

# Bounded and Eventual Adaptation for Evolvable Components

Mario Bravetti<sup>1</sup>, Cinzia Di Giusto<sup>2</sup>, Jorge A. Pérez<sup>3</sup>, and Gianluigi Zavattaro<sup>1</sup>

<sup>1</sup> Dipartimento di Scienze dell'Informazione, Università di Bologna, Italy

<sup>2</sup> INRIA Rhône-Alpes, Grenoble, France

<sup>3</sup> CITI - FCT New University of Lisbon, Portugal

**Abstract.** Deploying dynamically evolvable software applications is a common practice nowadays. This is typically achieved by means of mechanisms capable of *adapting* the system components to the modifications required by the external environment. *Correctness* and *evolvability* are closely related concerns: components might evolve along time, possibly in reaction to errors, but it is most desirable that the overall system exhibits a bounded or finite amount of error states. We propose a framework for reasoning about dynamically evolvable component systems. We introduce a basic calculus with evolvability capabilities and propose two correctness properties: *bounded* and *eventual* adaptation. While bounded adaptation ensures that at most  $k$  errors will arise in future states—including those reachable as a result of dynamic reconfigurations—, eventual adaptation ensures that the system will eventually reach a state from which no other error will arise (i.e., only finitely many errors can occur). We study the (un)decidability of these two adaptation properties in six different variants of the calculus, which represent different evolvability patterns under structural and behavioral criteria.

## 1 Introduction

The deployment of applications by the *aggregation* of elementary blocks (modules, components, web services, ...) is a long-standing principle in software engineering. Such a principle is still present in several emerging technologies and trends nowadays: one representative example is cloud computing, in which *clients* deploy their applications by the aggregation of predefined web services offered as virtualized resources by a *provider*; such a provider also offers the infrastructure to run the applications. In our view, the greater flexibility derived from these emerging paradigms suggests changes to the way in which traditional properties of software systems (notably, *correctness* and *reliability*) are conceived and ensured. For instance, in cloud computing applications correctness becomes a bi-directional property concerning clients and providers alike: providers would like to guarantee that the offered services are correct not only in isolation but also under every possible combination, customization or modification available to the clients; conversely, clients would like guarantees on the correctness of the execution of their aggregations once these are deployed in the provider's infrastructure (that is, guarantees that such an infrastructure does not introduce errors).

Most importantly, correctness and reliability requirements are unavoidably tied to *evolvability* issues. In effect, aggregations of blocks are usually subject to dynamic evolution as parts of the system can be replaced with new blocks, which provide new functionalities or represent updates for outdated blocks. Again, evolvability issues concern

both clients and providers: for instance, a client would like guarantees on the fact that updates/upgrades to the services performed by the provider preserve correct behavior (as in backward compatibility), or that the services deployed in exceptional circumstances (say, unanticipated peak loads or local malfunctions) are guaranteed to take the system back to a correct state or, at least, that the system will *eventually* reach a state in which it has *adapted* to such exceptional circumstances.

In this paper, we address the *correctness* of aggregations of *components* which are subject to *evolvability* and *adaptation* concerns as described above. We use the term *component* in a broad sense, as it refers to elementary blocks such as web services in cloud computing scenarios, but also to analogous concepts in related settings, such as services in service-oriented computing or long-running processes in workflow management. Our approach is as follows: we introduce a process calculus with primitives for components and evolvability, and use it to study the decidability of *verification problems* associated to the correctness of dynamically evolvable component aggregations. More precisely, we introduce a variant of Milner’s CCS [26] without restriction and relabeling, and extended with primitive notions of component and update. In this calculus, called *Evolvable CCS* ( $\mathcal{E}$  in the sequel),  $a[P]$  denotes the component called (or located at)  $a$  with state given by process  $P$ . Name  $a$  acts as a *transparent locality*:  $P$  can evolve on its own as well as interact freely with its surrounding environment. Components can be nested, and are sensible to interactions with *update actions*. An update action  $\tilde{a}\{U\}$  decrees the update of component  $a$  with the behavior defined by  $U$ , which can be thought of as a *context* with zero or more holes denoted by  $\bullet$ . The *evolution* of  $a[P]$  is realized by its interaction with the update action  $\tilde{a}\{U\}$ , which leads to process  $U\{P/\bullet\}$ , i.e., the process obtained by replacing every hole in  $U$  by  $P$ .

We propose two correctness properties for  $\mathcal{E}$  processes. The first one, *k-bounded adaptation* (abbreviated  $\mathcal{BA}$ ) ensures that, given a finite  $k$ , at most  $k$  errors can arise during the system evolution. The second property, *eventual adaptation* (abbreviated  $\mathcal{EA}$ ), is similar but weaker: it ensures that the system will eventually reach a state from which no other error will arise (that is, only finitely many errors can occur). We believe that  $\mathcal{BA}$  and  $\mathcal{EA}$  fit well in the kind of correctness analysis that is required in a number of emerging applications, including, but not limiting to, cloud computing scenarios. For instance, on the provider side of a cloud application, these properties allow to check whether a malicious client is able to assemble faulty systems via the aggregation of the provided services and the possible subsequent updates. On the client side, it is possible to carry out forms of *traceability analysis*, so as to prove that if the system exhibits an incorrect behavior, then it follows from the malfunctioning of the provider’s infrastructure and not from the initial aggregation and dynamic updates provided by the client.

The  $\mathcal{E}$  calculus provides a generic framework for reasoning about component aggregations and their correctness. We investigate the decidability of  $\mathcal{BA}$  and  $\mathcal{EA}$  in several variants of the calculus, which are obtained via two orthogonal characterizations. The first one is *structural*, and defines both *static* and *dynamic* topologies of aggregations. In a static topology, the number of components does not vary along the evolution of the system: components cannot be destroyed nor new components can appear. In contrast, in the more general dynamic topology this restriction is not considered. The second characterization is *behavioral*, and concerns *update patterns*—the context  $U$  in an update action  $\tilde{a}\{U\}$ . As hinted at above, update patterns determine the behavior of components after

an update action. In order to account for different evolvability patterns, we consider three kinds of update patterns, which determine three families of  $\mathcal{E}$  calculi—denoted  $\mathcal{E}^1$ ,  $\mathcal{E}^2$ , and  $\mathcal{E}^3$ , respectively. The first update pattern admits all kinds of contexts, and so it represents the most expressive form of update. In particular, holes  $\bullet$  can appear behind prefixes. The second update pattern forbids such guarded holes in contexts. In the third update pattern we additionally require contexts to have exactly one hole, thus preserving the existing behavior (and possibly adding new behaviors): this is the most restrictive form of update.

In our view, the variants of  $\mathcal{E}$  derived from these two characterizations capture a fairly ample spectrum of scenarios that arise in the joint analysis of correctness and adaptation concerns in component aggregations. They borrow inspiration from existing component models, development frameworks, and programming languages. Our structural characterization aims at capturing the distinctions between “flat” component models (such as, e.g., EJB [4] and CCM [29]) and “hierarchical” ones (such as, e.g., Fractal [1], SOFA 2 [10]). The behavioral characterization is inspired in the (restricted) forms of reconfiguration and/or update available in component models in which evolvability is specified in terms of patterns, such as SOFA 2 [22]. Update patterns are also related to functionalities present in programming languages such as Erlang [3,7] and in development frameworks such as the Windows Workflow Foundation (WWF) [2]. In fact, forms of dynamic update behavior for workflow services in the WWF include the possibility of replacing and removing service contracts (analogous to our first and second update patterns) and also the addition of new service contracts and operations (as in our third update pattern, which preserves existing behavior and operations).

As discussed before, a central issue when investigating the correctness of component-based systems together with possible dynamic update actions, is that the number of modifications that will be applied to the system is usually unknown. To this end, we propose the notion of *cluster* of a component-based system. Given a system  $P$  and the set  $M$  of the possible modifications that can be applied to the system, its associated cluster considers  $P$  together with an arbitrary amount of instances of the modifications in  $M$  that can be performed over (parts of)  $P$  at runtime. The notion of cluster is thus useful to formalize the adaptation and correctness properties of an initial system configuration (represented by a component aggregation) in the presence of arbitrarily many sources of update actions. Take, for instance, the cloud computing scenario sketched earlier: the notion of cluster captures the cloud application as initially deployed by the client along with the options offered by the provider for its evolution at runtime.

Component models typically consider stateful entities with *interfaces*, which specify conditions for composition and/or behavioral compatibility. We have chosen to formalize components by abstracting from the notion of interface (and its associated issues, e.g., interface mismatches), and have relied on transparent localities for describing containment/hierarchy. The reasons for this choice are twofold. On the one hand, we are interested in correctness and adaptation in dynamic reconfiguration from a general perspective, and so including particular notions of interface would have limited the generality of our results. By abstracting from interfaces in this way our results shed light on the nature of bounded and eventual adaptation in areas including—but not limiting to—component-based systems. On the other hand, the analysis of adaptation becomes substantially harder when considering interfaces. In fact, we conjecture undecidability of  $\mathcal{BA}$  and  $\mathcal{EA}$  for the six variants of  $\mathcal{E}$ , even with very simple interfaces. In our view, rather than investigating

(un)decidability issues in a fully-fledged formalism with interfaces, a more reasonable research strategy is to identify first the variants of  $\mathcal{E}$  in which adaptation is decidable—thus spotting where decidability lies—, and then to investigate the actual influence interfaces have in expressiveness and (un)decidability results.

Summing up, in the present paper we make the following contributions.

1. We propose  $\mathcal{E}$ , a formalism for component-based systems and identify different variants of it representing different evolvability patterns. We are not aware of other formalisms tailored to the evolution/adaptation properties of component systems.
2. We introduce bounded and eventual adaptation and study their (un)decidability in each of the variants of  $\mathcal{E}$ . To the best of our knowledge, ours is the first study of the (un)decidability of adaptation properties for dynamically evolvable components.

The table at the right summarizes our (un)decidability results. The decidability of eventual adaptation is proved by resorting to Petri nets, while for bounded adaptation we

	Dynamic Topology	Static Topology
$\mathcal{E}^1$	$\mathcal{BA}$ undec / $\mathcal{EA}$ undec	$\mathcal{BA}$ undec / $\mathcal{EA}$ undec
$\mathcal{E}^2$	$\mathcal{BA}$ dec / $\mathcal{EA}$ undec	$\mathcal{BA}$ dec / $\mathcal{EA}$ undec
$\mathcal{E}^3$	$\mathcal{BA}$ dec / $\mathcal{EA}$ undec	$\mathcal{BA}$ dec / $\mathcal{EA}$ dec

consider the theory of *well-structured transition systems* [17,5] and its associated results. transfer (un)decidability results in both directions. In our case, the theory of well-structured transition systems must be coupled with Kruskal’s theorem [24] (which allows 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 (which is necessary in order to deal with arbitrary component aggregations and dynamic updates that generate new nested components). This combination of techniques proved to be very challenging. In particular, the technique is more complex than that in [6], which relies on a bound on the depth of trees, or that in [36], where only topologies with bounded paths are taken into account. Kruskal’s theorem is also used in [8] for studying the decidability properties of calculi with exceptions and compensations. The calculi considered in [8] are *first-order*; in contrast, and as elaborated below,  $\mathcal{E}$  can be considered as a *higher-order* calculus. The undecidability results are obtained via encodings of Minsky machines [27], a well-known Turing complete model. In particular, for eventual adaptation the simulation that we provide does not reproduce faithfully the corresponding machine, but only finitely many steps are wrongly simulated. Similar techniques have been used to prove the undecidability of repeated coverability in reset Petri nets [14], but in our case their application revealed much more complex; this is particularly true for the case of  $\mathcal{E}^3$  where there is no native mechanism for *removing* an arbitrary amount of processes. Moreover, as in a cluster there is no a-priori knowledge of the number of modifications that will be applied to the system, we need to perform a parametric analysis. Parametric verification has been studied, e.g., in the context of broadcast protocols in both fully connected [15] and ad-hoc networks [13]. Differently from [15,13], 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  $M$ ).

*Related Work.* The earliest related work we are aware of is [21], which investigates *on-line software version change*. There, an on-line change is said to be *valid* if the updated

program eventually exhibits behavior of the new version. The problem of determining validity of an on-line change is shown to be undecidable by relating it to the halting problem. The study in [21], however, limits to restricted instances of imperative languages. The use of process calculi for giving formal grounds to component-based systems has been explored in, e.g., [25,28,32]. Darwin [25] is a language for describing the *structure* of static and dynamic component architectures which may evolve at runtime. The focus is then on the bindings of interacting components; the operational semantics of Darwin relies on a  $\pi$ -calculus model for handling such bindings. Darwin features a mechanism of dynamic instantiation which allows arbitrary changes in the system architecture; these changes concern the system topology rather than the state of the interconnected components, as in our case. The Piccola calculus [28] extends the asynchronous  $\pi$ -calculus with higher-order abstractions and passing of records, and allows to express components, connectors, and scripts. However, it does not feature constructs for dynamic reconfiguration as those in  $\mathcal{E}$ .

$\mathcal{E}$  is related to *higher-order* process calculi such as the higher-order  $\pi$ -calculus [31], Kell [33], and Homer [9]. In a higher-order setting, processes can be passed around in communications, and therefore they involve the instantiation of variables with terms, as in the  $\lambda$ -calculus. As hinted at above, evolvability actions in  $\mathcal{E}$  also involves term instantiation, and so  $\mathcal{E}$  can be considered as a very basic higher-order process calculus. Update capabilities in  $\mathcal{E}$  are also related to the *passivation* operator present in Kell and Homer, which allows to suspend the execution of a running process. In fact, passivation can be seen as a “decoupled” version of our update capabilities, in which process suspension and relocation take place in different stages; instead, our update actions perform suspension and relocation in a single transition. The expressiveness study of a core calculus with passivation detailed in [30, Chapter 5] is therefore loosely related to the present paper. MECo [32] is a model for evolvable components which relies on a form of passivation. In MECo, components feature a hierarchical structure, rich input/output interfaces, as well as channel communication. Evolvability in MECo is enforced by a construct that *suspends* a component and *extracts* its “skeleton”; adaptation is then concerned with changes in input/output interfaces of components. While the notion of component interface in [32] is richer and arguably more realistic than ours, in the light of the discussion above, we claim that our notion (based on transparent localities) is still interesting to represent component aggregations; it also allows us to concentrate in the fundamental aspects of adaptation/correctness inherent to dynamic reconfiguration.

*For the reviewer’s convenience, additional technical details and proofs, auxiliary definitions, and extended discussions on related work have been collected in the Appendix.*

## 2 The Calculi

*Basic Syntax.* Evolvable CCS (abbreviated  $\mathcal{E}$  in the sequel) is a variant of CCS [26] without restriction and relabeling, and extended with constructs for evolvability. 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  $a, b$ , possibly decorated as  $\bar{a}, \bar{b}$  and  $\tilde{a}, \tilde{b}$ . The syntax of  $\mathcal{E}$  prefixes and processes is the following:

$$\pi ::= a \mid \bar{a} \mid \tilde{a}\{P\} \quad P, Q, \dots ::= \sum_{i \in I} \pi_i.P_i \mid a[P] \mid P \parallel P \mid !\pi.P$$

As in CCS, we use  $a$  and  $\bar{a}$  to denote atomic input and output actions, respectively.  $\mathcal{E}$  extends CCS with *update prefixes* and a primitive notion of *component*. In the update prefix  $\tilde{a}\{P\}$ , process  $P$  represents a context, i.e., a process with a hole  $\bullet$ . The intention is that when an update prefix is able to interact, the current state of a component named  $a$  is used to fill the holes in  $P$ . Given a process (or program)  $P$ , process  $a[P]$  denotes the component  $a$  with state  $P$ . Alternatively, it can be seen as a process  $P$  located at  $a$ . Components act as *transparent* localities: a process inside a component can evolve on its own, and interact freely with external processes. The rest of the syntax follows standard lines. A process  $\pi.P$  performs prefix  $\pi$  and then behaves as  $P$ . Parallel composition  $P \parallel Q$  decrees the concurrent execution of  $P$  and  $Q$ . We abbreviate  $P_1 \parallel \dots \parallel P_n$  as  $\prod_{i=1}^n P_i$ . Given an index set  $I = \{1, \dots, n\}$ , the guarded sum  $\sum_{i \in I} \pi_i.P_i$  represents an exclusive choice over  $\pi_1.P_1, \dots, \pi_n.P_n$ . As usual, we write  $\pi_1.P_1 + \pi_2.P_2$  if  $|I| = 2$ , and  $\mathbf{0}$  if  $I$  is empty. Process  $! \pi.P$  defines guarded replication, which expresses infinitely many occurrences of  $P$  which are triggered by prefix  $\pi$ .

*A Structural Characterization of Update.* As anticipated in the Introduction, our structural characterization of update in  $\mathcal{E}$  defines two families of languages, namely  $\mathcal{E}$  with *dynamic topology* (denoted  $\mathcal{E}_d$ ) and  $\mathcal{E}$  with *static topology* (denoted  $\mathcal{E}_s$ ). Here, “dynamic” refers to the ability of creating and deleting new components, something allowed in languages in  $\mathcal{E}_d$  but not in those in  $\mathcal{E}_s$ . The definition of  $\mathcal{E}_d$  and  $\mathcal{E}_s$  is parametric on *update patterns*, denoted  $U$ .

**Definition 1 (Dynamic  $\mathcal{E} - \mathcal{E}_d$ ).** *The class of  $\mathcal{E}$  processes with dynamic topology ( $\mathcal{E}_d$ ), is described by the following grammar:*

$$P ::= a[P] \mid P \parallel P \mid !\pi.P \mid \sum_{i \in I} \pi_i.P_i \quad \pi ::= a \mid \bar{a} \mid \tilde{a}\{U\}$$

The definition of  $\mathcal{E}_s$  makes use of two distinct rule productions  $P$  and  $A$ :  $P$  are generic terms defining the static topology populated by terms  $A$  that do not include subterms of the kind  $a[Q]$ .

**Definition 2 (Static  $\mathcal{E} - \mathcal{E}_s$ ).** *The class of  $\mathcal{E}$  processes with static topology ( $\mathcal{E}_s$ ) is described by the following grammar:*

$$\begin{aligned} P &::= a[P] \mid P \parallel P \mid !\pi.A \mid \sum_{i \in I} \pi_i.A_i \\ A &::= A \parallel A \mid !\pi.A \mid \sum_{i \in I} \pi_i.A_i \quad \pi ::= a \mid \bar{a} \mid \tilde{a}\{a[U] \parallel A\} \end{aligned}$$

Definition 2 exploits syntactic restrictions to ensure that the structure of components in  $\mathcal{E}_s$  processes remains invariant. The first one concerns update prefixes: update prefixes are of the form  $a[U] \parallel A$ : since component structure must be preserved we require that the outermost component  $a$  is recreated. Moreover, for the same reason, the processes inside  $a$  cannot be relocated by putting  $\bullet$  outside the  $a$  component. The second restriction concerns the use of production rules for  $A$ , and ensures that no new component is create after a prefix. Apart from the above, the main difference between dynamic and static topologies is the semantics of update, as we will discuss below.

*Remark 1.* Observe that every term of a calculi in  $\mathcal{E}_s$  is also a term of a calculi in  $\mathcal{E}_d$ . In fact, in the dynamic case updates always include the case  $U = a[U'] \parallel A$ .

*A Behavioral Characterization of Update.* We now move on to consider concrete update patterns  $U$  and their associated variants of  $\mathcal{E}_d$  and  $\mathcal{E}_s$ . We begin by introducing a way of extending rule productions with process holes.

**Definition 3.** Given a rule production  $E$ , we denote with  $E_\bullet$  the production obtained from  $E$  by: (i) considering a rule “ $E_\bullet ::= \text{term}_\bullet$ ” for each rule “ $E ::= \text{term}$ ” of  $E$ , where “ $\text{term}_\bullet$ ” is obtained from “ $\text{term}$ ” by syntactically replacing all process categories  $F$  occurring in “ $\text{term}$ ” by  $F_\bullet$ ; (ii) adding a new rule “ $E_\bullet ::= \bullet$ ”.

**Full  $\mathcal{E}$  ( $\mathcal{E}_d^1$  and  $\mathcal{E}_s^1$ ).** The first update pattern admits all kinds of contexts for update prefixes, i.e.  $U = P_\bullet$ . The variants of  $\mathcal{E}_d$  and  $\mathcal{E}_s$  that adopt the above update pattern are denoted as  $\mathcal{E}_d^1$  and  $\mathcal{E}_s^1$ , respectively.

**Unguarded  $\mathcal{E}$  ( $\mathcal{E}_d^2$  and  $\mathcal{E}_s^2$ ).** In this update pattern, the hole  $\bullet$  cannot occur in the scope of prefixes in  $U$ :  $U ::= P \mid a[U] \mid U \parallel U \mid \bullet$

As before, the variants of  $\mathcal{E}_d$  and  $\mathcal{E}_s$  that adopt the patterns above are denoted  $\mathcal{E}_d^2$  and  $\mathcal{E}_s^2$ , respectively.

**Preserving  $\mathcal{E}$  ( $\mathcal{E}_d^3$  and  $\mathcal{E}_s^3$ ).** In the third update pattern, the current state of the component is preserved, and so it is only possible to add new components and/or behaviors in parallel or to relocate it:  $U ::= a[U] \mid U \parallel P \mid \bullet$

The variants of  $\mathcal{E}_d$  and  $\mathcal{E}_s$  that adopt the patterns above are denoted  $\mathcal{E}_d^3$  and  $\mathcal{E}_s^3$ , respectively.

*Semantics.* We now define a Labeled Transition System (LTS) for both  $\mathcal{E}_d$  and  $\mathcal{E}_s$ . We introduce some auxiliary definitions first.

**Definition 4 (Normal Form/Parallel Processes).** An  $\mathcal{E}$  process  $P$  is in normal form iff

$$P = \prod_{i=1}^m P_i \parallel \prod_{j=1}^n a_j[P'_j]$$

where, for  $i \in \{1, \dots, m\}$ ,  $P_i$  is not in the form  $Q \parallel Q'$  or  $a[Q]$ . Moreover, for  $j \in \{1, \dots, n\}$ ,  $P'_j$  is in normal form. Given  $P$  in normal form, we denote with  $\text{Par}(P)$  the set  $\{P_i \mid i \in [1..m]\} \cup \{a_i[P'_i] \mid i \in [1..n]\}$  of all processes in parallel at top level. The definition is extended to sets of processes in normal form in the expected way.

We make use of a structural congruence relation defined as the minimal congruence such that  $P \equiv P \parallel \mathbf{0}$ ,  $P \parallel Q \equiv Q \parallel P$  and  $P \parallel (Q \parallel R) \equiv (P \parallel Q) \parallel R$ . As usual,  $\equiv$  is preserved by the operational semantics, in the following sense: if  $P \equiv Q$  and  $P \xrightarrow{\alpha} P'$ , then also  $Q \xrightarrow{\alpha} Q'$  for some  $P' \equiv Q'$ . Due to structural congruence all processes can be rewritten in normal form, as stated in the following lemma.

**Lemma 1.** Every  $P \in \mathcal{E}$  is structurally congruent to a process in normal form.

Finally, we define the component structure denotation of a process.

$$\begin{array}{c}
\text{COMP } a[P] \xrightarrow{a[P]} \star \quad \text{SUM } \sum_{i \in I} \alpha_i. P_i \xrightarrow{\alpha_i} P_i \quad \text{REPL } !\alpha. P \xrightarrow{\alpha} P \parallel !\alpha. P \\
\\
\text{ACT1 } \frac{P_1 \xrightarrow{\alpha} P'_1}{P_1 \parallel P_2 \xrightarrow{\alpha} P'_1 \parallel P_2} \quad \text{TAU1 } \frac{P_1 \xrightarrow{\alpha} P'_1 \quad P_2 \xrightarrow{\bar{\alpha}} P'_2}{P_1 \parallel P_2 \xrightarrow{\tau} P'_1 \parallel P'_2} \\
\\
\text{LOC } \frac{P \xrightarrow{\alpha} P'}{a[P] \xrightarrow{\alpha} a[P']} \quad \text{TAU3 } \frac{P_1 \xrightarrow{a[Q]} P'_1 \quad P_2 \xrightarrow{\bar{a}\{U\}} P'_2 \quad \text{cond}(U, Q)}{P_1 \parallel P_2 \xrightarrow{\tau} P'_1\{U\{Q/\bullet\}/\star\} \parallel P'_2}
\end{array}$$

**Fig. 1.** LTS for  $\mathcal{E}$  (both for variants with static and dynamic topology). Rules ACT2, TAU2, and TAU4, the symmetric counterparts of ACT1, TAU1, and TAU3, have been omitted.

**Definition 5 (Component Structure).** Let  $P = \prod_{i=1}^m P_i \parallel \prod_{j=1}^n a_j[P'_j] \in \mathcal{E}$  be a process in normal form. The component structure denotation of  $P$ , denoted with  $St(P)$ , is built as follows. The root is labeled  $\epsilon$ , and has  $n$  children: the subtrees recursively built from processes  $P'_1, \dots, P'_n$  respectively, where the only difference is that the roots are labeled  $a_1, \dots, a_n$ .

We shall define two different LTSs for  $\mathcal{E}_s$  and  $\mathcal{E}_d$ . However, both LTSs are generated by the same set of rules, and they only differ on a condition associated to update actions. This is the content of the following definition.

**Definition 6 (LTS for  $\mathcal{E}_d$  and  $\mathcal{E}_s$ ).** The LTS for  $\mathcal{E}_d$ , denoted  $\longrightarrow_d$ , is defined by the rules in Figure 1 in which we decree  $\text{cond}(U, Q) = \text{true}$  in rules TAU3 and TAU4.

Similarly, the LTS for  $\mathcal{E}_s$ , denoted  $\longrightarrow_s$ , is defined by the rules in Figure 1 in which we decree  $\text{cond}(U, Q) = St(a[Q]) = St(U\{Q/\bullet\})$  in rules TAU3 and TAU4.

We now give some intuitions on the rules in 1, which are common to both  $\longrightarrow_d$  and  $\longrightarrow_s$ . There are five kinds of actions; we use  $\alpha$  to range over them. In addition to the standard input, output, and  $\tau$  actions, we consider two complementary actions for component update:  $\bar{a}\{U\}$  and  $a[P]$ . The former represents the offering of an update  $U$  for component  $a$ ; the latter expresses the fact that component  $a$  with current state  $P$  is ready to update. We often write  $\xrightarrow{\alpha}$  instead of  $\xrightarrow{\alpha}_d$  and  $\xrightarrow{\alpha}_s$ ; the actual LTS used in each case will be clear from the context. Similarly, we use  $\longrightarrow$  to denote  $\xrightarrow{\tau}$ . Intuitions for some rules of the LTS follow. Rule COMP represents the participation of a component  $a$  in an update operation; we use  $\star$  to denote a unique placeholder. Rule LOC formalizes transparency of components. Process evolvability is formalized by rule TAU3, and is the only point in which the LTS of  $\mathcal{E}_d$  and that of  $\mathcal{E}_s$  differs, as formalized by condition  $\text{cond}(U, Q)$ . The update action offers a process  $U$  for updating component  $a$  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. Provided that  $\text{cond}(U, Q)$  holds, the update action is completed by replacing all occurrences of  $\star$  in  $P'_1$  with  $U\{Q/\bullet\}$ .

*Remark 2.* It can be easily verified that transitions preserve the syntactical constraints and the component structure of static processes. That is, if  $P \in \mathcal{E}_s$  and  $P \xrightarrow{\alpha} P'$  then also  $P' \in \mathcal{E}_s$ . Moreover  $St(P') = St(P)$ .

*From Static to Dynamic Terms.* We now discuss and formalize the conditions under which an  $\mathcal{E}_s$  process can be transformed into an equivalent  $\mathcal{E}_d$  process. This formalization will be useful to transfer (un)decidability results between  $\mathcal{E}_s$  and  $\mathcal{E}_d$ . We first introduce some terminology. Given  $P \in \mathcal{E}_s$ , we denote with  $Sub_{st}(P)$  the set  $\{St(a[P']) \mid a[P'] \text{ subt } P\}$ , where *subt* is the subterm relation. Given a  $P_\bullet$  term, we use:  $num_\bullet(P_\bullet)$  to denote the number of  $\bullet$  syntactically occurring in  $P_\bullet$  not inside an update prefix, and  $num_c(P_\bullet)$  to denote the number of  $a[P']$  conents (for any  $a, P'$ ) syntactically occurring in  $P_\bullet$  not inside an update prefix. We also use  $a$  to denote  $St(a[\mathbf{0}])$ .

We have already remarked the fact that every process in  $\mathcal{E}_s$  is also a process in  $\mathcal{E}_d$ . We are interested in the class of  $\mathcal{E}_s$  processes that behaves accordingly to the semantics of  $\mathcal{E}_d$ . That is to say,  $\mathcal{E}_s$  processes for which the condition  $\text{cond}(U, Q)$  always holds. The following lemma characterizes this class of processes.

**Lemma 2.** *Let  $P, Q \in \mathcal{E}_s$ . We have  $St(a[Q]) = St(a[P_\bullet\{Q/\bullet\}] \parallel A)$  if and only if:*

1.  $num_\bullet(P_\bullet) = 0 \wedge St(a[Q]) = St(a[P_\bullet])$
2.  $num_\bullet(P_\bullet) = 1 \wedge num_c(P_\bullet) = 0$
3.  $num_\bullet(P_\bullet) > 1 \wedge num_c(P_\bullet) = 0 \wedge num_c(Q) = 0$

*Remark 3.* Using Lemma 2 it is possible to show that the class of  $\mathcal{E}_s$  processes that do not contain nested components (i.e., “flat” processes) behaves as processes in  $\mathcal{E}_d$ .

Based on the above lemma, it is possible in general to encode a static process in the above class into a process in  $\mathcal{E}_d$  that preserves its behavior. This is the objective of the encoding  $\llbracket \cdot \rrbracket_S^D$ , which we define next. Given a static process  $P$ , such an encoding relates the component structure of  $P$  (i.e., a denotation given by a set of names) with a single name in  $\mathcal{E}_d$ . Also, given a single-child component structure  $CS$ , we use  $l(CS)$  to denote the label of the only child of the root of  $CS$ .

**Definition 7.** *Let  $P \in \mathcal{E}_s$  and  $S$  be a set of single-child component structure denotations such that  $Sub_{st}(P) \subseteq S$ . Assume a set of names  $\mathcal{N}'$  such that  $S \cup \{\text{err}\} \subseteq \mathcal{N}'$ . We inductively define  $\llbracket P \rrbracket_S^D$  as a  $\mathcal{E}_d$  term over a  $\mathcal{N}'$ , as follows:*

- (1)  $\llbracket \xi \tilde{a}\{a[P] \parallel A\}. P'_\bullet \rrbracket_S^D = \xi St(\widetilde{a[P]})\{\llbracket a[P] \parallel A \rrbracket_S^D\}. \llbracket P'_\bullet \rrbracket_S^D$
- (2a)  $\llbracket \tilde{a}\{a[P_\bullet] \parallel A\}. P'_\bullet \rrbracket_S^D = \sum_{C \in S \mid l(C)=a} \tilde{C}\{C\{\llbracket P_\bullet \rrbracket_S^D \parallel A\}. \llbracket P'_\bullet \rrbracket_S^D\}$   
if  $num_\bullet(P_\bullet) = 1 \wedge num_c(P_\bullet) = 0$
- (2b)  $\llbracket !\tilde{a}\{a[P_\bullet] \parallel A\}. P'_\bullet \rrbracket_S^D = \prod_{C \in S \mid l(C)=a} \tilde{C}\{C\{\llbracket P_\bullet \rrbracket_S^D \parallel A\}. \llbracket P'_\bullet \rrbracket_S^D\}$   
if  $num_\bullet(P_\bullet) = 1 \wedge num_c(P_\bullet) = 0$
- (3)  $\llbracket \xi \tilde{a}\{a[P_\bullet] \parallel A\}. P'_\bullet \rrbracket_S^D = \xi \tilde{a}\{a\{\llbracket P_\bullet \rrbracket_S^D \parallel A\}. \llbracket P'_\bullet \rrbracket_S^D\}$  if  $num_\bullet(P_\bullet) > 1 \wedge num_c(P_\bullet) = 0$
- (4)  $\llbracket \xi \tilde{a}\{a[P_\bullet] \parallel A\}. P'_\bullet \rrbracket_S^D = \xi \tilde{err}\{\mathbf{0}\}. \llbracket P'_\bullet \rrbracket_S^D$  if  $num_\bullet(P_\bullet) \geq 1 \wedge num_c(P_\bullet) \neq 0$
- (5)  $\llbracket \xi a.P_\bullet \rrbracket_S^D = \xi a. \llbracket P_\bullet \rrbracket_S^D$
- (6)  $\llbracket \xi \bar{a}. P_\bullet \rrbracket_S^D = \xi \bar{a}. \llbracket P_\bullet \rrbracket_S^D$
- (7)  $\llbracket a[P] \rrbracket_S^D = St(a[P])\{\llbracket P \rrbracket_S^D\}$
- (8)  $\llbracket \sum_{i \in I} \pi_i.P^i \rrbracket_S^D = \sum_{i \in I} \llbracket \pi_i.P^i \rrbracket_S^D$
- (9)  $\llbracket P_\bullet^1 \parallel P_\bullet^2 \rrbracket_S^D = \llbracket P_\bullet^1 \rrbracket_S^D \parallel \llbracket P_\bullet^2 \rrbracket_S^D$

where  $\xi \in \{!, \varepsilon\}$  and  $\varepsilon$  stands for the empty string, i.e. a non-banged prefix in the above.

The following theorem states the correctness of the previous encoding: if  $P \in \mathcal{E}_s$  then  $\llbracket P \rrbracket_S^D \in \mathcal{E}_d$  preserves the behavior of  $P$ .

**Theorem 1.** *Let  $P \in \mathcal{E}_s$  and  $S$  a set of component structure denotations such that  $Sub_{St}(P) \subseteq S$ . We have:  $P \rightarrow_s P' \Leftrightarrow \llbracket P \rrbracket_S^D \rightarrow_d \llbracket P' \rrbracket_S^D$*

*Correctness Properties: Bounded and Eventual Adaptation.* Our correctness properties are stated in terms of observability predicates, or *barbs*. Our definition of barbs is parameterized on the number of repetitions of a given signal. We thus obtain a uniform definition for *bounded* and *repeated* weak barbs.

**Definition 8 (Barbs).** *Let  $P$  be an  $\mathcal{E}$  process, and let  $\alpha$  be an action in  $\{a, \bar{a} \mid a \in \mathcal{N}\}$ . We write  $P \downarrow_\alpha$  if there exists a  $P'$  such that  $P \xrightarrow{\alpha} P'$ . Given  $k > 0$ , we write  $P \Downarrow_\alpha^k$  if there exists a computation  $P = P_0 \rightarrow \dots \rightarrow P_l$  such that the cardinality of the set  $\{j \in [0..l] \mid P_j \xrightarrow{\alpha}\}$  is greater than or equal to  $k$ . Analogously, we write  $P \Downarrow_\alpha^\omega$  if there exist an infinite computation  $P = P_0 \rightarrow \dots \rightarrow P_i \rightarrow \dots$  such that the set  $\{j \in \mathbb{N} \mid P_j \xrightarrow{\alpha}\}$  is infinite. Furthermore, 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 shall consider two instances of the problem of reaching an error configuration in an aggregation of terms, or *cluster*. A *cluster* is a process obtained as the parallel composition of an initial process with an arbitrary amount of processes representing the subsequent modifications.

**Definition 9 (Cluster).** *Let  $P$  be the initial process and  $M$  be the set of processes  $\{P_1, \dots, P_n\}$  representing the possible subsequent modifications. The set of clusters is defined as:  $\mathcal{CS}_P^M = \{P \parallel \prod_{m_1} P_1 \parallel \dots \parallel \prod_{m_n} P_n \mid m_1, \dots, m_n \in \mathbb{N}\}$ .*

The *adaptation* properties we are about to introduce formalize correctness of clusters with respect to their ability for recovering from errors by means of evolvability actions. More precisely, given a set of clusters  $\mathcal{CS}_P^M$  and a distinguished barb  $e$  (signaling an error), we would like to know if all computations of processes in  $\mathcal{CS}_P^M$  (1) have *at most*  $k$  states that exhibit  $e$ , or (2) *eventually* reach a state from which no other barbs on  $e$  will be observed.

**Definition 10 (Bounded and Eventual Adaptation).**

- The bounded adaptation problem ( $\mathcal{BA}$  in the following) consists in checking whether given an initial process  $P$ , a set of processes  $M$ , a barb  $e$ , and  $k > 0$ , for all processes  $P \in \mathcal{CS}_P^M$ ,  $P \Downarrow_e^k$  holds.
- The eventual adaptation problem ( $\mathcal{EA}$  in the following) consists in checking whether given an initial process  $P$ , a set of processes  $M$  and a barb  $e$ , for all processes  $P \in \mathcal{CS}_P^M$ ,  $P \Downarrow_e^\omega$  holds.

Similarly as before static clusters can be encoded into equivalent dynamic ones.

**Definition 11.** *Given  $P \in \mathcal{E}_s$  and  $M = \{P_1, \dots, P_n\} \subset \mathcal{E}_s$  we turn the static cluster set  $\mathcal{CS}_P^M$  into a dynamic cluster set  $\llbracket \mathcal{CS}_P^M \rrbracket_S^D = \mathcal{CS}_{P'}^{M'}$  by taking  $P' = \llbracket P \rrbracket_S^D$  and  $M' = \{\llbracket P_1 \rrbracket_S^D, \dots, \llbracket P_n \rrbracket_S^D\}$ , where  $S = Sub_{St}(P) \cup \bigcup_{1 \leq i \leq n} Sub_{St}(P_i)$ .*

**Theorem 2.** *Given  $P \in \mathcal{E}_s$  and  $M = \{P_1, \dots, P_n\} \subset \mathcal{E}_s$ , we have  $[[\mathcal{CS}_P^M]]_S^{\mathcal{D}} = \{[C]_S^{\mathcal{D}} \mid C \in \mathcal{CS}_P^M\}$ , where  $S = \text{Sub}_{St}(P) \cup \bigcup_{1 \leq i \leq n} \text{Sub}_{St}(P_i)$ .*

Notice that, for every cluster  $C$  in  $[[\mathcal{CS}_P^M]]_S^{\mathcal{D}}$  we obviously have  $\text{Sub}_{St}(C) \subseteq S$ , hence Theorem 1 is individually applicable to each cluster.

### 3 Preliminaries

The following results and definitions are from [17], unless differently specified. Standard definitions and notation for well-quasi-orders (wqo), transition systems, and well-structured transition systems can be found in Appendix D.

Given a quasi-order  $\leq$  over  $X$ , an *upward-closed set* is a subset  $I \subseteq X$  such that the following holds:  $\forall x, y \in X : (x \in I \wedge x \leq y) \Rightarrow y \in I$ . Given  $x \in X$ , we define its upward closure as  $\uparrow x = \{y \in X \mid x \leq y\}$ . This notion can be extended to sets as expected: given a set  $Y \subseteq X$  we define its upward closure as  $\uparrow Y = \bigcup_{y \in Y} \uparrow y$ .

**Definition 12 (Finite basis).** *A finite basis of an upward-closed set  $I$  is a finite set  $B$  such that  $I = \bigcup_{x \in B} \uparrow x$ .*

The notion of basis is particularly important when considering the basis of the predecessor of a state in a transition system. More precisely, we are interested in effective pred-basis as defined below.

**Definition 13 (Effective pred-basis).** *A well-structured transition system has effective pred-basis if there exists an algorithm such that, for any any state  $s \in S$ , it returns the set  $pb(s)$  which is a finite basis of  $\uparrow \text{Pred}(\uparrow s)$ .*

The following proposition is a special case of Proposition 3.5 in [17].

**Proposition 1.** *Let  $TS = (S, \rightarrow, \leq)$  be a finitely branching, well-structured transition system with strong compatibility, decidable  $\leq$  and effective pred-basis. It is possible to compute a finite basis of  $\text{Pred}^*(I)$  for any upward-closed set  $I$  given via a finite basis.*

Finally we will use the following proposition, whose proof is immediate.

**Proposition 2.** *Let  $S$  be a finite set. Then the equality is a wqo over  $S$ .*

An extension to the theory of wqo is needed in the rest of the paper. In [24], Kruskal proved that a wqo on a set  $S$  can be extended to the set of finite trees whose nodes have labels ranging in  $S$ ; we refer to this as the set of trees *over*  $S$ . We first define how to extend a quasi order on a set  $S$  to the trees over  $S$ . If  $t$  is a tree and  $n$  a node in  $t$ , we denote with  $\text{label}(n)$  the label of the node  $n$ .

**Definition 14.** *Let  $S$  and  $\leq$  be a set and a wqo over  $S$ , respectively. The relation  $\leq^{tr}$  on the set of trees over  $S$  is defined as follows. Let  $t, u$  be trees over  $S$ . We have that  $t \leq^{tr} u$  iff there exists an injection  $f$  from the nodes of  $t$  to the ones of  $u$  such that:*

1. *Let  $m, n$  be nodes in  $t$ . If  $m$  is an ancestor of  $n$  then  $f(m)$  is an ancestor of  $f(n)$ .*

2. Let  $m, n, p$  be nodes in  $t$ . If  $p$  is the minimal common ancestor of  $m$  and  $n$  then  $f(p)$  is the minimal common ancestor of  $f(m)$  and  $f(n)$ .
3. Let  $n$  be a node in  $t$ . Then  $\text{label}(n) \leq \text{label}(f(n))$ .

The relation  $\leq^{tr}$  is a quasi-order over the trees over  $S$ . It is also a wqo, since we have the following result.

**Theorem 3 (Kruskal [24]).** *Let  $S$  be a set and  $\leq$  a wqo over  $S$ . Then, the relation  $\leq^{tr}$  is a wqo on the set of trees over  $S$ .*

## 4 Undecidability Results for $\mathcal{E}^1$

We prove that  $\mathcal{BA}$  is undecidable in both  $\mathcal{E}_d^1$  and  $\mathcal{E}_s^1$ ; undecidability of  $\mathcal{EA}$  in  $\mathcal{E}_d^1$  and  $\mathcal{E}_s^1$  is described in Section 5.2. The result relies on an encoding of Minsky machines into  $\mathcal{E}_s^1$  which satisfies the following: a Minsky machine terminates if and only if its encoding into  $\mathcal{E}_s^1$  evolves into at least  $k$  processes that can perform a distinguished barb  $e$ .

The encoding, denoted  $\llbracket \cdot \rrbracket_M$ , can be found in Appendix E. Next we give an informal description. A register  $j$  with value  $m$  is represented by a component  $r_j$  that contains the encoding of number  $m$ , denoted  $\langle \langle m \rangle \rangle_j$ . In turn,  $\langle \langle m \rangle \rangle_j$  consists of a chain of  $m$  nested output prefixes on name  $u_j$ . The encoding of zero is given by an output action on  $z_j$ . Instructions are encoded as replicated processes guarded by  $p_i$ , which represents the state of the Minsky machine when the program counter  $p = i$ . Once  $p_i$  is consumed, each instruction is ready to interact with the registers. The encoding of an increment operation on the value of register  $r_j$  consists in the enlargement of the chain of nested output prefixes it contains. The component  $r_j$  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 the value of register  $j$  consists of an internal, 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 internal choice is indeed deterministic: 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 Minsky machine reaches the HALT instruction the encoding can either exhibit a barb on  $e$ , or set the program counter again to  $p_1$  so as to repeat the whole computation from the beginning, thus passing through a state that exhibits  $e$  at least  $k > 0$  times. We can show the following lemma.

**Lemma 3.** *Let  $N$  be a Minsky machine and  $k \geq 1$ .  $N$  terminates iff  $\llbracket N \rrbracket_M \Downarrow_e^k$ .*

Lemma 3 allows to conclude that  $\mathcal{BA}$  is undecidable for processes in  $\mathcal{E}_s^1$ :

**Theorem 4.**  *$\mathcal{BA}$  is undecidable in  $\mathcal{E}_s^1$ .*

The proof of Theorem 4 proceeds by considering a Minsky machine  $N$  and its encoding  $\llbracket N \rrbracket_M$ . We have that  $\mathcal{CS}_{\llbracket N \rrbracket_M}^\emptyset = \{\llbracket N \rrbracket_M\}$ . Undecidability of  $\mathcal{BA}$  follows from undecidability of the termination problem in Minsky machines and Lemma 3. Notice now that since  $\llbracket N \rrbracket_M$  is a process in  $\mathcal{E}_s^1$  that does not contain any nested components, by means of Lemma 2 we can conclude that this result extends to  $\mathcal{E}_d^1$  as well.

**Corollary 1.**  *$\mathcal{BA}$  is undecidable in  $\mathcal{E}_d^1$ .*

## 5 (Un)decidability Results for $\mathcal{E}^2$

### 5.1 Decidability of Bounded Adaptation

Here we prove that  $\mathcal{BA}$  is decidable for  $\mathcal{E}_d^2$  processes. The proof appeals to the theory of well-structured transition systems [17,5]. We define  $\preceq$ , a quasi-ordering on processes which turns out to be a well-quasi-ordering that is strongly compatible wrt  $\rightarrow$ . This will allow us to compute finite basis for the set of predecessors of the processes exposing a given barb  $\alpha$  (which turns out to be a set upward-closed w.r.t.  $\preceq$ ).

The above strategy requires Kruskal's theorem on well-quasi-orderings on trees (Theorem 3). Unlike other works exploiting the theory of well-structured transition systems for obtaining decidability results (e.g. [11]) in the case of  $\mathcal{E}_d^2$ , it is not possible to find a bound on the "depth" of processes. As an example, consider the process  $R = a[P] \parallel \tilde{a}\{a[a[\bullet]]\}$ . One possible evolution of  $R$  is when it is always the innermost component which is updated; as a result, one obtains a process with an unbounded number of nested components:  $a[a[\dots a[P]]]$ . Nevertheless, not everything is lost and some regularity can be found also in our case. By mapping processes into particular forms of trees and then exploiting an ordering over those trees, it can be shown that this is indeed a well-quasi-ordering with strong compatibility, and has an effective pred-basis. This way, decidability of  $\mathcal{BA}$  can be shown as sketched above.

We start by introducing some auxiliary definitions.

**Definition 15 (Sequential Subprocesses).** Let  $P \in \mathcal{E}_d^2$ . Using  $\mu$  to stand for an input or output prefix, the set  $sub(P)$ , containing all the sequential subprocesses of  $P$ , is defined inductively as follows:

$$\begin{aligned} sub(\mu.P) &= \{\mu.P\} \cup sub(P) & sub(\tilde{a}\{P\}.Q) &= \{\tilde{a}\{P\}.Q\} \cup sub(P) \cup sub(Q) \\ sub(a[P]) &= sub(P) & sub(\sum_i \pi_i.P_i) &= \{\sum_i \pi_i.P_i\} \cup \bigcup_i sub(\pi_i.P_i) \\ sub(\bullet) &= \emptyset & sub(!\pi.P) &= \{!\pi.P\} \cup sub(P) \\ sub(P \parallel Q) &= sub(P) \cup sub(Q) \end{aligned}$$

The set  $Cn(P) = \{a[\ ] \mid a[Q] \text{ appears in } P\}$  stands for the names of components appearing in  $P$ . The definition of  $sub$  and  $Cn$  are extended to sets of processes in the expected way. Given a set of terms  $S$  we denote with  $Lab(S)$  the set containing all the sequential subterms and the component names appearing in  $S$ :  $Lab(S) = sub(S) \cup Cn(S)$ . We now define the tree denotation of a process.

**Definition 16 (Tree of a process).** Let  $P = \prod_{i=1}^m P_i \parallel \prod_{j=1}^n a_j[P'_j] \in \mathcal{E}_d^2$  be a process in normal form. The tree denotation of  $P$ , denoted  $Tr(P)$ , is built as follows. The root is labeled  $\epsilon$ , and has  $m + n$  children: the former  $m$  are singletons labeled  $P_1, \dots, P_m$ , while the latter  $n$  are subtrees recursively built from processes  $P'_1, \dots, P'_n$ , where the only difference is that their roots are labeled  $a_1[\ ], \dots, a_n[\ ]$ , respectively.

Notice that if  $P$  is a process in  $\mathcal{E}_d^2$  then  $Tr(P)$  is a tree over  $Lab(P) \cup \{\epsilon\}$ . We use  $\mathcal{T}_S$  to denote the set of all trees over  $Lab(S) \cup \{\epsilon\}$ . We now define the order  $\preceq$  on processes. It can be seen as the extension of  $=$  to trees as in Definition 14. More precisely, given  $P, Q \in \mathcal{E}_d^2$ , we decree  $P \preceq Q$  iff  $Tr(P) =^{tr} Tr(Q)$ .

By Proposition 2 and by noticing that set  $Lab(S)$  is finite we have that  $=$  is a wqo over  $Lab(S) \cup \{\epsilon\}$ . Moreover, by Lemma 3,  $=^{tr}$  is a wqo over  $\mathcal{T}_S$ .

**Theorem 5 (Wqo).** *Let  $S$  be a set of terms in  $\mathcal{E}_d^2$ . The relation  $=^{tr}$  is a wqo over  $\mathcal{T}_S$ .*

The next theorem states that  $\preceq$  is strongly compatible with respect to the LTS of  $\mathcal{E}_d^2$ .

**Theorem 6 (Strong Compatibility).** *Let  $P, Q, P' \in \mathcal{E}_d^2$ . If  $P \preceq Q$  and  $P \rightarrow P'$  then there exists  $Q'$  such that  $Q \rightarrow Q'$  and  $P' \preceq Q'$ .*

We now move on to define the finite basis for predecessors. To this end, we introduce a notion of *reduction contexts* over  $\mathcal{E}_d^2$  processes. Reduction contexts are trees extended with variables in a very restricted sense, and are useful to reason about (and characterize) the parts of a process that intervene in a reduction. Let  $(X_i)_{i \geq 1}$  be an infinite sequence of variables disjoint from  $\mathcal{N}$ . Consider the tree denotations for  $\mathcal{E}_d^2$  extended with labels including variables, in  $X_i$  respecting the following conditions: (i) each variable occurs at most once in a term; and (ii) each variable is the label of a leaf node. We denote this extended set of trees with  $\mathcal{E}_D^2$ . It might be clear now that we are considering contexts in which variables act as the holes. Given  $D \in \mathcal{E}_D^2$ , we sometimes write  $D[\tilde{X}]$  to indicate that the variables in  $D$  are exactly  $\tilde{X} = X_1, \dots, X_n$ . Hence, given  $\tilde{P} = P_1, \dots, P_n$  we denote by  $D[\tilde{P}]$  the tree obtained by replacing in  $D[\tilde{X}]$  the leaves labeled with  $X_i$  with the trees  $Tr(P_i)$ , respectively.

We show that the well-quasi-ordering given above has an effective pred-basis. The pred-basis function  $pb_S(Q)$  can be intuitively explained as follows. Given a process  $Q$ , we consider all the decompositions of  $Tr(Q)$  as  $D[\tilde{R}]$  (with  $|\tilde{R}| = 0, 1, \text{ or } 2$ ). There are finitely many such decompositions. Then, for each of such  $D$ , we build all contexts  $D'$ , where  $D'$  corresponds to the extension of  $D$  with at most two variables that can be added at the labels of nodes in  $D$ . We denote these contexts with  $Ext(D)$ . Each context  $D' \in Ext(D)$  is then filled in such a way that the corresponding processes have a reduction to  $Q$ , up to the ordering  $\preceq$ . Notice that considering at most 2 holes is related to the fact that in every reduction up to 2 subprocesses are involved.

Let  $P \in \mathcal{E}_d^2$ ; we define the updates occurring in  $P$  as  $Upd(P) = \{U \mid \tilde{a}\{U\} \text{ occurs in } P\}$ . This notation extends to sets of processes in the expected way.

**Definition 17.** *Let  $Q$  and  $S = \{T_1, \dots, T_n\}$  be a process in  $\mathcal{E}_d^2$  and a set of terms in  $\mathcal{E}_d^2$ , respectively. Let  $D, D'$  range over  $\mathcal{E}_D^2$ . We define  $pb_S(Q)$  as:*

$$\begin{aligned} & \bigcup_{Tr(Q)=D[\tilde{R}]} \{P \mid Tr(P) = D'[\tilde{G}], P \rightarrow_{\succeq} Q, D' \in Ext(D) \text{ and} \\ & \quad \tilde{G} \subseteq sub(S) \cup \{a[\mathbf{0}], a[H] \mid a \in \mathbf{Cn}(S) \text{ and } H \in sub(S)\} \cup \\ & \quad \{a[H] \mid R = U\{H/\bullet\}, R \in \tilde{R}, U \in Upd(S) \text{ and contains at least one } \bullet\} \} \end{aligned}$$

The construction of  $pb_S(Q)$  is effective. In particular, given  $D$ , there are finitely many ways of adding one or two holes to  $D$ , so as to obtain a  $D'$  such that  $D' \in Ext(D)$ . Notice that when filling the contexts with terms in  $\tilde{G}$  the sequential subprocesses are not enough. In fact, we also have to consider the cases when an update occurs: those cases are ruled out by  $\{a[\mathbf{0}], a[H]\}$ . Next, we let  $Pred_S(\cdot)$  be all processes  $Q$  that are in  $Pred(\cdot)$ —see Definition 22—and such that  $Tr(Q)$  is over  $\mathcal{T}_S$ .

**Theorem 7.** *Let  $S = \{T_1, \dots, T_n\}$  be a set of terms in  $\mathcal{E}_d^2$  and  $Q \in \mathcal{E}_d^2$  such that  $Tr(Q)$  is over  $\mathcal{T}_S$ . We then have that  $\uparrow pb_S(Q) = \uparrow Pred_S(\uparrow Q)$ . Moreover,  $pb_S(\cdot)$  is effective.*

Exploiting Proposition 1 we have the following.

**Theorem 8.** *Let  $S = \{T_1, \dots, T_n\}$  be a set of terms in  $\mathcal{E}_d^2$ . Given  $\mathcal{S} = \{Q \in \mathcal{E}_d^2 \mid \text{Tr}(Q) \in \mathcal{T}_S\}$ , then  $(\mathcal{S}, \rightarrow, \preceq)$  is a finitely branching, well-structured transition system with strong compatibility, decidable  $\preceq$ , and effective pred-basis  $\text{pb}_{\mathcal{S}}$ . Then it is possible to compute a finite basis of  $\text{Pred}^*(I)$  (and  $\text{Pred}^+(I)$ ) for any upward-closed set  $I$  given via a finite basis.*

The final step for proving the decidability is to define the set of processes that immediately exhibit a barb  $\alpha$ , i.e., a finite basis.

**Definition 18.** *Let  $P \in \mathcal{E}_d^2$  and  $\alpha \in \{a, \bar{a} \mid a \in \mathcal{N}\}$ . Then,  $\text{fb}_{\alpha}(P) = \{R \in \text{sub}(P) \mid R \downarrow_{\alpha}\}$ . The definition is extended to sets of processes in the expected way.*

To determine whether  $P \Downarrow_{\alpha}^k$  it is sufficient to check if  $P$  appears in the set of the predecessors of the processes that can perform  $\alpha$  a least  $k - 1$  times. Since  $\preceq$  imposes a well quasi order on the processes of  $\mathcal{E}_d^2$  it is enough to characterize the set of predecessors by means of finite basis as shown by Theorem 8. More precisely, if  $k = 1$  then it is sufficient to check if  $P$  is in the set of predecessors of the processes that can immediately perform  $\alpha$ . Otherwise, if  $k > 1$  then we need to determine the existence of  $k$  evolutions of  $P$  with the ability of performing  $\alpha$ : i.e.,  $P = P_1 \xrightarrow{*} P_2 \xrightarrow{+} \dots \xrightarrow{+} P_{k+1}$  and  $P_i \xrightarrow{\alpha}$  for  $i \in [2..k+1]$ . To do this, we proceed backwards: we first calculate the basis  $I$  of the set of processes that can immediately perform  $\alpha$ . Thus  $P_{k+1}$  should be in the upward closure of  $I$ . Then, as  $P_k$  should be in the upward closure of  $\text{Pred}^+(I)$ —the set of processes that can perform  $\alpha$  in one or more steps—and it also exposes the barb  $\alpha$ , we intersect the set  $\text{Pred}^+(I)$  with the set of processes that can immediately perform  $\alpha$ . Thus  $P_k$  should be in the upward closure of the obtained basis. This algorithm iterates  $k$  times. At this point it is sufficient to check whether  $P$  is in the upward closure of the finally obtained basis. More formally, we define  $\text{FB}_{\alpha,k}(P) = \text{Pred}^*(\text{FB}'_{\alpha,k}(P))$  where

$$\text{FB}'_{\alpha,k}(P) = \begin{cases} \text{fb}_{\alpha}(P) & \text{if } k = 1 \\ \text{Ib}_{\alpha}(\text{Pred}^+(\text{FB}'_{\alpha,k-1}(P)), \text{fb}_{\alpha}(P)) & \text{otherwise} \end{cases}$$

where  $\text{Ib}_{\alpha}(A, B) = \{Q \in A \mid Q \xrightarrow{\alpha}\} \cup \{\text{Add}_B(Q) \mid Q \in A \text{ and } Q \not\xrightarrow{\alpha}\}$  and where  $\text{Add}_B(Q) = \{Q \parallel R \mid R \in B\} \cup \{C[a[S \parallel R]] \mid Q = C[a[S]] \text{ and } R \in B\}$  is the set of processes obtained by adding in turn, each of the processes in  $B$ , in parallel at top level or inside every location appearing in  $Q$ . The definition is extended to sets of processes in the expected way. The effectiveness of  $\text{FB}_{\alpha,k}$  allows to prove the decidability of  $\mathcal{BA}$ , a more formal proof can be found in Appendix F.1

**Theorem 9.**  *$\mathcal{BA}$  is decidable for  $\mathcal{E}_d^2$ .*

Since  $\mathcal{E}_d^3$  is a subcalculus of  $\mathcal{E}_d^2$  and by means of Theorems 1 and 2, the decidability result extends to  $\mathcal{E}_s^2$ ,  $\mathcal{E}_d^3$ , and  $\mathcal{E}_s^3$ .

**Corollary 2.**  *$\mathcal{BA}$  is decidable for  $\mathcal{E}_s^2$ ,  $\mathcal{E}_d^3$ , and  $\mathcal{E}_s^3$ .*

REGISTER  $r_j$

$$\llbracket r_j = m \rrbracket_M = \begin{cases} r_j[!inc_j. \bar{u}_j \parallel \bar{z}_j] & \text{if } m = 0 \\ r_j[!inc_j. \bar{u}_j \parallel \prod_1^m \bar{u}_j \parallel \bar{z}_j] & \text{if } m > 0. \end{cases}$$

INSTRUCTIONS ( $i : I_i$ )

$$\begin{aligned} \llbracket (i : \text{INC}(r_j)) \rrbracket_M &= !p_i. b. \overline{inc_j. \bar{p}_{i+1}} \\ \llbracket (i : \text{DECJ}(r_j, s)) \rrbracket_M &= !p_i. (u_j. (\bar{b} \parallel \bar{p}_{i+1}) + z_j. \tilde{r}_j \{ r_j[!inc_j. \bar{u}_j \parallel \bar{z}_j] \}. \bar{p}_s) \\ \llbracket (i : \text{HALT}) \rrbracket_M &= !p_i. (e + \bar{p}_1) \end{aligned}$$

**Table 1.** Encoding of Minsky Machines into  $\mathcal{E}_s^2$

## 5.2 Undecidability of Eventual Adaptation

Here we show that  $\mathcal{EA}$  is undecidable in  $\mathcal{E}_s^2$  by relating it to termination in Minsky machines; this result carries over to  $\mathcal{E}_s^1$ ,  $\mathcal{E}_d^1$ , and  $\mathcal{E}_d^2$ —see Corollary 3. This relationship is obtained by defining an encoding tailored to the features of the property. In contrast to the encoding given in Section 4, the encoding presented here is *non faithful* as it may perform erroneous tests for zero on the registers (i.e. in the simulation of the Minsky machine a register is assumed to contain the value zero even if this is not the case). Nevertheless, we are able to guarantee that in an infinite computation the number of these erroneous steps is finite. This way, the Minsky machine terminates iff its encoding has a non terminating computation that exhibits a barb on  $e$  infinitely often. In fact, the barb  $e$  indicates that the simulation of the Minsky machine is restarted: so we have that infinitely many simulations are executed and, as only finitely many erroneous steps can be performed, at least one of the simulations is correct (to be more precise, infinitely many simulations are correct). It then follows that  $\mathcal{EA}$  is undecidable for  $\mathcal{E}_s^2$  processes.

The encoding of Minsky machines into  $\mathcal{E}_s^2$ , denoted by  $\llbracket \cdot \rrbracket_M$ , is presented in Table 1. We first give the encoding of a Minsky machine into  $\mathcal{E}_s^2$  and provide some intuitions on its behavior. Then, we discuss its correctness and state the undecidability result.

**Definition 19.** *Let  $N$  be a Minsky machine, with registers  $r_0, r_1$  and instructions  $(1 : I_1), \dots, (n : I_n)$ . Given the encodings in Table 1, the encoding of  $N$  in  $\mathcal{E}_s^2$  (denoted with  $\llbracket N \rrbracket_M$ ) is defined as  $\llbracket r_0 = 0 \rrbracket_M \parallel \llbracket r_1 = 0 \rrbracket_M \parallel \prod_{i=1}^n \llbracket (i : I_i) \rrbracket_M \parallel \bar{a} \parallel !a. (\bar{b} \parallel (\bar{a} + \bar{p}_1))$ .*

A register  $r_j$  that stores number  $m$  is encoded as a component that contains  $m$  copies of the unit process  $\bar{u}_j$ . Such a component also contains process  $!inc_j. \bar{u}_j$  which allows to create further copies of  $\bar{u}_j$  when an increment instruction is invoked. Instructions are encoded as replicated processes guarded by  $p_i$ . Once  $p_i$  is consumed, each instruction is ready to interact with the registers. This depends on the presence of suitable actions on  $b$ . In order to guarantee that at most a finite number of mistakes are performed, before the execution of the first instruction arbitrarily many processes  $\bar{b}$  are generated—this is the role of process  $\bar{a} \parallel !a. (\bar{b} \parallel (\bar{a} + \bar{p}_1))$  in Definition 19. In case of an increment, one of these provided processes is consumed and the instruction is actually performed; in case of a decrement, a process  $\bar{b}$  is restored. This way, after a correct simulation of the Minsky machine, the number of processes  $\bar{b}$  remains invariant as the number of decrements corresponds to the number of executed decrements. If an incorrect simulation is executed, we have that the number of increments is greater than the of decrements so the number of processes  $\bar{b}$  strictly decreases. A deadlock state is reached when performing an increment in the absence of processes  $\bar{b}$ .

REGISTER  $r_j$   
 $\llbracket r_j = 0 \rrbracket_M = r_j[Reg_j \parallel c_j[0]]$  with  $Reg_j = !inc_j. \tilde{c}_j\{c_j[\bullet]\}. \overline{ack}. u_j. \tilde{c}_j\{c_j[\bullet]\}. \overline{ack}$   
 INSTRUCTIONS ( $i : I_i$ )  
 $\llbracket (i : \text{INC}(r_j)) \rrbracket_M = !p_i. b. \overline{inc_j}. (ack. (\overline{w} \parallel \overline{p_{i+1}}))$   
 $\llbracket (i : \text{DECJ}(r_j, s)) \rrbracket_M = !p_i. (\overline{u_j}. ack. (\overline{b} \parallel \overline{p_{i+1}}) + \tilde{c}_j\{\bullet\}. \tilde{r}_j\{r_j[Reg_j \parallel c_j[\bullet]]\}. \overline{p_s})$   
 $\llbracket (i : \text{HALT}) \rrbracket_M = !p_i. (e + \overline{p_1})$

**Table 2.** Encoding of Minsky machines into  $\mathcal{E}_d^3$ .

While the encoding of an increment instruction of register  $r_j$  simply consists in creating additional copies of  $\overline{u_j}$ , the encoding of a decrement-and-jump instruction is slightly more involved. The instruction is implemented as an internal choice: the process can choose either to actually perform a decrement and proceed with the next instruction, or to jump. This choice can be seen as guess on the actual value stored by the register. We therefore have two situations. In the first case the process can choose to decrement the register, and a synchronization on  $u_j$  takes place. This is enough to implement a decrement on the value of the register, and to spawn the next instruction. In the second case, the process chooses to jump. Therefore, the content of the component  $r_j$  is immediately discarded, and the register is recreated with content zero, i.e., without copies of  $\overline{u_j}$ . Notice that if the value of the register is equal to zero, this is a right guess, and the instruction  $p_s$  is enabled. If the value of the register is greater than zero then this corresponds to a wrong guess. Finally, similarly to the encoding of Minsky machine into  $\mathcal{E}_s^1$ , as soon as the HALT instruction is reached, the barb on  $e$  is exposed and the computation can be restarted by setting the program counter to the first instruction. As a consequence we obtain that a Minsky machine  $N$  terminates iff its encoding into  $\mathcal{E}_s^2$  can reach a state that can exhibit a barb on the observable action  $e$  infinitely often.

**Lemma 4.** *Let  $N$  be a Minsky machine.  $N$  terminates iff  $\llbracket N \rrbracket_M \Downarrow_e^\omega$ .*

Lemma 4 allows to conclude that  $\mathcal{EA}$  is undecidable for processes in  $\mathcal{E}_s^2$ :

**Theorem 10.**  *$\mathcal{EA}$  is undecidable in  $\mathcal{E}_s^2$ .*

The proof of Theorem 10 proceeds exactly as the proof of Theorem 4 for  $\mathcal{E}_s^1$ . Similarly as in that case, undecidability extends also to  $\mathcal{E}_d^2$ ,  $\mathcal{E}_s^1$  and  $\mathcal{E}_d^1$ . This easily follows from the fact that  $\mathcal{E}_s^2$  is a subcalculus of  $\mathcal{E}_s^1$  and from Lemma 2, since  $\llbracket N \rrbracket_M$  is a process in  $\mathcal{E}_s^2$  that does not contain any nested components.

**Corollary 3.**  *$\mathcal{EA}$  is undecidable in  $\mathcal{E}_s^1$ ,  $\mathcal{E}_d^1$ , and  $\mathcal{E}_d^2$ .*

## 6 (Un)decidability Results for $\mathcal{E}^3$

### 6.1 Undecidability of Eventual Adaptation in $\mathcal{E}_d^3$

Here we prove that  $\mathcal{EA}$  is undecidable for  $\mathcal{E}_d^3$  processes. We obtain this result by means of a non-faithful encoding of Minsky machines.

Similarly to the encoding into  $\mathcal{E}_s^2$ , the encoding of Minsky machines into  $\mathcal{E}_d^3$  is based on the possibility of performing at most a finite amount of mistakes. Moreover, the distinctive feature of  $\mathcal{E}_d^3$  is that the previous state of the component is always preserved in update operations. This way the encoding of a decrement instruction becomes challenging. Our approach is to implement two mechanisms: the first one isolates certain processes and the second one avoids interaction with the rest of the encoding. For this purpose each register contains a component that collects all those pieces that one would like to isolate, and at the same time the collector acts as a disabling process.

The encoding of Minsky machines into  $\mathcal{E}_s^3$ , denoted by  $\llbracket \cdot \rrbracket_M$ , is presented in Table 2. We begin by giving the encoding of a Minsky machine into  $\mathcal{E}_s^3$  by assuming that before and after the execution of a Minsky program the registers are set to zero (this is a nonrestrictive assumption). Then, we provide some intuitions on the behavior of the encoding. Finally, we discuss its correctness and state the undecidability result.

**Definition 20.** *Let  $N$  be a Minsky machine, with registers  $r_0, r_1$  and instructions  $(1 : I_1), \dots, (n : I_n)$ . Given the encodings in Table 2, the encoding of  $N$  in  $\mathcal{E}_d^3$  (denoted with  $\llbracket N \rrbracket_M$ ) is defined as  $\llbracket r_0 = 0 \rrbracket_M \parallel \llbracket r_1 = 0 \rrbracket_M \parallel \prod_{i=1}^n \llbracket (i : I_i) \rrbracket_M \parallel \bar{a} \parallel !a.(\bar{b} \parallel (\bar{a} + \bar{p}_1))$ .*

A register  $r_j$  that stores number  $m$  is encoded as a component  $r_j$  that contains  $m$  copies of the unit process  $u_j. \tilde{c}_j\{c_j[\bullet]\}. \overline{ack}$ . Such a component also contains process  $Reg_j$  which allows us to create further copies of the unit process when an increment instruction is invoked. Furthermore, we use the collector  $c_j$  to store processes which are meant to be isolated. The instructions are defined taking into account this role of  $c_j$ . An increment instruction adds an occurrence of  $u_j. \tilde{c}_j\{c_j[\bullet]\}. \overline{ack}$ . Notice that it could occur that an output on *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 an internal choice. The process tests if the content of the register is equal or greater than zero. If the process guesses that the register is zero, before jumping to the given instruction, it proceeds at disabling its current content: this is done by putting the collector at top level in the register, the register is then recreated by placing all its previous content in the collector. A decrement instead removes one occurrence of  $u_j. \tilde{c}_j\{c_j[\bullet]\}. \overline{ack}$ . Notice that as before it could occur that the output on  $u_j$  could synchronize with the corresponding input inside a collected process. Again this immediately leads to deadlock. Moreover, by using a blocking mechanism on instructions similar to the one used in Section 5.2, we ensure that the number of mistakes is finite. Finally, in order to obtain an infinite sequence of states that can exhibit a barb on  $e$ , as soon as the computation reaches the HALT instruction the barb can be exposed or the Minsky machine is restarted by setting the program counter to the first instruction.

**Lemma 5.** *Let  $N$  be a Minsky machine.  $N$  terminates iff  $\llbracket N \rrbracket_M \Downarrow_e^\omega$ .*

Lemma 5 allows to conclude that  $\mathcal{EA}$  is undecidable for processes in  $\mathcal{E}_d^3$ . The proof of the following theorem proceeds as the proofs of Theorems 4 and 10.

**Theorem 11.**  *$\mathcal{EA}$  is undecidable in  $\mathcal{E}_d^3$ .*

## 6.2 Decidability of Eventual Adaptation in $\mathcal{E}_s^3$

We prove the decidability of  $\mathcal{EA}$  in  $\mathcal{E}_s^3$  by resorting to Petri nets, an infinite-state concurrent computational model for which several problems are decidable (see, e.g., [16]).

The idea is to use the markings of the Petri net to represent the active sequential subprocesses and the available components. Transitions are used to model the execution of actions. More precisely, each active sequential subprocess is represented by one token. Two tokens corresponding to two sequential subprocesses able to execute complementary actions can fire a transition, whose effect is to produce tokens representing the two continuations. As for update actions, they are represented by transitions that consume (at least) two tokens: one corresponding to the process executing the update and one representing the component target of the update operations. It is necessary to check that the process is not in the target component, as a process can update only components in parallel with it. To this aim, we need to keep track of the components in which a process is included by decorating its place with the list of outer components. Formally, let  $P$  be a process of  $\mathcal{E}_s^3$  and  $M = \{P_1, \dots, P_n\}$  be a set of processes of  $\mathcal{E}_s^3$ . In the light of Theorem 1 it is not restrictive to assume that all the update actions on  $a$  can be executed on every component having name  $a$  (even if under the static semantics an update action can be executed only if the update does not modify the component nesting structure). Let  $\mathcal{P}_{seq}(P, M)$  be the set of sequential subprocesses in  $P, P_1, \dots, P_n$  and let  $\mathcal{A}(P, M)$  be the set of component names nestings, i.e. strings composed of names of nested components, starting from the outermost component, occurring in one of the processes  $P, P_1, \dots, P_n$ . The set of places of the corresponding Petri net, denoted  $\text{Places}(P, M)$ , is defined as  $\{\langle P, \alpha \rangle \mid P \in \mathcal{P}_{seq}(P, M), \alpha \in \mathcal{A}(P, M)\} \uplus \{start, check, go, stop\}$ , where *start*, *check*, *go*, and *stop* are distinct auxiliary places.

We now move to the description of the Petri net computation. The initial marking include one token in the place *start* plus the tokens corresponding to the active sequential subprocesses of  $P$ . Formally, these are the places  $dec_\varepsilon(P)$  (with  $\varepsilon$  corresponding to the empty string) where  $dec_\alpha(P)$  is defined inductively as follows:

$$\begin{aligned} dec_\alpha(P \parallel P') &= dec_\alpha(P) \uplus dec_\alpha(P') & dec_\alpha(a[P]) &= dec_{\alpha a}(P) \uplus \{\alpha a\} \\ dec_\alpha(P) &= \{\langle P, \alpha \rangle\} & & \text{otherwise} \end{aligned}$$

where  $\uplus$  denotes multiset union. The initial marking is  $\text{Init}(P) = dec_\varepsilon(P) \uplus \{start\}$ .

The token in *start* allows to generate an arbitrary amount of copies of the processes  $P_1, \dots, P_n$ . This is simply achieved by considering  $n$  transitions, such that the  $i$ -th transition tests for the presence of the token in *start* and then produces the sequential subprocesses of  $P_i$ . Nondeterministically, the token is moved from *start* to *go*. At this point the simulation of the evolution of the generated configuration is started. As described above, synchronizations between complementary actions are modeled by transitions that consume the tokens corresponding to the two synchronizing processes and then produce the sequential subprocesses in the continuations. As for update actions, we need to check the availability of a target component, but this component should not enclose the updating process. Let  $Q$  be the process executing the update action on the component name  $a$ , and let  $\alpha$  be the string of the names of its enclosing components: the availability of the target component can be checked by verifying whether there is a token in a place  $\beta a$  which is not a prefix of  $\alpha$ . If  $\beta a$  is a prefix of  $\alpha$ , the component corresponding to  $\beta a$  could enclose

the updating process: in this case it is sufficient to check that the place  $\beta a$  contains at least two tokens, thus indicating the existence of a distinct component with the same path that does not enclose the updating process.

To check that a given barb  $\alpha$  is exposed infinitely often during the simulated computation, we consider one transition able to add one token to the *check* place every time the barb is exposed. This way, *check* is unbounded if and only if barb  $\alpha$  can be exposed infinitely often. The entire set of transitions  $\text{Trans}(P, M, \alpha)$  is given in Appendix G.1. So we can formally define the Petri net for the given process  $P$  and the set of processes  $M$  as the triple  $(\text{Places}(P, M), \text{Trans}(P, M, \alpha), \text{Init}(P, M))$ . We can now state the correspondence between  $\mathcal{E}_s^3$  processes and their corresponding Petri net. The decidability of  $\mathcal{EA}$  for  $\mathcal{E}_s^3$  follows from the decidability of place boundedness in Petri nets.

**Theorem 12.** *Let  $P$  be a process of  $\mathcal{E}_s^3$ , and let  $M$  be the set  $\{P_1, \dots, P_n\}$  of processes of  $\mathcal{E}_s^3$ . Consider  $S = \text{Sub}_{St}(P) \cup \text{Sub}_{St}(P_1) \cup \dots \cup \text{Sub}_{St}(P_n)$ , and let  $P' = \llbracket P \rrbracket_S^D$  and  $M' = \{\llbracket P_1 \rrbracket_S^D, \dots, \llbracket P_n \rrbracket_S^D\}$ . Let  $\alpha$  be a barb. We have that  $P$  and  $M$  satisfies  $\mathcal{EA}$  for the barb  $\alpha$  iff the place *check* is bound in the Petri net  $(\text{Places}(P', M'), \text{Trans}(P', M', \alpha), \text{Init}(P', M'))$ .*

## 7 Concluding Remarks

We have provided a complete investigation of the (un)decidability properties of *bounded* and *eventual* adaptation, two novel correctness properties for component-based systems. Our study has relied on the  $\mathcal{E}$  calculus, an extension of Milner's CCS with a construct for components and update capabilities. The (un)decidability of the adaptation properties is analyzed in different variants of  $\mathcal{E}$  which result from considering dynamic and static component topologies as well as three different possibilities for behavior associated to update capabilities. Our results shed light on the nature of verification in a number of application areas in which software construction proceeds by the systematic combination of predefined blocks. As discussed in the introduction, we plan to investigate the influence behavioral interfaces have on the expressiveness of process calculi for components and on the (un)decidability of associated correctness properties. In this spirit, investigating whether the constructs of (fragments of)  $\mathcal{E}$  can be suitably enriched with the kind of interfaces defined in MECo [32] appears interesting.

## References

1. *The Fractal Project Web Page*.
2. .NET Framework Developer Center, Windows Workflow Foundation. <http://msdn.microsoft.com/en-us/netframework/aa663328.aspx>.
3. Open source erlang. <http://www.erlang.org/>.
4. The Enterprise JavaBeans specification, Version 3.0. <http://java.sun.com/products/ejb/docs.html>.
5. 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.
6. 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.

7. J. Armstrong. *Making reliable distributed systems in the presence of software errors*. PhD thesis, SICS, 2003.
8. M. Bravetti and G. Zavattaro. On the expressive power of process interruption and compensation. *Math. Struct. in Comp. Sci.*, 19(3):565–599, 2009.
9. 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.
10. T. Bures, P. Hnetynka, and F. Plasil. Sofa 2.0: Balancing advanced features in a hierarchical component model. In *Proc. of SERA'06*, pages 40–48. IEEE Computer Society, 2006.
11. N. Busi, M. Gabbriellini, 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.
12. A. Cansado, C. Canal, G. Salaün, and J. Cubo. A formal framework for structural reconfiguration of components under behavioural adaptation. *Electr. Notes Theor. Comput. Sci.*, 263:95–110, 2010.
13. G. Delzanno, A. Sangnier, and G. Zavattaro. Parameterized verification of ad hoc networks. In *Proc. of CONCUR*, volume 6269 of *Lecture Notes in Computer Science*, pages 313–327. Springer, 2010.
14. J. Esparza. Some applications of petri nets to the analysis of parameterised systems, 2003. Talk at WISP'03.
15. J. Esparza, A. Finkel, and R. Mayr. On the verification of broadcast protocols. In *Proc. of LICS*, pages 352–359, 1999.
16. J. Esparza and M. Nielsen. Decidability issues for petri nets - a survey. *Bulletin of the EATCS*, 52:244–262, 1994.
17. A. Finkel and P. Schnoebelen. Well-structured transition systems everywhere! *Theor. Comput. Sci.*, 256(1-2):63–92, 2001.
18. G. Göbller, S. Graf, M. E. Majster-Cederbaum, M. Martens, and J. Sifakis. An approach to modelling and verification of component based systems. In *Proc. of SOFSEM'07*, volume 4362 of *Lecture Notes in Computer Science*, pages 295–308. Springer, 2007.
19. G. Göbller, S. Graf, M. E. Majster-Cederbaum, M. Martens, and J. Sifakis. Ensuring properties of interaction systems. In *Program Analysis and Compilation*, volume 4444 of *Lecture Notes in Computer Science*, pages 201–224. Springer, 2007.
20. G. Göbller and J. Sifakis. Composition for component-based modeling. *Sci. Comput. Program.*, 55(1-3):161–183, 2005.
21. D. Gupta, P. Jalote, and G. Barua. A formal framework for on-line software version change. *IEEE Trans. Softw. Eng.*, 22(2):120–131, 1996.
22. P. Hnetynka and F. Plasil. Dynamic reconfiguration and access to services in hierarchical component models. In *Proc. of CBSE'06*, volume 4063 of *Lecture Notes in Computer Science*, pages 352–359. Springer, 2006.
23. R. M. Karp and R. E. Miller. Parallel program schemata. *J. Comput. Syst. Sci.*, 3(2):147–195, 1969.
24. J. B. Kruskal. Well-quasi-ordering, the tree theorem, and vazsonyi's conjecture. *Transactions of the American Mathematical Society*, 95(2):210–225, 1960.
25. J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. In *Prod. of ESEC'95*, volume 989 of *Lecture Notes in Computer Science*, pages 137–153. Springer, 1995.
26. R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.
27. M. Minsky. *Computation: Finite and Infinite Machines*. Prentice-Hall, 1967.
28. O. Nierstrasz and F. Achemann. A calculus for modeling software components. In *Proc. of FMCO'02*, volume 2852 of *Lecture Notes in Computer Science*, pages 339–360. Springer, 2003.
29. OMG. CORBA Component Model, V4.0. <http://www.omg.org/spec/CCM/4.0/>.

30. J. A. Pérez. *Higher-Order Concurrency: Expressiveness and Decidability Results*. PhD thesis, University of Bologna, 2010. Draft in [www.japerez.phipages.com](http://www.japerez.phipages.com).
31. 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.
32. D. Sangiorgi and F. Montesi. A model of evolvable components. In *Proc. of 5th International Symposium on Trustworthy Global Computing (TGC 2010)*, Lecture Notes in Computer Science. Springer, 2010.
33. 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.
34. G. Stoyle, M. W. Hicks, G. M. Bierman, P. Sewell, and I. Neamtiu. *Mutatis Mutandis*: Safe and predictable dynamic software updating. *ACM Trans. Program. Lang. Syst.*, 29(4), 2007.
35. M. Wermelinger and J. L. Fiadeiro. A graph transformation approach to software architecture reconfiguration. *Sci. Comput. Program.*, 44(2):133–155, 2002.
36. T. Wies, D. Zufferey, and T. A. Henzinger. Forward analysis of depth-bounded processes. In *FOSSACS*, volume 6014 of *Lecture Notes in Computer Science*, pages 94–108. Springer, 2010.

## A More on Related Work

Other formal approaches to dynamic update and/or component-based systems include [35,34,20,12]. Wermelinger and Fiadeiro [35] use graph rewriting/category theory to formalize software evolution. Stoyle et al [34] proposed a calculus for *dynamic update* in typed, imperative languages. The focus is on *type-safe* updates—intuitively, the consistent update of type  $\tau$  with some new type  $\tau'$ . There is no knowledge about future software updates; type coercions mechanisms are then used to recast new (in principle, unknown) types to old types. In contrast, in our case “update code” is defined in advance. In fact, there is a conceptual difference between *update* (as in works such as [34]) and *dynamic adaptation*, as we have considered it here. In the model of interaction systems [20], the behavior of a set of components (given by a transition system for each one of them) is defined separately from the way in which such components interact (given by a set of connectors). The aim is to ensure properties such as deadlock-freedom by construction. Other properties—including liveness, local/global deadlock, and fairness—have been considered in subsequent works (see, e.g., [19,18]). Both the model of interaction systems and the nature of the studied properties studied in [19,18] are rather different from our framework and goals. Cansado et al [12] proposed a framework for structural reconfiguration with behavioral adaptation considerations. In [12], component architectures are given by *nets* of interacting components represented by Labeled Transition Systems (LTSs). Notice that the concept of “behavioral adaptation” in [12] is different from our notion of adaptation. The former refers to the changes required in component interfaces so as to achieve effective compositions. Instead, and as argued above, our notion of adaptation concerns a higher abstraction level, as we address dynamic reconfiguration by abstracting from interfaces.

## B Additional Results for Section 2

Here we prove the Theorems of Section 2

**Lemma 6 (Lemma 2).** *Let  $P, Q \in \mathcal{E}_s$ . We have  $St(a[Q]) = St(a[P_\bullet\{Q/\bullet\}]) \parallel A$  if and only if:*

1.  $num_\bullet(P_\bullet) = 0 \wedge St(a[Q]) = St(a[P_\bullet])$
2.  $num_\bullet(P_\bullet) = 1 \wedge num_c(P_\bullet) = 0$
3.  $num_\bullet(P_\bullet) > 1 \wedge num_c(P_\bullet) = 0 \wedge num_c(Q) = 0$

*Proof (Sketch).* The only non-obvious part is the "only if" one. In the case  $num_\bullet(P_\bullet) = 0$  than the given constraint reduces to the required one. In the case  $num_\bullet(P_\bullet) = 1$  than having  $num_c(P_\bullet) > 0$  would clearly deny the given constraint in that, for any  $Q$ , it would produce a component structure tree with additional nodes in the righthand side (and component structure trees of terms are always finite). For a similar motivation, in the case  $num_\bullet(P_\bullet) > 1$ , having  $num_c(P_\bullet) > 0$  or  $num_c(Q) > 0$  would deny the given constraint.  $\square$

**Theorem 13 (Theorem 1).** *Let  $P \in \mathcal{E}_s$  and  $S$  a set of component structure denotations such that  $Sub_{St}(P) \subseteq S$ . We have:*

$$P \rightarrow_s P' \Leftrightarrow \llbracket P \rrbracket_S^D \rightarrow_d \llbracket P' \rrbracket_S^D$$

*Proof (Sketch).* The only significant case concern reduction transitions obtained as update synchronization, i.e. that have rule TAU3 in their inference. We start with the  $\Leftarrow$  case. In this case, the synchronizing update prefix must be originated by one of the first three items in the transformation of Definition 7. It is immediate to verify that the three cases correspond (in the same order) to the three cases of Lemma 2, hence they all lead to satisfaction of the static constraint. The  $\Rightarrow$  case is shown similarly. Since the synchronizing update prefix (and the synchronizing component) satisfies the static constraint, one of the three cases of Lemma 2 must hold true. This guarantees that, in the transformation of Definition 7, the synchronizing update prefix is transformed according to one of the first three cases (and e.g. an "err" prefix is not generated) and that the corresponding synchronization can occur also in the transformed term (easily checkable for each case, given the corresponding assert in Lemma 2).

## C Minsky machines: Definition and Semantics

A Minsky machine [27] 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  $(1 : I_1), \dots, (n : I_n)$  can be of two kinds:  $INC(r_j)$  adds 1 to register  $r_j$  and proceeds to the next instruction;  $DECJ(r_j, s)$  jumps to instruction  $s$  if  $r_j$  is zero, otherwise it decreases register  $r_j$  by 1 and proceeds to the next instruction. A Minsky machine includes a program counter  $p$  indicating the label of the instruction being executed. In its initial state, the machine has both registers set to 0 and the program counter  $p$  set to the first instruction. The Minsky machine stops whenever the program counter is set to the HALT instruction. A *configuration* of a Minsky machine is a tuple  $(i, m_0, m_1)$ ; it consists of the current program counter and the values of the registers. Formally, the reduction relation over configurations of a Minsky machine, denoted  $\rightarrow_M$ , is defined in Figure 2. We shall exploit encodings into Minsky machines to prove undecidability of

$$\begin{array}{c}
\text{M-INC} \frac{i : \text{INC}(r_j) \quad m'_j = m_j + 1 \quad m'_{1-j} = m_{1-j}}{(i, m_0, m_1) \longrightarrow_{\text{M}} (i + 1, m'_0, m'_1)} \quad \text{M-JMP} \frac{i : \text{DECJ}(r_j, s) \quad m_j = 0}{(i, m_0, m_1) \longrightarrow_{\text{M}} (s, m_0, m_1)} \\
\text{M-DEC} \frac{i : \text{DECJ}(r_j, s) \quad m_j \neq 0 \quad m'_j = m_j - 1 \quad m'_{1-j} = m_{1-j}}{(i, m_0, m_1) \longrightarrow_{\text{M}} (i + 1, m'_0, m'_1)} \\
\text{M-HALT} \frac{i : \text{HALT}}{(i, m_0, m_1) \dashrightarrow_{\text{M}}}
\end{array}$$

**Fig. 2.** Semantics of Minsky machines

bounded and eventual adaptation. In our encodings, we sometimes make the unrestrictive assumption that at the beginning and at the end of the computation the registers contain the value zero.

## D Well-Structured Transition Systems: Auxiliary Definitions

Recall that a *quasi-order* (or, equivalently, preorder) is a reflexive and transitive relation.

**Definition 21 (Well-quasi-order).** A well-quasi-order (*wqo*) is a quasi-order  $\leq$  over a set  $X$  such that, for any infinite sequence  $x_0, x_1, x_2, \dots \in X$ , there exist indexes  $i < j$  such that  $x_i \leq x_j$ .

Note that if  $\leq$  is a wqo then any infinite sequence  $x_0, x_1, x_2, \dots$  contains an infinite increasing subsequence  $x_{i_0}, x_{i_1}, x_{i_2}, \dots$  (with  $i_0 < i_1 < i_2 < \dots$ ). Thus well-quasi-orders exclude the possibility of having infinite strictly decreasing sequences.

We also need a definition for (finitely branching) transition systems. Here and in the following  $\rightarrow^*$  denotes the reflexive and transitive closure of the relation  $\rightarrow$ .

**Definition 22 (Transition system).** A transition system is a structure  $TS = (S, \rightarrow)$ , where  $S$  is a set of states and  $\rightarrow \subseteq S \times S$  is a set of transitions. We define  $\text{Succ}(s)$  as the set  $\{s' \in S \mid s \rightarrow s'\}$  of immediate successors of  $s$ .  $TS$  is finitely branching if, for each  $s \in S$ ,  $\text{Succ}(s)$  is finite. We also define  $\text{Pred}(s)$  as the set  $\{s' \in S \mid s' \rightarrow s\}$  of immediate predecessors of  $s$ , while  $\text{Pred}^*(s)$  and  $\text{Pred}^+(s)$  denote the sets  $\{s \in S \mid s' \rightarrow^* s\}$  and  $\{s \in S \mid s' \rightarrow^+ s\}$ , respectively, of predecessors of  $s$ .

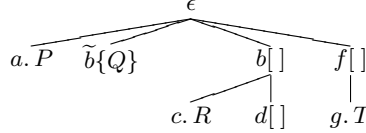
We will also use  $\text{Succ}$ ,  $\text{Pred}$ ,  $\text{Pred}^*$  and  $\text{Pred}^+$  on sets by assuming the point-wise extension of the above definitions.

The key tool to the decidability of several properties of computations is the notion of *well-structured transition system* [17,5]. This is a transition system equipped with a well-quasi-order on states which is (upward) compatible with the transition relation. Here we will use a strong version of compatibility; hence the following definition.

**Definition 23 (Well-structured transition system).** A well-structured transition system with strong compatibility is a transition system  $TS = (S, \rightarrow)$ , equipped with a quasi-order  $\leq$  on  $S$ , such that the two following conditions hold: (i)  $\leq$  is a well-quasi-order;

REGISTER $r_j$	$\llbracket r_j = n \rrbracket_M = r_j[\langle n \rangle_j]$	where	$\langle n \rangle_j = \begin{cases} \bar{z}_j & \text{if } n = 0 \\ \bar{u}_j. \langle n - 1 \rangle_j & \text{if } n > 0. \end{cases}$
INSTRUCTIONS ( $i : I_i$ )			
$\llbracket (i : \text{INC}(r_j)) \rrbracket_M$	$= !p_i. \tilde{r}_j \{ r_j[\bar{u}_j. \bullet] \}. \bar{p}_{i+1}$		
$\llbracket (i : \text{DECJ}(r_j, s)) \rrbracket_M$	$= !p_i. (u_j. \bar{p}_{i+1} + z_j. \tilde{r}_j \{ r_j[\bar{z}_j] \}. \bar{p}_s)$		
$\llbracket (i : \text{HALT}) \rrbracket_M$	$= !p_i. (e + \bar{p}_1)$		

**Table 3.** Encoding of Minsky machines into  $\mathcal{E}_s^1$ .



**Fig. 3.** The tree denotation of the process  $a. P \parallel \tilde{b}\{Q\} \parallel b[c. R \parallel d[]] \parallel f[g. T]$

(ii)  $\leq$  is strongly (upward) compatible with  $\rightarrow$ , that is, for all  $s_1 \leq t_1$  and all transitions  $s_1 \rightarrow s_2$ , there exists a state  $t_2$  such that  $t_1 \rightarrow t_2$  and  $s_2 \leq t_2$  holds.

## E Additional Material and Results for $\mathcal{E}^1$

The encoding of a Minsky machine into  $\mathcal{E}_s^1$  is defined as follows:

**Definition 24.** Let  $N$  be a Minsky machine, with registers  $r_0 = 0, r_1 = 0$  and instructions  $(1 : I_1), \dots, (n : I_n)$ . Given the encodings in Table 3, the encoding of  $N$  in  $\mathcal{E}_s^1$  (denoted with  $\llbracket N \rrbracket_M$ ) is defined as  $\llbracket r_0 = 0 \rrbracket_M \parallel \llbracket r_1 = 0 \rrbracket_M \parallel \prod_{i=1}^n \llbracket (i : I_i) \rrbracket_M \parallel \bar{p}_1$ .

Given this encoding, we have that a Minsky machine  $N$  terminates iff its encoding has at least  $k$  barbs on the distinguished action  $e$ , for every  $k \geq 1$ .

## F Additional Material and Results for $\mathcal{E}^2$

### F.1 Decidability of Bounded Adaptation in $\mathcal{E}_d^2$

Here can be found the proofs of the theorems in Section 5.1.

*Example 1.* Let  $P_0$  be the process  $a. P \parallel \tilde{b}\{Q\} \parallel b[c. R \parallel d[]] \parallel f[g. T]$ . The tree of  $P_0$  as defined in Definition 16 is depicted in Figure 3.

**Theorem 14 (Strong Compatibility - Theorem 6).** Let  $P, Q, P' \in \mathcal{E}_d^2$ . If  $P \preceq Q$  and  $P \rightarrow P'$  then there exists  $Q'$  such that  $Q \rightarrow Q'$  and  $P' \preceq Q'$ .

*Proof (Sketch).* By case analysis on the rule used to infer reduction  $P \rightarrow P'$ . We content ourselves with illustrating the case of update. We then assume  $P = \mathbb{C}_1(\mathbb{C}_2(a[P_1]) \parallel \mathbb{C}_3(\tilde{a}\{P_2\}. R))$ , where  $\mathbb{C}(Q)$  denotes a term  $\mathbb{C}$  with a hole which is filled with the term  $Q$ . Let  $m$  be the node labeled  $\tilde{a}\{P_2\}. R$ , and  $n$  the root labeled  $a$  of the tree  $a[P_1]$ .

We first consider the modifications to  $Tr(P)$  when  $P \rightarrow P'$ . The new  $Tr(P')$  is obtained from  $Tr(P)$  in the following way:

1. the node labeled  $\tilde{a}\{P_2\}.R$  is replaced with  $Tr(R)$ ;
2. the tree rooted in  $a[\ ]$  becomes  $Tr(P_2\{P_1/\bullet\})$  (thus replacing  $Tr(a[P_1])$ ).

We know that since  $P \preceq Q$ ,  $Tr(P) =^{tr} Tr(Q)$ . Hence, there exists a mapping  $f$  that associates nodes in  $Tr(P)$  to nodes in  $Tr(Q)$ . This way there is a node  $f(m)$  in  $Tr(Q)$  labeled  $\tilde{a}\{P_2\}.R$ . Moreover, there exists another node  $f(n)$  labeled with  $a[\ ]$  which has a common ancestor with node  $m$ . The update described above can therefore take place in  $Q$  as well and so  $Q \rightarrow Q'$ . Now,  $Tr(Q')$  is obtained from  $Tr(Q)$  by applying the same changes described above to the target nodes (of the component  $a$  and of the update  $\tilde{a}\{P_2\}.R$ ) according to  $f$ .

The last thing to show is that  $P' \preceq Q'$ , which follows by observing that the mapping between  $Tr(P')$  and  $Tr(Q')$  is the same mapping  $f$  between  $Tr(P)$  and  $Tr(Q)$ , for all the nodes that have not been modified by the reduction and that there is a correspondence one to one for the other nodes. More precisely:

1. Consider the labels in node  $f(m)$ : all nodes removed in  $Tr(P')$  have been removed in  $Tr(Q')$ , hence the nodes  $m$  and  $f(m)$  are still in relation.
2. Finally, we consider the two trees rooted in  $n$  and  $f(n)$ :  $S = Tr(P_2\{P_1/\bullet\})$  and  $T = Tr(P_2\{Q_1/\bullet\})$ , respectively.  $S$  is the same subtree as  $T$  apart from some subtrees of  $P_1$  and  $Q_1$  that can be put easily in relation as the subtrees  $Tr(P_1)$  and  $Tr(Q_1)$  are in relation with  $f$ .

Thus  $Tr(P') =^{tr} Tr(Q')$ . □

**Theorem 15 (Theorem 7).** *Let  $S = \{T_1, \dots, T_n\}$  be a set of terms in  $\mathcal{E}_d^2$  and  $Q \in \mathcal{E}_d^2$  such that  $Tr(Q)$  is over  $\mathcal{T}_S$ . We then have that  $\uparrow pb_S(Q) = \uparrow Pred_S(\uparrow Q)$ . Moreover,  $pb_S(\cdot)$  is effective.*

*Proof (Sketch).* Effectiveness follows from the finiteness of  $pb_P$  as discussed above. The inclusion  $\uparrow pb_P(Q) \subseteq \uparrow Pred_P(\uparrow Q)$  follows by construction. We consider the other inclusion, i.e.,  $\uparrow Pred_P(\uparrow Q) \subseteq \uparrow pb_P(Q)$ . Suppose there is an  $R$  such that  $R \rightarrow_{\succeq} Q$ ; we show that there is an  $S \in pb_P(Q)$  such that  $R \succeq S$ . This follows by an ease case analysis on the kind of reductions that can occur in  $R$ , as discussed above. □

**Lemma 7.** *Let  $S = \{T_1, \dots, T_n\}$  be a set of terms in  $\mathcal{E}_d^2$ . and  $\alpha \in \{a, \bar{a} \mid a \in \mathcal{N}\}$ .  $FB_{\alpha,k}(S)$  is effective*

*Proof (Sketch).* The effectiveness of the calculation of the finite basis of  $Pred^*(\cdot)$  and  $Pred^+(\cdot)$  follows from Theorem 8. The set  $Ib_{\alpha}(\cdot, \cdot)$  is finite and hence can be computed as defined above. Moreover, it is easy to see that it is a finite basis representing all the predecessors of  $fb_{\alpha}(S)$ , which in turn can immediately exhibit  $\alpha$ . □

We can finally conclude that:

**Theorem 16 (Theorem 9).**  *$\mathcal{BA}$  is decidable for  $\mathcal{E}_d^2$ .*

*Proof (Sketch).* Let  $P \in \mathcal{E}_d^2$  and  $M = \{P_1, \dots, P_n\}$  be an initial process and a set of processes, respectively. In order to show that  $\mathcal{BA}$  is decidable, it suffices to check whether there exists a process  $R \in \mathcal{CS}_P^M$  such that  $R \Downarrow_{\alpha}^k$ . More precisely, we have to check if there

exists a process  $Q \in FB_{\alpha,k}(\{P\} \cup M)$  such that  $Q \preceq R$ . From Lemma 7, we know that it is possible to compute the set  $FB_{\alpha,k}(\{P\} \cup M)$ . Then, for each  $Q_i \in FB_{\alpha,k}(\{P\} \cup M)$  we analyze the processes in  $Par(Q_i)$ . Let  $S$  be the set of the processes  $Q'_j$  in  $Par(Q_i)$  such that  $Q'_j \preceq T$ , for some  $T \in M$ . We now consider  $Q_i^*$ , the process obtained by  $Q_i$  by removing all the occurrences of the parallel processes in  $S$ . At this point, it is enough to check whether  $Q_i^* \preceq P$ . If this is the case, for at least one  $Q_i \in FB_{\alpha,k}(\{P\} \cup M)$ , we can conclude that there exists  $R \in \mathcal{CS}_P^S$  such that  $R \Downarrow_{\alpha}^k$ , otherwise there exists no  $R \in \mathcal{CS}_P^S$  such that  $R \Downarrow_{\alpha}^k$ .  $\square$

## G Additional Material and Results for $\mathcal{E}^3$

### G.1 Decidability of Eventual Adaptation in $\mathcal{E}_s^3$

A *Petri net* is a tuple  $N = (S, T, m_0)$ , where  $S$  and  $T$  are finite sets of *places* and *transitions*, respectively. A finite multiset over the set  $S$  of places is called a *marking*, and  $m_0$  is the initial marking. Given a marking  $m$  and a place  $p$ , we say that the place  $p$  contains  $m(p)$  *tokens* in the marking  $m$  if there are  $m(p)$  occurrences of  $p$  in the multiset  $m$ . A transition is a pair of markings written in the form  $m' \Rightarrow m''$ . The marking  $m$  of a Petri net can be modified by means of transitions firing: a transition  $m' \Rightarrow m''$  can fire if  $m(p) \geq m'(p)$  for every place  $p \in S$ ; upon transition firing the new marking of the net becomes  $n = (m \setminus m') \uplus m''$  where  $\setminus$  and  $\uplus$  are the difference and union operators for multisets, respectively. This is written as  $m \rightarrow n$ . A marking  $m_n$  is *reachable* if there exists a sequence  $m_0 \rightarrow m_1 \rightarrow \dots \rightarrow m_n$ . A place  $p$  is *bound* if there exists  $k$  such that for each reachable marking  $m$  we have that  $m(p) \leq k$ . It is well known that place boundedness is decidable in Petri nets (this can be checked, e.g., by verifying if all the markings in the coverability tree [23] do not contain  $\omega$  in the place to be checked).

Here, we give the formal definition of the transitions of the Petri net defined in Section 6.2. More precisely, we define the set of transitions  $\text{Trans}(P, M, \alpha)$  as the instantiations to places in  $\text{Places}(P, M)$  of the transitions schemata in Table 4. So we can formally define the Petri net for the given process  $P$  and the set of processes  $M$  as the triple  $(\text{Places}(P, M), \text{Trans}(P, M, \alpha), \text{Init}(P, M))$ .

$\{start\} \Rightarrow \{start\} \uplus dec_\varepsilon(T_i)$ with $T_i \in S$
$\{start\} \Rightarrow \{go\}$
$\{X, \sum_{i \in I} \langle \beta_i. A_i, \alpha \rangle, \sum_{j \in J} \langle \gamma_j. B_j, \beta \rangle\} \Rightarrow \{go\} \uplus dec_\alpha(A_l) \uplus dec_\beta(B_m)$ if $\beta_l = a$ and $\gamma_m = \bar{a}$ (for $l \in I, m \in J$ ) and $X \in \{go, stop\}$
$\{X, \langle !\beta. A, \alpha \rangle, \sum_{j \in J} \langle \gamma_j. B_j, \beta \rangle\} \Rightarrow \{go, \langle !\beta. A, \alpha \rangle\} \uplus dec_\alpha(A) \uplus dec_\beta(B_m)$ if $\beta = a$ (resp. $\bar{a}$ ) and $\gamma_m = \bar{a}$ (resp. $a$ ) (for $m \in J$ ) and $X \in \{go, stop\}$
$\{X, \langle !\beta. A, \alpha \rangle, \langle !\gamma. B, \beta \rangle\} \Rightarrow \{go, \langle !\beta. A, \alpha \rangle, \langle !\gamma. B, \beta \rangle\} \uplus dec_\alpha(A) \uplus dec_\beta(B_m)$ if $\beta = a$ (resp. $\bar{a}$ ) and $\gamma = \bar{a}$ (resp. $a$ ) and $X \in \{go, stop\}$
$\{X, \langle \sum_{i \in I} \beta_i. A_i, \alpha \rangle, \beta a\} \Rightarrow \{go\} \uplus dec_\alpha(A_l) \uplus dec_\beta(A) \uplus dec_{\beta a}(U) \uplus \{\beta a\}$ if $\beta a$ is not a prefix of $\alpha$ , $\beta_l = \tilde{a}\{a[U] \mid A\}$ (for $l \in I$ ) and $X \in \{go, stop\}$
$\{X, \langle \sum_{i \in I} \beta_i. A_i, \alpha \rangle, \beta a, \beta a\} \Rightarrow \{go\} \uplus dec_\alpha(A_l) \uplus dec_\beta(A) \uplus dec_{\beta a}(U) \uplus \{\beta a, \beta a\}$ if $\beta a$ is a prefix of $\alpha$ , $\beta_l = \tilde{a}\{a[U] \mid A\}$ (for $l \in I$ ) and $X \in \{go, stop\}$
$\{X, \langle !\beta. A', \alpha \rangle, \beta a\} \Rightarrow \{go, \langle !\beta. A', \alpha \rangle\} \uplus dec_\alpha(A') \uplus dec_\beta(A) \uplus dec_{\beta a}(U) \uplus \{\beta a\}$ if $\beta a$ is not a prefix of $\alpha$ , $\beta = \tilde{a}\{a[U] \mid A\}$ and $X \in \{go, stop\}$
$\{X, \langle !\beta. A', \alpha \rangle, \beta a, \beta a\} \Rightarrow \{go, \langle !\beta. A', \alpha \rangle\} \uplus dec_\alpha(A') \uplus dec_\beta(A) \uplus dec_{\beta a}(U) \uplus \{\beta a, \beta a\}$ if $\beta a$ is a prefix of $\alpha$ , $\beta = \tilde{a}\{a[U] \mid A\}$ and $X \in \{go, stop\}$
$\{go, \langle \sum_{i \in I} \beta_i. A_i, \alpha \rangle\} \Rightarrow \{stop, \langle \sum_{i \in I} \beta_i. A_i, \alpha \rangle, check\}$ if $\beta_l = \alpha$ (for some $l \in I$ )
$\{go, \langle !\alpha. A, \alpha \rangle\} \Rightarrow \{stop, \langle !\alpha. A, \alpha \rangle, check\}$

**Table 4.** Petri net transitions