HAL
archives-ouvertes.fr

Decidability Problems for Actor Systems

F. de Boer, Mohammad Mahdi Jaghoori, Cosimo Laneve, Gianluigi Zavattaro

▶ **To cite this version:**

F. de Boer, Mohammad Mahdi Jaghoori, Cosimo Laneve, Gianluigi Zavattaro. Decidability Problems for Actor Systems. Logical Methods in Computer Science, Logical Methods in Computer Science Association, 2014, 104, pp.1 - 29. 10.2168/LMCS-10(4:5)2014 . hal-01090952

HAL Id: hal-01090952
https://hal.inria.fr/hal-01090952

Submitted on 4 Dec 2014

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Logical Methods in Computer Science
Vol. 10(4:5)2014, pp. 1–29
www.lmcs-online.org

Submitted Jun. 28, 2013
Published Dec. 4, 2014

DECIDABILITY PROBLEMS FOR ACTOR SYSTEMS *

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

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

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

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

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

1. Introduction

Since their introduction in [20], actor languages have evolved as a powerful computational model for defining distributed and concurrent systems [2, 3]. Languages based on actors have been also designed for modelling embedded systems [23, 24], wireless sensor networks [11, 31], multi-core programming [22], and web services [9, 10]. The underlying concurrent model of actor languages also forms the basis of the programming languages Erlang [4] and Scala [19] that have recently gained in popularity, in part due to their support for scalable concurrency.

2012 ACM CCS: [**Software and its engineering**]: Software notations and tools—General programming languages—Language types—Concurrent programming languages; [**Theory of computation**]: Semantics and reasoning—Program reasoning—Program verification.

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

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

[a−d] This work has been has been supported by the HATS Project No. FP7-231620 (Highly Adaptable and Trustworthy Software using Formal Models) and by the ENVISAGE Project No. FP7-610582 (Engineering Virtualized Services) of the EC.

LOGICAL METHODS IN COMPUTER SCIENCE
DOI:10.2168/LMCS-10(4:5)2014
© F. S. de Boer, M. M. Jaghoori, C. Laneve, and G. Zavattaro
Creative Commons

In actor languages [2, 20, 32], actors use a queue for storing the invocations to their methods in a FIFO manner. The queued invocations are processed sequentially by executing the corresponding method bodies. The encapsulated memory of an actor is represented by a finite number of *fields* that can be read and set by its methods and as such exist throughout its life time.

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

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

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

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

Related Works. There exists a vast body of related work on decidability of infinite-state systems (see [1]). However, to the best of our knowledge, the specific characteristics of the

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

2. The language Actor

Four disjoint infinite sets of names are used: *actor classes*, ranged over by $\mathtt{C}$, $\mathtt{D}$, $\cdots$, *method names*, ranged over by m, m', n, n', $\cdots$, *field names*, ranged over by $\mathtt{f}$, $\mathtt{g}$, $\cdots$, and *variables*, ranged over by 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

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

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, denoted by $\mathtt{0}$, or a field update $(\mathtt{f} \leftarrowtail E).P$, 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 $(\mathtt{f} \leftarrowtail x).\mathtt{0}$ will be always shortened into $(\mathtt{f} \leftarrowtail x)$. We will also shorten $[E = E']P;\mathtt{0}$ into $[E = E']P$. Following the tradition of process calculi like

CCS [27], we model sequences of actions by exploiting the so-called prefix notation (like, e.g., in $(\mathtt{f} \leftarrowtail E).P$).

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 $\mathit{free}(P)$ be the set of variables of P that are not bound. As usual, we identify processes P and P' that are equal up-to *alpha-conversion* of bound names, written $P =_\alpha P'$. The substitution operation $P[{}^{y}/_{x}]$ returns the process P where the free occurrences of x are replaced by y. The substitution operation $P[{}^{\widetilde{y}}/_{\widetilde{x}}]$ returns the process P where the free occurrences of $\widetilde{x}$ are *simultaneously* replaced by $\widetilde{y}$. In case, alpha-renaming is required for avoiding name clashes. For example $\mathtt{let}\ z = x\ \mathtt{in}\ \mathtt{new}\ \mathtt{C}(y,z)[{}^{z,\,u}/_{x,\,y}]$ returns $\mathtt{let}\ z' = z\ \mathtt{in}\ \mathtt{new}\ \mathtt{C}(u,z')$.

In the following examples and encodings we shorten $\mathtt{let}\ x = \mathtt{f}\ \mathtt{in}\ x\,!\,m(\widetilde{E}).P$ into $\mathtt{f}\,!\,m(\widetilde{E}).P$ (we have preferred the simpler syntax to ease the descriptions).

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 C, m has at most one definition;
(2) *correct*, namely, let $\mathit{fields}(\cdot)$ 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 $\mathit{fields}(\mathtt{C})$ 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 $\mathit{fields}(\mathtt{C})$.

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.

2.1. **Examples.** To illustrate the features of the actor language we discuss four examples.

Example 2.1. The *merger* service is an application that forwards to a back office server the values carried by two invocations of clients to methods *first* and *second*. The *merger* freezes the forward as long as there is no invocation of either *first* or *second* – in the process calculi community, the *merger* is modelled by a join pattern [17]. In our actor language the *merger* is modelled by a class actor `Merger` with six fields: `t` and `f` storing the values *true* and *false*, respectively; `fst` and `snd` storing *true* or *false* according to the method *first* and *second* have been invoked or not, respectively; `g` storing the argument of the invocation; `srv` storing the actor name of the back office server.

The class actor `Merger` includes the following three methods:

$$\begin{array}{rcl}
\mathtt{Merger}.\mathit{init}(x) & = & (\mathtt{t} \leftarrowtail \mathit{tt}).(\mathtt{f} \leftarrowtail \mathit{ff}).(\mathtt{fst} \leftarrowtail \mathit{ff}).(\mathtt{snd} \leftarrowtail \mathit{ff}).(\mathtt{srv} \leftarrowtail x) \\[2ex]
\mathtt{Merger}.\mathit{first}(a) & = & [\mathtt{snd} = \mathtt{t}]\ \mathtt{srv}\,!\,m(a,\mathtt{g}).(\mathtt{snd} \leftarrowtail \mathtt{f});\\
 & & \quad [\mathtt{fst} = \mathtt{f}]\ (\mathtt{fst} \leftarrowtail \mathtt{t}).(\mathtt{g} \leftarrowtail a);\mathit{this}\,!\,\mathit{first}(a) \\[2ex]
\mathtt{Merger}.\mathit{second}(a) & = & [\mathtt{fst} = \mathtt{t}]\ \mathtt{srv}\,!\,m(\mathtt{g},a).(\mathtt{fst} \leftarrowtail \mathtt{f});\\
 & & \quad [\mathtt{snd} = \mathtt{f}]\ (\mathtt{snd} \leftarrowtail \mathtt{t}).(\mathtt{g} \leftarrowtail a);\mathit{this}\,!\,\mathit{second}(a)
\end{array}$$

The method *init* manifests a basic feature of our actor language: the creation of new variables. In particular, *tt* and *ff* are free variables in the method definition of *init*. When *init* will be invoked, they will be replaced by two different fresh variables (said in pi-calculus jargon [28], we are assuming an implicit $(\nu\ \mathit{tt}, \mathit{ff})$- at the beginning of the method body).

These variables are stored in the fields `t` and `f`, respectively and they will be used to update the fields `fst` and `snd` appropriately . The merger forwards two messages to the server, which is stored in the field `srv`. These messages have been received through invocations of *first* and *second*. No forward occurs if one of the two methods has not been invoked. In particular, *first* (the method *second* behaves in a similar way) performs the forward if *second* has been evaluated *and* its message has not been already forwarded (field `snd` equal to *tt*); in this case the parameter of the invocation of *second* has been stored into `g`. Otherwise, if there is no previous invocation to *first*, still to be forwarded, (field `fst` equal to *ff*) then `fst` is set to *tt* and the parameter stored into `g`. There is a possibility that *first* is evaluated and a previous evaluation of *first* has still to be forwarded (field `fst` equal to *tt*). In this case, the invocation is bounced back (it is enqueued in the actor queue – see the operational semantics).

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

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

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

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

$$\begin{array}{rcl} \texttt{S}.\mathit{init}(\mathit{authServer}) & = & (\texttt{auth} \leftarrow\!\!\shortmid\ \mathit{authServer}) \\ \texttt{S}.\mathit{login}(\mathit{client}) & = & \texttt{auth}\,!\,\mathit{open}(\mathit{this}, \mathit{clientToken}, \mathit{authToken})\,. \\ & & \mathit{client}\,!\,\mathit{token}(\texttt{auth}, \mathit{authToken}) \\ \texttt{S}.\mathit{succeeds}(\mathit{authToken}) & = & \ldots \\ \texttt{S}.\mathit{fails}(\mathit{authToken}) & = & \ldots \end{array}$$

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

$$\begin{array}{rcl} \texttt{A}.\mathit{open}(\mathit{server}, \mathit{cToken}, \mathit{aToken}) & = & (\texttt{server} \leftarrow\!\!\shortmid\ \mathit{server})\,.\,(\texttt{ctoken} \leftarrow\!\!\shortmid\ \mathit{cToken})\,. \\ & & (\texttt{atoken} \leftarrow\!\!\shortmid\ \mathit{aToken}) \\ \texttt{A}.\mathit{authenticate}(u, p, \mathit{cToken}) & = & [\texttt{ctoken} = \mathit{cToken}]\,\texttt{server}\,!\,\mathit{succeeds}(\texttt{atoken})\,; \\ & & \qquad\qquad \texttt{server}\,!\,\mathit{fails}(\texttt{atoken}) \end{array}$$

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

$$\begin{array}{l} \texttt{let}\ \mathit{client} = \texttt{new C in} \\ \quad \texttt{let}\ \mathit{server} = \texttt{new S in} \\ \quad\quad \texttt{let}\ \mathit{authProvider} = \texttt{new A in} \\ \quad\quad\quad \mathit{client}\,!\,\mathit{init}(\mathit{server})\,.\,\mathit{server}\,!\,\mathit{init}(\mathit{authProvider}) \end{array}$$

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

// R has a field called `dec`, $Ctrl$ is an actor in the context

$$R.inc(pc, tt, ff) = R\,!\,item(tt, ff)\,.\,Ctrl\,!\,run(pc, tt, ff)$$
$$R.dec(pc, pc', tt, ff) = (\texttt{dec} \leftarrowtail tt)\,.\,R\,!\,checkzero(pc, pc', tt, ff)$$
$$R.checkzero(pc, pc', tt, ff) = [\texttt{dec} = tt]Ctrl\,!\,run(pc', tt, ff)\,.\,(\texttt{dec} \leftarrowtail ff)\,;\,Ctrl\,!\,run(pc, tt, ff)$$
$$R.init(tt, ff) = (\texttt{dec} \leftarrowtail ff)$$
$$R.item(tt, ff) = [\texttt{dec} = ff]R\,!\,item(tt, ff)\,;(\texttt{dec} \leftarrowtail ff)$$

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

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

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

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

// C has no fields

$$C.init(TManager) = TManager\,!\,tasksExec(t_1, \cdots, t_n)$$

The task manager is an instance of a class providing a method called *tasksExec* able to pass to the workers the description of their relative tasks:

// TM has n fields called W1, $\cdots$, Wn,

$$TM.tasksExec(p_1, \cdots, p_n) = \texttt{W1!}\, exec(p_1)\texttt{.} \cdots \texttt{.Wn!}\, exec(p_n)$$

Finally, a worker is an instance of a class W with one method *exec* able to execute the passed task (we leave this method unspecified):

// W has no fields and one method $exec(p)$

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

$$\begin{array}{l} \texttt{let}\ w_1 = \texttt{new W in} \quad \cdots \quad \texttt{let}\ w_n = \texttt{new W in} \\ \qquad \texttt{let}\ taskManager = \texttt{new TMs}(w_1, \cdots, w_n)\ \texttt{in} \\ \qquad\quad \texttt{let}\ client = \texttt{new C in}\ client\,!\,init(taskManager) \end{array}$$

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

// TM has no fields

$$\begin{array}{rl} TM.tasksExec(p_1, \cdots, p_n) = & this\,!\,nextTask(p_1, \cdots, p_n, x_1, x_2, \cdots, x_n, x_1) \\ TM.nextTask(p_1, \cdots, p_n, v_1, \cdots, v_{n+1}) = & [v_1 = v_2]\texttt{0;let}\ w = \texttt{new W in}\ w\,!\,exec(p_1)\texttt{.} \\ & this\,!\,nextTask(p_2, \cdots, p_n, p_1, v_2, \cdots, v_{n+1}, v_1) \end{array}$$

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

2.2. **The operational semantics.** The operational semantics of the language 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 extended 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, $\cdots$. Similarly for processes that, at run-time, may also have actor names. The extended set will be addressed by P. We notice that $free(P)$, when P belongs to this extended set, may returns variables *and* actor names. We will also apply $free(\cdot)$ to tuples of (extended) expressions: $free(\widetilde{E})$ returns the set of variables and actor names in $\widetilde{E}$.

The semantics is defined in terms of a *transition relation* $\texttt{S} \longrightarrow \texttt{S}'$, where S, 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 $fields(\texttt{C})$ to values, where $A \in \texttt{C}$, 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. As usual, let $\longrightarrow^*$ be the transitive closure of $\longrightarrow$ and $\longrightarrow^+$ be $\longrightarrow\longrightarrow^*$.

$$U \overset{\varphi}{\rightsquigarrow} U \;;\; \varnothing \qquad \mathtt{f} \overset{\varphi}{\rightsquigarrow} \varphi(\mathtt{f}) \;;\; \varnothing$$

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

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

Table 1: The evaluation relation $E \overset{\varphi}{\rightsquigarrow} U \;;\; \mathtt{S}$

The operational semantics of `Actor` is defined in Table 2, where the *evaluation function* $E \overset{\varphi}{\rightsquigarrow} U \;;\; \mathtt{S}$ is used (defined in Table 1). 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 $\mathit{fresh}(\cdot)$ used in the evaluation function takes a class actor and returns an actor name of that class that is fresh. In order to have a finitely branching transition system (see the remark at the end of this section), we assume actor names in classes are totally ordered and $\mathit{fresh}(\mathtt{C})$ always returns the first unused name of the class `C`. In this way, fresh names are selected in a deterministic manner. 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. Also in this case, we assume a total order of variables and $\mathit{fresh}(\widetilde{x})$, where $\widetilde{x}$ has length n, returns the least n unused variables. 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).

The initial configuration of a program with main process P 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 *finitely branching* (every state has a finite number of successor states) because the choices of fresh actor names (in the evaluation of `new C`) and of fresh variables (in the instantiation of the bodies of methods) are deterministic.

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

- $\mathtt{Actor}_{\mathtt{ba}}$ 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 ($\mathtt{f} \leftarrowtail E$) (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).

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

(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)
$$A \triangleright (P + Q, \varphi, q) \;\longrightarrow\; A \triangleright (P, \varphi, q)$$

(PLUS-R)
$$A \triangleright (P + Q, \varphi, q) \;\longrightarrow\; A \triangleright (Q, \varphi, q)$$

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

Table 2: The transition relation $\mathtt{S} \longrightarrow \mathtt{S}'$

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

We will use a reduction technique from a Turing-complete model to our actor model. In particular, the Turing-complete models we consider are the 2 Counter Machines (2CMs) [29] and a faulty variation of them. 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. The transition relation of the 2CM will be denoted by $\Longmapsto$.

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

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

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

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.

$$
\begin{array}{rl}
\texttt{R} & \texttt{// R has fields dec, ctr, loop and stop} \\
\texttt{R}.\mathit{item}(\mathit{tt},\mathit{ff}) & = [\texttt{stop} = \mathit{ff}]\big([\texttt{dec} = \mathit{ff}]\mathit{this}\,!\,\mathit{item}(\mathit{tt},\mathit{ff})\,;(\texttt{dec} \leftarrowtail \mathit{ff})\big) \\
\texttt{R}.\mathit{inc}(\mathit{pc},\mathit{tt},\mathit{ff}) & = [\texttt{stop} = \mathit{ff}](\texttt{loop} \leftarrowtail \mathit{ff})\,.\,\mathit{this}\,!\,\mathit{item}(\mathit{tt},\mathit{ff})\,.\,\texttt{ctr}\,!\,\mathit{run}(\mathit{pc},\mathit{tt},\mathit{ff}) \\
\texttt{R}.\mathit{decjump}(\mathit{pc},\mathit{pc}',\mathit{tt},\mathit{ff}) & = [\texttt{stop} = \mathit{ff}](\texttt{loop} \leftarrowtail \mathit{ff})\,.(\texttt{dec} \leftarrowtail \mathit{tt})\,.\,\mathit{this}\,!\,\mathit{checkzero}(\mathit{pc},\mathit{pc}',\mathit{tt},\mathit{ff}) \\
\texttt{R}.\mathit{checkzero}(\mathit{pc},\mathit{pc}',\mathit{tt},\mathit{ff}) & = [\texttt{stop} = \mathit{ff}](\texttt{loop} \leftarrowtail \mathit{ff})\,. \\
 & \quad \big([\texttt{dec} = \mathit{tt}]\texttt{ctr}\,!\,\mathit{run}(\mathit{pc}',\mathit{tt},\mathit{ff})\,.(\texttt{dec} \leftarrowtail \mathit{ff})\,;\texttt{ctr}\,!\,\mathit{run}(\mathit{pc},\mathit{tt},\mathit{ff})\big) \\
\texttt{R}.\mathit{init}(\mathit{tt},\mathit{ff},\mathit{Ctrl}) & = (\texttt{dec} \leftarrowtail \mathit{ff})\,.(\texttt{ctr} \leftarrowtail \mathit{Ctrl})\,.(\texttt{loop} \leftarrowtail \mathit{ff})\,.(\texttt{stop} \leftarrowtail \mathit{ff})\,. \\
 & \quad \mathit{this}\,!\,\mathit{bottom}(\mathit{tt},\mathit{ff}) \\
\texttt{R}.\mathit{bottom}(\mathit{tt},\mathit{ff}) & = [\texttt{loop} = \mathit{ff}](\texttt{loop} \leftarrowtail \mathit{tt}).\mathit{this}\,!\,\mathit{bottom}(\mathit{tt},\mathit{ff});(\texttt{stop} \leftarrowtail \mathit{tt})
\end{array}
$$

$$
\begin{array}{rl}
\texttt{Ctrl} & \texttt{// Ctrl has fields } \texttt{stm}_1\texttt{, } \cdots\texttt{, } \texttt{stm}_n \texttt{ and } \texttt{r}_1 \texttt{ and } \texttt{r}_2 \\
\texttt{Ctrl}.\mathit{run}(\mathit{pc},\mathit{tt},\mathit{ff}) & = [\mathit{pc} = \texttt{stm}_1][\![\mathit{Instruction\_1}]\!]_{1,\mathit{tt},\mathit{ff}} \\
 & \quad \cdots \\
 & \quad [\mathit{pc} = \texttt{stm}_n][\![\mathit{Instruction\_n}]\!]_{n,\mathit{tt},\mathit{ff}} \\
\texttt{Ctrl}.\mathit{init}() & = \texttt{r}_1\,!\,\mathit{init}(\mathit{tt},\mathit{ff},\mathit{this})\,.\,\texttt{r}_2\,!\,\mathit{init}(\mathit{tt},\mathit{ff},\mathit{this})\,.\,\mathit{this}\,!\,\mathit{run}(\texttt{stm}_1,\mathit{tt},\mathit{ff}) \\
\texttt{Ctrl}.\mathit{halt}() & = \texttt{0}
\end{array}
$$

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 $\mathit{Instruction\_i} = \mathsf{DecJump}(R_j,k)$;
- $\mathit{this}\,!\,\mathit{halt}$ if $\mathit{Instruction\_i} = \mathsf{Halt}$.

The main process is

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

Figure 1: Encoding a 2FTM in $\texttt{Actor}_{\texttt{ba}}$ ("_" denotes an irrelevant initialization parameter)

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

In case (ii), registers have a field `dec` that is set to 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* is 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.

The Halt instruction is simulated by invoking a method *halt* that does nothing.

As in the examples of Section 2, booleans are implemented by two variables – see the method $\texttt{Ctrl}.\mathit{init}$ – that are distributed during the invocations. With a similar machinery, in the actor class Ctrl, the labels of the instructions are 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 3.2. *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 2FTM diverges if and only if the corresponding actor program has an infinite computation. As regards process reachability, we need a smooth refinement of the encoding in Figure 1 where the Halt instruction is simulated by a specific process.

Proof. Let us assume to have a fixed 2FTM and let $[\![\mathit{Instruction\_i}]\!]_{i,tt,ff}$ be defined in Figure 1. Let also

$$
\begin{array}{rcll}
[\![(i,v_1,v_2)]\!]_{tt,ff} & \stackrel{def}{=} & \aleph \triangleright (0,\varnothing,\varepsilon), & (v_1 \neq \bot \text{ and } v_2 \neq \bot)\\
& & C \triangleright ([\![\mathit{Instruction\_i}]\!]_{i,tt,ff}, \varphi_{\texttt{ctrl}}, \varepsilon),\\
& & R_1 \triangleright (0, \varphi_{\texttt{reg}}, \mathit{bottom}(tt,ff)\cdot \mathit{item}(tt,ff)^{v_1}),\\
& & R_2 \triangleright (0, \varphi_{\texttt{reg}}, \mathit{bottom}(tt,ff)\cdot \mathit{item}(tt,ff)^{v_2})\\
[\![(i,\bot,v)]\!]_{tt,ff} & \stackrel{def}{=} & \aleph \triangleright (0,\varnothing,\varepsilon), & (v \neq \bot)\\
& & C \triangleright ([\![\mathit{Instruction\_i}]\!]_{i,tt,ff}, \varphi_{\texttt{ctrl}}, \varepsilon),\\
& & R_1 \triangleright (0, \varphi^{\bot}_{\texttt{reg}}, \varepsilon),\\
& & R_2 \triangleright (0, \varphi_{\texttt{reg}}, \mathit{bottom}(tt,ff)\cdot \mathit{item}(tt,ff)^{v})\\
[\![(0,\bot,v)]\!]_{tt,ff} & \stackrel{def}{=} & \aleph \triangleright (0,\varnothing,\varepsilon), & (v \neq \bot)\\
& & C \triangleright (0, \varphi_{\texttt{ctrl}}, \varepsilon),\\
& & R_1 \triangleright (0, \varphi^{\bot}_{\texttt{reg}}, \varepsilon),\\
& & R_2 \triangleright (0, \varphi_{\texttt{reg}}, \mathit{bottom}(tt,ff)\cdot \mathit{item}(tt,ff)^{v})
\end{array}
$$

where

- $\varphi_{\texttt{ctrl}} \stackrel{def}{=} [\texttt{stm}_i \mapsto x_i{}^{i\in 1..n}, \texttt{r}_1 \mapsto R_1, \texttt{r}_2 \mapsto R_2]$;
- $\varphi_{\texttt{reg}} \stackrel{def}{=} [\texttt{stop} \mapsto ff, \texttt{loop} \mapsto ff, \texttt{dec} \mapsto ff, \texttt{ctr} \mapsto C]$;
- $\varphi^{\bot}_{\texttt{reg}} \stackrel{def}{=} [\texttt{stop} \mapsto tt, \texttt{loop} \mapsto tt, \texttt{dec} \mapsto ff, \texttt{ctr} \mapsto C]$;
- $\mathit{item}(tt,ff)^{v} \stackrel{def}{=} \underbrace{\mathit{item}(tt,ff)\cdots\mathit{item}(tt,ff)}_{v \text{ times}}$.

(the definitions of $[\![(i,v,\bot)]\!]_{tt,ff}$ and $[\![(0,v,\bot)]\!]_{tt,ff}$ and $[\![(0,\bot,\bot)]\!]_{tt,ff}$ follow the same patterns).

We first observe that $\aleph \triangleright (P,\varnothing,\varepsilon) \longrightarrow^* [\![(1,0,0)]\!]_{tt,ff}$, where P is the main process in Figure 1. Then we demonstrate the following properties:

(1) if $(i,v_1,v_2) \Longrightarrow_{\texttt{F}} (j,v_1',v_2')$ then $[\![(i,v_1,v_2)]\!]_{tt,ff} \longrightarrow^+ [\![(j,v_1',v_2')]\!]_{tt,ff}$;

(2) if $[\![(i,v_1,v_2)]\!]_{tt,ff}$ has an infinite computation then the computation has infinitely many configurations like

$$\aleph \triangleright (0,\varnothing,\varepsilon), C \triangleright ([\![\mathit{Instruction\_i}]\!]_{i,tt,ff}, \varphi_{\texttt{ctrl}}, \varepsilon), R_1 \triangleright (P_1,\varphi_1,q_1), R_2 \triangleright (P_2,\varphi_2,q_2) \quad (3.1)$$

(3) if

$$\aleph \triangleright (0, \varnothing, \varepsilon), C \triangleright (\llbracket \mathit{Instruction\_i} \rrbracket_{i,tt,ff}, \varphi_{\texttt{ctrl}}, \varepsilon), R_1 \triangleright (P_1, \varphi_1, q_1), R_2 \triangleright (P_2, \varphi_2, q_2)$$
$$\longrightarrow^*$$
$$\aleph \triangleright (0, \varnothing, \varepsilon), C \triangleright (\llbracket \mathit{Instruction\_j} \rrbracket_{j,tt,ff}, \varphi_{\texttt{ctrl}}, \varepsilon), R_1 \triangleright (P_1', \varphi_1', q_1'), R_2 \triangleright (P_2', \varphi_2', q_2')$$

is a computation with every intermediate configuration having the process of the actor C equal to `0` then there are two computations

$$\aleph \triangleright (0, \varnothing, \varepsilon), C \triangleright (\llbracket \mathit{Instruction\_i} \rrbracket_{i,tt,ff}, \varphi_{\texttt{ctrl}}, \varepsilon), R_1 \triangleright (P_1, \varphi_1, q_1), R_2 \triangleright (P_2, \varphi_2, q_2)$$
$$\longrightarrow^* \quad \llbracket (i, v_1, v_2) \rrbracket_{tt,ff}$$

and

$$\aleph \triangleright (0, \varnothing, \varepsilon), C \triangleright (\llbracket \mathit{Instruction\_j} \rrbracket_{j,tt,ff}, \varphi_{\texttt{ctrl}}, \varepsilon), R_1 \triangleright (P_1', \varphi_1', q_1'), R_2 \triangleright (P_2', \varphi_2', q_2')$$
$$\longrightarrow^* \quad \llbracket (j, v_1', v_2') \rrbracket_{tt,ff}$$

where the actor C never moves such that

$$(i, v_1, v_2) \Longrightarrow_{\texttt{F}} (j, v_1', v_2') \; .$$

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

The proof of (2) uses an argument by contradiction. Assume that there are finitely many configurations like 3.1. Then, there is an infinite suffix of this computation in which the actor C does not perform actions. This means that at least one actor R_i performs infinitely many actions by executing the methods *item* and *bottom*. But this is not possible because *bottom* blocks the actor if it is invoked twice without executing update actions on the register in between.

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

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

3.2. **The language $\texttt{Actor}^{\texttt{ro}}$.** We show that $\texttt{Actor}^{\texttt{ro}}$ is Turing-complete by means of an encoding of a 2CM – see Figure 2. In this encoding the two registers are represented by two disjoint stacks of actors linked by the `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 3.3. *Termination and process reachability are undecidable in* $\texttt{Actor}^{\texttt{ro}}$.

Proof. It is easy to verify that if a 2CM has a computation

$$(1,0,0) \Longmapsto (i_1, v_1, v_1') \Longmapsto \cdots \Longmapsto (i_n, v_n, v_n')$$

then there is a computation

$$\begin{array}{rl} \aleph \triangleright (P, \emptyset, \varepsilon) \longrightarrow^* & \aleph \triangleright (\mathtt{0}, \emptyset, \varepsilon), C \triangleright ([\![\mathit{Instruction}_1]\!]_{nil,nil}, \varphi, \varepsilon) \\ \longrightarrow^* & \aleph \triangleright (\mathtt{0}, \emptyset, \varepsilon), C \triangleright ([\![\mathit{Instruction\_i}_1]\!]_{r_1,r_1'}, \varphi, \varepsilon), \mathcal{R}_{r_1}, \mathcal{R}_{r_1'}, \mathcal{G}_1 \\ \longrightarrow^* & \aleph \triangleright (\mathtt{0}, \emptyset, \varepsilon), C \triangleright ([\![\mathit{Instruction\_i}_n]\!]_{r_n,r_n'}, \varphi, \varepsilon), \mathcal{R}_{r_n}, \mathcal{R}_{r_n'}, \mathcal{G}_n \end{array}$$

where P is the main process of Figure 2, $[\![\mathit{Instruction\_i}_j]\!]_{r_j,r_j'}$ are defined in Figure 2, $\varphi = [\mathtt{stm}_1 \mapsto x_1, \cdots, \mathtt{stm}_n \mapsto x_n, \mathtt{nil} \mapsto \mathit{nil}]$, $\mathcal{R}_{r_i}$ and $\mathcal{R}_{r_i'}$ are stacks of register actors whose length is v_i and v_i', respectively. For instance, $\mathcal{R}_{r_1}$ of length k is

$$r_1 \triangleright (\mathtt{0}, [\mathtt{next} \mapsto r_2], \varepsilon), r_2 \triangleright (\mathtt{0}, [\mathtt{next} \mapsto r_3], \varepsilon), \cdots, r_k \triangleright (\mathtt{0}, [\mathtt{next} \mapsto \mathit{nil}], \varepsilon)\ .$$

The configurations $\mathcal{G}_i$ only contain register terms $r \triangleright (\mathtt{0}, [\mathtt{next} \mapsto r'], \varepsilon)$ and represent *garbage* (they are inactive).

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

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

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, 16]. This will allow us to decide a number of properties, such as termination. We begin with some background on well-structured transition systems.

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

Definition 4.1. A *well-structured transition system* is a finitely branching 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 $\mathtt{S}_1 \preceq \mathtt{S}_1'$ and $\mathtt{S}_1 \longrightarrow \mathtt{S}_2$, there exists $\mathtt{S}_1' \longrightarrow^* \mathtt{S}_2'$ such that $\mathtt{S}_2 \preceq \mathtt{S}_2'$.

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

According to the theory of well-structured transition systems [1, 16], we have that several properties are decidable for such transition systems (under some conditions discussed below). We will consider the following properties.

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

where $[\![Instruction\_i]\!]_{r_1,r_2}$ 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 $Instruction\_i = \mathsf{DecJump}(R_2, k)$;
- $\texttt{0}$ if $Instruction\_i = \mathsf{Halt}$.

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

Figure 2: Encoding a 2CM in $\texttt{Actor}^{\texttt{ro}}$

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

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

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

$$
\rho \overset{\bullet}{=} \rho' \quad \overset{def}{=} \quad \text{for every } x, y: \quad (i)\ \ \rho(x) = \rho(y) \quad \text{if and only if} \quad \rho'(x) = \rho'(y)
$$
$$
(ii)\ \ \rho(x) = \rho'(x) \quad \text{if } \rho(x) \text{ or } \rho'(x) \text{ is an actor name or a free variable of the main process}
$$

Namely, two renamings are in the relation $\overset{\bullet}{=}$ 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. By definition,

renamings in relation according to $\stackrel{\bullet}{=}$ never apply to free variables of the main process. This because these variables are possibly stored in fields of actors and their renaming might change the behaviours of actors in a way that breaks the upward compatibility of the following relation $\preceq$ and $\longrightarrow$ (*c.f.* proof of Theorem 4.4, part **(2)**). From this also follows that the above renamings *do not change the main process* (because they do not apply to its free variables).

Let $dom(\rho)$ be the domain of the renaming ρ. We denote by $P\rho$ the result of $P[^{\rho(\widetilde{x})}/_{\widetilde{x}}]$, where $\widetilde{x} = x_1, \cdots, x_n$ is a tuple containing the variables in $dom(\rho)$ (without repetitions) and $\rho(\widetilde{x}) = \rho(x_1), \cdots, \rho(x_n)$.

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 it identifies 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 4.3. *Let T be either a method invocation $m(U_1, \cdots, U_n)$ or a process 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 $free(P)$ that are different according to $\stackrel{\bullet}{=}$. In fact, the values of renamings on variables that are different from $free(P)$ 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 $free(P)$: 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 $\stackrel{\bullet}{=}$ if they yield different partitions. The number of such renaming is the *Bell number* of the cardinality of $free(P)$, let it be $\mathtt{Bell}(\kappa)$, where κ is the cardinality of $free(P)$. 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 $\stackrel{\bullet}{=}$. 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 $\stackrel{\bullet}{=}$ is finite.

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

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

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

Then, let

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

It is worth noticing 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. For this reason there are finitely many of them and the relation $\preceq$ is parametric with respect to them.

Theorem 4.4. *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.* 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 of states in $\mathcal{S}$ such that, for every $i < j$, $\mathtt{S}_i \not\preceq \mathtt{S}_j$. Let $y_1, \cdots, y_m$ be a sequence of variables not free in the main process. Consider:

$$\begin{array}{rll} \mathtt{subterms}(\mathtt{C}) = & \{P & | \quad \text{there exists a method } m \text{ s.t. } P \text{ is a subterm of } \mathtt{C}.m(\widetilde{x})\} \cup \\ & \{m(y_1, \cdots, y_g) & | \quad \text{there exists a method } m \text{ s.t. } \mathtt{C}.m(\widetilde{x}) \text{ with } |\widetilde{x}| = g\} \end{array}$$

The set $\mathtt{subterms}(\mathtt{C})$ is finite, but all the –possibly infinitely many– processes that can be executed (or the messages that can be received) by an actor of class $\mathtt{C}$ are renamings $P\rho$ (or $m(\widetilde{y})\rho$) of these terms. Notice that by Lemma 4.3, the number of terms $P\rho$ (and $m(\widetilde{y})\rho$) which are different according to $\simeq$ is finite as well. It is thus 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, in the elements $A \triangleright (P^A_{i_j}\rho_{i_j}, \varphi^A_{i_j}, q^A_{i_j})$ and $A \triangleright (P^A_{i_k}\rho_{i_k}, \varphi^A_{i_k}, q^A_{i_k})$ of $\mathtt{S}_{i_j}$ and $\mathtt{S}_{i_k}$, respectively, we have that $P^A_{i_j}\rho_{i_j} \simeq P^A_{i_k}\rho_{i_k}$. Moreover, as we are considering $\mathtt{Actor}^{\mathtt{ro}}_{\mathtt{ba}}$ the actor state cannot be modified, hence $\varphi^A_{i_j} = \varphi^A_{i_k}$.

As we are considering $\mathtt{Actor}^{\mathtt{ro}}_{\mathtt{ba}}$, the set of actor is bound. Let $A_1, \cdots, A_\ell$ be such actor names. 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:

$$(q^{A_1}_{i_1}, \cdots, q^{A_\ell}_{i_1}),\ (q^{A_1}_{i_2}, \cdots, q^{A_\ell}_{i_2}),\ (q^{A_1}_{i_3}, \cdots, q^{A_\ell}_{i_3}),\ \cdots$$

such that $\mathtt{S}_{i_j} \preceq \mathtt{S}_{i_k}$ if and only if $(q^{A_1}_{i_j}, \cdots, q^{A_\ell}_{i_j}) \sqsubseteq^\ell (q^{A_1}_{i_k}, \cdots, q^{A_\ell}_{i_k})$, 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 tuples of queues such that every tuple cannot be in relation according to $\sqsubseteq^\ell$ with any of the subsequent ones. This fact contradicts the

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

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

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

and from the following statement

• if $(X, \leq)$ is a well-quasi-ordering then $(X^\ell, \leq^\ell)$ is a well-quasi-ordering.

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

We first demonstrate that, if $\widetilde{E} \stackrel{\varphi}{\rightsquigarrow} \widetilde{U}\ ;\ \mathtt{S}$ with φ normed, then

(exp-i) if $\widetilde{E} \simeq \widetilde{E'}$ and $\mathtt{S} = \varnothing$ then $\widetilde{E'} \stackrel{\varphi}{\rightsquigarrow} \widetilde{U'}\ ;\ \varnothing$ and $\widetilde{U} \simeq \widetilde{U'}$;

(exp-ii) if $\widetilde{E}$ (respectively P) only contain free variables in the main process and actor names then $\widetilde{E} \simeq \widetilde{E}'$ (respectively $P \simeq P'$) implies $\widetilde{E} = \widetilde{E}'$ (respectively $P =_\alpha P'$) and $\widetilde{E} \stackrel{\varnothing}{\rightsquigarrow} \widetilde{U}\ ;\ \mathtt{S}$ implies that $\widetilde{U}$ contain free variables in the main process and actor names and $\mathtt{S}$ is normed.

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

(exp-ii) is an immediate consequence of the definition of $\stackrel{\bullet}{=}$ and $\stackrel{\varnothing}{\rightsquigarrow}$.

Let $\mathtt{S}_1 \longrightarrow \mathtt{S}_2$. We demonstrate that

(i) if $\mathtt{S}_1$ is normed then $\mathtt{S}_2$ is normed as well (this means that the transition system $(\mathcal{S}, \longrightarrow)$ of a program of $\mathtt{Actor}^{\mathtt{ro}}_{\mathtt{ba}}$ has normed configurations because the initial state is normed);

(ii) if $\mathtt{S}_1 \preceq \mathtt{S}'_1$ then there exists $\mathtt{S}'_1 \longrightarrow^* \mathtt{S}'_2$ such that $\mathtt{S}_2 \preceq \mathtt{S}'_2$.

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

As regards *(ii)*, its proof is a case analysis on the proof-tree of $\mathtt{S}_1 \longrightarrow \mathtt{S}_2$ where the cases correspond to the unique rule appearing in the tree that is not an instance of (CONTEXT). 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 cases are discussed in order.

(1) $\mathtt{S}_1 \longrightarrow \mathtt{S}_2$ contains an instance of (LET), namely

$$A \triangleright (\mathtt{let}\ x = E\ \mathtt{in}\ P, \varphi, q),\ \longrightarrow\ A \triangleright (P[^U/_x], \varphi, q), \mathtt{S}_3\ .$$

where $E \stackrel{\varphi}{\rightsquigarrow} U\ ;\ \mathtt{S}$. By $\mathtt{S}_1 \preceq \mathtt{S}'_1$, $\mathtt{S}'_1$ must contain $A \triangleright (\mathtt{let}\ x = E'\ \mathtt{in}\ P', \varnothing, \varepsilon)$ such that $\mathtt{let}\ x = E\ \mathtt{in}\ P \simeq \mathtt{let}\ x = E'\ \mathtt{in}\ P'$ (without loss of generality, we are assuming the two bound variables are the same) and $q \leq q'$. There are two subcases: (1.1) $A = \aleph$ and (1.2) $A \neq \aleph$. In (1.1), By *(exp-ii)*, this is possible provided $E = E'$ and $P =_\alpha P'$. It is easy to verify that $\mathtt{S}'_1 \longrightarrow \mathtt{S}'_2$ and $\mathtt{S}_2 \preceq \mathtt{S}'_2$ because their unique difference with $\mathtt{S}_1$ and $\mathtt{S}'_1$ is due to the two processes P and P'. In (1.2), $\mathtt{S}_3 = \varnothing$ because no `new` can occur in E. Additionally, by definition of $\simeq$, $E \simeq E'$ and $P \simeq P'$. Let $E' \stackrel{\varphi}{\rightsquigarrow} U', \varnothing$. By *(exp-i)* we have $U \simeq U'$ and it is easy to verify that $P[^U/_x] \simeq P'[^{U'}/_x]$. Henceforth $\mathtt{S}'_1 \longrightarrow \mathtt{S}'_2$ and $\mathtt{S}_2 \preceq \mathtt{S}'_2$ because their unique difference with $\mathtt{S}_1$ and $\mathtt{S}'_1$ is due to the two processes $P[^U/_x]$ and $P'[^{U'}/_x]$.

(2) $\mathtt{S}_1 \longrightarrow \mathtt{S}_2$ contains an instance of (INVK-S), namely

$$A \triangleright (A\,!\,m(\widetilde{E})\,.\,P, \varphi, q), \;\longrightarrow\; A \triangleright (P, \varphi, q \cdot m(\widetilde{U}))\;.$$

Since $\mathtt{S}_1 \preceq \mathtt{S}_1'$ then $\mathtt{S}_1'$ contains $A \triangleright (A\,!\,m(\widetilde{E'})\,.\,P', \varphi, q')$ with $\widetilde{E} \simeq \widetilde{E'}$, $P \simeq P'$, and $q \leq q'$. We observe that $A \neq \aleph$ and if $\widetilde{E} \overset{\varphi}{\rightsquigarrow} \widetilde{U}, \varnothing$ and $\widetilde{E'} \overset{\varphi}{\rightsquigarrow} \widetilde{U'}, \varnothing$ then $\widetilde{U} \simeq \widetilde{U'}$ by *(exp-i)*. Therefore $q \cdot m(\widetilde{U}) \leq q' \cdot m(\widetilde{U'})$ and $\mathtt{S}_1' \longrightarrow \mathtt{S}_2'$ with $\mathtt{S}_2 \preceq \mathtt{S}_2'$ because their unique difference with $\mathtt{S}_1$ and $\mathtt{S}_1'$ is due to the two terms $A \triangleright (P, \varphi, q \cdot m(\widetilde{U}))$ and $A \triangleright (P', \varphi, q' \cdot m(\widetilde{U'}))$.

(3) $\mathtt{S}_1 \longrightarrow \mathtt{S}_2$ contains an instance of (INVK), namely

$$A \triangleright (B\,!\,m(\widetilde{E})\,.\,P, \varphi, q), B \triangleright (Q, \psi, p) \;\longrightarrow\; A \triangleright (P, \varphi, q), B \triangleright (Q, \psi, p \cdot m(\widetilde{U})), \mathtt{S}_3\;.$$

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

(4) $\mathtt{S}_1 \longrightarrow \mathtt{S}_2$ contains an instance of (INST), namely

$$A \triangleright (\mathtt{0}, \varphi, m(\widetilde{U}) \cdot q) \longrightarrow A \triangleright (P[{}^{A}/_{\mathit{this}}][\widetilde{y'}/\widetilde{y}][\widetilde{U}/\widetilde{x}], \varphi, q)\;,$$

where $\mathtt{C}\,.\,m(\widetilde{x}) = P$, $\mathtt{C}$ 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 $\mathtt{S}_2 = A \triangleright (P[{}^{A}/_{\mathit{this}}][\widetilde{y'}/\widetilde{y}][\widetilde{U}/\widetilde{x}], q), \mathtt{T}_1$. 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 $\mathtt{S}_1' \longrightarrow^* A \triangleright (\mathtt{0}, \varphi, m(\widetilde{V}) \cdot q' \cdot q''), \mathtt{T}_1''$ by performing transitions of the actor A, with $\mathtt{T}_1' \preceq \mathtt{T}_1''$ and, by definition, $q \leq q' \cdot q''$. At this stage, we notice that $A \triangleright (\mathtt{0}, \varphi, m(\widetilde{V}) \cdot q' \cdot q''), \mathtt{T}_1'' \longrightarrow A \triangleright (P[{}^{A}/_{\mathit{this}}][\widetilde{z}/\widetilde{y}][\widetilde{V}/\widetilde{x}], \varphi, q' \cdot q''), \mathtt{T}_1''$. We notice that $P[{}^{A}/_{\mathit{this}}][\widetilde{y'}/\widetilde{y}][\widetilde{U}/\widetilde{x}] = P[{}^{A}/_{\mathit{this}}][\widetilde{y} \mapsto \widetilde{y'}, \widetilde{x} \mapsto \widetilde{U}]$ and $P[{}^{A}/_{\mathit{this}}][\widetilde{z}/\widetilde{y}][\widetilde{V}/\widetilde{x}] = P[{}^{A}/_{\mathit{this}}][\widetilde{y} \mapsto \widetilde{z}, \widetilde{x} \mapsto \widetilde{V}]$ and $[\widetilde{y} \mapsto \widetilde{y'}, \widetilde{x} \mapsto \widetilde{U}] \overset{\bullet}{=} [\widetilde{y} \mapsto \widetilde{z}, \widetilde{x} \mapsto \widetilde{V}]$. Therefore

$$P[{}^{A}/_{\mathit{this}}][\widetilde{z}/\widetilde{y}][\widetilde{V}/\widetilde{x}] \simeq P[{}^{A}/_{\mathit{this}}][\widetilde{z}/\widetilde{y}][\widetilde{V}/\widetilde{x}]$$

which implies that $\mathtt{S}_1' \longrightarrow^* \longrightarrow \mathtt{S}_2'$ and $\mathtt{S}_2 \preceq \mathtt{S}_2'$ because their unique difference with $\mathtt{S}_1$ and $\mathtt{S}_1'$ is due to the two above processes.

(5) $\mathtt{S}_1 \longrightarrow \mathtt{S}_2$ contains an instance of (MATCH), namely

$$A \triangleright ([E = E']P\,;Q, \varphi, q) \;\longrightarrow\; A \triangleright (P, \varphi, q), \mathtt{S}_3\;.$$

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

(6) $\mathtt{S}_1 \longrightarrow \mathtt{S}_2$ contains an instance of (MMATCH). Similar to (5).

(7) $\mathtt{S}_1 \longrightarrow \mathtt{S}_2$ contains an instance of (PLUS-L) or of (PLUS-R). Straightforward. □

We notice that the well-structured transition system $(\mathcal{S}, \longrightarrow, \preceq)$ has decidable algorithms for computing $\preceq$ and for computing the next states. Then decidability of termination directly follows from the above mentioned results of the theory of well-structured transition systems that we have previously recalled.

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

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

Lemma 4.6. *Let* $(\mathcal{S}, \longrightarrow, \preceq)$ *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 4.6 and the above mentioned results on well-structured transition systems allow us to decide the *control-state reachability problem*.

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

Proof. Let $\uparrow \mathtt{S} = \{\mathtt{S}' \in \mathcal{S} \mid \mathtt{S} \preceq \mathtt{S}'\}$. Let also $\mathit{Pred}(\uparrow \mathtt{S}) = \{\mathtt{T} \mid \mathtt{T} \longrightarrow \mathtt{S}' \text{ and } \mathtt{S}' \succeq \mathtt{S}\}$. 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 4.6 that is effectively computable. □

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

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

It is worth noticing that the language `Actor` also owns the more classical diamond property: if in a configuration there are two transitions inferred by two distinct actors, then it is possible to perform them in any order reaching the same configurations *up-to bijective renaming*. We omit the formalization of this property since it is not needed in the rest of the paper.

Corollary 4.9. *The process reachability problem is decidable in* $\mathtt{Actor}^{\mathtt{ro}}_{\mathtt{ba}}$.

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

$$\{\ \aleph \triangleright (\mathtt{0}, \varnothing, \varepsilon), A_1 \triangleright (Q_1[{}^{A_1}/_{this}][{}^{\widetilde{z}_1}/_{\widetilde{y}_1}][{}^{\widetilde{U_1}}/_{\widetilde{x_1}}], \varphi_1, \varepsilon), \cdots, A_\ell \triangleright (Q_\ell[{}^{A_\ell}/_{this}][{}^{\widetilde{z}_\ell}/_{\widetilde{y}_\ell}][{}^{\widetilde{U_\ell}}/_{\widetilde{x}_\ell}], \varphi_\ell, \varepsilon)$$
$$\mid \textit{ for every } 1 \leq i \leq \ell,\ Q_i \textit{ is a suffix of the body of } \mathtt{m}_i \textit{ in } \mathtt{C}_i,\ \textit{where } A_i \in \mathtt{C}_i,$$
$$\textit{formal parameters and free variables of } \mathtt{m}_i \textit{ are } \widetilde{x}_i \textit{ and } \widetilde{y}_i$$
$$\widetilde{U}_i \textit{ is a tuple in } \{A_1, \ldots, A_\ell, \widetilde{u}, \widetilde{z}\} \quad (\widetilde{z}, \widetilde{z}_1, \cdots, \widetilde{z}_\ell \textit{ are fresh})$$
$$\textit{there exists } 1 \leq j \leq \ell \textit{ such that } Q_j \textit{ is equal to } P \textit{ up-to renaming } \}$$

Then the corollary follows by Theorem 4.7. □

We conclude this section by recalling that we have already proved the undecidability of termination in programs with unboundedly many actors and read-only fields. Note that if we remove from $\mathtt{Actor}^{\mathtt{ro}}_{\mathtt{ba}}$ the constraint on bounded actor names then the relation $\preceq$ is no longer a well-quasi-ordering. Consider, for instance, an actor (with empty state) having a method that first creates a new instance of the same class and then invokes on this new instance the same method. Among the reachable configurations it is possible to select a sequence $\mathtt{S}_1, \mathtt{S}_2, \mathtt{S}_3, \cdots$ such that the configuration $\mathtt{S}_n$ is defined as follows:

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

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

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

The *decorated evaluation relation* $E \rightsquigarrow_d U \ ; \ \mathtt{S}$:

$$U \rightsquigarrow_d U \ ; \ \varnothing \qquad \frac{\widetilde{E} \rightsquigarrow_d \widetilde{U} \ ; \ \mathtt{S} \quad A = \mathit{fresh}(\mathtt{C})}{\mathtt{new}\ \mathtt{C}(\widetilde{E}) \rightsquigarrow_d A \ ; \ A \triangleright (\varepsilon : \mathtt{0}, \varnothing, \varepsilon), \mathtt{S}}$$

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

The *decorated transition relation* $\mathtt{S} \xrightarrow{\sigma} \mathtt{S}'$:

$$\text{(LET}_d\text{)}\ \frac{E \rightsquigarrow_d U \ ; \ \mathtt{S}}{A \triangleright (\sigma \cdot n : \mathtt{let}\ x = E\ \mathtt{in}\ P, \varnothing, q) \xrightarrow{\sigma \cdot n+1} A \triangleright (\sigma \cdot n+1 : P[^U/_x], \varnothing, q), \mathtt{S}}$$

$$\text{(INVK-S}_d\text{)}\ \frac{\widetilde{E} \rightsquigarrow_d \widetilde{U} \ ; \ \mathtt{S}}{A \triangleright (\sigma \cdot n : A\,!\,m(\widetilde{E})\,.\,P, \varnothing, q) \xrightarrow{\sigma \cdot n+1 | m(\widetilde{U}, A)} A \triangleright (\sigma \cdot n+1 : P, \varnothing, q \cdot m(\widetilde{U}, \sigma \cdot n+1)), \mathtt{S}}$$

$$\text{(INVK}_d\text{)}\ \frac{\widetilde{E} \rightsquigarrow_d \widetilde{U} \ ; \ \mathtt{S}}{A \triangleright (\sigma \cdot n : A'\,!\,m(\widetilde{E})\,.\,P, \varnothing, q), A' \triangleright (\sigma' : P', \varnothing, q') \xrightarrow{\sigma \cdot n+1 | m(\widetilde{U}, A')} A \triangleright (\sigma \cdot n+1 : P, \varnothing, q), A' \triangleright (\sigma' : P', \varnothing, q' \cdot m(\widetilde{U}, \sigma \cdot n+1)), \mathtt{S}}$$

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

$$\text{(MATCH}_d\text{)}\ \frac{E, E' \rightsquigarrow_d U, U \ ; \ \mathtt{S}}{A \triangleright (\sigma \cdot n : [E = E']P\,;Q, \varnothing, q) \xrightarrow{\sigma \cdot n+1} A \triangleright (\sigma \cdot n+1 : P, \varnothing, q), \mathtt{S}}$$

$$\text{(MMATCH}_d\text{)}\ \frac{E, E' \rightsquigarrow_d U, V \ ; \ \mathtt{S} \quad U \neq V}{A \triangleright (\sigma \cdot n : [E = E']P\,;Q, \varnothing, q) \xrightarrow{\sigma \cdot n+1} A \triangleright (\sigma \cdot n+1 : Q, \varnothing, q), \mathtt{S}}$$

$$\text{(PLUS-L}_d\text{)}\ A \triangleright (\sigma \cdot n : P + Q, \varnothing, q) \xrightarrow{\sigma \cdot n+1} A \triangleright (\sigma \cdot n+1 : P, \varnothing, q)$$

$$\text{(PLUS-R}_d\text{)}\ A \triangleright (\sigma \cdot n : P + Q, \varnothing, q) \xrightarrow{\sigma \cdot n+1} A \triangleright (\sigma \cdot n+1 : Q, \varnothing, q)$$

$$\text{(CONTEXT}_d\text{)}\ \frac{\mathtt{S} \xrightarrow{\sigma} \mathtt{S}'}{\mathtt{S}, \mathtt{S}'' \xrightarrow{\sigma} \mathtt{S}', \mathtt{S}''}$$

Table 3: The decorated operational semantics of the language $\mathtt{Actor}^{\mathtt{sl}}$

identifies the action of emission of the message that caused the method instantiation from which P was generated, and n is a counter indicating that P is actually generated by the method instantiation after n steps. Notice that for the main process executed by the actor $\aleph$ the sequence σ' is empty, and that when a method is instantiated the counter n is initialized

$$\text{(INVK-S}_a\text{)}\quad \frac{\widetilde{E} \rightsquigarrow_d \widetilde{U}\ ;\ \mathtt{S} \quad A, A' \in \mathtt{C}}{A \triangleright (\sigma \cdot n : A' \,!\, m(\widetilde{E})\,.\,P, \varnothing, q) \ \stackrel{\sigma\cdot n+1 | m(\widetilde{U}, A')}{\longrightarrow_{\mathtt{a}}}\ A \triangleright (\sigma \cdot n + 1 : P, \varnothing, q \cdot m(\widetilde{U}, \sigma \cdot n + 1, A')), \mathtt{S}}$$

$$\text{(INVK}_a\text{)}\quad \frac{\widetilde{E} \rightsquigarrow_d \widetilde{U}\ ;\ \mathtt{S} \qquad A', A'' \in \mathtt{C}}{\begin{array}{l} A \triangleright (\sigma \cdot n : A' \,!\, m(\widetilde{E})\,.\,P, \varnothing, q), A'' \triangleright (\sigma' : P', \varnothing, q') \\ \qquad \stackrel{\sigma\cdot n+1 | m(\widetilde{U}, A')}{\longrightarrow_{\mathtt{a}}}\ A \triangleright (\sigma \cdot n + 1 : P, \varnothing, q), A'' \triangleright (\sigma' : P', \varnothing, q' \cdot m(\widetilde{U}, \sigma \cdot n + 1, A')), \mathtt{S} \end{array}}$$

$$\text{(INST}_a\text{)}\quad \frac{A \in \mathtt{C} \quad \mathtt{C}\,.\,m(\widetilde{x}) = P \quad \widetilde{y} = \mathit{free}(P) \setminus \widetilde{x} \quad \widetilde{y'} = \mathit{fresh}(\widetilde{y})}{A \triangleright (\mathtt{0}, \varnothing, m(\widetilde{U}, \sigma, A') \cdot q) \ \stackrel{\sigma\cdot 0}{\longrightarrow_{\mathtt{a}}}\ A \triangleright (\sigma \cdot 0 : P[{}^{A'}/_{\mathit{this}}][{}^{\widetilde{y'}}/_{\widetilde{y}}][{}^{\widetilde{U}}/_{\widetilde{x}}], \varnothing, q)}$$

Table 4: Abstract transition rules for method invocations and instantiations

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

It is trivial to see that the operational semantics in Table 3 and the decorated semantics coincide, in the sense that given one configuration $\mathtt{S}_1$ of $\mathtt{Actor}^{\mathtt{sl}}$ we have $\mathtt{S}_1 \longrightarrow \mathtt{S}_2$ if and only if there exist a label α and two decorations $\mathtt{S}'_1$ and $\mathtt{S}'_2$ of $\mathtt{S}_1$ and $\mathtt{S}_2$, respectively, such that $\mathtt{S}'_1 \stackrel{\alpha}{\longrightarrow} \mathtt{S}'_2$.

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

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

We now introduce few notations:

– Let $\Omega()$ be a map from "concrete" to "abstract" configurations: given a configuration $\mathtt{S}$, we denote with $\Omega(\mathtt{S})$ the configuration obtained from $\mathtt{S}$ by replacing each of its actors

$A \triangleright (\sigma : P, \varnothing, q)$ with $A \triangleright (\sigma : P, \varnothing, q')$ where q' is obtained from q by adding to each method invocation the parameter A.

– Given a decorated configuration $\mathtt{S}$ and a label α, such that $\alpha = \sigma$ or $\alpha = \sigma|m(\widetilde{U}, A)$, we use $\mathtt{S}\lfloor_\alpha$ to denote the process decorated with σ in $\mathtt{S}$: $\mathtt{S}\lfloor_\alpha = P$ if $\mathtt{S}$ contains the actor $A \triangleright (\sigma : P, \varnothing, q)$, for some A and q.

– Let $\overset{\bullet}{=}_{\mathsf{a}}$ be the following relation on variable renamings (not applied to variables that are free in the main process)

$$\rho \overset{\bullet}{=}_{\mathsf{a}} \rho' \quad \overset{def}{=} \quad \text{for every } x, y:$$

$$\begin{array}{lll} (i) & \rho(x) = \rho(y) & \text{if and only if} \quad \rho'(x) = \rho'(y) \\ (ii) & \rho(x) \in \mathtt{C} & \text{if and only if} \quad \rho'(x) \in \mathtt{C} \\ (iii) & \rho(x) = \rho'(x) & \text{if } \rho(x) \text{ or } \rho'(x) \text{ is a free variable of the main process} \end{array}$$

Differently from the definition of $\overset{\bullet}{=}$, $\overset{\bullet}{=}_{\mathsf{a}}$ does not care of the identity of actor names (it is sufficient that they belong to the same class).

– Let $\simeq_{\mathsf{a}}$ be the relation defined as $\simeq$ in Section 4, with $\overset{\bullet}{=}_{\mathsf{a}}$ instead of $\overset{\bullet}{=}$. We extend it to messages containing sequences and actor names as follows: $m(\widetilde{U}, \sigma, A) \simeq_{\mathsf{a}} m(\widetilde{U}', \sigma', A')$ iff $m(\widetilde{U}) \simeq_{\mathsf{a}} m(\widetilde{U}')$, $\sigma = \sigma'$ and there exists $\mathtt{C}$ such that $A, A' \in \mathtt{C}$. We extend it also to labels: $\sigma \simeq_{\mathsf{a}} \sigma$ and $\sigma|m(\widetilde{U}, A) \simeq_{\mathsf{a}} \sigma|m(\widetilde{U}', A')$ iff $m(\widetilde{U}) \simeq_{\mathsf{a}} m(\widetilde{U}')$ and there exists $\mathtt{C}$ such that $A, A' \in \mathtt{C}$.

The following Proposition formalizes the correspondence between $\xrightarrow{\alpha}$ and $\xrightarrow{\alpha}_{\mathsf{a}}$: *1.* all $\xrightarrow{\alpha}$ transitions are present also in $\xrightarrow{\alpha}_{\mathsf{a}}$ (up-to application of the abstraction function $\Omega()$ to configurations) and *2.* all the abstract computations $\mathtt{S}_0 \xrightarrow{\alpha_1}_{\mathsf{a}} \ldots \xrightarrow{\alpha_n}_{\mathsf{a}} \mathtt{S}_n$ have a corresponding concrete computation $\mathtt{S}'_0 \xrightarrow{\alpha'_1} \ldots \xrightarrow{\alpha'_m} \mathtt{S}'_m$ in which they can be embedded.

Proposition 5.1. *Let* $\mathtt{S}$ *and* $\mathtt{S}_0$ *be a decorated configuration and an initial decorated configuration* $\aleph \triangleright (0 : P, \varnothing, \varepsilon)$ *of* $\mathtt{Actor}^{\mathtt{sl}}$*, respectively.*

(1) *If* $\mathtt{S} \xrightarrow{\alpha} \mathtt{S}'$ *then* $\Omega(\mathtt{S}) \xrightarrow{\alpha}_{\mathsf{a}} \Omega(\mathtt{S}')$*;*

(2) *if* $\mathtt{S}_0 \xrightarrow{\alpha_1}_{\mathsf{a}} \ldots \xrightarrow{\alpha_n}_{\mathsf{a}} \mathtt{S}_n$ *then there exists a computation* $\mathtt{S}'_0 \xrightarrow{\alpha'_1} \mathtt{S}'_1 \ldots \xrightarrow{\alpha'_m} \mathtt{S}'_m$ *and an injection* I *such that, for all* $1 \leq i \leq n$*, we have* $\alpha_i \simeq_{\mathsf{a}} \alpha'_{I(i)}$ *and* $\mathtt{S}_i\lfloor_{\alpha_i} \simeq_{\mathsf{a}} \mathtt{S}'_{I(i)}\lfloor_{\alpha'_{I(i)}}$*.*

Proof. The first item trivially holds because the new rules used in the definition of $\xrightarrow{\alpha}_{\mathsf{a}}$ are (strictly) more general than the corresponding rules used in the definition of $\xrightarrow{\alpha}$.

The second item is proved by induction on the length of the computation $\mathtt{S}_0 \xrightarrow{\alpha_1}_{\mathsf{a}} \ldots \xrightarrow{\alpha_n}_{\mathsf{a}} \mathtt{S}_n$.

If $n = 1$ then $\mathtt{S}_0 \xrightarrow{\alpha_1}_{\mathsf{a}} \mathtt{S}_1$ with $\alpha_1 = 1$ or $\alpha_1 = 1|m(\widetilde{U}, A)$ and $\mathtt{S}_1\lfloor_\alpha = P'$, where P' is an immediate derivative of the main process P. It is trivial to see that the same transition is present in the concrete decorated semantics: namely, $\aleph \triangleright (0 : P, \varnothing, \varepsilon) \xrightarrow{\alpha'_1} \mathtt{S}'_1$ with $\alpha_1 \simeq_{\mathsf{a}} \alpha'_1$ and $\mathtt{S}'_1\lfloor_{\alpha'_1} \simeq_{\mathsf{a}} P'$.

If $n > 1$ we consider $\mathtt{S}_0 \xrightarrow{\alpha_1}_{\mathsf{a}} \ldots \xrightarrow{\alpha_n}_{\mathsf{a}} \mathtt{S}_n \xrightarrow{\alpha_{n+1}}_{\mathsf{a}} \mathtt{S}_{n+1}$. The inductive hypothesis guarantees the existence of the concrete computation $\mathtt{S}'_0 \xrightarrow{\alpha'_1} \ldots \xrightarrow{\alpha'_m} \mathtt{S}'_m$ and of the injection I such that, for all $1 \leq i \leq n$, we have $\alpha_i \simeq_{\mathsf{a}} \alpha'_{I(i)}$ and $\mathtt{S}_i\lfloor_{\alpha_i} \simeq_{\mathsf{a}} \mathtt{S}'_{I(i)}\lfloor_{\alpha'_{I(i)}}$. We now proceed by case analysis on the last number of the sequence in α_{n+1}.

If the number is 0, then $\alpha_{n+1} = \sigma \cdot 0$ and the transition is obtained by applying rule (INST$_a$) on a message $m(\widetilde{U}, \sigma, A')$. The presence of this message in one of the queues in

$\mathtt{S}_n$ guarantees the existence of $1 \leq j \leq n$ such that $\alpha_j = \sigma | m(\widetilde{U}, A')$. In the concrete computation we have $\alpha'_{I(j)} \simeq_{\mathsf{a}} \sigma | m(\widetilde{U}, A')$. This means that the same message (up-to $\simeq_{\mathsf{a}}$) is in the queue of an actor A'' such that $A', A'' \in \mathtt{C}$ in the concrete state $\mathtt{S}'_{I(j)}$. We have two subcases: either such method invocation is instantiated by the actor A'' during the concrete computation $\mathtt{S}'_0 \xrightarrow{\alpha'_1} \ldots \xrightarrow{\alpha'_m} \mathtt{S}'_m$ or not. In the first case, there exists $I(j) < l \leq m$ such that $\alpha'_l = \sigma \cdot 0$ that instantiates the method. The thesis is proved simply by extending the injection I with $I(n+1) = l$ and observing that $\mathtt{S}_{n+1}\!\downarrow_{\sigma\cdot 0} \simeq_{\mathsf{a}} \mathtt{S}'_l\!\downarrow_{\sigma\cdot 0}$. If the method invocation was not instantiated, it is in the queue of the actor A'' in the configuration $\mathtt{S}'_m$. It is sufficient to apply the same reasoning on an extension of the concrete computation that consumes the messages in front of the method invocation with sequence σ and that finally instantiates it. Such extension exists because processes are finite and non-blocking.

If the number is not 0, we discuss only the case in which $\alpha_{n+1} = \sigma \cdot k$ (with $k > 0$) because the case $\alpha = \sigma \cdot k | m(\widetilde{U}, A)$ is treated similarly. In the computation $\mathtt{S}_0 \xrightarrow{\alpha_1}_{\mathsf{a}} \ldots \xrightarrow{\alpha_n}_{\mathsf{a}} \mathtt{S}_n$ is guaranteed the presence of a label containing $\sigma \cdot k - 1$, i.e. there exists $1 \leq j \leq n$ such that the label α_j contains $\sigma \cdot k - 1$. The process $\mathtt{S}_j\!\downarrow_{\alpha_j}$ is the process that has just performed the action labeled with $\sigma \cdot k - 1$ and that performs the action in the transition $\mathtt{S}_n \xrightarrow{\alpha_{n+1}}_{\mathsf{a}} \mathtt{S}_{n+1}$ because $\alpha_{n+1} = \sigma \cdot k$. By inductive hypothesis $\mathtt{S}_j\!\downarrow_{\alpha_j} \simeq_{\mathsf{a}} \mathtt{S}'_{I(j)}\!\downarrow_{\alpha'_{I(j)}}$ hence a process ready to execute an action labeled with the sequence $\sigma \cdot k$ occurs also in the concrete state $\mathtt{S}'_{I(j)}$. We now consider two subcases.

- There exists no label α'_l containing $\sigma \cdot k$. In this case the process $\mathtt{S}'_{I(j)}\!\downarrow_{\alpha'_{I(j)}}$ still occurs in $\mathtt{S}'_m$. Hence it is possible to extend the computation $\mathtt{S}'_0 \xrightarrow{\alpha'_1} \ldots \xrightarrow{\alpha'_m} \mathtt{S}'_m$ with $\mathtt{S}'_m \xrightarrow{\sigma\cdot k} \mathtt{S}'_{m+1}$ in such a way that $\mathtt{S}'_{m+1}\!\downarrow_{\sigma\cdot k} \simeq_{\mathsf{a}} \mathtt{S}_{n+1}\!\downarrow_{\sigma\cdot k}$. The thesis is proved simply by extending the injection I with $I(n+1) = m+1$.
- There exists $I(j) < l \leq m$ such that α'_l contains $\sigma \cdot k$. In this case it is not guaranteed that $\mathtt{S}'_l\!\downarrow_{\alpha'_l} \simeq_{\mathsf{a}} \mathtt{S}_{n+1}\!\downarrow_{\sigma\cdot k}$, due to nondeterminism. For this reason we construct from $\mathtt{S}'_0 \xrightarrow{\alpha'_1} \ldots \xrightarrow{\alpha'_m} \mathtt{S}'_m$ another concrete computation that satisfies our thesis. The first transformation that we apply to $\mathtt{S}'_0 \xrightarrow{\alpha'_1} \ldots \xrightarrow{\alpha'_m} \mathtt{S}'_m$ consists of the cancellation of the transition α'_l and of all the other transitions that depends on it. Namely, there are two kinds of transitions that must be removed: (i) those labeled with a sequence having a prefix $\sigma \cdot r$ such that $r \geq k$ and (ii) those causally dependent on the instantiation of messages that are in $\mathtt{S}'_l$ inside the queue of the actor containing the process decorated with $\sigma \cdot k$. Let $\mathtt{S}'_0 \xrightarrow{\alpha'_1} \ldots \xrightarrow{\alpha'_{l-1}} \mathtt{S}'_{l-1} \xrightarrow{\alpha''_l} \mathtt{S}''_l \ldots \xrightarrow{\alpha''_s} \mathtt{S}''_s$ be the concrete computation obtained after this elimination of transitions. We now extend such computation by letting the process labeled with $\sigma \cdot k - 1$ to execute the expected action labeled with $\sigma \cdot k$. Namely, we add the transition $\mathtt{S}''_s \xrightarrow{\sigma\cdot k} \mathtt{S}''_{s+1}$ such that $\mathtt{S}''_{s+1}\!\downarrow_{\sigma\cdot k} \simeq_{\mathsf{a}} \mathtt{S}_{n+1}\!\downarrow_{\sigma\cdot k}$. Then, we extend the computation by performing at least all the transitions removed for the reason (ii) above. This extension exists because all processes are finite and nonblocking and because the considered transitions causally depend on messages that are in $\mathtt{S}''_{s+1}$ inside the queue of the actor containing the process decorated with $\sigma \cdot k$. Let $\mathtt{S}'_0 \xrightarrow{\alpha'_1} \ldots \xrightarrow{\alpha'_{l-1}} \mathtt{S}'_{l-1} \xrightarrow{\alpha''_l} \mathtt{S}''_l \ldots \xrightarrow{\alpha''_s} \mathtt{S}''_s \xrightarrow{\alpha''_{s+1}} \ldots \xrightarrow{\alpha''_t} \mathtt{S}''_t$ be the obtained computation. The thesis is proved by considering this last concrete computation, a rearrangement of the injection I that maps to their new positions the transitions in its codomain that belong to the group (ii), and by extending it with $I(n+1) = l$. □

As a direct consequence we have that the abstract semantics preserves both termination and control-state reachability. It remains to prove that termination and process reachability is decidable for the abstract semantics. To this aim, we consider a transition system $\longrightarrow_{\mathsf{a}}$ obtained by removing the labels from the transitions $\xrightarrow{\alpha}_{\mathsf{a}}$. On this transition system we define $\preceq_{\mathsf{a}}$ as a variant of the ordering $\preceq$ defined in the previous section in such a way that $(\mathcal{S}, \longrightarrow_{\mathsf{a}}, \preceq_{\mathsf{a}})$ turns out to be a well-structured transition system (for configurations of stateless programs). Let:

– Let $\leq_{\mathsf{a}}$ be the following relation on message queues:

$$\frac{\textit{there exist } i_1 < i_2 < \cdots < i_k \leq h \textit{ s.t. for } j \in 1..k,\ \ m_j(\widetilde{U_j}, \sigma_j, A_j) \simeq_{\mathsf{a}} n_{i_j}(\widetilde{V_{i_j}}, \sigma'_{i_j}, A'_{i_j})}{m_1(\widetilde{U_1}, \sigma_1, A_1) \ldots m_k(\widetilde{U_k}, \sigma_k, A_k) \leq_{\mathsf{a}} n_1(\widetilde{V_1}, \sigma'_1, A'_1) \ldots n_h(\widetilde{V_h}, \sigma'_h, A'_h)}$$

– Let $\preceq_{\mathsf{a}}$ be the ordering:

$$\frac{A_i, A'_{j_i} \in \mathtt{C}_i \quad P_i \simeq_{\mathsf{a}} P'_{j_i} \ \text{ and } \ q_i \leq_{\mathsf{a}} q'_{j_i} \ \text{ for } i \in 1..\ell,\ 1 \leq j_1 < j_2 < \cdots < j_\ell \leq \kappa}{A_1 \triangleright (\sigma_1 : P_1, \varnothing, q_1), \cdots, A_\ell \triangleright (\sigma_\ell : P_\ell, \varnothing, q_\ell) \ \preceq_{\mathsf{a}} \ A'_1 \triangleright (\sigma'_1 : P'_1, \varnothing, q'_1), \cdots, A'_\kappa \triangleright (\sigma'_\kappa : P'_\kappa, \varnothing, q'_\kappa)}$$

Next, we observe that Lemma 4.3 can be adapted to the case of unbounded actors by using $\simeq_{\mathsf{a}}$ instead of $\simeq$. Namely, let T be either a process or a method invocation $m(U_1, \cdots, U_n, \sigma, A)$ 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 $T\rho_i \not\simeq_{\mathsf{a}} T\rho_j$. Proceeding as in the proof of Lemma 4.3, we prove that $\mathcal{T}$ is finite.

Theorem 5.2. *Given a stateless program $\mathcal{S}$ we have that $(\mathcal{S}, \longrightarrow_{\mathsf{a}}, \preceq_{\mathsf{a}})$ is a well-structured transition system.*

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

In part **(1)** the unique difference is in the last part where the coordinatewise order $\sqsubseteq^\ell$ on sequences (of length 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 $\sqsubseteq_{\mathsf{a}}$ defined as follows:

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

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

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

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

Lemma 5.3. *Let* $(\mathcal{S}, \longrightarrow_{\mathtt{a}}, \preceq_{\mathtt{a}})$ *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* $\mathtt{S}' \succeq_{\mathtt{a}} \mathtt{S}$ *and* $\mathtt{S}'' \in \mathit{Pred}(\mathtt{S}')$*, there is* $\mathtt{T} \in \mathcal{X}$ *with* $\mathtt{T} \preceq_{\mathtt{a}} \mathtt{S}''$*.* $\mathcal{X}$ *can be effectively computed.*

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

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

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

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

The decidability of control-state reachability entails the decidability of process reachability. In fact, given a process P, the reachability of a configuration $A \triangleright (P', \varphi, q), \mathtt{S}$ 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:

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

6. Conclusions

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

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

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

References

[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] A. Bouajjani, S. Fratani, and S. Qadeer. Context-bounded analysis of multithreaded programs with dynamic linked structures. In *CAV*, pages 207–220, 2007.

[6] D. Brand and P. Zafiropulo. On communicating finite-state machines. *J. ACM*, 30(2):323–342, Apr. 1983.

[7] E. Broch Johnsen and O. Owe. An asynchronous communication model for distributed concurrent objects. *Software and System Modeling*, 6(1):39–58, 2007.

[8] N. Busi, R. Gorrieri, and G. Zavattaro. On the expressiveness of linda coordination primitives. *Information and Computation*, 156(12):90 – 121, 2000.

[9] P.-H. Chang and G. Agha. Supporting reconfigurable object distribution for customized web applications. In *SAC*, pages 1286–1292, 2007.

[10] P.-H. Chang and G. Agha. Towards context-aware web applications. In *DAIS*, pages 239–252, 2007.

[11] 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.

[12] J. Cleese and C. Booth. Fawlty towers. See `en.wikipedia.org/wiki/Fawlty_Towers`, 1975.

[13] F. de Boer, M. Jaghoori, C. Laneve, and G. Zavattaro. Decidability Problems for Actor Systems. In *Concurrency Theory - 23rd International Conference, CONCUR 2012*, volume 7454 of *Lecture Notes in Computer Science*, pages 562–577. Springer, 2012.

[14] F. S. de Boer, D. Clarke, and E. Broch Johnsen. A complete guide to the future. In *ESOP*, pages 316–330, 2007.

[15] F. S. de Boer, I. Grabe, and M. Steffen. Termination detection for active objects. *Journal of Logic and Algebraic Programming*, 2012.
[16] A. Finkel and P. Schnoebelen. Well-structured transition systems everywhere! *Theoretical Computer Science*, 256:63–92, 2001.
[17] C. Fournet and G. Gonthier. The reflexive cham and the join-calculus. In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 372–385, New York, NY, USA, 1996. ACM.
[18] E. Giachino and C. Laneve. Analysis of deadlocks in object groups. In *FMOODS/FORTE*, pages 168–182, 2011.
[19] P. Haller and M. Odersky. Scala actors: Unifying thread-based and event-based programming. *Theoretical Computer Science*, 410(2-3):202–220, 2009.
[20] C. Hewitt. Procedural embedding of knowledge in planner. In *Proc. the 2nd International Joint Conference on Artificial Intelligence*, pages 167–184, 1971.
[21] G. Higman. Ordering by Divisibility in Abstract Algebras. *Proc. London Math. Soc.*, s3-2(1):326–336, 1952.
[22] R. K. Karmani, A. Shali, and G. Agha. Actor frameworks for the jvm platform: a comparative analysis. In *PPPJ*, pages 11–20. ACM, 2009.
[23] E. A. Lee, X. Liu, and S. Neuendorffer. Classes and inheritance in actor-oriented design. *ACM Transactions in Embedded Computing Systems*, 8(4), 2009.
[24] 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.
[25] G. Memmi and A. Finkel. An introduction to fifo nets monogeneous nets: A subclass of fifo nets. *Theoretical Computer Science*, 35(0):191 – 214, 1985.
[26] R. Meyer. On boundedness in depth in the pi-calculus. In *IFIP TCS*, volume 273 of *IFIP*, pages 477–489. Springer, 2008.
[27] R. Milner. *Communication and concurrency*. PHI Series in computer science. Prentice Hall, 1989.
[28] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, ii. *Inf. and Comput.*, 100:41–77, 1992.
[29] M. Minsky. *Computation: finite and infinite machines*. Prentice Hall, 1967.
[30] OpenID. Openid specifications. `http://openid.net/developers/specs/`.
[31] N. Razavi, R. Behjati, H. Sabouri, E. Khamespanah, A. Shali, and M. Sirjani. Sysfier: Actor-based formal verification of systemc. *ACM Trans. Embedded Comput. Syst.*, 10(2):19, 2010.
[32] M. Sirjani. Rebeca: Theory, applications, and tools. In *FMCO*, pages 102–126, 2006.
[33] 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.

This work is licensed under the Creative Commons Attribution-NoDerivs License. To view a copy of this license, visit `http://creativecommons.org/licenses/by-nd/2.0/` or send a letter to Creative Commons, 171 Second St, Suite 300, San Francisco, CA 94105, USA, or Eisenacher Strasse 2, 10777 Berlin, Germany