Decidability Problems for Actor Systems

F.S. de Boer[1], M. M. Jaghoori[1], C. Laneve[2], and G. Zavattaro[2]

[1] CWI, Amsterdam, The Netherlands
[2] University of Bologna, INRIA Focus Research Team, Bologna, Italy
{f.s.de.boer,jaghoori}@cwi.nl, {laneve,zavattar}@cs.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 relevant 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. Such decidability result holds in actors with states when the number of actors is finite and the state is read-only.

1 Introduction

Since their introduction in [12], 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 [16,17], wireless sensor networks [7], multi-core programming [15], and web services [5,6]. The underlying concurrent model of actor languages also forms the basis of the programming languages Erlang [4] and Scala [11] that have recently gained in popularity, in part due to their support for scalable concurrency.

In actor languages [2,12], 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 relevant 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 control-state 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 systems consisting of finitely many actors with read- only fields as a well-structured transition system [9] – henceforth the decidability of termination, control-state maintainability and the inevitability problem. Further, we show that a termination and process reachability preserving abstraction of systems of unboundedly many stateless actors (i.e., actors without fields) 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 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 exist a vast body of related work on decidability of infinite-state systems (see [1]) that however does not address the specific characteristics of the pure asynchronous mechanism of queued and dequeued method calls in actor-based languages. 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 [18]. In [21] 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. Recent work on actor-based language focusses on deadlock analysis:

In [10], 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 [8] for detecting deadlock in an actor-like subset of Creol [14] is based on suitable over-approximations.

2 The language `Actor`

Four disjoint infinite sets of names are used: *actor classes*, ranged over $\mathtt{C}$, $\mathtt{D}$, $\cdots$, *method names*, ranged over m, m', n, n', $\cdots$, *field names*, ranged over $\mathtt{f}$, $\mathtt{g}$, $\cdots$, and *variables*, ranged over x, y, z, $\cdots$. For notational convenience, we use $\widetilde{x}$ 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

$$\begin{array}{lcl} E & ::= & \mathtt{f} \mid x \mid \mathtt{new}\ \mathtt{C}(\widetilde{E}) \\ P & ::= & \mathtt{0} \mid (\mathtt{f} \hookleftarrow E).P \mid \mathtt{let}\ x = E\ \mathtt{in}\ P \mid x\,!\,m(\widetilde{E}).P \mid \\ & & [E = E]P;P \mid P + P \end{array}$$

An expression E either denotes a value stored in a field $\mathtt{f}$, or a variable x, or a new actor of class $\mathtt{C}$ with fields initialized to the values of $\widetilde{E}$. A process may be either the terminated one $\mathtt{0}$, or a field update $(\mathtt{f} \hookleftarrow E).P$, or the assignment $\mathtt{let}\ x = E\ \mathtt{in}\ P$ of a value to a variable, or an invocation $x\,!\,m(\widetilde{E}).P$ of a method m of the actor x with arguments $\widetilde{E}$, 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 $\mathtt{0}$ in processes; for example $(\mathtt{f} \hookleftarrow x).\mathtt{0}$ will be always shortened into $(\mathtt{f} \hookleftarrow x)$. We will also shorten $[E = E']P;\mathtt{0}$ into $[E = E']P$.

The operation $\mathtt{let}\ x = E\ \mathtt{in}\ P$ is a binder of the occurrences of the variable x in the process P that are not already bound by a nested `let` operation of x; the occurrences of x in E are *free*. Let *free*(P) be the set of variables of P that are not bound. As usual, the substitution operation $P[{}^{y}/_{x}]$ returns the process P where the free occurrences of x are replaced by y.

A *program* is a *main process* P and a finite set of *actor class definitions* $\mathtt{C}.m(\widetilde{x}) = P_{\mathtt{C},m}$, where $P_{\mathtt{C},m}$ may contain the special variable *this* (which can be seen as an implicit formal parameter of each method). In the following we restrict to programs that are

1. *unambiguous*, namely, every pair $\mathtt{C}$, m has at most one definition;
2. *correct*, namely, let *fields*$(\cdot)$ be a map that associates a tuple of field names to every actor class. Then, (i) in every expression $\mathtt{new}\ \mathtt{C}(\widetilde{E})$, the length of the tuples $\widetilde{E}$ and *fields*$(\mathtt{C})$ are the same; (ii) in every definition $\mathtt{C}.m(\widetilde{x}) = P_{\mathtt{C},m}$, the field names occurring in $P_{\mathtt{C},m}$ are in the tuple *fields*$(\mathtt{C})$.

In this paper, we abstract from types and type-correctness because we are only interested in expressive power issues. However, it is straightforward to equip the above language with a type discipline.

Example 1. As a second example we model the OpenID authentication protocol [20]. 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, 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:

$$\begin{aligned} \texttt{C}.\mathit{init}(\mathit{server}) &= \mathit{server}\,!\,\mathit{login}(\mathit{this}) \\ \texttt{C}.\mathit{token}(\mathit{authServer}, \mathit{clientToken}) &= \mathit{authServer}\,!\,\mathit{authenticate}(\texttt{u}, \texttt{p}, \mathit{clientToken}) \end{aligned}$$

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

$$\begin{aligned} \texttt{S}.\mathit{init}(\mathit{authServer}) &= (\texttt{auth} \leftarrowtail \mathit{authServer}) \\ \texttt{S}.\mathit{login}(\mathit{client}) &= \texttt{auth}\,!\,\mathit{open}(\mathit{self}, \mathit{clientToken}, \mathit{authToken})\,. \\ &\quad\ \mathit{client}\,!\,\mathit{token}(\texttt{auth}, \mathit{authToken}) \\ \texttt{S}.\mathit{succeeds}(\mathit{authToken}) &= \ldots \\ \texttt{S}.\mathit{fails}(\mathit{authToken}) &= \ldots \end{aligned}$$

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

$$\begin{aligned} \texttt{A}.\mathit{open}(\mathit{server}, \mathit{cToken}, \mathit{aToken}) &= (\texttt{server} \leftarrowtail \mathit{server})\,.\,(\texttt{ctoken} \leftarrowtail \mathit{cToken})\,. \\ &\quad\ (\texttt{atoken} \leftarrowtail \mathit{aToken}) \\ \texttt{A}.\mathit{authenticate}(u, p, \mathit{cToken}) &= [\texttt{ctoken} = \mathit{cToken}]\texttt{server}\,!\,\mathit{succeeds}(\texttt{atoken})\texttt{;} \\ &\quad\ \texttt{server}\,!\,\mathit{fails}(\texttt{atoken}) \end{aligned}$$

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

$$\texttt{let}\ \mathit{client} = \texttt{new C in}\ \big(\texttt{let}\ \mathit{server} = \texttt{new S in}\ \big(\texttt{let}\ \mathit{authProvider} = \texttt{new A in}\ \mathit{client}\,!\,\mathit{init}(\mathit{server})\,.\,\mathit{server}\,!\,\mathit{init}(\mathit{authProvider})\big)\big)$$

The operational semantics. The operational semantics of the language `Actor` will use an infinite set of *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 $A \in \texttt{C}$ to say that A belongs to the partition of `C`. In the following, the (*run-time*) expressions will also include actor names and, with an abuse of notation, this estended set of expressions will be ranged over by E. The

set of terms that are variables or actor names, called *values*, will be addressed by U, V, $\cdots$.

The semantics is defined in terms of a *transition relation* $\mathtt{S} \longrightarrow \mathtt{S}'$, where $\mathtt{S}$, $\mathtt{S}'$, called *configurations*, are sets of terms $A \triangleright (P, \varphi, q)$ with A being an actor name, φ, the *state* of A, being a map from *fields*($\mathtt{C}$) to values, where $A \in \mathtt{C}$, and q being a queue of terms $m(\widetilde{U})$. The empty queue will be denoted with ε. Configurations contain at most one $A \triangleright (P, \varphi, q)$ for each actor name A.

The operational semantics of `Actor` is defined in Table 1, where the *evaluation function* $E \overset{\varphi}{\rightsquigarrow} U \; ; \; \mathtt{S}$ is used. This function takes an expression E and a store φ and returns a value U and a possibly empty configuration $\mathtt{S}$ of terms $A \triangleright (\mathtt{0}, \varphi, \varepsilon)$. These terms represent actors created during the evaluation – the names A are *fresh* – and φ 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. The same auxiliary function is used in rule (INST) on a tuple of variables. In this case it returns a tuple of the same length of variables that are fresh. For notational convenience, we always omit the standard curly brackets in the set notation and we use "," both to separate elements inside sequences and for set union (the actual meaning is made clear by the context).

Given a program, with main process P, the initial configuration is $\aleph \triangleright (P, \varnothing, \varepsilon)$, 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 *not finitely branching* because of the choice of actor names (in the evaluation of `new C`) and the choice of fresh variables (in the instantiation of the bodies of methods). For example, if $\mathtt{C}.m() = [x = x]P$ then $A \triangleright (\mathtt{0}, \varnothing, m()) \longrightarrow A \triangleright ([z = z]P, \varnothing, m())$ for every z. Additionally, every configuration $A \triangleright ([z = z]P, \varnothing, m())$ transits to $A \triangleright (P, \varnothing, m())$. Said otherwise, the sets $\mathit{Succ}(\mathtt{S}) = \{\mathtt{S}' \in \mathcal{S} \mid \mathtt{S} \longrightarrow \mathtt{S}'\}$, called the *successor configurations* of $\mathtt{S}$, and $\mathit{Pred}(\mathtt{S}) = \{\mathtt{S}' \in \mathcal{S} \mid \mathtt{S}' \longrightarrow \mathtt{S}\}$, called the *predecessor configurations* of $\mathtt{S}$, are not finite, in general.

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

- $\mathtt{Actor}_{\mathtt{ba}}$ 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).
- $\mathtt{Actor}^{\mathtt{ro}}$ is the sublanguage without the field update operation ($\mathtt{f} \hookleftarrow E$) (fields are read-only as they cannot be modified after the initialization).
- $\mathtt{Actor}^{\mathtt{ro}}_{\mathtt{ba}}$ is the intersection of $\mathtt{Actor}_{\mathtt{ba}}$ and $\mathtt{Actor}^{\mathtt{ro}}$.
- $\mathtt{Actor}^{\mathtt{sl}}$ is the sublanguage with classes without fields (objects are stateless).

The *evaluation relation* $E \overset{\varphi}{\rightsquigarrow} U \ ; \ \mathtt{S}$:

$$U \overset{\varphi}{\rightsquigarrow} U \ ; \ \varnothing \qquad \mathtt{f} \overset{\varphi}{\rightsquigarrow} \varphi(\mathtt{f}) \ ; \ \varnothing \qquad \frac{\widetilde{E} \overset{\varphi}{\rightsquigarrow} \widetilde{U} \ ; \ \mathtt{S} \quad \widetilde{\mathtt{f}} = \mathit{fields}(\mathtt{C}) \quad A = \mathit{fresh}(\mathtt{C})}{\mathtt{new}\ \mathtt{C}(\widetilde{E}) \overset{\varphi}{\rightsquigarrow} A \ ; \ A \triangleright (\mathtt{0}, [\widetilde{\mathtt{f}} \mapsto \widetilde{U}], \varepsilon), \mathtt{S}}$$

$$\frac{E_i \overset{\varphi}{\rightsquigarrow} U_i \ ; \ \mathtt{S}_i, \quad \text{for} \quad i \in 1..n}{E_1, \cdots, E_n \overset{\varphi}{\rightsquigarrow} U_1, \cdots, U_n \ ; \ \mathtt{S}_1, \cdots, \mathtt{S}_n}$$

The *transition relation* $\mathtt{S} \longrightarrow \mathtt{S}'$:

(UPD)
$$\frac{E \overset{\varphi}{\rightsquigarrow} U \ ; \ \mathtt{S}}{A \triangleright ((\mathtt{f} \leftarrow E).P, \varphi, q) \longrightarrow A \triangleright (P, \varphi[\mathtt{f} \leftarrow U], q), \mathtt{S}}$$

(LET)
$$\frac{E \overset{\varphi}{\rightsquigarrow} U \ ; \ \mathtt{S}}{A \triangleright (\mathtt{let}\ x = E\ \mathtt{in}\ P, \varphi, q) \longrightarrow A \triangleright (P[^{U}/_{x}], \varphi, q), \mathtt{S}}$$

(INVK-S)
$$\frac{\widetilde{E} \overset{\varphi}{\rightsquigarrow} \widetilde{U} \ ; \ \mathtt{S}}{A \triangleright (A!m(\widetilde{E}).P, \varphi, q) \longrightarrow A \triangleright (P, \varphi, q \cdot m(\widetilde{U})), \mathtt{S}}$$

(INVK)
$$\frac{\widetilde{E} \overset{\varphi}{\rightsquigarrow} \widetilde{U} \ ; \ \mathtt{S}}{A \triangleright (A'!m(\widetilde{E}).P, \varphi, q), A' \triangleright (P', \varphi', q') \longrightarrow A \triangleright (P, \varphi, q), A' \triangleright (P', \varphi', q' \cdot m(\widetilde{U})), \mathtt{S}}$$

(INST)
$$\frac{A \in \mathtt{C} \quad \mathtt{C}.m(\widetilde{x}) = P \quad \widetilde{y} = \mathit{free}(P) \setminus \widetilde{x} \quad \widetilde{y'} = \mathit{fresh}(\widetilde{y})}{A \triangleright (\mathtt{0}, \varphi, m(\widetilde{U}) \cdot q) \longrightarrow A \triangleright (P[^{A}/_{\mathit{this}}][^{\widetilde{y'}}/_{\widetilde{y}}][^{\widetilde{U}}/_{\widetilde{x}}], \varphi, q)}$$

(MATCH)
$$\frac{E, E' \overset{\varphi}{\rightsquigarrow} U, U \ ; \ \mathtt{S}}{A \triangleright ([E = E']P; Q, \varphi, q) \longrightarrow A \triangleright (P, \varphi, q), \mathtt{S}}$$

(MMATCH)
$$\frac{E, E' \overset{\varphi}{\rightsquigarrow} U, V \ ; \ \mathtt{S} \quad U \neq V}{A \triangleright ([E = E']P; Q, \varphi, q) \longrightarrow A \triangleright (Q, \varphi, q), \mathtt{S}}$$

(PLUS-L)
$$\frac{A \triangleright (P, q), \mathtt{S} \longrightarrow \mathtt{S}'}{A \triangleright (P + Q, q), \mathtt{S} \longrightarrow \mathtt{S}'}$$

(PLUS-R)
$$\frac{A \triangleright (P, q), \mathtt{S} \longrightarrow \mathtt{S}'}{A \triangleright (Q + P, q), \mathtt{S} \longrightarrow \mathtt{S}'}$$

(CONTEXT)
$$\frac{\mathtt{S} \longrightarrow \mathtt{S}'}{\mathtt{S}, \mathtt{S}'' \longrightarrow \mathtt{S}', \mathtt{S}''}$$

Table 1. The operational semantics of the language `Actor`

3 Undecidability results for $\texttt{Actor}_{\texttt{ba}}$ and $\texttt{Actor}^{\texttt{ro}}$

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

Definition 1. *An actor program* terminates *if it has no infinite computation; it* reaches a process P *if it has a computation 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.*

Actually, in order to convey a stronger result, we consider two sublanguages: (i) where methods never use the `new` expression – actors may be only created by the main process –, therefore the actor names are bounded, and (ii) where fields cannot be updated – the fields are read-only after the initialization.

We will use a reduction technique of the halting and reachability problems in 2 Counter Machines (2CMs) [19] – a well-known Turing-complete model – to that of our actor model. A 2CM is a machine with *two registers* R_1 and R_2 holding arbitrary large natural numbers and a *program* P consisting of a finite sequence of numbered instructions of the following type:

- $j : \mathsf{Inc}(R_i)$: increments R_i and goes to the instruction $j + 1$;
- $j : \mathsf{DecJump}(R_i, l)$: 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;
- $j : \mathsf{Halt}$: 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. In the sequel, we consider 2CMs in which registers are initially set to zero.

3.1 The language $\texttt{Actor}_{\texttt{ba}}$

We encode the value n stored in a register as n messages (of the same type) that are enqueued in an actor – see Figure 1. Namely, let R_1 and R_2 be two actors of class `R` and let the number of messages *item* in R_1 and R_2 be their value. The instruction Inc is implemented by inserting one *item* message in the queue of the corresponding register. In our formalism, this is done by invoking the method *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 $\mathsf{DecJump}$ operation, as described next.

In case (i), to avoid an infinite sequence of *item* dequeues and enqueues, the queue of the registers is initialized with a *bottom* message. The execution of *bottom* updates the field `loop` to *tt* (it is initialized to *ff*). This field is reset to *ff* when either *inc*, or *decjump*, or *checkzero* is executed. If the *bottom* method is executed with `loop` set to *tt*, the register becomes inactive by setting another

$$
\begin{array}{rl}
\texttt{R} & \texttt{// R has fields dec, ctr, loop and stop} \\[2ex]
& \texttt{R}.\mathit{item}(\mathit{tt},\mathit{ff}) = [\texttt{stop} = \mathit{ff}]\big([\texttt{dec} = \mathit{ff}]\mathit{this}\,!\,\mathit{item}(\mathit{tt},\mathit{ff});(\texttt{dec} \leftarrow \mathit{ff})\big) \\[2ex]
& \texttt{R}.\mathit{inc}(\mathit{pc},\mathit{tt},\mathit{ff}) = [\texttt{stop} = \mathit{ff}](\texttt{loop} \leftarrow \mathit{ff})\,.\,\mathit{this}\,!\,\mathit{item}(\mathit{tt},\mathit{ff})\,.\,\texttt{ctr}\,!\,\mathit{run}(\mathit{pc},\mathit{tt},\mathit{ff}) \\[2ex]
& \texttt{R}.\mathit{decjump}(\mathit{pc},\mathit{pc}',\mathit{tt},\mathit{ff}) = [\texttt{stop} = \mathit{ff}](\texttt{loop} \leftarrow \mathit{ff})\,.\,(\texttt{dec} \leftarrow \mathit{tt})\,.\,\mathit{this}\,!\,\mathit{checkzero}(\mathit{pc},\mathit{pc}',\mathit{tt},\mathit{ff}) \\[2ex]
& \texttt{R}.\mathit{checkzero}(\mathit{pc},\mathit{pc}',\mathit{tt},\mathit{ff}) = [\texttt{stop} = \mathit{ff}](\texttt{loop} \leftarrow \mathit{ff})\,. \\
& \qquad\qquad \big([\texttt{dec} = \mathit{tt}]\texttt{ctr}\,!\,\mathit{run}(\mathit{pc}',\mathit{tt},\mathit{ff});\texttt{ctr}\,!\,\mathit{run}(\mathit{pc},\mathit{tt},\mathit{ff})\big) \\[2ex]
& \texttt{R}.\mathit{init}(\mathit{tt},\mathit{ff},\mathit{Ctrl}) = (\texttt{dec} \leftarrow \mathit{ff})\,.\,(\texttt{ctr} \leftarrow \mathit{Ctrl})\,.\,(\texttt{loop} \leftarrow \mathit{ff})\,.\,(\texttt{stop} \leftarrow \mathit{ff})\,. \\
& \qquad\qquad \mathit{this}\,!\,\mathit{bottom}(\mathit{tt},\mathit{ff}) \\[2ex]
& \texttt{R}.\mathit{bottom}(\mathit{tt},\mathit{ff}) = [\texttt{loop} = \mathit{ff}](\texttt{loop} \leftarrow \mathit{tt}).\mathit{this}\,!\,\mathit{bottom}(\mathit{tt},\mathit{ff});(\texttt{stop} \leftarrow \mathit{tt}) \\[2ex]
\texttt{Ctrl} & \texttt{// Ctrl has fields } \texttt{stm}_1\texttt{, } \cdots\texttt{, } \texttt{stm}_n \texttt{ and } \texttt{r}_1 \texttt{ and } \texttt{r}_2 \\[2ex]
& \texttt{Ctrl}.\mathit{run}(\mathit{pc},\mathit{tt},\mathit{ff}) = [\mathit{pc} = \texttt{stm}_1][\![\mathit{Instruction\_1}]\!]_{1,\mathit{tt},\mathit{ff}} \\
& \qquad\qquad\qquad\qquad \cdots \\
& \qquad\qquad\qquad\qquad [\mathit{pc} = \texttt{stm}_n][\![\mathit{Instruction\_n}]\!]_{n,\mathit{tt},\mathit{ff}} \\[2ex]
& \texttt{Ctrl}.\mathit{init}() = \texttt{r}_1\,!\,\mathit{init}(\mathit{tt},\mathit{ff},\mathit{this})\,.\,\texttt{r}_2\,!\,\mathit{init}(\mathit{tt},\mathit{ff},\mathit{this})\,.\,\mathit{this}\,!\,\mathit{run}(\texttt{stm}_1,\mathit{tt},\mathit{ff})
\end{array}
$$

where $[\![\mathit{Instruction\_i}]\!]_{i,\mathit{tt},\mathit{ff}}$ is equal to

- $\texttt{r}_j\,!\,\mathit{inc}(\texttt{stm}_{i+1},\mathit{tt},\mathit{ff})$ if $\mathit{Instruction\_i} = \mathsf{Inc}(R_j)$;
- $\texttt{r}_j\,!\,\mathit{decjump}(\texttt{stm}_{i+1},\texttt{stm}_k,\mathit{tt},\mathit{ff})$ if $\mathit{Instruction\_i} = \mathsf{DecJump}(R_j,k)$;
- $\texttt{0}$ if $\mathit{Instruction\_i} = \mathsf{Halt}$.

The main process is $\texttt{let } x = \texttt{new Ctrl}(x_1,\cdots,x_n,\texttt{new R}(\_,\_,\_,\_),\texttt{new R}(\_,\_,\_,\_))\ \texttt{in}\ x\,!\,\mathit{init}()$.

Fig. 1. Encoding a 2CM in $\texttt{Actor}_{\texttt{ba}}$ ("_" denotes an irrelevant initialization parameter)

field `stop`. This value of `stop` possibly makes the overall computation block as soon as an instruction concerning that register is performed.

In case (ii), registers have a field `dec` that is set to *tt* 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* in not enqueued, then the register is actually decremented and the field `dec` is set to *ff*. When *checkzero* will be executed, since $\texttt{dec} = \mathit{ff}$ then the next instruction of the 2CM is simulated. On the contrary, when *checkzero* is executed with $\texttt{dec} = \mathit{tt}$ then the decrement has not been performed (the register is 0) and the simulation jumps.

As in the examples of Section 2, booleans are implemented by two variables – see the method $\texttt{Ctrl}.\mathit{init}$ – that are distributed during the invocations. With a similar machinery, in the actor class `Ctrl`, the labels of the instructions are rep-

resented by the variables $x_1, \cdots, x_n$, which are stored in the fields $\mathtt{stm}_1, \ldots, \mathtt{stm}_n$ of *Ctrl*.

Theorem 1. *Termination and process reachability are undecidable in* $\mathtt{Actor}_{\mathtt{ba}}$.

The undecidability of termination in $\mathtt{Actor}_{\mathtt{ba}}$ follows by the property that a 2CM 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 P' (see Definition 1).

Proof. We reduce the termination problem of the 2CMs to the termination of the actor programs. It is possible to verify that if the 2CM executes the instructions $i_1, \cdots, i_n$ then the corresponding actor program has a computation that executes the processes $[\![\mathit{Instruction\_i_1}]\!]_{i_1,\mathtt{tt},\mathtt{ff}}, \cdots, [\![\mathit{Instruction\_i_n}]\!]_{i_n,\mathtt{tt},\mathtt{ff}}$ and conversely. Therefore the termination of the termination of the actor program implies the termination of the 2CM. The converse is not immediate because the actor program might have an infinite computation performing finitely many instructions $[\![\mathit{Instruction\_i}]\!]_{i,\mathtt{tt},\mathtt{ff}}$. This is excluded by the *bottom* messages in registers' queues that block the entire computation in case a register consumes and reintroduces in its queue all its *item* messages without an actual progress of the 2CM simulation. Hence if the actor program has an infinite computation then the 2CM does not terminate.

The undecidability result can be easily extended to the process reachability problem. It suffices to modify the process modeling the Halt instructions by using Q instead of 0, where Q is a process different from all the other processes in Figure 1. We have that Q is reachable if and only if the 2CM terminates. □

3.2 The language $\mathtt{Actor}^{\mathtt{ro}}$

We show that $\mathtt{Actor}^{\mathtt{ro}}$ is Turing-complete by delivering another encoding of a 2CM – see Figure 2. In this encoding the two registers are represented by two disjoint stacks of actors linked by the `next` field. The top elements of the two stacks are passed as parameters r_1 and r_2 of the *run* method of the controller. As before, this actor encodes the control of the 2CM.

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

The instruction DecJump is implemented by popping the corresponding stack. In particular, the method *run* of the controller is invoked with the field `next` of the register being decreased. This pop operation is performed provided the register that is argument of *run* is different from *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.

$$
\begin{array}{lll}
\texttt{R} & & \texttt{// R has a field next} \\
& \texttt{R}.dec_1(ctrl, r, stm) = & ctrl\,!\,run(\texttt{next}, r, stm) \\
& \texttt{R}.dec_2(ctrl, r, stm) = & ctrl\,!\,run(r, \texttt{next}, stm) \\
\texttt{Ctrl} & & \texttt{// Ctrl has fields stm}_1\texttt{, } \cdots\texttt{, stm}_n \texttt{ and nil} \\
& \texttt{Ctrl}.run(r_1, r_2, pc) = & [pc = \texttt{stm}_1][\![Instruction\_1]\!]\texttt{;} \\
& & \cdots \\
& & [pc = \texttt{stm}_n][\![Instruction\_n]\!]
\end{array}
$$

where $[\![Instruction\_i]\!]$ is equal to

- $this\,!\,run(\texttt{new R}(r_1), r_2, \texttt{stm}_{i+1})$ if $Instruction\_i = \mathsf{Inc}(R_1)$;
- $this\,!\,run(r_1, \texttt{new R}(r_2), \texttt{stm}_{i+1})$ if $Instruction\_i = \mathsf{Inc}(R_2)$;
- $[r_1 = \texttt{nil}]this\,!\,run(r_1, r_2, \texttt{stm}_k)\texttt{;}\, r_1\,!\,dec_1(this, r_2, \texttt{stm}_{i+1})$ if $Instruction\_i = \mathsf{DecJump}(R_1, k)$;
- $[r_2 = \texttt{nil}]this\,!\,run(r_1, r_2, \texttt{stm}_k)\texttt{;}\, r_2\,!\,dec_2(this, r_1, \texttt{stm}_{i+1})$ if $Instruction\_i = \mathsf{DecJump}(R_2, k)$;
- $\mathsf{0}$ if $Instruction\_i = \mathsf{Halt}$.

The program is invoked with $\texttt{let}\ x = \texttt{new Ctrl}(x_1, \cdots, x_n, nil)\ \texttt{in}\ x\,!\,run(nil, nil, x_1)$.

Fig. 2. Encoding a 2CM in $\texttt{Actor}^{\texttt{ro}}$

Theorem 2. *Termination and process reachability are undecidable in* $\texttt{Actor}^{\texttt{ro}}$.

Proof. It is easy to verify that a 2CM has a computation executing the instructions $i_1, \cdots, i_n$ if and only if the corresponding actor system has a run that executes the processes $[\![Instruction\_i_1]\!], \cdots, [\![Instruction\_i_n]\!]$. In this case, it is also guaranteed that the computation of the actor system terminates if and only if a Halt instruction is reached. The undecidability of process reachability is proved by using the same arguments of Theorem 1. □

4 Decidability results for $\texttt{Actor}^{\texttt{ro}}_{\texttt{ba}}$

We demonstrate that programs in $\texttt{Actor}^{\texttt{ro}}_{\texttt{ba}}$ are well-structured transition systems [1,9]. 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 $x_1, x_2, x_3, \cdots$, there exist $i < j$ with $x_i \leq x_j$.

Definition 2. *A* well-structured transition system *is a transition system* $(\mathcal{S}, \longrightarrow, \preceq)$ *where* $\preceq$ *is a quasi-ordering relation on states such that*

1. $\preceq$ *is a well-quasi-ordering*
2. $\preceq$ *is upward compatible with* $\longrightarrow$*, i.e., for every* $\mathtt{S}_1 \preceq \mathtt{S}'_1$ *such that* $\mathtt{S}_1 \longrightarrow \mathtt{S}_2$*, there exists* $\mathtt{S}'_1 \longrightarrow^* \mathtt{S}'_2$ *such that* $\mathtt{S}_2 \preceq \mathtt{S}'_2$*.*

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 $\overset{\bullet}{=}$ that relates renamings of variables *that are not free in the main process* into either actor names or variables *that are not free in the main process*. Let

$$\rho \overset{\bullet}{=} \rho' \quad \overset{def}{=} \quad \text{for every } x, y : \begin{array}{ll} (i) & \rho(x) = \rho(y) \quad \text{if and only if} \quad \rho'(x) = \rho'(y) \\ (ii) & \rho(x) = \rho'(x) \quad \text{if} \quad \rho(x) \text{ or } \rho'(x) \text{ is an actor name} \end{array}$$

Namely, two renamings are in the relation $\overset{\bullet}{=}$ if they identify the same variables, regardless the value they associate when such a value is a variable. For example, $[x \mapsto y, y \mapsto z] \overset{\bullet}{=} [x \mapsto x, y \mapsto z]$ and $[x \mapsto y, y \mapsto y, z \mapsto A] \overset{\bullet}{=} [x \mapsto x', y \mapsto x', z \mapsto A]$. However $[x \mapsto y, y \mapsto z] \overset{\bullet}{\neq} [x \mapsto x, y \mapsto x]$ and $[x \mapsto A] \overset{\bullet}{\neq} [x \mapsto B]$. In general, if ρ and ρ' are injective renamings that always return variables then $\rho \overset{\bullet}{=} \rho'$. The requirements of $\overset{\bullet}{=}$ are stronger for actor names: in this case the two renamings should be identical. We also notice that renamings never apply to free variables of the main process and never return free variables of the main processes. This because these variables are possibly stored in fields of actors and their renamings might change the behaviours of actors in a way that breaks the upward compatibility of the following relation $\preceq$ and $\longrightarrow$ (*c.f.* proof of Theorem 3, part **(2)**). We finally notice that the above renamings *do not change the main process* (because they do not apply to its free variables).

We denote by $P\rho$ the result of the application of ρ to P.

Next, let $\simeq$ be the least relation on terms $m(U_1, \cdots, U_n)$ and on processes such that

$$\frac{\rho \overset{\bullet}{=} \rho'}{m(\rho(x_1), \cdots, \rho(x_k)) \simeq m(\rho'(x_1), \cdots, \rho'(x_k))} \qquad \frac{\rho \overset{\bullet}{=} \rho'}{P\rho \simeq P\rho'}$$

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

Lemma 1. *Let* T *be either a process or a method invocation* $m(U_1, \cdots, U_n)$ *of a program in* $\mathtt{Actor}_{\mathtt{ba}}$ *(and therefore in* $\mathtt{Actor}^{\mathtt{ro}}_{\mathtt{ba}}$*). Let* $\mathcal{T} = \{T\rho_1, T\rho_2, T\rho_3, \cdots\}$ *be such that* $i \neq j$ *implies* $T\rho_i \not\simeq T\rho_j$*. Then* $\mathcal{T}$ *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 ρ on $\mathit{free}(P)$ that are different according to $\overset{\bullet}{=}$. In fact, the values of renamings on variables that are different from $\mathit{free}(P)$ do not play any role in the definition of $\mathcal{T}$.

The basic remark is that a renaming ρ generates a *partition* of the set $\mathit{free}(P)$: 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 $\overset{\bullet}{=}$ if they yield different partitions. The number of such renaming is the *Bell number* of the cardinality of $\mathit{free}(P)$, let it be $\mathtt{Bell}(\kappa)$, where κ is the cardinality of $\mathit{free}(P)$. In addition, in our case, renamings may map a variable to an actor name into a finite set $\{A_1, \cdots, A_\ell\}$. In this case the identity of the actor name is relevant. If $\kappa \geq \ell$ then $(\binom{\kappa}{\ell} \times \ell! + 1) \times \mathtt{Bell}(\kappa)$ is an upper bound to the different renamings according $\overset{\bullet}{=}$. If $\kappa < \ell$ then the upper bound is $(\ell!/\kappa! + 1) \times \mathtt{Bell}(\kappa)$. In any case the number of different renamings according to $\overset{\bullet}{=}$ is finite.

Henceforth the set $\mathcal{T}$ is finite as well. □

In order to define a well-quasi ordering on states, we consider the following *embedding relation* $\leq$ on queues (except the part about $\simeq$, it is almost standard [9]):

$$\frac{\textit{there exist } i_1 < i_2 < \cdots < i_k \leq h \textit{ such that, for } j \in 1..k,\ \ m_j(\widetilde{U_j}) \simeq n_{i_j}(\widetilde{V}_{i_j})}{m_1(\widetilde{U_1}) \ldots m_k(\widetilde{U_k}) \leq n_1(\widetilde{V_1}) \ldots n_h(\widetilde{V_h})}$$

Then we define the following relation on states:

$$\frac{P_i \simeq P'_i \quad \text{and} \quad q_i \leq q'_i \quad \text{for } i \in 1..\ell}{A_1 \triangleright (P_1, \varphi_1, q_1), \cdots, A_\ell \triangleright (P_\ell, \varphi_\ell, q_\ell) \ \preceq\ A_1 \triangleright (P'_1, \varphi_1, q'_1), \cdots, A_\ell \triangleright (P'_\ell, \varphi_\ell, q'_\ell)}$$

It is worth to notice that the relation $\preceq$ constraints corresponding elements $A \triangleright (P, \varphi, q)$ and $A \triangleright (P', \varphi, q')$ to have the same states. In fact these states are defined by the main process using either its free variables or the actor names that it has created.

Theorem 3. *Let $(\mathcal{S}, \longrightarrow)$ be a transition system of a program of $\mathtt{Actor}^{\mathtt{ro}}_{\mathtt{ba}}$. Then $(\mathcal{S}, \longrightarrow, \preceq)$ is a well-structured transition system.*

Proof. **(1)** *$\preceq$ is a well-quasi-ordering.* It is easy to prove that $\preceq$ is a quasi-ordering. To prove that $\preceq$ is a well-quasi-ordering, we reason by contradiction. Let $\mathtt{S}_1, \mathtt{S}_2, \mathtt{S}_3, \cdots$ be an infinite sequence such that, for every $i < j$, $\mathtt{S}_i \not\preceq \mathtt{S}_j$. It is easy to verify that without loss of generality we may assume that the sequence is such that for every $i < j$, $\mathtt{S}_i \not\succeq \mathtt{S}_j$ (all the states are incomparable). Note that an infinite strictly decreasing sequence, i.e., for every i, $\mathtt{S}_i \succeq \mathtt{S}_{i+1}$ and $\mathtt{S}_i \neq \mathtt{S}_{i+1}$ is not possible. Let

$$\mathtt{subterms}(\mathtt{C}) = \{P \quad | \quad \text{there exists a method } m \text{ such that } P \text{ is a subterm of } \mathtt{C}.m(\widetilde{x})\}\ .$$

The set $\mathtt{subterms}(\mathtt{C})$ is finite. Therefore, by Proposition 1, the number of terms $P\rho$ which are different according to $\overset{\bullet}{=}$ is finite as well. This means that it is possible to extract a subsequence $\mathtt{S}_{i_1}, \mathtt{S}_{i_2}, \mathtt{S}_{i_3}, \cdots$ from $\mathtt{S}_1, \mathtt{S}_2, \mathtt{S}_3, \cdots$ such that,

for every A, elements $A \triangleright (P_{i_j}\rho_{i_j}, \varphi_{i_j}, q_{i_j})$ and $A \triangleright (P_{i_k} i_k, \varphi_{i_k}, q_{i_k})$ in $\mathtt{S}_{i_j}$ and $\mathtt{S}_{i_k}$, respectively, we have $P_{i_j}\rho_{i_j} \stackrel{\bullet}{=} P_{i_k}\rho_{i_k}$.

Due to the above arguments, the sequence $\mathtt{S}_{i_1}, \mathtt{S}_{i_2}, \mathtt{S}_{i_3}, \cdots$ 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 $\mathtt{S}_{i_j} \preceq \mathtt{S}_{i_k}$ if and only if $(q_{i_j}^{A_1}, \cdots, q_{i_j}^{A_\ell}) \sqsubseteq^\ell (q_{i_k}^{A_1}, \cdots, q_{i_k}^{A_\ell})$, where $\sqsubseteq^\ell$ is the coordinatewise order defined by

$$(q_1, \cdots, q_\ell) \sqsubseteq^\ell (q'_1, \cdots, q'_\ell) \quad \stackrel{def}{=} \quad \textit{for every } h \,:\, q_h \leq q'_h$$

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

We are finally reduced to an infinite sequence of tuple of queues that are pairwise incomparable according to $\sqsubseteq^\ell$. This fact contradicts the

> *Higman's Lemma [13]: 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 $(X, \leq)$ is a well-quasi-ordering then $(X^\ell, \leq^\ell)$ is a well-quasi-ordering.

(2) $\preceq$ *is upward compatible with* $\longrightarrow$. Let $\mathtt{S}_1 \preceq \mathtt{S}'_1$ and $\mathtt{S}_1 \longrightarrow \mathtt{S}_2$. We demonstrate that there exists $\mathtt{S}'_1 \longrightarrow^* \mathtt{S}'_2$ such that $\mathtt{S}_2 \preceq \mathtt{S}'_2$. The proof is by induction on the height of the proof-tree of $\mathtt{S}_1 \longrightarrow \mathtt{S}_2$. Let $\mathtt{S}_1 = A_1 \triangleright (P_1\rho_1, \varphi_1, q_1), \cdots, A_\ell \triangleright (P_\ell\rho_\ell, \varphi_\ell, q_\ell)$. Since $\mathtt{S}_1 \preceq \mathtt{S}'_1$ then $\mathtt{S}'_1 = A_1 \triangleright (P_1\rho'_1, \varphi_1, q'_1), \cdots, A_\ell \triangleright (P_\ell\rho'_\ell, \varphi_\ell, q'_\ell)$ such that, for every i, $P_i\rho_i \stackrel{\bullet}{=} P_i\rho'_i$ and $q_i \leq q'_i$.

The basic case is when the height is 0. There are four subcases (the fourth case has been moved to the inductive case because, when the context is present, the case is more complex):

1. $\mathtt{S}_1 \longrightarrow \mathtt{S}_2$ is produced by the axiom $A \triangleright ((A\,!\,m(\widetilde{E})\,.\,P)\rho, \varphi, q),\ \longrightarrow\ A \triangleright (P\rho, \varphi, q \cdot m(\widetilde{x})\rho)$. Since $\mathtt{S}_1 \preceq \mathtt{S}'_1$ then $\mathtt{S}'_1 = A \triangleright ((A\,!\,m(\widetilde{x})\,.\,P)\rho', \varphi, q')$ and $\rho \stackrel{\bullet}{=} \rho'$ and $q \leq q'$. It is straightforward to verify that $\mathtt{S}'_1 \longrightarrow (P\rho', \varphi, q' \cdot m(\widetilde{x})\rho')$ and $A \triangleright (P\rho, \varphi, q \cdot m(\widetilde{x})\rho) \preceq A \triangleright (P\rho', \varphi, q' \cdot m(\widetilde{x})\rho')$.

2. $\mathtt{S}_1 \longrightarrow \mathtt{S}_2$ is produced by the axiom $A \triangleright ((B\,!\,m(\widetilde{E})\,.\,P)\rho, \varphi, q), B \triangleright (P', \varphi', q')\ \longrightarrow\ A \triangleright (P\rho, \varphi, q), B \triangleright (P', \varphi', q' \cdot m(\widetilde{x})\rho)$. Similar to the previous case.

3. $\mathtt{S}_1 \longrightarrow \mathtt{S}_2$ is produced by the axiom $A \triangleright (([E = E']P\,;Q)\rho, \varphi, q) \longrightarrow A \triangleright (P\rho, \varphi, q)$. There are three subcases (a) both E and E' are variables; (b) E is a variable and E' is a field; (c) E and E' are both fields. In case (a), if $\rho(x) = \rho(y)$ then, since $\mathtt{S}_1 \preceq \mathtt{S}'_1$, we have $\mathtt{S}'_1 = A \triangleright (([x = y]P\,;Q)\rho', \varphi, q')$ and $\rho \overset{\bullet}{=} \rho'$, $q \leq q'$. By $\rho \overset{\bullet}{=} \rho'$ and $\rho(x) = \rho(y)$, we obtain $\rho'(x) = \rho'(y)$. Therefore we may apply the same axiom to $\mathtt{S}'_1$ and derive $\mathtt{S}'_1 \longrightarrow A \triangleright (Q\rho', \varphi, q')$ with $\mathtt{S}_2 \preceq A \triangleright (Q\rho', \varphi, q')$. If $\rho(x) \neq \rho(y)$, the argument is similar.
   In case (b), let $E = x$ and $E' = \mathtt{f}$. We first notice that $\rho \overset{\bullet}{=} \rho'$ implies $[x = \mathtt{f}]\rho$ is true if and only if $[x = \mathtt{f}]\rho'$ is true. This because $\rho(x)$ if and only if $\rho'(x)$ is a variable. In this case they must be both different from the value of $\mathtt{f}$, which is either an actor name or a free variable in the main process. If $\rho(x)$ is an actor name then $\rho'(x) = \rho(x)$. Then the argument follows as in case (a).
   In case (c) we observe that $[\mathtt{f} = \mathtt{f}']\rho = [\mathtt{f} = \mathtt{f}'] = [\mathtt{f} = \mathtt{f}']\rho'$ and argue as before.
4. Let $\mathtt{S}_1 \longrightarrow \mathtt{S}_2$ be produced by the axiom $A \triangleright (\mathtt{0}, \varphi, m(\widetilde{U}) \cdot q) \longrightarrow A \triangleright (P[{}^{A}/_{this}][\widetilde{y'}/\widetilde{y}][\widetilde{U}/\widetilde{x}], \varphi, q)$. We discuss this case below.

The inductive cases deal with the choice operator and contexts. The proof when the axiom is different from the dequeue operation is omitted because straightforward. We detail the case of dequeue plus the contextual rules. Let $\mathtt{S}_1 \longrightarrow \mathtt{S}_2$ be produced by the axiom $A \triangleright (\mathtt{0}, \varphi, m(\widetilde{U}) \cdot q) \longrightarrow A \triangleright (P[{}^{A}/_{this}][\widetilde{y'}/\widetilde{y}][\widetilde{U}/\widetilde{x}], \varphi, q)$, with $\mathtt{C}.m(\widetilde{x}) = P$, $\mathtt{C}$ being the class of A, $\widetilde{y} = \mathit{free}(P) \setminus \widetilde{x}$ and $\widetilde{y'} = \mathit{fresh}(\widetilde{y})$. Therefore $\mathtt{S}_1 = A \triangleright (\mathtt{0}, m(\widetilde{U}) \cdot q), \mathtt{T}_1$ and $\mathtt{S}_2 = A \triangleright (P[{}^{A}/_{this}][\widetilde{y'}/\widetilde{y}][\widetilde{U}/\widetilde{x}], q), \mathtt{T}_1$ [in this subcase we are reasoning on more complex proof trees].

Since $\mathtt{S}_1 \preceq \mathtt{S}'_1$ then $\mathtt{S}'_1 = A \triangleright (\mathtt{0}, \varphi, n_1(\widetilde{V_1}) \cdots n_h(\widetilde{V_h}) \cdot m(\widetilde{V}) \cdot q'), \mathtt{T}'_1$ and $m(\widetilde{U}) \simeq m(\widetilde{V})$ and $q \leq q'$ and $\mathtt{T}_1 \preceq \mathtt{T}'_1$. By the operational semantics rules, we get $\mathtt{S}'_1 \longrightarrow^* A \triangleright (\mathtt{0}, \varphi, m(\widetilde{V}) \cdot q' \cdot q''), \mathtt{T}''_1$ with $\mathtt{T}'_1 \preceq \mathtt{T}''_1$ and, by definition, $q \leq q' \cdot q''$. At this stage, we notice that $A \triangleright (\mathtt{0}, \varphi, m(\widetilde{V}) \cdot q' \cdot q''), \mathtt{T}''_1 \longrightarrow A \triangleright (P[{}^{A}/_{this}][\widetilde{y'}/\widetilde{y}][\widetilde{V}/\widetilde{x}], \varphi, q' \cdot q''), \mathtt{T}''_1$, where $P[{}^{A}/_{this}][\widetilde{y'}/\widetilde{y}][\widetilde{U}/\widetilde{x}] \overset{\bullet}{=} P[{}^{A}/_{this}][\widetilde{y'}/\widetilde{y}][\widetilde{V}/\widetilde{x}]$ because $m(\widetilde{U}) \simeq m(\widetilde{V})$. Hence $A \triangleright (P[{}^{A}/_{this}][\widetilde{y'}/\widetilde{y}][\widetilde{U}/\widetilde{x}], q), \mathtt{T}_1 \preceq A \triangleright (P[{}^{A}/_{this}][\widetilde{y'}/\widetilde{y}][\widetilde{V}/\widetilde{x}], \varphi, q' \cdot q''), \mathtt{T}''_1$. □

We notice that the well-structured transition system $(\mathcal{S}, \longrightarrow, \preceq)$ has transitive and stuttering compatibility (see [9], pp 9, 10). Additionally, $(\mathcal{S}, \longrightarrow, \preceq)$ has decidable algorithms for computing $\preceq$ and for computing the next states. Then decidability of termination follows directly from Theorems 4.6 in [9].

Theorem 4. *In* $\mathtt{Actor}^{\mathtt{ro}}_{\mathtt{ba}}$ *termination is decidable.*

As discussed in Section 2, the transition systems of the actor language are not finite branching. This is also the case for programs in $\mathtt{Actor}^{\mathtt{ro}}_{\mathtt{ba}}$ (due to the presence of fresh variables in method body instantiations). However, in this case, the sets $\mathit{Succ}(\mathtt{S})$ and $\mathit{Pred}(\mathtt{S})$ are finite if we reason up-to the well-quasi ordering relation $\preceq$.

Lemma 2. *Let $(\mathcal{S}, \longrightarrow, \preceq)$ be a well-structured transition system of a program in $\mathtt{Actor}^{\mathtt{ro}}_{\mathtt{ba}}$, and let $\mathtt{S} \in \mathcal{S}$. Then there is a finite set $\mathcal{X} \subseteq \mathit{Pred}(\mathtt{S})$ such that, for every $\mathtt{S}' \in \mathit{Pred}(\mathtt{S})$, there is $\mathtt{T} \in \mathcal{X}$ with $\mathtt{T} \preceq \mathtt{S}'$. $\mathcal{X}$ can be effectively computed.*

Proof. We show how to compute $\mathcal{X}$. Let $\mathtt{S} = A \triangleright (P, \varphi, q), \mathtt{S}'$. The *predecessor processes* of P are the following ones: (i) $\mathtt{let}\ x = E\ \mathtt{in}\ P'$, with $P = P'[\widetilde{U}/\widetilde{x}]$, for some $\widetilde{U}$ and some $\widetilde{x}$; (ii) $x\,!\,m(E_1, \cdots, E_n)\,.\,P$; (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 $\mathtt{C}$ then we take all the method bodies of $\mathtt{C}$ 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 $A = \aleph$ 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 $\mathtt{C}$, then $E = y$, for some y. If $x \in \mathit{free}(P')$ then $\mathcal{X}$ contains the configuration $A \triangleright (\mathtt{let}\ x = y\ \mathtt{in}\ P', \varphi, q), \mathtt{S}'$ with $P = P'[y/x]$. Otherwise $\mathcal{X}$ contains the configuration $A \triangleright (\mathtt{let}\ x = z\ \mathtt{in}\ P', \varphi, q), \mathtt{S}'$, for $z \in \mathit{free}(P')$ and for a unique $z \notin \mathit{free}(P')$. When $A = \aleph$ then E may be $\mathtt{new}\ C$ (orherwise the argument is as before). If $x \in \mathit{free}(P')$ and $\mathtt{S}' = A' \triangleright (\mathtt{0}, \varphi, \varepsilon), \mathtt{S}''$ with $A' \in \mathtt{C}$ then $\mathcal{X}$ contains the configuration $A \triangleright (\mathtt{let}\ x = \mathtt{new}\ C\ \mathtt{in}\ P', q), \mathtt{S}''$ (and this for every possible $A' \in \mathtt{C}$ such that $A' \triangleright (\mathtt{0}, \varepsilon)$ is in $\mathtt{S}'$). □

Lemma 2 and Theorem 4.8 in [9] allow us to decide the so-called *control-state reachability problem*: given two states $\mathtt{S}$ and $\mathtt{T}$ of a well-structured transition system with well-quasi ordering $\preceq$, decide whether there is $\mathtt{T}' \succeq \mathtt{T}$ such that $\mathtt{S} \longrightarrow^* \mathtt{T}'$.

Theorem 5. *In $\mathtt{Actor}^{\mathtt{ro}}_{\mathtt{ba}}$ process reachability is decidable.*

Proof. Let $\uparrow \mathtt{S} = \{\mathtt{S}' \in \mathcal{S} \mid \mathtt{S} \preceq \mathtt{S}'\}$. Let also $\mathit{Pred}(\uparrow \mathtt{S}) = \{\mathtt{T} \mid \mathtt{T} \longrightarrow \mathtt{S}'$ and $\mathtt{S}' \succeq \mathtt{S}\}$. By definition of $\preceq$, $\mathit{Pred}(\uparrow \mathtt{S}) \subseteq \mathit{Pred}(\mathtt{S})$. Therefore $\uparrow \mathit{Pred}(\uparrow \mathtt{S}) \subseteq \uparrow \mathit{Pred}(\mathtt{S}) \subseteq \uparrow \mathcal{X}$, where $\mathcal{X}$ is the finite set of Lemma 2 that is effectively computable. The theorem follows from Theorem 3.6 in [9]. □

In addition to the above decidability results, the process reachability problem – see Definition 1 – is decidable in the sublanguage of the present section. In fact, in order to verify whether a configuration $A \triangleright (P', \varphi, q), \mathtt{S}$ is reachable with P' equal to P up-to renaming of variables and actor names, we proceed as follows. First, consider a configuration $\mathtt{T}$ reachable after the complete execution of the main process. Therefore, in $\mathtt{T}$, every possible actor has been created (with the corresponding initialization performed). Let $\mathtt{T} = A_1 \triangleright (P_1, \varphi_1, q_1), \cdots, A_\ell \triangleright (P_\ell, \varphi_\ell, q_\ell)$. If this part of the computation already traverses a configuration with a term $A \triangleright (P', \varphi, q)$, then the reply is positive. Otherwise, we check control-state reachability from $\mathtt{T}$ to at least one of the states in the following finite set:

$$\mathcal{S} = \{\ A_1 \triangleright (Q_1, \varphi_1, \varepsilon), \cdots, A_\ell \triangleright (Q_\ell, \varphi_\ell, \varepsilon)\ \mid \text{for every } 1 \leq i \leq \ell,\ Q_i \text{ is a suffix of a method definition and there exists } 1 \leq j \leq \ell \text{ such that } Q_j \text{ is equal to } P \text{ up-to renaming } \}$$

We conclude this section by observing that we have already proved the undecidability of termination in programs with finitely many actors and field updates. If we remove the constraint of finite actor names then the relation $\preceq$ is not a well-quasi ordering anymore. Consider for instance, the configuration $\mathtt{S}_n$ defined as follows:

$$\mathtt{S}_n \stackrel{def}{=} \quad A_1 \triangleright (\mathtt{0}, \varnothing, \varepsilon) \, , \cdots , A_n \triangleright (\mathtt{0}, \varnothing, \varepsilon)$$

The infinite sequence $\mathtt{S}_1, \mathtt{S}_2, \mathtt{S}_3, \cdots$ is such that, for every $i < j$, $\mathtt{S}_i \not\preceq \mathtt{S}_j$. This trivial counterexample seems to suggest the following patch of $\preceq$:

$$\mathtt{S} \preceq' \mathtt{T} \quad \stackrel{def}{=} \quad \text{there exists } \mathtt{S}' \subseteq \mathtt{T} \text{ such that } \mathtt{S} \preceq \mathtt{S}'$$

However, the infinite sequence $\mathtt{S}_2, \mathtt{S}_3, \mathtt{S}_4, \cdots$ where $\mathtt{S}_n$ is defined as

$$\begin{array}{rll} \mathtt{S}_n & \stackrel{def}{=} & A_0 \triangleright (\mathtt{0}, \varnothing, m(A_{n-1}, A_1)) \ , \ A_1 \triangleright (\mathtt{0}, \varnothing, m(A_n, A_2)) \ , \\ & & \bigcup_{i \in 2..n-1} A_i \triangleright (\mathtt{0}, \varnothing, m(A_{i-2}, A_{i+1})) \ , \ A_n \triangleright (\mathtt{0}, \varnothing, m(A_{n-2}, A_0)) \end{array}$$

is such that, for every $i < j$, $\mathtt{S}_i \not\preceq' \mathtt{S}_j$.

5 Decidability results for $\mathtt{Actor}^{\mathtt{sl}}$

We prove that in $\mathtt{Actor}^{\mathtt{sl}}$ termination and process reachability are decidable, too. As discussed at the end of Section 4, we have not succeeded in demonstrating these decidability results by patching the definition of $\preceq$ in Section 4. The reason is that $\mathtt{Actor}^{\mathtt{sl}}$ programs may produce unboundedly many actor names. 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 interesting decidability properties for the original language.

Since we need a model with inexact message deliveries, we change the operational semantics in Table 1 in order to decouple the evaluation of the body of a method from the actor name of that method. Let $\mathtt{S} \longrightarrow_\alpha \mathtt{S}'$ be the *abstract transition relation* defined as $\mathtt{S} \longrightarrow \mathtt{S}'$ in Table 1 except the two rules (INVK) and (INVK-A) for method invocation and the rule (INST) for the instantiation of method bodies, which are replaced by the following ones:

$$\text{(INK-SA)} \quad \frac{\widetilde{E} \stackrel{\varphi}{\rightsquigarrow} \widetilde{U} \ ; \ \mathtt{S} \quad A, A' \in \mathtt{C}}{A \triangleright (A' \,!\, m(\widetilde{E}) \,.\, P, \varphi, q) \ \longrightarrow_\alpha \ A \triangleright (P, \varphi, q \cdot m(\widetilde{U}, A')), \mathtt{S}}$$

(INVK-A)

$$\frac{\widetilde{E} \overset{\varphi}{\rightsquigarrow} \widetilde{U}\ ;\ \mathtt{S} \qquad A', A'' \in \mathtt{C}}{A \triangleright (A' \,!\, m(\widetilde{E})\,.\,P, \varphi, q), A'' \triangleright (P', \varphi', q') \ \longrightarrow_\alpha\ A \triangleright (P, \varphi, q), A'' \triangleright (P', \varphi', q' \cdot m(\widetilde{U}, A')), \mathtt{S}}$$

(INST-A)

$$\frac{A' \in \mathtt{C} \quad \mathtt{C}.m(\widetilde{x}) = P \quad \widetilde{y} = \mathit{free}(P) \setminus \widetilde{x} \quad \widetilde{y'} = \mathit{fresh}(\widetilde{y})}{A \triangleright (\mathtt{0}, \varphi, m(\widetilde{U}, A') \cdot q) \ \longrightarrow_\alpha\ A \triangleright (P[{}^{A'}/_{\mathit{this}}][{}^{\widetilde{y'}}/_{\widetilde{y}}][{}^{\widetilde{U}}/_{\widetilde{x}}], \varphi, q)}$$

In the abstract transition relation, an item $m(\widetilde{U})$ is added in a queue of an actor name *nondeterministically selected* among those names belonging to the same class of the target actor. The item $m(\widetilde{U})$ is enqueued with an additional argument – the actor name of the target actor. This additional argument is used when a 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.

The next proposition formalizes the correspondence between $\longrightarrow$ and $\longrightarrow_\alpha$ (for stateless programs). We first introduce few notations:

- Let $\alpha()$ be a map from "concrete" to "abstract" configurations: given a configuration $\mathtt{S}$, we denote with $\alpha(\mathtt{S})$ the configuration obtained from $\mathtt{S}$ by replacing each of its actor $A \triangleright (P, \varnothing, q)$ with $A \triangleright (P, \varnothing, q')$ where q' is obtained from q by adding to each of its method invocations the parameter A.
- We use $\mathcal{M}$, $\mathcal{M}'$ to denote multisets of terms $m(\widetilde{U})$. We extend $\simeq$ to such multisets: $\mathcal{M} \simeq \mathcal{M}'$ iff there exists a bijection ρ from $\mathcal{M}$ to $\mathcal{M}'$ such that $m(\widetilde{U}) \simeq \rho(m(\widetilde{U}))$.
- Let $\mathtt{S} \xrightarrow{\mathcal{M}} \mathtt{S}'$ be the least relation such that

$$\mathtt{S} \xrightarrow{\varnothing} \mathtt{S} \qquad \frac{\mathtt{S} \xrightarrow{\mathcal{M}} \mathtt{S}' \qquad (\mathtt{S}' \longrightarrow \mathtt{S}'' \quad \text{proved without (INVK) or (INVK-S)})}{\mathtt{S} \xrightarrow{\mathcal{M}} \mathtt{S}''}$$

$$\frac{\mathtt{S} \xrightarrow{\mathcal{M}} A \triangleright (P, \varnothing, q), \mathtt{S}' \qquad A \triangleright (P, \varnothing, q), \mathtt{S}' \longrightarrow A \triangleright (P', \varnothing, q \cdot m(\widetilde{U})), \mathtt{S}''}{\mathtt{S} \xrightarrow{\mathcal{M} \uplus \{m(\widetilde{U}, A)\}} A \triangleright (P', \varnothing, q \cdot m(\widetilde{U})), \mathtt{S}''}$$

  Namely, this transition $\mathtt{S} \xrightarrow{\mathcal{M}} \mathtt{S}'$ collects in $\mathcal{M}$ all the method invocations that have been performed during the computation $\mathtt{S} \longrightarrow \mathtt{S}'$. These method invocations are extended with the target actor name as last parameter.
- Let $\mathtt{S} \xrightarrow{\mathcal{M}}_\alpha \mathtt{S}'$ be the least relation such that

$$\mathtt{S} \xrightarrow{\varnothing}_\alpha \mathtt{S} \qquad \frac{\mathtt{S} \xrightarrow{\mathcal{M}}_\alpha \mathtt{S}' \qquad (\mathtt{S}' \rightarrow_\alpha \mathtt{S}'' \quad \text{proved without (INVK-A) or (INVK-SA)})}{\mathtt{S} \xrightarrow{\mathcal{M}}_\alpha \mathtt{S}''}$$

$$\frac{\mathtt{S} \xrightarrow{\mathcal{M}}_\alpha A \triangleright (P, \varnothing, q), \mathtt{S}' \qquad A \triangleright (P, \varnothing, q), \mathtt{S}' \rightarrow_\alpha A \triangleright (P', \varnothing, q \cdot m(\widetilde{U})), \mathtt{S}''}{\mathtt{S} \xrightarrow{\mathcal{M} \uplus \{m(\widetilde{U})\}}_\alpha A \triangleright (P', \varnothing, q \cdot m(\widetilde{U})), \mathtt{S}''}$$

  Note that in this case the additional argument A is not explicitly added as it is already introduced as argument by the transition system $\longrightarrow_\alpha$.

Proposition 1. *Let* $\mathtt{S}$ *be a state of a transition system of a stateless program.*

1. *If* $\mathtt{S} \longrightarrow \mathtt{S}'$ *then* $\alpha(\mathtt{S}) \longrightarrow_\alpha \alpha(\mathtt{S}')$*;*
2. *if* $\alpha(\mathtt{S}) \xrightarrow{\mathcal{M}}_\alpha A \triangleright (P, \varnothing, q), \mathtt{T}$ *then* $\mathtt{S} \xrightarrow{\mathcal{M}'} A' \triangleright (P', \varnothing, q'), \mathtt{S}' \xrightarrow{\mathcal{M}''} \mathtt{S}''$ *such that* $P \simeq P'$ *and there exists* $\mathcal{M}'''$ *such that* $\mathcal{M} \simeq \mathcal{M}'''$ *and* $\mathcal{M}''' \subseteq \mathcal{M}' \uplus \mathcal{M}''$*.*

Proof. The first item trivially holds because the new rules used in the definition of $\longrightarrow_\alpha$ are (strictly) more general than the corresponding rules used in the definition of $\longrightarrow$.

The second item is proved by induction on the height of the proof tree of $\alpha(\mathtt{S}) \xrightarrow{\mathcal{M}}_\alpha A \triangleright (P, \varnothing, q), \mathtt{T}$.

The base case is trivial as $\mathtt{S} \xrightarrow{\emptyset} \mathtt{S}$ and as P occurs in $\alpha(\mathtt{S})$ then it also occurs in $\mathtt{S}$.

In the inductive case there are two sub-cases, according to the last rule used in the proof tree of $\alpha(\mathtt{S}) \xrightarrow{\mathcal{M}}_\alpha A \triangleright (P, \varnothing, q), \mathtt{T}$. The first sub-case is when the last rule is

$$\frac{\alpha(\mathtt{S}) \xrightarrow{\mathcal{M}}_\alpha \mathtt{T}' \qquad (\mathtt{T}' \to_\alpha A \triangleright (P, \varnothing, q), \mathtt{T} \quad \text{proved without (INVK-A) or (INVK-SA)})}{\alpha(\mathtt{S}) \xrightarrow{\mathcal{M}}_\alpha A \triangleright (P, \varnothing, q), \mathtt{T}}$$

We proceed by case analysis on $\mathtt{T}' \to_\alpha A \triangleright (P, \varnothing, q), \mathtt{T}$.

- If the process P of $A \triangleright (P, \varnothing, q)$ is not evaluated in this transition then $A \triangleright (P, \varnothing, q')$ is already present in $\mathtt{T}'$ for some q'. Therefore $\mathtt{T}' = A \triangleright (P, \varnothing, q'), \mathtt{T}''$ (for some $\mathtt{T}''$) and we may apply the inductive hypothesis to $\alpha(\mathtt{S}) \xrightarrow{\mathcal{M}}_\alpha A \triangleright (P, \varnothing, q''), \mathtt{T}''$ and conclude.
- If the process P of $A \triangleright (P, \varnothing, q)$ is evaluated in $\mathtt{T}' \to_\alpha A \triangleright (P, \varnothing, q), \mathtt{T}$ then we only discuss when the rule used is an instance of (INST-A) (the other cases are simpler). In this case we have that $\alpha(\mathtt{S}) \xrightarrow{\mathcal{M}}_\alpha \mathtt{T}'$ and $\mathtt{T}' = A \triangleright (\mathtt{0}, \varnothing, m(\widetilde{U}, A') \cdot q), \mathtt{T}''$. Moreover, $m(\widetilde{U}, A') \in \mathcal{M}$.

  By inductive hypothesis $\mathtt{S} \xrightarrow{\mathcal{M}'} \mathtt{S}' \xrightarrow{\mathcal{M}''} \mathtt{S}''$ and there exists $\mathcal{M}'''$ such that $\mathcal{M} \simeq \mathcal{M}'''$ and $\mathcal{M}''' \subseteq \mathcal{M}' \uplus \mathcal{M}''$. As $\mathcal{M} \simeq \mathcal{M}'''$, we have that $\mathcal{M}'''$ contains a term $m(\widetilde{U'}, A'')$ such that $m(\widetilde{U'}, A'') \simeq m(\widetilde{U}, A')$, hence $A'' = A'$.
  The thesis follows from the existence of an extension of this last computation that completes the execution of the main program and (at least) of all the method invocations in $\mathcal{M}'''$. Notice that the execution of the method invocation $m(\widetilde{U'}, A')$ will instantiate a process P' such that $P \simeq P'$.
- If the process $A \triangleright (P, \varnothing, q)$ has been created by the last transition, we have that $P = \mathtt{0}$, $\alpha(\mathtt{S}) \xrightarrow{\mathcal{M}}_\alpha \mathtt{T}'$ and $\mathtt{T}' = A' \triangleright (\mathtt{new\ C()}.P', \varnothing, q'), \mathtt{T}''$. By inductive hypothesis $\mathtt{S} \xrightarrow{\mathcal{M}'} A'' \triangleright (\mathtt{new\ C()}.P'', \varnothing, q''), \mathtt{S}' \xrightarrow{\mathcal{M}''} \mathtt{S}''$ and there exists $\mathcal{M}'''$ such that $\mathcal{M} \simeq \mathcal{M}'''$ and $\mathcal{M}''' \subseteq \mathcal{M}' \uplus \mathcal{M}''$. Thesis follows from the existence of an extension of this last computation in which the $\mathtt{new\ C()}.P''$ process performs the creation of the new object that will be initialized with an empty $\mathtt{0}$ process.

The second sub-case is when the last rule of $\alpha(\mathtt{S}) \xrightarrow{\mathcal{M}}_\alpha A \triangleright (P, \varnothing, q), \mathtt{T}$ is

$$\frac{\alpha(\mathtt{S}) \xrightarrow{\mathcal{M}_1}_\alpha A \triangleright (P', \varnothing, q'), \mathtt{T}' \qquad A \triangleright (P', \varnothing, q'), \mathtt{T}' \rightarrow_\alpha A \triangleright (P, \varnothing, q' \cdot m(\widetilde{U}, A')), \mathtt{T}}{\alpha(\mathtt{S}) \xrightarrow{\mathcal{M}_1 \uplus \{m(\widetilde{U}, A')\}}_\alpha A \triangleright (P, \varnothing, q' \cdot m(\widetilde{U}, A')), \mathtt{T}}$$

with $\mathcal{M} = \mathcal{M}_1 \uplus \{m(\widetilde{U}, A')\}$ and $q = q' \cdot m(\widetilde{U}, A')$. In this case we have that $A \triangleright (P', \varnothing, q'), \mathtt{T}' = B \triangleright (A' \,!\, m(\widetilde{E}) \,.\, Q, \varnothing, q''), \mathtt{T}''$ with $\widetilde{E} \overset{\varnothing}{\rightsquigarrow} \widetilde{U}$, that is, the last rule is a method invocation. We can apply the inductive hypothesis to $\alpha(\mathtt{S}) \xrightarrow{\mathcal{M}_1}_\alpha B \triangleright (A' \,!\, m(\widetilde{E}) \,.\, Q, \varnothing, q''), \mathtt{T}''$. Hence, we have $\mathtt{S} \xrightarrow{\mathcal{M}'} B' \triangleright (A'' \,!\, m(\widetilde{E'}) \,.\, Q', \varnothing, q'''), \mathtt{S}' \xrightarrow{\mathcal{M}''} \mathtt{S}''$ with $A' \,!\, m(\widetilde{E}) \,.\, Q \simeq A'' \,!\, m(\widetilde{E'}) \,.\, Q'$ and there exists $\mathcal{M}_1'$ such that $\mathcal{M}_1 \simeq \mathcal{M}_1'$ and $\mathcal{M}_1' \subseteq \mathcal{M}' \uplus \mathcal{M}''$. Also in this case the thesis follows from the existence of an extension of this last computation that completes the execution of the main program and (at least) of all the method invocations in $\mathcal{M}_1'$. Notice that the execution of the method invocation $A'' \,!\, m(\widetilde{E'})$ will have the effect of adding $m(\widetilde{U}, A')$ to the multiset because $A'' \,!\, m(\widetilde{E'}) \simeq A' \,!\, m(\widetilde{E})$. Moreover, also notice that the process P' in $\alpha(\mathtt{S}) \xrightarrow{\mathcal{M}_1}_\alpha A \triangleright (P', \varnothing, q'), \mathtt{T}'$ is a suffix of either the main process or the body of a method invocation in $\mathcal{M}_1$, so the considered extended computation surely traverses a configuration including a process P'' such that $P' \simeq P''$. □

As a consequence we have that the abstract semantics preserves both termination and process reachability.

Proposition 2. *Let* $\mathtt{S}$ *be a state of a transition system of a program in* $\mathtt{Actor}^{\mathtt{sl}}$.

- $\mathtt{S}$ terminates *in the concrete transition system if and only if* $\alpha(\mathtt{S})$ terminates *in the abstract transition system;*
- *given a process* P*, there exist* A'*,* q'*, and* $\mathtt{S}'$ *such that* $\mathtt{S} \longrightarrow^* A' \triangleright (P, \varnothing, q'), \mathtt{S}'$ *if and only if there exist* A''*,* q''*, and* $\mathtt{S}''$ *such that* $\alpha(\mathtt{S}) \longrightarrow_\alpha^* A'' \triangleright (P, \varnothing, q''), \mathtt{S}''$.

Proof. The second item is a direct consequence of Proposition 1. For the first item we first observe that if $\mathtt{S}$ has an infinite computation then the same also holds for $\alpha(\mathtt{S})$ (see the first item of Proposition 1). We now assume that $\mathtt{S}$ terminates. This implies the existence of an upper bound k such that if $\mathtt{S} \xrightarrow{\mathcal{M}} \mathtt{S}'$ then $|\mathcal{M}| \leq k$. We now proceed by contradiction assuming that $\alpha(\mathtt{S})$ does not terminate. If this is the case, there exists $\mathtt{T}$ such that $\alpha(\mathtt{S}) \xrightarrow{\mathcal{M}'} \mathtt{T}$ with $|\mathcal{M}'| > k$. By the second item of Proposition 1 we have that there exists $\mathtt{S}''$ such that $\mathtt{S} \xrightarrow{\mathcal{M}''} \mathtt{S}''$ such that $|\mathcal{M}''| \geq |\mathcal{M}'| > k$. But this contradicts the assumption on the upper bound k. □

We now move to the definition of $\preceq_\alpha$, a variant of the ordering $\preceq$ defined in the previous section, such that $(\mathcal{S}, \longrightarrow_\alpha, \preceq_\alpha)$ turns out to be a well-structured transition system (for configurations of stateless programs). To this aim, we redefine the notions of Section 4. Let

– $\overset{\bullet}{=}_\alpha$ be the least relation such that
$\rho \overset{\bullet}{=}_\alpha \rho' \quad \overset{def}{=} \quad$ for every x, y :
$(i)\ \rho(x) = \rho(y) \quad$ if and only if $\quad \rho'(x) = \rho'(y)$
$(ii)\ \rho(x) \in \mathtt{C} \quad$ if and only if $\quad \rho'(x) \in \mathtt{C}$

Differently from the definition of $\overset{\bullet}{=}$, $\overset{\bullet}{=}_\alpha$ does not care of the identity of actor names. Moreover, $\overset{\bullet}{=}_\alpha$ identifies two renamings that "have matching types", letting the type of variable being distinct from those of class actors.
– $\simeq_\alpha$ be the relation defined as $\simeq$ in Section 4, with $\overset{\bullet}{=}_\alpha$ instead of $\overset{\bullet}{=}$.
– $\leq_\alpha$ be the relation defined as $\leq$ in Section 4, with $\simeq_\alpha$ instead of $\simeq$.
– $\preceq_\alpha$ be the ordering:

$$\frac{A_i, A'_{j_i} \in \mathtt{C}_i \quad P_i \simeq_\alpha P'_{j_i} \ \text{ and } \ q_i \leq q'_{j_i} \ \text{ for } i \in 1..\ell,\ 1 \leq j_1 < j_2 < \cdots < j_\ell \leq \kappa}{A_1 \triangleright (P_1, \varnothing, q_1), \cdots, A_\ell \triangleright (P_\ell, \varnothing, q_\ell) \ \leq_\alpha \ A'_1 \triangleright (P'_1, \varnothing, q'_1), \cdots, A'_\kappa \triangleright (P'_\kappa, \varnothing, q'_\kappa)}$$

Next, we observe that Lemma 1 can be adapted to the case of unbounded actors by using $\simeq_\alpha$ instead of $\simeq$. Let T be either a process or a method invocation $m(U_1, \cdots, U_n)$ of a stateless program and let $\mathcal{T} = \{T\rho_1, T\rho_2, T\rho_3, \cdots\}$ be such that $i \neq j$ implies $T\rho_i \not\simeq_\alpha T\rho_j$. By proceeding as in the proof of Lemma 1, we prove that $\mathcal{T}$ is finite.

Theorem 6. *Let $(\mathcal{S}, \longrightarrow_\alpha)$ be the abstract transition system of a program in $\mathtt{Actor}^{\mathtt{sl}}$. Then $(\mathcal{S}, \longrightarrow_\alpha, \preceq_\alpha)$ is a well-structured transition system.*

Proof. The proof is as in Theorem 3 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 ℓ) of queues of terms is used. As we now consider configurations with an unbounded number of actors, instead of configurations with a bounded number ℓ of actors, we need to resort to the embedding $\sqsubseteq_\alpha$ defined as follows:

$$\frac{q_i \leq_\alpha q'_{j_i} \quad \text{for } \ i \in 1..\ell,\ 1 \leq j_1 < j_2 < \cdots < j_\ell \leq \kappa}{(q_1, \cdots, q_\ell) \ \sqsubseteq_\alpha \ (q'_1, \cdots, q'_\kappa)}$$

The final contradiction of part **(1)** is now reached by observing that by Higman's lemma, also $\sqsubseteq_\alpha$ is a well-quasi ordering, as a consequence of the well-quasi ordering $\leq_\alpha$.

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 $\preceq_\alpha$ does not preserve actor names as it was for $\preceq$ in the proof of Theorem 3. But $\preceq_\alpha$ preserves at least actor classes. As the abstract transition system $\longrightarrow_\alpha$ allows a term $m(\widetilde{U})$ to be introduced in the queue of any of the actor belonging to the same class, 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. □

In the light of Theorem 6, it is possible to decide the termination for the abstract transition system of a stateless program. As termination is preserved by the abstract semantics (see Proposition 2) we can conclude that termination is also decidable for the concrete transition system of a stateless program.

We complete this section by demonstrating the decidability of control-state reachability for the well-structured transition system $(\mathcal{S}, \longrightarrow_\alpha, \preceq_\alpha)$ of a stateless program (see the definition after Lemma 2). The proof is similar to the one of Theorem 5, with the difference that it is needed a more sophisticated algorithm for computing the predecessors of a configuration.

Lemma 3. *Let $(\mathcal{S}, \longrightarrow_\alpha, \preceq_\alpha)$ be a well-structured transition system of a program in $\texttt{Actor}^{\texttt{sl}}$, and let $\texttt{S} \in \mathcal{S}$. Then there is a finite set $\mathcal{X}$ such that, for every $\texttt{S}' \succeq_\alpha \texttt{S}$ and $\texttt{S}'' \in Pred(\texttt{S}')$, there is $\texttt{T} \in \mathcal{X}$ with $\texttt{T} \preceq_\alpha \texttt{S}''$. $\mathcal{X}$ can be effectively computed.*

Proof. The computation of $\mathcal{X}$ must extend the construction presented in the proof of Lemma 2 in two ways.

The extension **(1)** 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.

The extension **(2)** derives from the fact that, differently from the ordering $\preceq$ considered in Lemma 2, if $\texttt{S} \preceq_\alpha \texttt{S}'$ it could be possible for $\texttt{S}'$ to have strictly more actors than $\texttt{S}$. We can cope with this problem by applying the procedure described in the proof of Lemma 2 not only to the configuration $\texttt{S}$, but to all the configurations in a computable set representing a finite basis for all the configurations $\texttt{S}' \succeq_\alpha \texttt{S}$ with strictly more actors than $\texttt{S}$. This set includes all the extensions of $\texttt{S}$ with one or two additional actors belonging to one of the finite classes of the considered program. There finitely many combination of actor names up-to $\preceq_\alpha$. Each of these additional actors executes a process obtained by applying a renaming ρ 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 $\preceq_\alpha$. Finally, the additional actors have a queue including at most one method invocation. 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 $\preceq_\alpha$.

The set $\mathcal{X}$ is computed by applying the procedure of the proof of Lemma 2 with the extension **(1)**, to $\texttt{S}$ as well as to all the other configurations computed as in the extension **(2)**. □

We can conclude that control-state reachability is decidable for the abstract transition system of a stateless program. This entails the decidability of process reachability. In fact, given a process P, the reachability of a configuration $A \triangleright (P', \varphi, q), \texttt{S}$ 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 $\texttt{C}_1, \ldots, \texttt{C}_n$ be the actor classes of the considered actor system and let $A_1, \cdots, A_n$ be such that $A_i \in \texttt{C}_i$.

We consider the following finite set of states:

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

From the decidability of the process reachability problem for the abstract transition system we can conclude its decidability for the concrete semantics. By Proposition 2, this problem is preserved by the abstract semantics. Note that control-state reachability is not preserved by the abstract semantics. In fact, the abstract transition system is guaranteed to execute the same method invocations, but this can be done in a different order and also by different actors.

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 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.

References

1. P. A. Abdulla, K. Cerans, B. Jonsson, and Y.-K. Tsay. General decidability theorems for infinite-state systems. In *LICS*, pages 313–321. IEEE, 1996.
2. G. Agha. The structure and semantics of actor languages. In *REX Workshop*, pages 1–59, 1990.
3. G. Agha, I. Mason, S. Smith, and C. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7:1–72, 1997.
4. J. Armstrong. Erlang. *Communications of ACM*, 53(9):68–75, 2010.
5. P.-H. Chang and G. Agha. Supporting reconfigurable object distribution for customized web applications. In *SAC*, pages 1286–1292, 2007.
6. P.-H. Chang and G. Agha. Towards context-aware web applications. In *DAIS*, pages 239–252, 2007.
7. E. Cheong, E. A. Lee, and Y. Zhao. Viptos: a graphical development and simulation environment for tinyos-based wireless sensor networks. In *SenSys*, pages 302–302, 2005.
8. F. S. de Boer, I. Grabe, and M. Steffen. Termination detection for active objects. *Journal of Logic and Algebraic Programming*, 2012.
9. A. Finkel and P. Schnoebelen. Well-structured transition systems everywhere! *Theoretical Computer Science*, 256:63–92, 2001.
10. E. Giachino and C. Laneve. Analysis of deadlocks in object groups. In *FMOOD-S/FORTE*, pages 168–182, 2011.
11. P. Haller and M. Odersky. Scala actors: Unifying thread-based and event-based programming. *Theoretical Computer Science*, 410(2-3):202–220, 2009.
12. C. Hewitt. Procedural embedding of knowledge in planner. In *Proc. the 2nd International Joint Conference on Artificial Intelligence*, pages 167–184, 1971.
13. G. Higman. Ordering by Divisibility in Abstract Algebras. *Proc. London Math. Soc.*, s3-2(1):326–336, 1952.

14. E. B. Johnsen and O. Owe. An asynchronous communication model for distributed concurrent objects. *Software and System Modeling*, 6(1):39–58, 2007.
15. R. K. Karmani, A. Shali, and G. Agha. Actor frameworks for the jvm platform: a comparative analysis. In *PPPJ*, pages 11–20. ACM, 2009.
16. E. A. Lee, X. Liu, and S. Neuendorffer. Classes and inheritance in actor-oriented design. *ACM Transactions in Embedded Computing Systems*, 8(4), 2009.
17. E. A. Lee, S. Neuendorffer, and M. J. Wirthlin. Actor-oriented design of embedded hardware and software systems. *Journal of Circuits, Systems, and Computers*, 12(3):231–260, 2003.
18. R. Meyer. On boundedness in depth in the pi-calculus. In *IFIP TCS*, volume 273 of *IFIP*, pages 477–489. Springer, 2008.
19. M. Minsky. *Computation: finite and infinite machines*. Prentice Hall, 1967.
20. OpenID. Openid specifications. `http://openid.net/developers/specs/`.
21. T. Wies, D. Zufferey, and T. A. Henzinger. Forward analysis of depth-bounded processes. In *FOSSACS*, volume 6014 of *LNCS*, pages 94–108. Springer, 2010.