Adaptable Processes (Extended Abstract)

Mario Bravetti, Cinzia Di Giusto, Jorge Pérez, Gianluigi Zavattaro

Adaptable Processes (Extended Abstract)*

Mario Bravetti[1], Cinzia Di Giusto[2], Jorge A. Pérez[3], and Gianluigi Zavattaro[1]

[1] Laboratory FOCUS (Università di Bologna / INRIA), Italy
[2] INRIA Rhône-Alpes, Grenoble, France
[3] CITI - Dept. of Computer Science, FCT New University of Lisbon, Portugal

Abstract. We propose the concept of *adaptable processes* as a way of overcoming the limitations that process calculi have for describing patterns of dynamic *process evolution*. Such patterns rely on direct ways of controlling the behavior and location of *running* processes, and so they are at the heart of the *adaptation* capabilities present in many modern concurrent systems. Adaptable processes have a location and are sensible to actions of *dynamic update* at runtime. This allows to express a wide range of evolvability patterns for processes. We introduce a core calculus of adaptable processes and propose two verification problems for them: *bounded* and *eventual adaptation*. While the former ensures that at most k consecutive errors will arise in future states, the latter ensures that if the system enters into an error state then it will eventually reach a correct state. We study the (un)decidability of these two problems in different fragments of the calculus. Rather than a specification language, our calculus intends to be a basis for investigating the fundamental properties of evolvable processes and for developing richer languages with evolvability capabilities.

1 Introduction

Process calculi aim at describing formally the behavior of concurrent systems. A leading motivation in the development of process calculi has been properly capturing the *dynamic character* of concurrent behavior. In fact, much of the success of the π-calculus can be fairly attributed to the way it departs from CCS [16] so as to describe mobile systems in which communication topologies can change dynamically. Subsequent developments can be explained similarly. For instance, the introduction of the Ambient calculus can be justified by the need of describing mobility with considerations of space/context awareness, frequent in distributed systems. A commonality to these developments is that dynamic behavior in a system is realized through a number of *local changes* in its topology: mobility in the π-calculus arises from local changes in single linkages, while spatial mobility in the Ambient calculus arises from local changes in the containment relations in the hierarchy of ambients. This way, the combined effect of local changes suffices to explain dynamic behavior at the global (system) level.

There are, however, interesting forms of dynamic behavior that cannot be satisfactorily described as a combination of local changes, in the sense just discussed. These are

* Supported by the French project ANR-2010-SEGI-013 - AEOLUS, the EU integrated project HATS, the Fondation de Coopération Scientifique Digiteo Triangle de la Physique, and FCT / MCTES - Carnegie Mellon Portugal Program, grant NGN-44-2009-12 - INTERFACES.

behavioral patterns which concern change at the *process* level, and thus describe *process evolution* along time. In general, forms of process evolvability are characterized by an enhanced control/awareness over the current behavior and location of running processes. Remarkably, this increased control is central to the *adaptation* capabilities by which processes modify their behavior in response to exceptional circumstances in their environment. As a simple example, consider the behavior of a process scheduler in an operating system, which executes, suspends, and resumes a given set of threads. In order to model the scheduler, the threads, and their evolution, we would need mechanisms for *direct* process manipulation, which appear quite difficult to represent by means of link mobility only. More precisely, it is not clear at all how to represent the *intermittent evolution* of a thread under the scheduler's control: that is, precise ways of describing that its behavior "disappears" (when the scheduler suspends the thread) and "appears" (when the scheduler resumes the thread). Emerging applications and programming paradigms provide challenging examples of evolvable processes. In workflow applications, we would like to be able to replace or update a running activity, without affecting the rest of the workflow. We might also like to suspend the execution of a set of activities, or even suspend and relocate the whole workflow. Similarly, in component-based systems we would like to reconfigure parts of a component, a whole component, or even groups of components. Also, in cloud computing scenarios, applications rely on scaling policies that remove and add instances of computational power at runtime. These are context-aware policies that dynamically adapt the application to the user's demand. (We discuss these examples in detail in Section 3.) At the heart of these applications are patterns of process evolution and adaptation which we find very difficult (if not impossible) to represent in existing process calculi.

In an attempt to address these shortcomings, in this paper we introduce the concept of *adaptable processes*. Adaptable processes have a location and are sensible to actions of *dynamic update* at runtime. While locations are useful to designate and structure processes into hierarchies, dynamic update actions implement a sort of built-in adaptation mechanism. We illustrate this novel concept by introducing $\mathcal{E}$, a process calculus of adaptable processes (Section 2). The $\mathcal{E}$ calculus arises as a variant of CCS without restriction and relabeling, and extended with primitive notions of location and dynamic update. In $\mathcal{E}$, $a[P]$ denotes the adaptable process P located at a. Name a acts as a *transparent locality*: P can evolve on its own as well as interact freely with its surrounding environment. Localities can be nested, and are sensible to interactions with *update actions*. An update action $\widetilde{a}\{U\}$ decrees the update of the adaptable process at a with the behavior defined by U, a *context* with zero or more holes, denoted by $\bullet$. The *evolution* of $a[P]$ is realized by its interaction with the update action $\widetilde{a}\{U\}$, which leads to process $U\{^{P}/_{\bullet}\}$, i.e., the process obtained by replacing every hole in U by P.

Rather than a specification language, the $\mathcal{E}$ calculus intends to be a basis for investigating the fundamental properties of evolvable processes. In this presentation, we focus on two *verification problems* associated to $\mathcal{E}$ processes and their (un)decidability. The first one, *k-bounded adaptation* (abbreviated $\mathcal{BA}$) ensures that, given a finite k, at most k consecutive error states can arise in computations of the system—including those reachable as a result of dynamic updates. The second problem, *eventual adaptation* (abbreviated $\mathcal{EA}$), is similar but weaker: it ensures that if the system enters into an

error state then it will eventually reach a correct state. In addition to error occurrence, the correctness of adaptable processes must consider the fact that the number of modifications (i.e. update actions) that can be applied to the system is typically *unknown*. For this reason, we consider $\mathcal{BA}$ and $\mathcal{EA}$ in conjunction with the notion of *cluster* of adaptable processes. Given a system P and a set M of possible updates that can be applied to it at runtime, its associated cluster considers P together with an arbitrary number of instances of the updates in M. This way, a cluster formalizes adaptation and correctness properties of an initial system configuration (represented by an *aggregation* of adaptable processes) in the presence of arbitrarily many sources of update actions.

A summary of our main results follows. $\mathcal{E}$ is shown to be Turing complete, and both $\mathcal{BA}$ and $\mathcal{EA}$ are shown to be *undecidable* for $\mathcal{E}$ processes (Section 4). Turing completeness of $\mathcal{E}$ says much on the expressive power of update actions. In fact, it is known that fragments of CCS without restriction can be translated into finite Petri nets (see for instance the discussion in [7]), and so they are not Turing complete. Update actions in $\mathcal{E}$ thus allow to "jump" from finite Petri nets to a Turing complete model. We then move to $\mathcal{E}^-$, the fragment of $\mathcal{E}$ in which *update patterns*—the context U in $\widetilde{a}\{U\}$—are restricted in such a way that holes $\bullet$ cannot appear behind prefixes. In $\mathcal{E}^-$, $\mathcal{BA}$ turns out to be *decidable* (Section 5), while $\mathcal{EA}$ remains *undecidable* (Section 6). Interestingly, $\mathcal{EA}$ is already undecidable in two fragments of $\mathcal{E}^-$: one fragment in which adaptable processes cannot be neither removed nor created by update actions—thus characterizing systems in which the topology of nested adaptable processes is *static*—and another fragment in which update patterns are required to have exactly one hole—thus characterizing systems in which running processes cannot be neither deleted nor replicated.

The undecidability results are obtained via encodings of Minsky machines [17]. While the encoding in $\mathcal{E}$ *faithfully* simulates the behavior of the Minsky machine, this is not the case for the encoding in $\mathcal{E}^-$, in which only finitely many steps are wrongly simulated. The decidability of $\mathcal{BA}$ in $\mathcal{E}^-$ is proved by resorting to the theory of *well-structured transition systems* [1,13] and its associated results. In our case, such a theory must be coupled with Kruskal's theorem [15] (so as to deal with terms whose syntactical tree structure has an unbounded depth), and with the calculation of the predecessors of target terms in the context of trees with unbounded depth (so as to deal with arbitrary aggregations and dynamic updates that may generate new nested adaptable processes). This combination of techniques and results proved to be very challenging.

In Section 7 we give some concluding remarks and review related work. Detailed definitions and proofs can be found in [5].

2 The $\mathcal{E}$ Calculus: Syntax, Semantics, Adaptation Problems

The $\mathcal{E}$ calculus extends the fragment of CCS without restriction and relabeling (and with replication instead of recursion) with *update prefixes* $\widetilde{a}\{U\}$ and a primitive notion of *adaptable processes* $a[P]$. As in CCS, in $\mathcal{E}$ processes can perform actions or synchronize on them. We presuppose a countable set $\mathcal{N}$ of names, ranged over by $a, b \ldots$, possibly decorated as $\overline{a}, \overline{b} \ldots$ and $\widetilde{a}, \widetilde{b} \ldots$. As customary, we use a and $\overline{a}$ to denote atomic input and output actions, respectively.

$$\textsc{Comp}\ \ a[P] \xrightarrow{a[P]} \star \qquad \textsc{Sum}\ \ \textstyle\sum_{i\in I} \alpha_i.\, P_i \xrightarrow{\alpha_i} P_i \qquad \textsc{Repl}\ \ !\alpha.\, P \xrightarrow{\alpha} P \parallel\, !\alpha.\, P$$

$$\textsc{Act1}\ \ \frac{P_1 \xrightarrow{\alpha} P_1'}{P_1 \parallel P_2 \xrightarrow{\alpha} P_1' \parallel P_2} \qquad \textsc{Tau1}\ \ \frac{P_1 \xrightarrow{a} P_1' \qquad P_2 \xrightarrow{\overline{a}} P_2'}{P_1 \parallel P_2 \xrightarrow{\tau} P_1' \parallel P_2'}$$

$$\textsc{Loc}\ \ \frac{P \xrightarrow{\alpha} P'}{a[P] \xrightarrow{\alpha} a[P']} \qquad \textsc{Tau3}\ \ \frac{P_1 \xrightarrow{a[Q]} P_1' \qquad P_2 \xrightarrow{\widetilde{a}\{U\}} P_2'}{P_1 \parallel P_2 \xrightarrow{\tau} P_1'\{U\{^{Q}/_{\bullet}\}/_{\star}\} \parallel P_2'}$$

Fig. 1. LTS for $\mathcal{E}$. Symmetric counterparts of ACT1, TAU1, and TAU3 have been omitted.

Definition 1. *The class of $\mathcal{E}$ processes, is described by the following grammar:*

$$P ::= a[P] \ \mid\ P \parallel P \ \mid\ \sum_{i\in I} \pi_i.\, P_i \ \mid\ !\pi.\, P \qquad \pi ::= a \ \mid\ \overline{a} \ \mid\ \widetilde{a}\{U\}$$

where the U in $\widetilde{a}\{U\}$ represents a context*, i.e., a process with zero or more holes $\bullet$.*

In $\mathcal{E}$, $a[P]$ denotes the *adaptable process* P located at a. Notice that a acts as a *transparent* locality: process P can evolve on its own, and interact freely with external processes. Given $a[P]$, we often refer to P as the process *residing at* a. Adaptable processes can be *updated*: when an update prefix $\widetilde{a}\{U\}$ is able to interact with the adaptable process $a[P]$, the current state of the adaptable process at a is used to fill the holes in context U (see below). The rest of the syntax follows standard lines. Parallel composition $P \parallel Q$ decrees the concurrent execution of P and Q. A process $\pi.\, P$ performs prefix π and then behaves as P. Given an index set $I = \{1, \ldots, n\}$, the guarded sum $\sum_{i\in I} \pi_i.\, P_i$ represents an exclusive choice over $\pi_1.\, P_1, \ldots, \pi_n.\, P_n$. As usual, we write $\pi_1.\, P_1 + \pi_2.\, P_2$ if $|I| = 2$, $\pi_1.\, P_1$ if $|I| = 1$, and $\mathbf{0}$ if I is empty. Process $!\,\pi.\, P$ defines (guarded) replication, by which infinitely many copies of P are triggered by prefix π.

The operational semantics of $\mathcal{E}$ is given in terms of a Labeled Transition System (LTS). It is denoted $\xrightarrow{\alpha}$, and defined by the rules in Figure 1. There are five kinds of actions; we use α to range over them. In addition to the standard input, output, and τ actions, we consider two complementary actions for process update: $\widetilde{a}\{U\}$ and $a[P]$. The former represents the offering of an update U for the adaptable process at a; the latter expresses the fact that the adaptable process at a, with current state P, is ready to update. We often use $\longrightarrow$ to denote $\xrightarrow{\tau}$. Intuitions for some rules of the LTS follow. Rule COMP represents the contribution of a process at a in an update operation; we use $\star$ to denote a unique placeholder. Rule LOC formalizes the above mentioned transparency of localities. Process evolvability is formalized by rule TAU3 and its symmetric. The update action offers a process U for updating the process at a which, by virtue of rule COMP, is represented in P_1' by $\star$. Process Q—the current state of a— is then used to fill the holes in U that do not appear inside other update prefixes: we use $U\{^{Q}/_{\bullet}\}$ to denote the process U in which every occurrence of $\bullet$ has been replaced by Q in this way. The update action is completed by replacing all occurrences of $\star$ in P_1' with $U\{^{Q}/_{\bullet}\}$.

Notice that nested update prefixes are allowed in our language and treated consistently by the semantics. This way, for instance $\widetilde{a}\{\bullet \parallel \widetilde{b}\{\bullet\}\}$ is an allowed update prefix,

and the holes at a and at b are actually different. In fact, and as detailed above, the hole inside b is not instantiated in case of an update operation at a.

We move on to formally define the adaptation problems that we shall consider for $\mathcal{E}$ processes. They formalize the problem of reaching error states in a system with adaptable processes. Both problems are stated in terms of observability predicates, or *barbs*. Our definition of barbs is parameterized on the number of repetitions of a given signal.

Definition 2 (Barbs). *Let P be an $\mathcal{E}$ process, and let α be an action in $\{a, \overline{a} \mid a \in \mathcal{N}\}$. We write $P \downarrow_\alpha$ if there exists a P' such that $P \xrightarrow{\alpha} P'$. Moreover:*

- *Given $k > 0$, we write $P \Downarrow_\alpha^k$ iff there exist $Q_1, \ldots, Q_k$ such that $P \longrightarrow^* Q_1 \longrightarrow \ldots \longrightarrow Q_k$ with $Q_i \downarrow_\alpha$, for every $i \in \{1, \ldots, k\}$.*
- *We write $P \Downarrow_\alpha^\omega$ iff there exists an infinite computation $P \longrightarrow^* Q_1 \longrightarrow Q_2 \longrightarrow \ldots$ with $Q_i \downarrow_\alpha$ for every $i \in \mathbb{N}$.*

We use $\Downarrow_\alpha^k$ and $\Downarrow_\alpha^\omega$ to denote the negation of $\Downarrow_\alpha^k$ and $\Downarrow_\alpha^\omega$, with the expected meaning.

We consider two instances of the problem of reaching an error in an aggregation of terms, or *cluster*. A *cluster* is a process obtained as the parallel composition of an initial process P with an arbitrary set of processes M representing its possible subsequent modifications. That is, processes in M may contain update actions for the adaptable processes in P, and therefore may potentially lead to its modification (evolution).

Definition 3 (Cluster). *Let P be an $\mathcal{E}$ process and $M = \{P_1, \ldots, P_n\}$. The set of clusters is defined as: $\mathcal{CS}_P^M = \{P \parallel \prod^{m_1} P_1 \parallel \cdots \parallel \prod^{m_n} P_n \mid m_1, \ldots, m_n \in \mathbb{N}\}$.*

The *adaptation* problems below formalize correctness of clusters with respect to their ability for recovering from errors by means of update actions. More precisely, given a set of clusters $\mathcal{CS}_P^M$ and a barb e (signaling an error), we would like to know if all computations of processes in $\mathcal{CS}_P^M$ (1) have *at most* k consecutive states exhibiting e, or (2) *eventually* reach a state in which the barb on e is no longer observable.

Definition 4 (Adaptation Problems). *The* bounded adaptation *problem ($\mathcal{BA}$) consists in checking whether given an initial process P, a set of processes M, a barb e, and $k > 0$, for all $R \in \mathcal{CS}_P^M$, $R \Downarrow_e^k$ holds.*

Similarly, the eventual adaptation *problem ($\mathcal{EA}$) consists in checking whether given an initial process P, a set of processes M and a barb e, for all $R \in \mathcal{CS}_P^M$, $R \Downarrow_e^\omega$ holds.*

3 Adaptable Processes, By Examples

Next we discuss concrete instances of adaptable processes in several settings.

Mode Transfer Operators. In [3], dynamic behavior at the process level is defined by means of two so-called *mode transfer* operators. Given processes P and Q, the *disrupt* operator starts executing P but at any moment it may abandon P and execute Q. The *interrupt* operator is similar, but it returns to execute P once Q emits a termination signal. We can represent similar mechanisms in $\mathcal{E}$ as follows:

$$\mathsf{disrupt}_a(P, Q) \stackrel{\mathsf{def}}{=} a[P] \parallel \widetilde{a}\{Q\} \qquad \mathsf{interrupt}_a(P, Q) \stackrel{\mathsf{def}}{=} a[P] \parallel \widetilde{a}\{Q \parallel t_Q.\bullet\}$$

Assuming that P evolves on its own to P', the semantics of $\mathcal{E}$ decrees that $\mathsf{disrupt}_a(P,Q)$ may evolve either to $a[P'] \parallel \widetilde{a}\{Q\}$ (as locality a is transparent) or to Q (which represents disruption at a). By assuming that P was able to evolve into P'' just before being interrupted, process $\mathsf{interrupt}_a(P,Q)$ evolves to $Q \parallel t_Q.\, P''$. Notice how defining P as an adaptable process at a is enough to formalize its potential disruption/interruption. Above, we assume that a is not used in P and Q, and that termination of Q is signaled at the designated name t_Q.

Dynamic Update in Workflow Applications. Designing business/enterprise applications in terms of *workflows* is a common practice nowadays. A workflow is a conceptual unit that describes how a number of *activities* coordinate to achieve a particular task. A workflow can be seen as a container of activities; such activities are usually defined in terms of simpler ones, and may be software-based (such as, e.g., "retrieve credit information from the database") or may depend on human intervention (such as, e.g., "obtain the signed authorization from the credit supervisor"). As such, workflows are typically long-running and have a transactional character. A workflow-based application usually consists of a *workflow runtime engine* that contains a number of workflows running concurrently on top of it; a *workflow base library* on which activities may rely on; and of a number of *runtime services*, which are application dependent and implement things such as transaction handling and communication with other applications. A simple abstraction of a workflow application is the following $\mathcal{E}$ process:

$$App \stackrel{\mathsf{def}}{=} \mathsf{wfa}\Big[\,\mathsf{we}\big[\mathtt{WE} \parallel \mathsf{W}_1 \parallel \cdots \parallel \mathsf{W}_k \parallel \mathsf{wbl}[\mathtt{BL}]\big] \parallel \mathtt{S}_1 \parallel \cdots \parallel \mathtt{S}_j\,\Big]$$

where the application is modeled as an adaptable process wfa which contains a workflow engine we and a number of runtime services $\mathtt{S}_1, \ldots, \mathtt{S}_j$. In turn, the workflow engine contains a number of workflows $\mathsf{W}_1, \ldots, \mathsf{W}_k$, a process WE (which represents the engine's behavior and is left unspecified), and an adaptable process wbl representing the base library (also left unspecified). As described before, each workflow is composed of a number of activities. We model each W_i as an adaptable process w_i containing a process $\mathtt{WL}_i$ —which specifies the workflow's logic—, and n activities. Each of them is formalized as an adaptable process a_j and an *execution environment* env_j:

$$\mathsf{W}_i = \mathsf{w}_i\Big[\,\mathtt{WL}_i \parallel \prod_{j=1}^{n} \big(\mathsf{env}_j[\mathtt{P}_j] \parallel \mathsf{a}_j\big[!u_j.\, \widetilde{\mathsf{env}_j}\{\mathsf{env}_j[\bullet \parallel \mathtt{A}_j]\}\big]\big)\,\Big]$$

The current state of the activity j is represented by process $\mathtt{P}_j$ running in env_j. Locality a_j contains an update action for env_j, which is guarded by u_j and always available. As defined above, such an update action allows to add process $\mathtt{A}_j$ to the current state of the execution environment of j. It can also be seen as a procedure that is yet not active, and that becomes active only upon reception of an output at u_j from, e.g., $\mathtt{WL}_i$. Notice that by defining update actions on a_j (inside $\mathtt{WL}_i$, for instance) we can describe the evolution of the execution environment. An example of this added flexibility is the process

$$U_1 = !\,\mathsf{replace}_j.\, \widetilde{\mathsf{a}_j}\big\{a_j[!u_j.\, \widetilde{\mathsf{env}_j}\{\mathsf{env}_j[\bullet \parallel \mathtt{A}_j^2]\}]\big\}$$

Hence, given an output at $\mathsf{replace}_j$, process $a_j[!u_j.\, \widetilde{\mathsf{env}_j}\{\mathsf{env}_j[\bullet \parallel \mathtt{A}_j]\}] \parallel U_1$ evolves to $a_j[!u_j.\, \widetilde{\mathsf{env}_j}\{\mathsf{env}_j[\bullet \parallel \mathtt{A}_j^2]\}]$ thus discarding $\mathtt{A}_j$ in a *future* evolution of env_j. This kind

of *dynamic update* is available in commercial workflow engines, such as the Windows Workflow Foundation (WWF). Above, for simplicity, we have abstracted from lock mechanisms that keep consistency between concurrent updates on env_j and a_j.

In the WWF, dynamic update can also take place at the level of the workflow engine. This way, e.g., the engine may *suspend* those workflows which have been inactive for a certain amount of time. This optimizes resources at runtime, and favors active workflows. We can implement this policy as part of the process WE as follows:

$$U_2 = !\,\mathsf{suspend}_i.\,\widetilde{\mathsf{w}_i}\{!\,\mathsf{resume}_i.\,\mathsf{w}_i[\bullet]\}$$

This way, given an output signal at $\mathsf{suspend}_i$, process $\mathsf{w}_i[\mathtt{W}_i] \parallel U_3$ evolves to the persistent process $!\,\mathsf{resume}_i.\,\mathsf{w}_i[\mathtt{W}_i]$ which can be reactivated at a later time.

Scaling in Cloud Computing Applications. In the cloud computing paradigm, Web applications are deployed in the infrastructure offered by external providers. Developers only pay for the resources they consume (usually measured as the processor time in remote *instances*) and for additional services (e.g., services that provide performance metrics). Central to this paradigm is the goal of optimizing resources for both clients and provider. An essential feature towards that goal is *scaling*: the capability that cloud applications have for expanding themselves in times of high demand, and for reducing themselves when the demand is low. Scaling can be appreciated in, e.g., the number of running instances supporting the web application. Tools and services for *autoscaling* are provided by cloud providers such as Amazon's Elastic Cloud Computing (EC2) and by vendors who build on the public APIs cloud providers offer.

Here we draw inspiration from the autoscaling library provided by EC2. For scaling purposes, applications in EC2 are divided into *groups*, each defining different scaling policies for different parts of the application. This way, e.g., the part of the application deployed in Europe can have different scaling policies from the part deployed in the US. Each group is then composed of a number of identical instances implementing the web application, and of active processes implementing the scaling policies. This scenario can be abstracted in $\mathcal{E}$ as the process $App \stackrel{\mathsf{def}}{=} G_1 \parallel \cdots \parallel G_n$, with

$$G_i = g_i\big[\, I \parallel \cdots \parallel I \parallel S_{dw} \parallel S_{up} \parallel \mathtt{CTRL}_i \,\big]$$

where each group G_i contains a fixed number of running instances, each represented by $I = \mathtt{mid}[\mathsf{A}]$, a process that abstracts an instance as an adaptable process with unique identification mid and state A. Also, S_{dw} and S_{up} stand for the processes implementing scaling down and scaling up policies, respectively. Process $\mathtt{CTRL}_i$ abstracts the part of the system which controls scaling policies for group i. In practice, this control relies on external services (such as, e.g., services that monitor cloud usage and produce appropriate *alerts*). A simple way of abstracting scaling policies is the following:

$$S_{dw} = s_d\big[\,!\,\mathsf{alert}^{\mathsf{d}}.\prod^{j}\widetilde{\mathtt{mid}}\{\mathbf{0}\}\,\big] \qquad S_{up} = s_u\big[\,!\,\mathsf{alert}^{\mathsf{u}}.\prod^{k}\widetilde{\mathtt{mid}}\{\mathtt{mid}[\bullet] \parallel \mathtt{mid}[\bullet]\}\,\big]$$

Given proper alerts from $\mathtt{CTRL}_i$, the above processes modify the number of running instances. In fact, given an output at $\mathsf{alert}^{\mathsf{d}}$ process S_{dw} destroys j instances. This is achieved by leaving the inactive process as the new state of locality mid. Similarly, an output at $\mathsf{alert}^{\mathsf{u}}$ process S_{up} spawns k update actions, each creating a new instance.

Autoscaling in EC2 also includes the possibility of *suspending* and *resuming* the scaling policies themselves. To represent this, we proceed as we did for process U_2 above. This way, for the scale down policy, one can assume that $\mathtt{CTRL}_i$ includes a process $U_{dw} = !\,\mathsf{susp}_{\mathsf{down}}.\,\widetilde{s_d}\{!\,\mathsf{resume}_{\mathsf{dw}}.\,s_d[\bullet]\}$ which, provided an output signal on $\mathsf{susp}_{\mathsf{down}}$, captures the current policy, and evolves into a process that allows to resume it at a later stage. This idea can be used to enforce other modifications to the policies (such as, e.g., changing the number of instances involved).

4 Bounded and Eventual Adaptation are Undecidable in $\mathcal{E}$

We prove that $\mathcal{BA}$ and $\mathcal{EA}$ are undecidable in $\mathcal{E}$ by defining an encoding of Minsky machines (MMs) into $\mathcal{E}$ which satisfies the following: a MM terminates if and only if its encoding into $\mathcal{E}$ evolves into at least $k > 0$ processes that can perform a barb e.

A MM [17] is a Turing complete model composed of a set of sequential, labeled instructions, and two registers. Registers r_j ($j \in \{0, 1\}$) can hold arbitrarily large natural numbers. Instructions $(1 : I_1), \ldots, (n : I_n)$ can be of three kinds: $\mathtt{INC}(r_j)$ adds 1 to register r_j and proceeds to the next instruction; $\mathtt{DECJ}(r_j, s)$ jumps to instruction s if r_j is zero, otherwise it decreases register r_j by 1 and proceeds to the next instruction; $\mathtt{HALT}$ stops the machine. A MM includes a program counter p indicating the label of the instruction to be executed. In its initial state, the MM has both registers set to 0 and the program counter p set to the first instruction.

The encoding, denoted $[\![\cdot]\!]_1$, is given in Table 1. A register j with value m is represented by an adaptable process at r_j that contains the encoding of number m, denoted $(\!|\ m\ |\!)_j$. In turn, $(\!|\ m\ |\!)_j$ consists of a sequence of m output prefixes on name u_j ending with an output action on z_j (the encoding of zero). Instructions are encoded as replicated processes guarded by p_i, which represents the MM when the program counter $p = i$. Once p_i is consumed, each instruction is ready to interact with the registers. To encode the increment of register r_j, we enlarge the sequence of output prefixes it contains. The adaptable process at r_j is updated with the encoding of the incremented value (which results from putting the value of the register behind some prefixes) and then the next instruction is invoked. The encoding of a decrement of register j consists of an exclusive choice: the left side implements the decrement of the value of a register, while the right one implements the jump to some given instruction. This choice is indeed exclusive: the encoding of numbers as a chain of output prefixes ensures that both an input prefix on u_j and one on z_j are never available at the same time. When the MM reaches the $\mathtt{HALT}$ instruction the encoding can either exhibit a barb on e, or set the program counter again to the $\mathtt{HALT}$ instruction so as to pass through a state that exhibits e at least $k > 0$ times. The encoding of a MM into $\mathcal{E}$ is defined as follows:

Definition 5. *Let N be a MM, with registers $r_0 = 0$, $r_1 = 0$ and instructions $(1 : I_1) \ldots (n : I_n)$. Given the encodings in Table 1, the encoding of N in $\mathcal{E}$ (written $[\![N]\!]_1$) is defined as $[\![r_0 = 0]\!]_1 \parallel [\![r_1 = 0]\!]_1 \parallel \prod_{i=1}^{n} [\![(i : I_i)]\!]_1 \parallel \overline{p_1}$.*

It can be shown that a MM N terminates iff its encoding has at least k consecutive barbs on the distinguished action e, for every $k \geq 1$, i.e. $[\![N]\!]_1 \Downarrow_e^k$. By considering the cluster $\mathcal{CS}^{\emptyset}_{[\![N]\!]_1} = \{[\![N]\!]_1\}$, we can conclude that $\mathcal{BA}$ is undecidable for $\mathcal{E}$ processes.

REGISTER r_j $\quad [\![r_j = n]\!]_1 = r_j[(\!|\ n\ |\!)_j]$ $\quad$ where $\quad (\!|\ n\ |\!)_j = \begin{cases} \overline{z_j} & \text{if } n = 0 \\ \overline{u_j}.\,(\!|\ n-1\ |\!)_j & \text{if } n > 0. \end{cases}$

INSTRUCTIONS $(i : I_i)$

$[\![(i : \mathtt{INC}(r_j))]\!]_1 = !p_i.\,\widetilde{r_j}\{r_j[\overline{u_j}.\,\bullet]\}.\,\overline{p_{i+1}}$

$[\![(i : \mathtt{DECJ}(r_j, s))]\!]_1 = !p_i.\,(u_j.\,\overline{p_{i+1}} + z_j.\,\widetilde{r_j}\{r_j[\overline{z_j}]\}.\,\overline{p_s})$

$[\![(i : \mathtt{HALT})]\!]_1 = !p_i.\,(e + \overline{p_i})$

Table 1. Encoding of MMs into $\mathcal{E}$.

Moreover, since the number of consecutive barbs on e can be unbounded (i.e., there exists a computation where $[\![N]\!]_1 \Downarrow_e^\omega$), we can also conclude that $\mathcal{EA}$ is undecidable.

Theorem 1. *$\mathcal{BA}$ and $\mathcal{EA}$ are undecidable in $\mathcal{E}$.*

5 The Subcalculus $\mathcal{E}^-$ and Decidability of Bounded Adaptation

Theorem 1 raises the question as whether there are fragments of $\mathcal{E}$ in which the problems are decidable. A natural way of restricting the language is by imposing limitations on *update patterns*, the behavior of running processes as a result of update actions. We now consider $\mathcal{E}^-$, the fragment of $\mathcal{E}$ in which update prefixes are restricted in such a way that the hole $\bullet$ cannot occur in the scope of prefixes. More precisely, $\mathcal{E}^-$ processes are those $\mathcal{E}$ processes in which the context U in $\widetilde{a}\{U\}$ respects the following grammar:

$$U ::= P \ \mid\ a[U] \ \mid\ U \parallel U \ \mid\ \bullet$$

In [5], we have shown that there exists an algorithm to determine whether there exists $R \in \mathcal{CS}_P^M$ such that $R \Downarrow_\alpha^k$ holds. We therefore have the following result.

Theorem 2. *$\mathcal{BA}$ is decidable in $\mathcal{E}^-$.*

We now provide intuitions on the proof of Theorem 2; see [5] for details. The algorithm checks whether one of such R appears in the (possibly infinite) set of processes from which it is possible to reach a process that can perform α at least k consecutive times. The idea is to introduce a *preorder* (or, equivalently, quasi-order) $\preceq$ on processes so as to characterize such a set by means of a so-called *finite basis*: finitely many processes that generate the set by upward closure, i.e., by taking all processes which are greater or equal to some process (wrt $\preceq$) in the finite basis.

The proof appeals to the theory of well-structured transition systems [1,13]. We define the preorder $\preceq$ by resorting to a tree representation, in such a way that: (i) $\preceq$ is a *well-quasi-order*: given any infinite sequence $x_i, i \in \mathbb{N}$, of elements there exist $j < k$ such that $x_j \preceq x_k$; (ii) $\preceq$ is *strongly compatible* wrt reduction in $\mathcal{E}$: for all $x_1 \preceq y_1$ and all reductions $x_1 \longrightarrow x_2$, there exists y_2 such that $y_1 \longrightarrow y_2$ and $x_2 \preceq y_2$; (iii) $\preceq$ is *decidable*; and (iv) $\preceq$ has an *effective pred-basis*, i.e., for any x it is possible to compute a basis of the set of states y such that there exists $y' \longrightarrow x'$ with $y' \preceq y$ and $x \preceq x'$. It is known [1,13] that given any target set I which is characterizable by a finite basis, an algorithm exists to compute a finite basis FB for the set of processes from which it is possible to reach a process belonging to I.

The algorithm to determine if there exists $R \in \mathcal{CS}_P^M$ such that $R \Downarrow_\alpha^k$ consists of three steps: (1) We restrict the set of terms that we consider to those reachable by any $R \in \mathcal{CS}_P^M$. We characterize this set by (a) considering the *sequential subterms* in $\mathcal{CS}_P^M$, i.e., terms not having parallel or locations as their topmost operator, included in the term P or in the terms in M and (b) introducing $\preceq$ over terms with the above properties. (2) We then show that it is possible to compute I (i.e. the finite basis of the set mentioned before) such that it includes all processes that expose α at least k consecutive times. (3) Finally, we show that it is possible to determine whether or not some $R \in \mathcal{CS}_P^M$ is included in the set generated by the finite basis FB.

Intuitions on these three steps follow. As for (1), we exploit Kruskal's theorem on well-quasi-orderings on trees [15]. Unlike other works appealing to well-structured transition systems for obtaining decidability results (e.g. [9]), in the case of $\mathcal{E}^-$ it is not possible to find a bound on the "depth" of processes. Consider, for instance, process $R = a[P] \parallel !\widetilde{a}\{a[a[\bullet]]\}$. One possible evolution of R is when it is always the innermost adaptable process which is updated; as a result, one obtains a process with an unbounded number of nested adaptable processes: $a[a[\ldots a[P]]]$. Nevertheless, some regularity can be found also in the case of $\mathcal{E}^-$, by mapping processes into trees labeled over location names and sequential subterms in $\mathcal{CS}_P^M$. The tree of a process P is built as follows. We set a root node labeled with the special symbol ϵ, and which has as many children nodes as parallel subterms in P. For those subterms different from an adaptable process, we obtain leaf nodes labeled with the subterms. For each subterm $a[P']$, we obtain a node which is labeled with a and has as many children nodes as parallel subterms in P'; tree construction then proceeds recursively on each of these parallel subterms. By mapping processes into this kind of trees, and by using Kruskal's ordering over them, it can be shown that our preorder $\preceq$ is a well-quasi-ordering with strong compatibility, and has an effective pred-basis. The pred-basis of a process P is computed by taking all the *minimal* terms that reduce to P in a single reduction. These terms are obtained by extending the tree of P with at most two additional subtrees, which represent the subprocess(es) involved in a reduction.

As for (2), we proceed backwards. We first determine the finite basis FB' of the set of processes that can *immediately* perform α. This corresponds to the set of sequential subterms of $\mathcal{CS}_P^M$ that can immediately perform α. If $k > 1$ (otherwise we are done) we do a backward step by applying the effective pred-basis procedure described above to each term of FB', with a simple modification: if an element Q of the obtained basis cannot immediately perform α, we replace it with all the terms $Q \parallel Q'$, where Q' is a sequential subterm of $\mathcal{CS}_P^M$ that can immediately perform α. To ensure minimality of the finite basis FB', we remove from it all R' such that $R \preceq R'$, for some R already in FB'. We repeat this procedure $k-1$ times to obtain FB—the finite basis of I.

As for (3), we verify if there exists a $Q \in FB$ for which the following check succeeds. Let $Par(Q)$ be the multiset of terms Q_i such that Q is obtained as the parallel composition of such Q_i (notice that it could occur that $Par(Q) = \{Q\}$). We first remove from $Par(Q)$ all those Q_i such that there exists $T \in M$ with $Q_i \preceq T$, thus obtaining the multiset $Par'(Q)$. Let S be the parallel composition of the processes in $Par'(Q)$. Then, we just have to check whether $S \preceq P$ or not.

$\text{CONTROL} \;=\; !a.\,(\overline{f} \parallel \overline{b} \parallel \overline{a}) \parallel \overline{a}.\,a.\,(\overline{p_1} \parallel e) \parallel\; !h.\,(g.\,\overline{f} \parallel \overline{h})$

REGISTER r_j

$$[\![r_j = m]\!]_2 = \begin{cases} r_j[!inc_j.\,\overline{u_j} \parallel \overline{z_j}] & \text{if } m = 0 \\ r_j[!inc_j.\,\overline{u_j} \parallel \prod_1^m \overline{u_j} \parallel \overline{z_j}] & \text{if } m > 0. \end{cases}$$

INSTRUCTIONS $(i : I_i)$

$[\![(i : \texttt{INC}(r_j))]\!]_2 \quad = !p_i.\,f.\,(\overline{g} \parallel b.\,\overline{inc_j}.\,\overline{p_{i+1}})$

$[\![(i : \texttt{DECJ}(r_j, s))]\!]_2 = !p_i.\,f.\,\big(\overline{g} \parallel (u_j.\,(\overline{b} \parallel \overline{p_{i+1}}) + z_j.\,\widetilde{r_j}\{r_j[!inc_j.\,\overline{u_j} \parallel \overline{z_j}]\}.\,\overline{p_s})\big)$

$[\![(i : \texttt{HALT})]\!]_2 \quad = !p_i.\,\overline{h}.\,h.\,\widetilde{r_0}\{r_0[!inc_0.\,\overline{u_0} \parallel \overline{z_0}]\}.\,\widetilde{r_1}\{r_1[!inc_1.\,\overline{u_1} \parallel \overline{z_1}]\}.\,\overline{p_1}$

Table 2. Encoding of MMs into $\mathcal{E}^-$ - Static case.

6 Eventual Adaptation is Undecidable in $\mathcal{E}^-$

We show that $\mathcal{EA}$ is undecidable in $\mathcal{E}^-$ by relating it to termination in MMs. In contrast to the encoding given in Section 4, the encodings presented here are *non faithful*: when mimicking a test for zero, the encoding may perform a jump even if the tested register is not zero. Nevertheless, we are able to define encodings that repeatedly simulate finite computations of the MM, and if the repeated simulation is infinite, then we have the guarantee that the number of erroneous steps is finite. This way, the MM terminates iff its encoding has a non terminating computation. As during its execution the encoding continuously exhibits a barb on e, it then follows that $\mathcal{EA}$ is undecidable in $\mathcal{E}^-$.

We show that $\mathcal{EA}$ is already undecidable in two fragments of $\mathcal{E}^-$. While in the *static* fragment we assume that the topology of nested adaptable processes is fixed and cannot change during the computation, in the *dynamic* fragment we assume that such a topology can change, but that processes cannot be neither removed nor replicated.

Undecidability in the Static Case. The encoding relies on finitely many output prefixes acting as *resources* on which instructions of the MM depend in order to be executed. To repeatedly simulate finite runs of the MM, at the beginning of the simulation the encoding produces finitely many instances of these resources. When HALT is reached, the registers are reset, some of the consumed resources are restored, and a new simulation is restarted from the first instruction. In order to guarantee that an infinite computation of the encoding contains only finitely many erroneous jumps, finitely many instances of a second kind of resource (different from that required to execute instructions) are produced. Such a resource is consumed by increment instructions and restored by decrement instructions. When the simulation performs a jump, the tested register is reset: if it was not empty (i.e., an erroneous test) then some resources are permanently lost. When the encoding runs out of resources, the simulation will eventually block as increment instructions can no longer be simulated. We make two non restrictive assumptions. First, we assume that a MM computation contains at least one increment instruction. Second, in order to avoid resource loss at the end of a correct simulation run, we assume that MM computations terminate with both the registers empty.

We now discuss the encoding defined in Table 2. We first comment on CONTROL, the process that manages the resources. It is composed of three processes in parallel. The first replicated process produces an unbounded amount of processes $\overline{f}$ and $\overline{b}$, which represent the two kinds of resources described above. The second process starts and

stops a resource production phase by performing $\overline{a}$ and a, respectively. Then, it starts the MM simulation by emitting the program counter $\overline{p_1}$. The third process is used at the end of the simulation to restore some of the consumed resources $\overline{f}$ (see below).

A register r_j that stores number m is encoded as an adaptable process at r_j containing m copies of the unit process $\overline{u_j}$. It also contains process $!inc_j.\,\overline{u_j}$ which allows to create further copies of $\overline{u_j}$ when an increment instruction is executed. Instructions are encoded as replicated processes guarded by p_i. Once p_i is consumed, increment and decrement instructions consume one of the resources $\overline{f}$. If such a resource is available then it is renamed as $\overline{g}$, otherwise the simulation blocks. The simulation of an increment instruction also consumes an instance of resource $\overline{b}$.

The encoding of a decrement-and-jump instruction is slightly more involved. It is implemented as a choice: the process can either perform a decrement and proceed with the next instruction, or to jump. In case the decrement can be executed (the input u_j is performed) then a resource $\overline{b}$ is restored. The jump branch can be taken even if the register is not empty. In this case, the register is reset via an update that restores the initial state of the adaptable process at r_j. Note that if the register was not empty, then some processes $\overline{u_j}$ are lost. Crucially, this causes a permanent loss of a corresponding amount of resources $\overline{b}$, as these are only restored when process $\overline{u_j}$ are present.

The simulation of the `HALT` instruction performs two tasks before restarting the execution of the encoding by reproducing the program counter p_1. The first one is to restore some of the consumed resources $\overline{f}$: this is achieved by the third process of CONTROL, which repeatedly consumes one instance of $\overline{g}$ and produces one instance of $\overline{f}$. This process is started/stopped by executing the two prefixes $\overline{h}.\,h$. The second task is to reset the registers by updating the adaptable processes at r_j with their initial state.

The full definition of the encoding is as follows.

Definition 6. *Let N be a MM, with registers r_0, r_1 and instructions $(1 : I_1) \ldots (n : I_n)$. Given the* CONTROL *process and the encodings in Table 2, the encoding of N in $\mathcal{E}^-$ (written $[\![N]\!]_2$) is defined as $[\![r_0 = 0]\!]_2 \parallel [\![r_1 = 0]\!]_2 \parallel \prod_{i=1}^{n} [\![(i : I_i)]\!]_2 \parallel$* CONTROL.

The encoding has an infinite sequence of simulation runs if and only if the corresponding MM terminates. As the barb e is continuously exposed during the computation (the process e is spawn with the initial program counter and is never consumed), we can conclude that a MM terminates if and only if its encoding does not eventually terminate.

Lemma 1. *Let N be a MM. N terminates iff $[\![N]\!]_2 \Downarrow_e^\omega$.*

Exploiting Lemma 1, we can state the following:

Theorem 3. *$\mathcal{EA}$ is undecidable in $\mathcal{E}^-$.*

Note that the encoding $[\![\cdot]\!]_2$ uses processes that do not modify the topology of nested adaptable processes; update prefixes do not remove nor create adaptable processes: they simply remove the processes currently in the updated locations and replace them with the predefined initial content. One could wonder whether the ability to remove processes is necessary for the undecidability result: next we show that this is not the case.

Undecidability in the Dynamic Case. We now show that $\mathcal{EA}$ is still undecidable in $\mathcal{E}^-$ even if we consider updates that do not remove running processes. The proof relies

REGISTER r_j

$[\![r_j = 0]\!]_3 = r_j[Reg_j \parallel c_j[\mathbf{0}]]$ with $Reg_j = !inc_j.\, \widetilde{c_j}\{c_j[\bullet]\}.\, \overline{ack}.\, u_j.\, \widetilde{c_j}\{c_j[\bullet]\}.\, \overline{ack}$

INSTRUCTIONS $(i : I_i)$

$[\![(i : \texttt{INC}(r_j))]\!]_3 = !p_i.\, f.\, (\overline{g} \parallel b.\, \overline{inc_j}.\, ack.\, \overline{p_{i+1}})$

$[\![(i : \texttt{DECJ}(r_j, s))]\!]_3 = !p_i.\, f.\, \big(\overline{g} \parallel (\overline{u_j}.\, ack.\, (\overline{b} \parallel \overline{p_{i+1}}) + \widetilde{c_j}\{\bullet\}.\, \widetilde{r_j}\{r_j[Reg_j \parallel c_j[\bullet]]\}.\, \overline{p_s})\big)$

$[\![(i : \texttt{HALT})]\!]_3 = !p_i.\, \overline{h}.\, h.\, \widetilde{c_0}\{\bullet\}.\, \widetilde{r_0}\{r_0[Reg_0 \parallel c_0[\bullet]]\}.\, \widetilde{c_1}\{\bullet\}.\, \widetilde{r_1}\{r_1[Reg_1 \parallel c_1[\bullet]]\}.\, \overline{p_1}$

Table 3. Encoding of MMs into $\mathcal{E}^-$ - Dynamic case.

on a nondeterministic encoding of MMs, similar to the one presented before. In that encoding, process deletion was used to restore the initial state inside the adaptable processes representing the registers. In the absence of process deletion, we use a more involved technique based on the possibility of moving processes to a different context: processes to be removed are guarded by an update prefix $\widetilde{c_j}\{c_j[\bullet]\}$ that simply tests for the presence of a parallel adaptable process at c_j; when a process must be deleted, it is "collected" inside c_j, thus disallowing the possibility to execute such an update prefix.

The encoding is as in Definition 6, with registers and instructions encoded as in Table 3. A register r_j that stores number m is encoded as an adaptable process at r_j that contains m copies of the unit process $u_j.\, \widetilde{c_j}\{c_j[\bullet]\}.\, \overline{ack}$. It also contains process Reg_j, which creates further copies of the unit process when an increment instruction is invoked, as well as the collector c_j, which is used to store the processes to be removed.

An increment instruction adds an occurrence of $u_j.\, \widetilde{c_j}\{c_j[\bullet]\}.\, \overline{ack}$. Note that an output $\overline{inc}$ could synchronize with the corresponding input inside a collected process. This immediately leads to deadlock as the containment induced by c_j prevents further interactions. The encoding of a decrement-and-jump instruction is implemented as a choice, following the idea discussed for the static case. If the process guesses that the register is zero then, before jumping to the given instruction, it proceeds at disabling its current content: this is done by (i) removing the boundary of the collector c_j leaving its content at the top-level, and (ii) updating the register placing its previous state in the collector. A decrement simply consumes one occurrence of $u_j.\, \widetilde{c_j}\{c_j[\bullet]\}.\, \overline{ack}$. Note that as before the output $\overline{u_j}$ could synchronize with the corresponding input inside a collected process. Again, this immediately leads to deadlock. The encoding of HALT exploits the same mechanism of collecting processes to simulate the reset of the registers.

This encoding has the same properties of the one discussed for the static case. In fact, in an infinite simulation the collected processes are never involved, otherwise the computation would block. We can conclude that process deletion is not necessary for the undecidability of $\mathcal{EA}$ in $\mathcal{E}^-$. Nevertheless, in the encoding in Table 3 we need to use the possibility to remove and create adaptable processes (namely, the collectors c_j are removed and then reproduced when the registers must be reset). One could therefore wonder whether $\mathcal{EA}$ is still undecidable if we remove from $\mathcal{E}^-$ *both* the possibility to remove processes and to create/destroy adaptable processes. In the extended version of this paper [5] we have defined a fragment of $\mathcal{E}^-$ obtained by (i) disallowing creation and destruction of adaptable processes and (ii) eliminating the possibility of removing or relocating a running process to a different adaptable process. By resorting to the theory of Petri nets, we have proved that $\mathcal{EA}$ for processes in this fragment is *decidable*.

7 Concluding Remarks

We have proposed the concept of *adaptable process* as a way of describing concurrent systems that exhibit complex evolvability patterns at runtime. We have introduced $\mathcal{E}$, a calculus of adaptable processes in which processes can be updated/relocated at runtime. We also proposed the *bounded* and *eventual* adaptation problems, and provided a complete study of their (un)decidability for $\mathcal{E}$ processes. Our results shed light on the expressive power of $\mathcal{E}$ and on the verification of concurrent processes that may evolve at runtime. As for future work, it would be interesting to develop variants of $\mathcal{E}$ tailored to concrete application settings, to determine how the proposed adaptation problems fit in such scenarios, and to study how to transfer our decidability results to such variants.

Related Work. As argued before, the combination of techniques required to prove decidability of $\mathcal{BA}$ in $\mathcal{E}^-$ is non trivial. In particular, the technique is more complex than that in [2], which relies on a bound on the depth of trees, or that in [23], where only topologies with bounded paths are taken into account. Kruskal's theorem is also used in [7] for studying the decidability properties of calculi with exceptions and compensations. The calculi considered in [7] are *first-order*; in contrast, $\mathcal{E}$ is a *higher-order* calculus (see below). We showed the undecidability of $\mathcal{EA}$ in $\mathcal{E}^-$ by means of an encoding of MMs that does not reproduce faithfully the corresponding machine. Similar techniques have been used to prove the undecidability of repeated coverability in reset Petri nets [11], but in our case their application revealed much more complex. Notice that since in a cluster there is no a-priori knowledge on the number of modifications that will be applied to the system, the analysis needs to be *parametric*. Parametric verification has been studied, e.g., in the context of broadcast protocols in fully connected [12] and ad-hoc networks [10]. Differently from [12,10], in which the number of nodes (or the topology) of the network is unknown, we consider systems in which there is a known part (the initial system P), and there is another part composed of an unknown number of instances of processes (taken from the set of possible modifications M).

$\mathcal{E}$ is related to *higher-order* process calculi such as, e.g., the higher-order π-calculus [21], Kell [22], and Homer [8]. In such calculi, processes can be passed around, and so communication involves term instantiation, as in the λ-calculus. Update actions in $\mathcal{E}$ are a form of term instantiation: as we elaborate in [6], they can be seen as a streamlined version of the *passivation* operator of Kell and Homer, which allows to suspend a running process. The encoding given in Section 4 is inspired in the encoding of MMs into a core higher-order calculus with passivation presented in [19, Ch 5]. In [19], however, no adaptation concerns are studied. Also related to $\mathcal{E}$ are process calculi for distributed algorithms/systems (e.g., [4,20,18,14]) which feature located processes and notions of failure. In [4], a higher-order operation that defines *savepoints* is proposed: process $\mathsf{save}\langle P\rangle.\, Q$ defines the savepoint P for the current location; if the location crashes, then it will be restarted with state P. The calculus in [20] includes constructs for killing a located process, spawning a new located process, and checking the status of a location. In [18,14], the language includes a *failure detector* construct $\mathcal{S}(k).\, P$ which executes P if location k is *suspected* to have failed. Crucially, while in the languages in [4,20,18,14] the *post-failure* behavior is defined statically, in $\mathcal{E}$ it can be defined dy-

namically, exploiting running processes. Moreover, differently from our work, neither of [4,20,18,14] addresses expressiveness/decidability issues, as we do here.

References

1. P. A. Abdulla, K. Cerans, B. Jonsson, and Y.-K. Tsay. Algorithmic analysis of programs with well quasi-ordered domains. *Inf. Comput.*, 160(1-2):109–127, 2000.
2. L. Acciai and M. Boreale. Deciding safety properties in infinite-state pi-calculus via behavioural types. In *Proc. of ICALP*, volume 5556 of *LNCS*, pages 31–42. Springer, 2009.
3. J. Baeten and J. Bergstra. Mode transfer in process algebra. Technical Report Report 00/01, Eindhoven University of Technology, 2000.
4. M. Berger and K. Honda. The two-phase commitment protocol in an extended pi-calculus. *Electr. Notes Theor. Comput. Sci.*, 39(1), 2000.
5. M. Bravetti, C. Di Giusto, J. A. Pérez, and G. Zavattaro. Adaptable processes. Technical report, Univ. of Bologna, 2011. Draft in `www.cs.unibo.it/~perez/ap/`.
6. M. Bravetti, C. Di Giusto, J. A. Pérez, and G. Zavattaro. Steps on the Road to Component Evolvability. In *Post-proceedings of FACS'10*, LNCS. Springer, 2011. To appear.
7. M. Bravetti and G. Zavattaro. On the expressive power of process interruption and compensation. *Math. Struct. in Comp. Sci.*, 19(3):565–599, 2009.
8. M. Bundgaard, J. C. Godskesen, and T. Hildebrandt. Bisimulation congruences for homer — a calculus of higher order mobile embedded resources. Technical Report TR-2004-52, IT University of Copenhagen, 2004.
9. N. Busi, M. Gabbrielli, and G. Zavattaro. On the expressive power of recursion, replication and iteration in process calculi. *Math. Struct. in Comp. Sci.*, 19(6):1191–1222, 2009.
10. G. Delzanno, A. Sangnier, and G. Zavattaro. Parameterized verification of ad hoc networks. In *Proc. of CONCUR*, volume 6269 of *LNCS*, pages 313–327. Springer, 2010.
11. J. Esparza. Some applications of petri nets to the analysis of parameterised systems, 2003. Talk at WISP'03.
12. J. Esparza, A. Finkel, and R. Mayr. On the verification of broadcast protocols. In *Proc. of LICS*, pages 352–359, 1999.
13. A. Finkel and P. Schnoebelen. Well-structured transition systems everywhere! *Theor. Comput. Sci.*, 256(1-2):63–92, 2001.
14. A. Francalanza and M. Hennessy. A fault tolerance bisimulation proof for consensus (extended abstract). In *Proc. of ESOP*, volume 4421 of *LNCS*, pages 395–410. Springer, 2007.
15. J. B. Kruskal. Well-quasi-ordering, the tree theorem, and vazsonyi's conjecture. *Transactions of the American Mathematical Society*, 95(2):210–225, 1960.
16. R. Milner. *Comunication and Concurrency*. Prentice Hall, 1989.
17. M. Minsky. *Computation: Finite and Infinite Machines*. Prentice-Hall, 1967.
18. U. Nestmann, R. Fuzzati, and M. Merro. Modeling consensus in a process calculus. In *Proc. of CONCUR*, volume 2761 of *LNCS*, pages 393–407. Springer, 2003.
19. J. A. Pérez. *Higher-Order Concurrency: Expressiveness and Decidability Results*. PhD thesis, University of Bologna, 2010. Draft in `www.japerez.phipages.com`.
20. J. Riely and M. Hennessy. Distributed processes and location failures. *Theor. Comput. Sci.*, 266(1-2):693–735, 2001. An extended abstract appeared in Proc. of ICALP'97.
21. D. Sangiorgi. *Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms*. PhD thesis CST–99–93, University of Edinburgh, Dept. of Comp. Sci., 1992.
22. A. Schmitt and J.-B. Stefani. The kell calculus: A family of higher-order distributed process calculi. In *Global Computing*, volume 3267 of *LNCS*, pages 146–178. Springer, 2004.
23. 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.