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 [14], 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 [17,18], wireless sensor networks [7], multi-core programming [16], and web services [5,6]. The underlying concurrent model of actor languages also forms the basis of the programming languages Erlang [4] and Scala [13] that have recently gained in popularity, in part due to their support for scalable concurrency.

In actor languages [2,14], 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 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 [11] – henceforth the decidability of termination. 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 [19]. 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 [12], 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 [10] for detecting deadlock in an actor-like subset of Creol [15] is based on suitable over-approximations.

Disclaimer. Due to space limitations, proofs have been removed; they are in [8].

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.

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 $A \in \mathtt{C}$ to say that A belongs to the partition of $\mathtt{C}$. 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:

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`

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

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) [20] – 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

$$
\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 $\mathit{Instruction\_i} = \mathsf{DecJump}(R_j,k)$;
- $\mathsf{0}$ 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)

again; (ii) the invocation is discarded because we are in the presence of a residual of a 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 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.

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

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

The undecidability of termination in $\texttt{Actor}_{\texttt{ba}}$ 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).

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

We show that $\texttt{Actor}^{\texttt{ro}}$ 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.

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

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 [1,11]. 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 $x_i \leq x_j$.

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

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 $\texttt{Actor}^{\texttt{ro}}$

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* $\mathsf{S}_1 \preceq \mathsf{S}'_1$ *such that* $\mathsf{S}_1 \longrightarrow \mathsf{S}_2$*, there exists* $\mathsf{S}'_1 \longrightarrow^* \mathsf{S}'_2$ *such that* $\mathsf{S}_2 \preceq \mathsf{S}'_2$*.*

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

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

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

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

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 $\texttt{Actor}_{\texttt{ba}}$ (and therefore in $\texttt{Actor}^{\texttt{ro}}_{\texttt{ba}}$). 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.*

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 [11]):

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

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 $\texttt{Actor}^{\texttt{ro}}_{\texttt{ba}}$. Then $(\mathcal{S}, \longrightarrow, \preceq)$ is a well-structured transition system.*

We notice that the well-structured transition system $(\mathcal{S}, \longrightarrow, \preceq)$ has transitive and stuttering compatibility (see [11], 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 [11].

Theorem 4. *In* $\texttt{Actor}^{\texttt{ro}}_{\texttt{ba}}$ *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 $\texttt{Actor}^{\texttt{ro}}_{\texttt{ba}}$ (due to the presence of fresh variables in method body instantiations). However, in this case, the sets $\mathit{Succ}(\texttt{S})$ and $\mathit{Pred}(\texttt{S})$ 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* $\texttt{Actor}^{\texttt{ro}}_{\texttt{ba}}$*, and let* $\texttt{S} \in \mathcal{S}$*. Then there is a finite set* $\mathcal{X} \subseteq \mathit{Pred}(\texttt{S})$ *such that, for every* $\texttt{S}' \in \mathit{Pred}(\texttt{S})$*, there is* $\texttt{T} \in \mathcal{X}$ *with* $\texttt{T} \preceq \texttt{S}'$*.* $\mathcal{X}$ *can be effectively computed.*

Lemma 2 and Theorem 4.8 in [11] allow us to decide the so-called *control-state reachability problem*: given two states $\texttt{S}$ and $\texttt{T}$ of a well-structured transition system with well-quasi ordering $\preceq$, decide whether there is $\texttt{T}' \succeq \texttt{T}$ such that $\texttt{S} \longrightarrow^* \texttt{T}'$.

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

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 $A \triangleright (P', \varphi, q), \texttt{S}$ is reachable with P' equal to P up-to renaming of variables and actor names, we proceed as follows. First, consider a configuration $\texttt{T}$ reachable after the complete execution of the main process. Therefore, in $\texttt{T}$, every possible actor has been created (with the corresponding initialization performed). Let $\texttt{T} = A_1 \triangleright (P_1, \varphi_1, q_1), \cdots, A_\ell \triangleright (P_\ell, \varphi_\ell, q_\ell)$. 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 $\texttt{T}$ to at least one of the states in the following finite set:

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

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 $\texttt{S}_n$ defined as follows:

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

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

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

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

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

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

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

We prove that in $\texttt{Actor}^{\texttt{sl}}$ 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 $\texttt{Actor}^{\texttt{sl}}$ 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 $\texttt{S} \longrightarrow_\alpha \texttt{S}'$ be the *abstract transition relation* defined as $\texttt{S} \longrightarrow \texttt{S}'$ 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:

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

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

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

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 $\texttt{S}$, we denote with $\alpha(\texttt{S})$ the configuration obtained from $\texttt{S}$ 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 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* $\alpha(\mathtt{S}) \longrightarrow_\alpha^* A'' \triangleright (P, \varnothing, q''), \mathtt{S}''$.

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

$$\rho \overset{\bullet}{=}_\alpha \rho' \quad \overset{def}{=} \quad \text{for every } x, y: \quad (i)\ \rho(x) = \rho(y) \iff \rho'(x) = \rho'(y) \quad (ii)\ \rho(x) \in \mathtt{C} \iff \rho'(x) \in \mathtt{C}$$

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.*

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 1) 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 $\mathtt{Actor}^{\mathtt{sl}}$, and let $\mathtt{S} \in \mathcal{S}$. Then there is a finite set $\mathcal{X}$ such that, for every $\mathtt{S}' \succeq_\alpha \mathtt{S}$ and $\mathtt{S}'' \in Pred(\mathtt{S}')$, there is $\mathtt{T} \in \mathcal{X}$ with $\mathtt{T} \preceq_\alpha \mathtt{S}''$. $\mathcal{X}$ can be effectively computed.*

It turns out that control-state reachability is decidable for the abstract transition system of $\mathtt{Actor}^{\mathtt{sl}}$. This entails the decidability of process reachability. In fact, given a process P, the reachability of a configuration $A \triangleright (P', \varphi, q), \mathtt{S}$ 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 $\mathtt{C}_1, \ldots, \mathtt{C}_n$ be the actor classes of the considered actor system and let $A_1, \cdots, A_n$ be such that $A_i \in \mathtt{C}_i$. 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 1, 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. de Boer, M. Jaghoori, C. Laneve, and G. Zavattaro. Decidability Problems for Actor Systems. Technical report, 2012. Available at `cs.unibo.it/~laneve`.
9. F. S. de Boer, I. Grabe, and M. Steffen. Termination detection for active objects. *Journal of Logic and Algebraic Programming*, 2012.
10. A. Finkel and P. Schnoebelen. Well-structured transition systems everywhere! *Theoretical Computer Science*, 256:63–92, 2001.
11. E. Giachino and C. Laneve. Analysis of deadlocks in object groups. In *FMOOD-S/FORTE*, pages 168–182, 2011.
12. P. Haller and M. Odersky. Scala actors: Unifying thread-based and event-based programming. *Theoretical Computer Science*, 410(2-3):202–220, 2009.
13. C. Hewitt. Procedural embedding of knowledge in planner. In *Proc. the 2nd International Joint Conference on Artificial Intelligence*, pages 167–184, 1971.
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. 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.