

\[ \{ N \triangleright (0, \emptyset, \varepsilon), A_1 \triangleright (Q_1[A_1/\text{this}][\bar{z}_1/\bar{y}_1][\bar{U}_1/\bar{x}_1], \varphi_1, \varepsilon), \ldots, A_\ell \triangleright (Q_\ell[A_\ell/\text{this}][\bar{z}_\ell/\bar{y}_\ell][\bar{U}_\ell/\bar{x}_\ell], \varphi_\ell, \varepsilon) \mid \text{for every } 1 \leq i \leq \ell, Q_i \text{ is a suffix of the body of } m_i \text{ in } C_i, \text{ where } A_i \in C_i, \text{ formal parameters and free variables of } m_i \text{ are } \bar{x}_i \text{ and } \bar{y}_i, \bar{U}_i \text{ is a tuple in } \{ A_1, \ldots, A_\ell, \bar{u}, \bar{z}_i \} \text{ (} \bar{z}_i, \ldots, \bar{z}_\ell \text{ are fresh)} \text{ there exists } 1 \leq j \leq \ell \text{ such that } Q_j \text{ is equal to } P \text{ up-to renaming} \} \]

Then the corollary follows by Theorem 4.7.

We conclude this section by recalling that we have already proved the undecidability of termination in programs with unboundedly many actors and read-only fields. Note that if we remove from Actor$^{ba}$ the constraint on bounded actor names then the relation $\preceq$ is no longer a well-quasi-ordering. Consider, for instance, an actor (with empty state) having $i<j$ It is easy to see that, for every $\bar{x}, \bar{z}_i, \ldots, \bar{z}_\ell$ are fresh)

there exists $1 \leq j \leq \ell$ such that $Q_j$ is equal to $P$ up-to renaming

We prove that in Actor$^{al}$ termination and process reachability are decidable, too. As discussed at the end of Section 4, the ordering defined for Actor$^{ba}$ is not appropriate for Actor$^{al}$ because in the latter it is possible to dynamically produce unboundedly many actors. Therefore, in order to compute an upper bound to the instances of method bodies, which is the basic argument for the model of Section 4 to be a well-structured transition system, we need to abstract from the identity of these names – as we have done with variables. However, in case of actor names, the abstractions we have devised all break the delivering of messages. Therefore, we decided to apply our arguments to an abstraction of the operational model where the delivery of messages is inexact: it may be enqueued in every actor of the same class. Yet, this abstract model allows us to derive decidability of termination and process reachability for the original language.

In order to formalize the correspondence between the concrete and the abstract operational semantics, we need to add decorations to processes and transitions at the concrete level. Such decorations are used to keep track of the causal dependencies among processes. The decorated syntax adds a sequence of natural numbers in front of the process of an actor, namely, we use $A \triangleright (\sigma : P, \emptyset, q)$ where $\sigma$ has the following meaning: if $\sigma = \sigma' \cdot n$, then $\sigma'$

5. Decidability results for Actor$^{al}$

We prove that in Actor$^{al}$ termination and process reachability are decidable, too. As discussed at the end of Section 4, the ordering defined for Actor$^{ba}$ is not appropriate for Actor$^{al}$ because in the latter it is possible to dynamically produce unboundedly many actors. Therefore, in order to compute an upper bound to the instances of method bodies, which is the basic argument for the model of Section 4 to be a well-structured transition system, we need to abstract from the identity of these names – as we have done with variables. However, in case of actor names, the abstractions we have devised all break the delivering of messages. Therefore, we decided to apply our arguments to an abstraction of the operational model where the delivery of messages is inexact: it may be enqueued in every actor of the same class. Yet, this abstract model allows us to derive decidability of termination and process reachability for the original language.

In order to formalize the correspondence between the concrete and the abstract operational semantics, we need to add decorations to processes and transitions at the concrete level. Such decorations are used to keep track of the causal dependencies among processes. The decorated syntax adds a sequence of natural numbers in front of the process of an actor, namely, we use $A \triangleright (\sigma : P, \emptyset, q)$ where $\sigma$ has the following meaning: if $\sigma = \sigma' \cdot n$, then $\sigma'$

21
identifies the action of emission of the message that caused the method instantiation from which \( P \) was generated, and \( n \) is a counter indicating that \( P \) is actually generated by the method instantiation after \( n \) steps. Notice that for the main process executed by the actor \( \aleph \) the sequence \( \sigma' \) is empty, and that when a method is instantiated the counter \( n \) is initialized.
with an additional argument – the actor name of the target act or. This additional argument is used when the method body is instantiated. In fact, it replaces the variable this with a fresh variable.

Also under the abstract semantics, the item \( m(\bar{U}, \sigma) \) is enqueued with an additional argument – the actor name of the target actor. This additional argument is used when the method body is instantiated. In fact, it replaces the variable this with a fresh variable.

As an example, consider the task manager specified in Example 2.4. The task manager has the following rules for method invocation and instantiation:

\[
\begin{align*}
  & \text{(invk-a)} \quad \bar{E} \sim_d \bar{U} ; S \quad A, A' \in C \\
  & \text{(invk-s)} \quad A \triangleright (\sigma \cdot n : A'! m(\bar{E}), P, \varnothing, q) \sigma^{n+1}[m(\bar{U}, A')] \quad A \triangleright (\sigma \cdot n + 1 : P, \varnothing, q \cdot m(\bar{U}, \sigma \cdot n + 1, A')), S \\
  & \text{(inst-a)} \quad A \in C \quad C, m(x) = P \quad \bar{y} = \text{free}(P) \setminus \bar{x} \quad \bar{y}' = \text{fresh}(\bar{y}) \\
  & \quad A \triangleright (0, \varnothing, m(\bar{U}, \sigma, A') \cdot q) \xrightarrow{\sigma^0_a} A \triangleright (\sigma \cdot 0 : P[A'/\text{this}]\bar{y}'/\bar{y}[\bar{U}/\bar{x}], \varnothing, q)
\end{align*}
\]

Table 4: Abstract transition rules for method invocations and instantiations

To 0. In order to transfer the sequence from the message emitter to the receiving actor, we add \( \sigma \) at the end of messages. Namely, messages are now denoted with \( m(\bar{U}, \sigma) \). The decorated operational semantics \( S \overset{\alpha}{\rightarrow} S' \) is defined in Table 3, where the label \( \alpha \) can be either a sequence \( \sigma \) or a pair \( \sigma | m(\bar{U}, \sigma) \) where the second element identifies the message issued during the transition. The decorated operational semantics increments the last number of the sequence of a process every time it performs an action, adds to messages the current sequence of the emitter, and use the sequences inside messages to initialize the sequence of the method instantiations (by extending it with 0).

It is trivial to see that the operational semantics in Table 3 and the decorated semantics coincide, in the sense that given one configuration \( S_1 \) of Actor\^1 we have \( S_1 \rightarrow S_2 \) if and only if there exist a label \( \alpha \) and two decorations \( S'_1 \) and \( S'_2 \) of \( S_1 \) and \( S_2 \), respectively, such that \( S'_1 \overset{\alpha}{\rightarrow} S'_2 \).

As discussed at the beginning of this Section, we need a more abstract semantics with inexact message deliveries. This is obtained by changing the operational semantics in Table 3 by decoupling the evaluation of the body of a method from the actor name of that method. Let \( S \overset{\alpha}{\rightarrow} S' \) be the abstract transition relation defined as \( S \overset{\alpha}{\rightarrow} S' \) in Table 3 except the two rules (invk-s and inst) for method invocation and the rule (inst-d) for the instantiation of method bodies, which are replaced by those in Table 4. In the abstract transition relation, a message is added in a queue of an actor nondeterministically selected among those belonging to the same class of the target actor. The item \( m(\bar{U}, \sigma) \) is enqueued with an additional argument – the actor name of the target actor. This additional argument is used when the method body is instantiated. In fact, it replaces the variable this, thus making the execution of a body invariant regardless the actor that actually performs it.

As an example, consider the task manager specified in Actor\^1 in the Example 2.4. Also under the abstract semantics, the two distinct workers are instantiated, but it is possible for two distinct tasks to be delivered to the same worker.

We now introduce few notations:

- Let \( \Omega() \) be a map from “concrete” to “abstract” configurations: given a configuration \( S \), we denote with \( \Omega(S) \) the configuration obtained from \( S \) by replacing each of its actors
A \triangleright (\sigma : P, \emptyset, q) \text{ with } A \triangleright (\sigma : P, \emptyset, q') \text{ where } q' \text{ is obtained from } q \text{ by adding to each method invocation the parameter } A.

- Given a decorated configuration \( S \) and a label \( \alpha \), such that \( \alpha = \sigma \) or \( \alpha = \sigma|m(\tilde{U}, A) \), we use \( S|_{\alpha} \) to denote the process decorated with \( \sigma \) in \( S \): \( S|_{\alpha} = P \) if \( S \) contains the actor \( A \triangleright (\sigma : P, \emptyset, q) \), for some \( A \) and \( q \).

- Let \( \equiv_{\alpha} \) be the following relation on variable renamings (not applied to variables that are free in the main process)
  \[
  \rho \equiv_{\alpha} \rho' = \text{def for every } x, y:
  \begin{align*}
  (i) & \quad \rho(x) = \rho(y) \text{ if and only if } \rho'(x) = \rho'(y) \\
  (ii) & \quad \rho(x) \in C \text{ if and only if } \rho'(x) \in C \\
  (iii) & \quad \rho(x) = \rho'(x) \text{ if } \rho(x) \text{ or } \rho'(x) \text{ is a free variable of the main process}
  \end{align*}
  \]

Differently from the definition of \( \equiv \), \( \equiv_{\alpha} \) does not care of the identity of actor names (it is sufficient that they belong to the same class).

- Let \( \simeq_{\alpha} \) be the relation defined as \( \simeq \) in Section 4 with \( \equiv_{\alpha} \) instead of \( \equiv \). We extend it to messages containing sequences and actor names as follows: \( m(\tilde{U}, \sigma, A) \simeq_{\alpha} m(\tilde{U}', \sigma', A') \) if \( m(\tilde{U}) \simeq_{\alpha} m(\tilde{U}') \), \( \sigma = \sigma' \) and there exists \( C \) such that \( A, A' \in C \). We extend it also to labels: \( \sigma \simeq_{\alpha} \sigma \) and \( \sigma|m(\tilde{U}, A) \simeq_{\alpha} \sigma|m(\tilde{U}', A') \) if \( m(\tilde{U}) \simeq_{\alpha} m(\tilde{U}') \) and there exists \( C \) such that \( A, A' \in C \).

The following Proposition formalizes the correspondence between the \( \alpha \rightarrow \) and \( \alpha_{\alpha} \rightarrow \) transitions: 1. all \( \alpha \rightarrow \) transitions are present also in \( \alpha_{\alpha} \rightarrow \) (up-to application of the abstraction function \( \Omega() \) to configurations) and 2. all the abstract configurations \( S_0 \rightarrow_{\alpha_1} \ldots \rightarrow_{\alpha_3} \rightarrow_{\alpha_3} S_n \) have a corresponding concrete computation \( S'_0 \rightarrow_{\alpha'_1} \ldots \rightarrow_{\alpha'_m} S'_m \) in which they can be embedded.

**Proposition 5.1.** Let \( S \) and \( S_0 \) be a decorated configuration and an initial decorated configuration \( \pi \triangleright (0 : P, \emptyset, \varepsilon) \) of Actort, respectively.

1. If \( S \rightarrow_{\alpha} S' \) then \( \Omega(S) \rightarrow_{\alpha_{\alpha}} \Omega(S') \);
2. if \( S_0 \rightarrow_{\alpha_1} \ldots \rightarrow_{\alpha_3} S_n \) then there exists a computation \( S'_0 \rightarrow_{\alpha'_1} \ldots \rightarrow_{\alpha'_i} S'_i \rightarrow_{\alpha'_m} S'_m \) and an injection \( I \) such that, for all \( 1 \leq i \leq n \), we have \( \alpha_i \simeq_{\alpha} \alpha'_i \) and \( S_i|_{\alpha_i} \simeq_{\alpha} S'_i|_{\alpha'_i} \).

**Proof.** The first item trivially holds because the new rules used in the definition of \( \alpha_{\alpha} \rightarrow \) are (strictly) more general than the corresponding rules used in the definition of \( \alpha \rightarrow \).

The second item is proved by induction on the length of the computation \( S_0 \rightarrow_{\alpha_1} \ldots \rightarrow_{\alpha_3} \rightarrow_{\alpha_3} S_n \).

If \( n = 1 \) then \( S_0 \rightarrow_{\alpha_1} S_1 \) with \( \alpha_1 = 1 \) or \( \alpha_1 = 1|m(\tilde{U}, A) \) and \( S_1|_{\alpha_1} = P' \), where \( P' \) is an immediate derivative of the main process \( P \). It is trivial to see that the same transition is present in the concrete decorated semantics: namely, \( \pi \triangleright (0 : P, \emptyset, \varepsilon) \rightarrow_{\alpha'_1} S'_1 \) with \( \alpha_1 \simeq_{\alpha} \alpha'_1 \) and \( S'_1|_{\alpha'_1} \simeq_{\alpha} P' \).

If \( n > 1 \) we consider \( S_0 \rightarrow_{\alpha_1} \ldots \rightarrow_{\alpha_3} S_n \rightarrow_{\alpha_3} S_{n+1} \). The inductive hypothesis guarantees the existence of the concrete computation \( S'_0 \rightarrow_{\alpha'_1} \ldots \rightarrow_{\alpha'_i} S'_i \rightarrow_{\alpha'_i} S'_m \) and of the injection \( I \) such that, for all \( 1 \leq i \leq n \), we have \( \alpha_i \simeq_{\alpha} \alpha'_i \) and \( S_i|_{\alpha_i} \simeq_{\alpha} S'_i|_{\alpha'_i} \). We now proceed by case analysis on the last number of the sequence in \( \alpha_{n+1} \).

If the number is 0, then \( \alpha_{n+1} = \sigma \cdot 0 \) and the transition is obtained by applying rule (\text{INST}_a) on a message \( m(\tilde{U}, \sigma, A') \). The presence of this message in one of the queues in
 guarantees the existence of 1 \leq j \leq n such that \alpha_j = \sigma|m(U, A'). In the concrete computation we have \alpha'I(j) \equiv_3 \sigma|m(U, A'). This means that the same message (up-to \equiv_3) is in the queue of an actor A'' such that A', A'' \in C in the concrete state S'I(j). We have two subcases: either such method invocation is instantiated by the actor A'' during the concrete computation S'_0 \xrightarrow{\alpha'_1} \ldots \xrightarrow{\alpha'_{m}} S'_m or not. In the first case, there exists I(j) < l \leq m such that \alpha'_l = \sigma \cdot 0 that instantiates the method. The thesis is proved simply by extending the injection I with I(n + 1) = l and observing that S_{n+1}|_{\sigma \cdot 0} \equiv_3 S'_{n+1}|_{\sigma \cdot 0}. If the method invocation was not instantiated, it is in the queue of the actor A'' in the configuration S'_m.

It is sufficient to apply the same reasoning on an extension of the concrete computation that consumes the messages in front of the method invocation with sequence \sigma and that finally instantiates it. Such extension exists because processes are finite and non-blocking.

If the number is not 0, we discuss only the case in which \alpha_{n+1} = \sigma \cdot k (with k > 0) because the case \sigma = \sigma \cdot k|m(U, A) is treated similarly. In the computation S_0 \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_k} S_{n+1} is guaranteed the presence of a label containing \sigma \cdot k - 1, i.e. there exists 1 \leq j \leq n such that the label \alpha_j contains \sigma \cdot k - 1. The process S_j|_{\alpha_j} is the process that has just performed the action labeled with \sigma \cdot k - 1 and that performs the action in the transition S_n \xrightarrow{\alpha_{n+1}} S_{n+1} because \alpha_{n+1} = \sigma \cdot k. By inductive hypothesis S_j|_{\alpha_j} \equiv_3 S'_j|_{\alpha'_j} hence a process ready to execute an action labeled with the sequence \sigma \cdot k occurs also in the concrete state S'_j. We now consider two subcases.

- There exists no label \alpha'_l containing \sigma \cdot k. In this case the process S'_j|_{\alpha'_j} still occurs in S'_m. Hence it is possible to extend the computation S'_0 \xrightarrow{\alpha'_1} \ldots \xrightarrow{\alpha'_{m}} S'_m in such a way that S'_{m+1}|_{\sigma \cdot k} \equiv_3 S_{n+1}|_{\sigma \cdot k}. The thesis is proved simply by extending the injection I with I(n + 1) = m + 1.

- There exists I(j) < l \leq m such that \alpha'_l contains \sigma \cdot k. In this case it is not guaranteed that S_j|_{\alpha'_j} \equiv_3 S_{n+1}|_{\sigma \cdot k}, due to nondeterminism. For this reason we construct from S'_0 \xrightarrow{\alpha'_1} \ldots \xrightarrow{\alpha'_{m}} S'_m another concrete computation that satisfies our thesis. The first transformation that we apply to S'_0 \xrightarrow{\alpha'_1} \ldots \xrightarrow{\alpha'_{m}} S'_m consists of the cancellation of the transition \alpha'_l and of all the other transitions that depends on it. Namely, there are two kinds of transitions that must be removed: (i) those labeled with a sequence having a prefix \sigma \cdot r such that r \geq k and (ii) those causally dependent on the instantiation of messages that are in S'_l inside the queue of the actor containing the process decorated with \sigma \cdot k. Let S'_0 \xrightarrow{\alpha'_1} \ldots \xrightarrow{\alpha'_{l-1}} S'_{l-1} \xrightarrow{\alpha'_l} S''_{l-1} \xrightarrow{\alpha''_l} S''_{l} \ldots \xrightarrow{\alpha''_{m}} S''_m be the concrete computation obtained after this elimination of transitions. We now extend such computation by letting the process labeled with \sigma \cdot k - 1 to execute the expected action labeled with \sigma \cdot k. Namely, we add the transition S''_s \xrightarrow{\sigma \cdot k} S''_{s+1} such that S''_{s+1}|_{\sigma \cdot k} \equiv_3 S_{n+1}|_{\sigma \cdot k}. Then, we extend the computation by performing at least all the transitions removed for the reason (ii) above. This extension exists because all processes are finite and nonblocking and because the considered transitions causally depend on messages that are in S''_l inside the queue of the actor containing the process decorated with \sigma \cdot k. Let S'_0 \xrightarrow{\alpha'_1} \ldots \xrightarrow{\alpha'_{l-1}} S'_{l-1} \xrightarrow{\alpha'_l} S''_{l-1} \xrightarrow{\alpha''_l} S''_{l} \ldots \xrightarrow{\alpha''_{m}} S''_m \xrightarrow{\alpha''_{m+1}} \ldots \xrightarrow{\alpha''_{m+\ell}} S''_m be the obtained computation. The thesis is proved by considering this last concrete computation, a rearrangement of the injection I that maps to their new positions the transitions in its codomain that belong to the group (ii), and by extending it with I(n + 1) = l.
As a direct consequence we have that the abstract semantics preserves both termination and control-state reachability. It remains to prove that termination and process reachability is decidable for the abstract semantics. To this aim, we consider a transition system $\rightarrow_a$ obtained by removing the labels from the transitions $\alpha\gamma_a$. On this transition system we define $\preceq_a$ as a variant of the ordering $\preceq$ defined in the previous section in such a way that $(\mathcal{S}, \rightarrow_a, \preceq_a)$ turns out to be a well-structured transition system (for configurations of stateless programs). Let:

- Let $\preceq_a$ be the following relation on message queues:

\[
\text{there exist } i_1 < i_2 < \cdots < i_k \leq h \text{ s.t. for } j \in 1..k, \text{ } m_j(U_j, \sigma_j, A_j) \preceq_a n_i(V_i, \sigma_i', A_i')
\]

\[
m_1(U_1, \sigma_1, A_1) \ldots m_k(U_k, \sigma_k, A_k) \preceq_a n_1(V_1, \sigma_1', A_1') \ldots n_k(V_k, \sigma_k', A_k')
\]

- Let $\preceq_a$ be the ordering:

\[
A_i, A_j \in \mathcal{C}, \text{ } P_i \preceq_a P_j \text{ and } q_i \preceq_a q_j \text{ for } i \in 1..\ell, 1 \leq j_1 < j_2 < \cdots < j_\ell \leq \kappa
\]

\[
A_1 \triangleright (\sigma_1 : P_1, \emptyset, q_1), \ldots, A_\ell \triangleright (\sigma_\ell : P_\ell, \emptyset, q_\ell) \preceq_a A'_1 \triangleright (\sigma'_1 : P'_1, \emptyset, q'_1), \ldots, A'_\kappa \triangleright (\sigma'_\kappa : P'_\kappa, \emptyset, q'_\kappa)
\]

Next, we observe that Lemma 4.3 can be adapted to the case of unbounded actors by using $\preceq_a$ instead of $\preceq$. Namely, let $T$ be either a process or a method invocation $m(U_1, \ldots, U_n, \sigma, A)$ of a stateless program and let $T = \{T_{\rho_1}, T_{\rho_2}, T_{\rho_3}, \ldots\}$ be such that $i \neq j$ implies $T_{\rho_i} \preceq_a T_{\rho_j}$. Proceeding as in the proof of Lemma 4.3, we prove that $T$ is finite.

**Theorem 5.2.** Given a stateless program $S$ we have that $(S, \rightarrow_a, \preceq_a)$ is a well-structured transition system.

**Proof.** The proof is as in Theorem 4.4 with few differences that are discussed below.

In part (1) the unique difference is in the last part where the coordinatewise order $\sqsubseteq^\ell$ on sequences (of length $\ell$) of queues of terms is used. As we now consider configurations with an unbounded number of actors, instead of configurations with a bounded number $\ell$ of actors, we need to resort to the embedding $\sqsubseteq_a$ defined as follows:

\[
q_i \preceq_a q_j' \text{ for } i \in 1..\ell, 1 \leq j_1 < j_2 < \cdots < j_\ell \leq \kappa
\]

\[
(q_1, \ldots, q_\ell) \sqsubseteq_a (q'_1, \ldots, q'_\ell)
\]

The final contradiction of part (1) is now reached by observing that by Highman’s lemma, also $\sqsubseteq_a$ is a well-quasi-ordering, as a consequence of the well-quasi-ordering $\preceq_a$.

In part (2) the unique difference is for the monotonicity transitions due to rules (invk-A) and (invk-sa). The greater configuration is guaranteed to have a program ready to perform a corresponding method invocation, but this could be addressed to a different actor. In fact, the ordering $\preceq_a$ does not preserve actor names as it was for $\preceq$ in the proof of Theorem 4.4. But $\preceq_a$ preserves at least actor classes. As the abstract transition system $\rightarrow_a$ allows a term $m(U, \sigma, A)$ to be introduced in the queue of any of the actor belonging to the same class of $A$, the method invocation executed by the greater configuration can be introduced in the queue of the actor corresponding to the target of the method invocation executed by the smaller configuration.

$\square$

26
In the light of the results on well-structured transition systems recalled at the beginning of Section 4, this theorem proves the decidability of termination for the abstract semantics. To prove the decidability of process reachability we need to prove that a finite basis for predecessors is effectively computable.

**Lemma 5.3.** Let $(S, \rightarrow, \preceq_a)$ be a well-structured transition system of a program in Actor$^{1}$, and let $S \in S$. Then there is a finite set $\mathcal{X}$ such that, for every $S' \preceq_a S$ and $S'' \in \text{Pred}(S')$, there is $T \in \mathcal{X}$ with $T \preceq_a S''$. $\mathcal{X}$ can be effectively computed.

**Proof.** The computation of $\mathcal{X}$ must extend the construction presented in the proof of Lemma 4.6 in two ways.

The first extension derives from the fact that, differently from the ordering $\preceq$ considered in Lemma 4.6, if $S \preceq_a S'$ it could be possible for $S'$ to have strictly more actors than $S$. In these cases, it is possible that the predecessor differs from its successor $S'$ for actors that are not present in $S$. We can cope with this problem by applying the procedure described in the proof of Lemma 4.6 not only to the configuration $S$, but to all the configurations that can be obtained by extending $S$ with one or two additional actors belonging to one of the finite classes of the considered program. In fact, at most two actors are modified by one transition. Each of these additional actors executes a process obtained by applying a renaming $\rho$ to a suffix of one of the method definitions of the corresponding class. As observed above, there are finitely many processes that can be obtained up-to $\simeq_a$. Finally, the additional actors have a queue including at most one method invocation (in fact, at most one message can be consumed in one transition). Also in this case, by considering the method definitions of the actor class, it is easy to see that there are finitely many different method invocations up-to $\simeq_a$.

The second extension is trivial and deals with the fact that in the abstract semantics a method invocations can be introduced in the queue of any of the actors belonging to the same class of the expected target actor. So the procedure of the proof of Lemma 4.6 for computing $\mathcal{X}$ must be extended to consider also this kind of transitions.

From Theorem 5.2, this last Lemma and the results on well-structured transition systems recalled at the beginning of Section 4, we can conclude that also control-state reachability, besides termination as already commented above, is decidable for the abstract semantics. From Proposition 5.1 we have already concluded that the abstract semantics preserves termination and control-state reachability w.r.t. the concrete semantics. Hence, we have that termination and control-state reachability are decidable for stateless actor program.

The decidability of control-state reachability entails the decidability of process reachability. In fact, given a process $P$, the reachability of a configuration $A \triangleright (P', \varphi, q), S$ with $P'$ equal to $P$ up-to renaming of variables and actor names can be solved in the abstract transition system simply by checking the control-state reachability of at least one of the following states. Let $C_1, \ldots, C_n$ be the actor classes of the considered actor system and let $A_1, \ldots, A_n$ be such that $A_i \in C_i$. We consider the following finite set of states:

$$S = \{ A_i \triangleright (Q_i, \varnothing, \varepsilon) \mid 1 \leq i \leq n, \text{ } Q_i \text{ is a suffix of a method definition in the class } C_i \text{ and it is equal to } P \text{ up-to renaming} \}$$

27
6. Conclusions

To the best of our knowledge this paper contains a first systematic study on the computational power of Actor-based languages. We have focussed on the pure asynchronous FIFO queueing and dequeuing of method calls between actors in the context of a nominal calculus which features the dynamic creation of variable names that can be passed around. The results proved in this paper can be summarized as follows:

• we identified two small but Turing powerful fragments of our Actor language: the fragment in which only boundedly many actors can be created, and the fragment in which fields cannot be updated;
• we have proved that the fragment obtained as intersection of the two above sublanguages is not Turing complete, as properties like termination and control-state reachability turn out to be decidable;
• if Actors are stateless, the language has decidable termination and control-state reachability even if we consider unboundedly many Actors.

We conclude by mentioning relevant lines for future research. Recent work have identified more expressive Actor interaction mechanisms based an asynchronous method calls implemented by means of the so-called future variables \[14\]: we plan to investigate the impact of this Actor-based synchronization mechanism on our (un)decidability results. We also plan to extend our study of expressiveness to primitives like the release statements in \[7\]. These statements support the so-called cooperative scheduling of method invocations: method executions can release the control of the Actor in such a way that other method executions can be instantiated or resumed.

References


