

ADAPTABLE PROCESSES

MARIO BRAVETTI^a, CINZIA DI GIUSTO^b, JORGE A. PÉREZ^c, AND GIANLUIGI ZAVATTARO^d

^{a,d} Laboratory Focus (University of Bologna/INRIA), Italy
e-mail address: bravetti@cs.unibo.it, zavattar@cs.unibo.it

^b CEA, List, France
e-mail address: cinzia.digiusto@gmail.com

^c CITI and Department of Computer Science, FCT New University of Lisbon, Portugal
e-mail address: japerezp@gmail.com

ABSTRACT. We propose the concept of *adaptable processes* as a way of overcoming the limitations that process calculi have for describing patterns of dynamic *process evolution*. Such patterns rely on direct ways of controlling the behavior and location of *running* processes, and so they are at the heart of the *adaptation* capabilities present in many modern concurrent systems. Adaptable processes have a location and are sensible to actions of *dynamic update* at runtime; this allows to express a wide range of evolvability patterns for concurrent processes. We introduce a core calculus of adaptable processes and propose two verification problems for them: *bounded* and *eventual adaptation*. While the former ensures that the number of consecutive erroneous states that can be traversed during a computation is bound by some given number k , the latter ensures that if the system enters into a state with errors then a state without errors will be eventually reached. We study the (un)decidability of these two problems in several variants of the calculus, which result from considering dynamic and static topologies of adaptable processes as well as different evolvability patterns. Rather than a specification language, our calculus intends to be a basis for investigating the fundamental properties of evolvable processes and for developing richer languages with evolvability capabilities.

1. INTRODUCTION

Process calculi aim at describing formally the behavior of concurrent systems. A leading motivation in the development of process calculi has been properly capturing the *dynamic character* of concurrent behavior. In fact, much of the success of the π -calculus [46] can be fairly attributed to the way it departs from CCS [45] so as to describe mobile systems in which communication topologies can change dynamically. Subsequent developments can be explained similarly. For instance, the Ambient calculus [20] builds on π -calculus mobility to describe the dynamics of interaction within boundaries and hierarchies, as required in distributed systems. A commonality in these calculi is that the dynamic behavior of a

1998 ACM Subject Classification: D.2.4, F.3.1, F.3.2, F.4.3.

Key words and phrases: Process calculi, dynamic evolution, expressiveness and decidability, adaptation, verification, evolvable processes.

system is realized through a number of *local changes*, usually formalized by reduction steps. Indeed, while in the π -calculus mobility is enforced by the reconfiguration of individual linkages in the communication topology, in the Ambient calculus spatial mobility is obtained by individual modifications to the containment relations within the ambient hierarchy. This way, the combined effect of a series of changes at a local level (links, containment relations) suffices to explain dynamic behavior at the global (system) level.

There are, however, interesting forms of dynamic behavior that cannot be satisfactorily described as a combination of local changes, in the above sense. These are behavioral patterns which concern change at the *process* level (i.e., the process as a whole), and describe *process evolution* along time. In general, forms of process evolvability are characterized by an enhanced control/awareness over the current behavior and location of running processes. Crucially, this increased control is central to the *adaptation* capabilities by which processes modify their behavior in response to exceptional circumstances in their environment. As a simple example, consider a scheduler in an operating system which manages the execution of a set of processes. To specify the behavior of the scheduler, the processes, and their evolution, we would need mechanisms for *direct* process manipulation, which appear hard to represent in calculi enforcing local changes only. More precisely, it is not clear at all how to represent the *intermittent evolution* of a process under the scheduler’s control: that is, precise ways of describing that its behavior “disappears” (when the scheduler suspends the process) and “appears” (when the scheduler resumes the process). Emerging applications and programming paradigms provide challenging examples of evolvable processes. In workflow applications, we would like to be able to replace a running activity, suspend the execution of a set of activities, or even suspend and relocate the whole workflow. Similarly, in component-based systems we would like to reconfigure parts of a component, a whole component, or groups of components. Also, we would like to specify the context-aware policies that dynamically adapt the computational power of cloud computing applications. At the heart of these applications we find forms of process evolution and adaptation which appear very difficult (if not impossible) to express in existing process calculi.

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

We consider several variants of \mathcal{E} , obtained via two orthogonal characterizations. The first one is *structural*, and defines *static* and *dynamic* topologies of adaptable processes. In a static topology, the number of adaptable processes does not vary along the evolution of the system: they cannot be destroyed nor new ones can appear. In contrast, in the more general

dynamic topology this restriction is lifted. We will use the subscripts s and d to denote the variants of \mathcal{E} with static and dynamic topologies, respectively. The second characterization is *behavioral*, and concerns *update patterns*—the context U in an update prefix $\tilde{a}\{U\}$. As hinted at above, update patterns determine the behavior of running processes 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 by the superscripts 1, 2, and 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 further require contexts to have exactly one hole, thus preserving the current behavior (and possibly adding new behaviors): this is the most restrictive form of update.

In our view, these variants capture a fairly ample spectrum of scenarios that arise in the joint analysis of correctness and adaptation concerns in evolvable systems. They borrow inspiration from existing programming languages, development frameworks, and component models. The structural characterization follows the premise that while it is appealing to define the runtime evolution of the structures underlying aggregations of behaviors, in some scenarios it is also sensible to specify evolution and adaptation preserving such structures. For instance, we would like software updates to preserve the main architecture of our operating system; conversely, an operating system could be designed to disallow runtime updates that alter its basic organization in dangerous ways. A static topology is also consistent with settings in which adaptable processes represent *located resources*, whose creation is disallowed or comes with a cost (as in cloud computing scenarios). 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 [34]. Update patterns are also related to functionalities present in programming languages such as Erlang [1, 7] and in development frameworks such as the Windows Workflow Foundation (WWF) [44]. 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).

Verification of Adaptable Processes. Rather than a specification language, the \mathcal{E} calculus intends to be a basis for investigating the fundamental properties of evolvable processes. In this presentation, we study two *verification problems* associated to \mathcal{E} processes and their (un)decidability. They are defined in terms of standard observability predicates (*barbs*), which indicate the presence of a designated error signal. We thus distinguish between *correct states* (i.e., states in which no error barbs are observable) and *error states* (i.e., states exhibiting error barbs). The first verification problem, *bounded adaptation* (abbreviated BA) ensures that, given a finite k , at most k consecutive error states can arise in computations of the system—including those reachable as a result of dynamic updates. The second one, *eventual adaptation* (abbreviated EA), is similar but weaker: it ensures that if the system enters into an error state then it will eventually reach a correct state. We believe that BA and EA fit well in the kind of correctness analysis that is required in a number of emerging applications. For instance, on the provider side of a cloud computing application, these properties allow to check whether a client is able to assemble faulty systems via the aggregation of the provided services and the possible subsequent updates. On the client

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

Table 1: Summary of (un)decidability results for dialects of \mathcal{E} .

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 a bug in the provider’s infrastructure and not from the initial aggregation and dynamic updates provided by the client.

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

Contributions. The main technical results of the paper are summarized in Table 1. The calculus \mathcal{E}^1 is shown to be Turing complete, and both BA and EA are shown to be *undecidable* for \mathcal{E}^1 processes. The Turing completeness of \mathcal{E}^1 says much on the expressive power of update actions. In fact, it is known that fragments of CCS without restriction can be translated into finite Petri nets (see, e.g., the discussion in [17]), and so they are not Turing complete. Update actions in \mathcal{E} thus allow to “jump” from finite Petri nets to a Turing complete model. We then show that in \mathcal{E}^2 BA is *decidable*, while EA remains *undecidable*. Interestingly, EA is already undecidable in \mathcal{E}_d^3 , while it is *decidable* in \mathcal{E}_s^3 .

We now comment on the proof techniques. The decidability of EA in \mathcal{E}_s^3 is proved by resorting to Petri nets: EA is reduced to a problem on Petri nets that we call *infinite visits* which, in turn, can be reduced to *place boundedness*—a decidable problem for Petri nets. For the decidability of BA we appeal to the theory of *well-structured transition systems* [30, 2] and its associated results. In our case, such a theory must be coupled with Kruskal’s theorem [39] (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 aggregations and dynamic updates that may generate new adaptable processes). This combination of techniques proved to be very challenging. In particular, the technique is more complex than the one given in [3], which relies on a bound on the depth of trees, or the one in [63], where only topologies with bounded paths are taken into account. Kruskal’s theorem is also used in [17] for studying the decidability properties of calculi with exceptions and compensations. The calculi considered in [17] are *first-order*; in contrast, \mathcal{E} can be considered as a *higher-order* process calculus (see Section 9).

The undecidability results are obtained via encodings of Minsky machines [47], a well-known Turing complete model. In particular, the encodings that we provide for showing undecidability of EA in \mathcal{E}^2 and \mathcal{E}_d^3 do 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 [28], but in our case their application revealed much more complex; this is particularly true for the case of \mathcal{E}_d^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 [29] and ad-hoc networks [26]. Differently from [29, 26], 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 process instances (taken from M , the set of possible modifications).

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

- (1) We introduce \mathcal{E} , a core calculus of adaptable processes. \mathcal{E} allows to express a wide range of patterns of process evolution and runtime adaptation. By means of structural and behavioral characterizations, we identify different meaningful variants of the language. We are not aware of other process calculi tailored to the joint representation of evolution and adaptation concerns in concurrent systems.
- (2) We introduce bounded and eventual adaptation—two correctness properties for adaptable processes—and study their (un)decidability in each of the variants of \mathcal{E} . We do so by considering systems as part of clusters which define their evolvability along time. To the best of our knowledge, ours is the first study of the (un)decidability of adaptation properties for dynamically evolvable processes.

This paper is an extended, revised version of the conference paper [14]. In addition to provide full details of the technical results, here we thoroughly develop the structural and behavioral characterizations of adaptable processes. This way, we present a unified treatment of the distinction between the static and dynamic topologies of adaptable processes, as well as of the three different update patterns. These ideas were treated only partially in [14], where the family \mathcal{E}^2 was called \mathcal{E}^- . In particular, new results not presented in [14] include the relationship between static and dynamic topologies (Section 2.3) and the decidability of EA in \mathcal{E}_s^3 by resorting to Petri nets (Section 5.3). Moreover, several examples and extended discussions are included. Section 9.1 is based on the short paper [15].

Structure of the document. The rest of this paper is structured as follows. The \mathcal{E} calculus, its different variants, and several associated results are presented in Section 2. The two verification problems are defined in Section 3. Section 4 presents extended examples of modeling in \mathcal{E} , in several emerging applications. Section 5 collects basic definitions and results on Minsky machines, well-structured transition systems, and Petri nets. (Un)decidability results for \mathcal{E}^1 , \mathcal{E}^2 , and \mathcal{E}^3 are detailed in Sections 6, 7, and 8, respectively. Section 9 presents some additional discussions, and reviews some related works. Section 10 concludes. While proofs of the main results are included in the main text, technical details for some other results (most notably, correctness proofs for the encodings) are collected in the Appendix.

2. A CALCULUS OF ADAPTABLE PROCESSES

We begin by presenting the \mathcal{E} calculus and its different variants. Then, we introduce the operational semantics of the calculus, and establish the relationship between static and dynamic topologies of adaptable processes.

2.1. Syntax. The \mathcal{E} calculus is a variant of CCS [45] 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 $\bar{a}, \bar{b} \dots$ and $\tilde{a}, \tilde{b} \dots$. As customary, we use a and \bar{a} to denote atomic input and output actions, respectively. The syntax of \mathcal{E} processes extends that of CCS with primitive notions of *adaptable processes* $a[P]$ and *update prefixes* $\tilde{a}\{U\}$:

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

Above, the U in the update prefix $\tilde{a}\{U\}$ is an *update pattern*: it represents a context, i.e., a process with zero or more *holes* (see Definition 2.1 below). The intention is that when an update prefix is able to interact, the current state of an adaptable process named a is used to fill the holes in the update pattern U . Given a process P , process $a[P]$ denotes the adaptable process P located at a . Notice that a acts as a *transparent* locality: process P can evolve on its own, and interact freely with external processes. Localities can be nested, so as to form suitable hierarchies of adaptable processes. The rest of the syntax follows standard lines. A process $\pi.P$ performs prefix π 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 $\prod_{i=1}^n P_i$, and use $\prod^k P$ to denote the parallel composition of k instances of process P . 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, i.e., infinitely many occurrences of P in parallel, which are triggered by prefix π .

We now define a general way of extending the grammar of process languages with holes, so as to define update patterns. Intuitively, we extend rule productions with a hole (denoted \bullet), distinguishing between rule productions for process expressions (so-called *process categories*) from the rest. In particular, we would like to avoid adding holes to rule productions for prefixes (i.e., productions for π in the syntax).

Definition 2.1. Given a process category E , we denote with E_\bullet the process category with rule productions obtained from those of E by:

- (1) adding a new rule “ $E_\bullet ::= \bullet$ ”;
- (2) replacing every rule “ $E ::= term$ ” of E with a rule “ $E_\bullet ::= term_\bullet$ ”, where “ $term_\bullet$ ” is obtained from “ $term$ ” by syntactically replacing all process categories F occurring in “ $term$ ” by F_\bullet .

Given an update pattern U and a process Q , we define $U\langle\langle Q \rangle\rangle$ as the process obtained by filling in those holes in U not occurring inside update prefixes with Q .

Definition 2.2. The effect of replacing the holes in an update pattern U with a process Q , denoted $U\langle\langle Q \rangle\rangle$, is defined inductively on U as follows:

$$\begin{aligned} \bullet \langle\langle Q \rangle\rangle &= Q & (U_1 \parallel U_2)\langle\langle Q \rangle\rangle &= U_1 \langle\langle Q \rangle\rangle \parallel U_2 \langle\langle Q \rangle\rangle \\ a[U]\langle\langle Q \rangle\rangle &= a[U\langle\langle Q \rangle\rangle] & \left(\sum_{i \in I} \pi_i. U_i \right) \langle\langle Q \rangle\rangle &= \sum_{i \in I} \pi_i. U_i \langle\langle Q \rangle\rangle \\ (!\pi. U)\langle\langle Q \rangle\rangle &= !\pi. (U\langle\langle Q \rangle\rangle) \end{aligned}$$

This way, $\{\cdot\}$ can be intuitively seen as a scope delimiter for holes \bullet in $\tilde{a}\{U\}$. Indeed, it is worth observing that Definition 2.2 does not replace holes inside prefixes; this ensures a consistent treatment of nested update actions.

We now move on to consider different variants of this basic syntax by means of two different characterizations.

2.1.1. *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 adaptable processes, 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 U .

Definition 2.3 (Dynamic $\mathcal{E} - \mathcal{E}_d$). The class of \mathcal{E} processes with dynamic topology (\mathcal{E}_d) is described by the following grammar:

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

where $U ::= P_\bullet$, as in Definition 2.1.

The definition of \mathcal{E}_s makes use of two distinct process categories: P and A . Intuitively, P correspond to processes defining the (static) topology of adaptable processes; these are populated by terms A , which do not include subprocesses of the kind $a[Q]$.

Definition 2.4 (Static $\mathcal{E} - \mathcal{E}_s$). The class of \mathcal{E} processes with static topology (\mathcal{E}_s) is described by the following grammar:

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

where the syntax $U ::= P_\bullet$, as in Definition 2.1, considering both P and A as process categories.

Definition 2.4 relies on syntactic restrictions to ensure that the nesting structure of adaptable processes in \mathcal{E}_s remains invariant. The first restriction (i.e., no adaptable process is removed) is manifest in update prefixes, which are always of the form $a[U] \parallel A$; this forces the recreation of the adaptable process a after every update, thus maintaining the static structure of adaptable processes invariant. For the same reason, holes can only occur inside the recreated adaptable process: this way, processes cannot be relocated outside $a[U]$. The second restriction (i.e., no adaptable process is created) appears in the definition of A , which decrees that no new adaptable processes occur behind a prefix. As we will discuss below, the operational semantics ensures that these syntactic restrictions are preserved along process execution.

Remark 2.5. Observe that every \mathcal{E}_s process is, from a syntactic point of view, also an \mathcal{E}_d process. In fact, the update pattern $U = a[U'] \parallel A$ is a particular case of the possible update patterns for \mathcal{E}_d processes. The correspondence between processes in \mathcal{E}_s and \mathcal{E}_d from the point of view of their operational semantics will be made more precise by Lemma 2.18.

2.1.2. *A Behavioral Characterization of Update.* We now move on to consider three concrete instances of update patterns U and their associated variants of \mathcal{E}_d and \mathcal{E}_s .

Definition 2.6 (Update Patterns). We shall consider the following three instances of update patterns for \mathcal{E}_s and \mathcal{E}_d :

- (1) **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., $U ::= P \bullet$. These variants, corresponding to the above \mathcal{E}_d and \mathcal{E}_s , are denoted also with \mathcal{E}_d^1 and \mathcal{E}_s^1 , respectively.
- (2) **Unguarded \mathcal{E} (\mathcal{E}_d^2 and \mathcal{E}_s^2).** In the second update pattern, holes cannot occur in the scope of prefixes in U :

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

The variants of \mathcal{E}_d and \mathcal{E}_s that adopt this update pattern are denoted \mathcal{E}_d^2 and \mathcal{E}_s^2 , respectively.

- (3) **Preserving \mathcal{E} (\mathcal{E}_d^3 and \mathcal{E}_s^3).** In the third update pattern, the current state of the adaptable process is always preserved. Hence, it is only possible to add new adaptable processes 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 this update pattern are denoted \mathcal{E}_d^3 and \mathcal{E}_s^3 , respectively.

2.2. **Semantics.** The semantics of \mathcal{E} processes is given in terms of a Labeled Transition System (LTS). We introduce some auxiliary definitions first.

Definition 2.7. *Structural congruence* is the smallest congruence relation generated by the following laws: $P \parallel Q \equiv Q \parallel P$; $P \parallel (Q \parallel R) \equiv (P \parallel Q) \parallel R$.

Definition 2.8 (Normal Form). An \mathcal{E} process P is said to be 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 $a[Q]$, and, for all $j \in \{1, \dots, n\}$, P'_j is in normal form. Note that if $m = 0$ then the normal form is simply $P = \prod_{j=1}^n a_j[P'_j]$; similarly, if $n = 0$ then the normal form is $P = \prod_{i=1}^m P_i$.

Lemma 2.9. *Every \mathcal{E} process is structurally congruent to a process in normal form.* \square

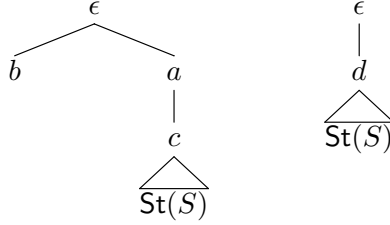


Figure 1: Containment structure denotation for $P = P_1 \parallel b[P_2] \parallel a[P_3 \parallel c[S]]$ and $R = P_1 \parallel d[S]$, as in Example 2.11.

We now define the *containment structure denotation* of a process. Intuitively, it captures the tree-like structure induced by the nesting of adaptable processes.

Definition 2.10 (Containment Structure). Let $P = \prod_{i=1}^m P_i \parallel \prod_{j=1}^n a_j[P'_j]$ be an \mathcal{E} process in normal form. The *containment structure denotation* of P , denoted $\text{St}(P)$, is built as follows. The root is labeled ϵ , and has n children: the subtrees recursively built from processes P'_1, \dots, P'_n with roots labeled a_1, \dots, a_n (instead of ϵ), respectively. When $n = 1$, we say that the containment structure $\text{St}(P)$ is *single-child*.

Example 2.11. Consider the processes P , Q , and R defined as

$$P = P_1 \parallel b[P_2] \parallel a[P_3 \parallel c[S]] \quad Q = a.P_2 \parallel b[P_3] \parallel a[c[S]] \quad R = P_1 \parallel d[S]$$

where P_1, P_2, P_3 do not contain adaptable processes. Then, P and Q have the same containment structure denotation; it is depicted in Figure 1 (left). As for R , the containment structure denotation $\text{St}(R)$, that is single-child, is depicted in Figure 1 (right).

Given an update pattern U , the following two definitions on \mathcal{E}_s processes indicate the number of holes and adaptable processes which syntactically occur in U , respectively. In both cases, we do not consider occurrences inside nested update prefixes.

Definition 2.12. Let U denote an \mathcal{E}_s update pattern or an \mathcal{E}_s process. The number of adaptable processes which occur in U , denoted $|U|_{\text{ap}}$, is inductively defined as follows:

$$\begin{aligned} |\bullet|_{\text{ap}} &= 0 & |U_1 \parallel U_2|_{\text{ap}} &= |U_1|_{\text{ap}} + |U_2|_{\text{ap}} & |!\pi.U|_{\text{ap}} &= 0 \\ |a[U]|_{\text{ap}} &= 1 + |U|_{\text{ap}} & |\sum_{i \in I} \pi_i.U_i|_{\text{ap}} &= 0 \end{aligned}$$

Notice that in the above definition, as we are considering \mathcal{E}_s processes, the number of adaptable processes after a prefix is necessarily 0.

Definition 2.13. Let U be an \mathcal{E}_s update pattern. The number of holes which occur in U , denoted $|U|_{\bullet}$, is inductively defined as follows:

$$\begin{aligned} |\bullet|_{\bullet} &= 1 & |U_1 \parallel U_2|_{\bullet} &= |U_1|_{\bullet} + |U_2|_{\bullet} & |!\pi.U|_{\bullet} &= |U|_{\bullet} \\ |a[U]|_{\bullet} &= |U|_{\bullet} & |\sum_{i \in I} \pi_i.U_i|_{\bullet} &= \sum_{i \in I} |U_i|_{\bullet} \end{aligned}$$

The following auxiliary notation will be useful to formalize the properties of \mathcal{E}_s processes along reductions.

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

Figure 2: LTS for \mathcal{E}_s and \mathcal{E}_d . Rules (ACT2), (TAU2), and (TAU4)—the symmetric counterparts of (ACT1), (TAU1), and (TAU3)—have been omitted.

Definition 2.14. Let U be an \mathcal{E}_s update pattern. The number of prefixed holes occurring in U , denoted $|U|_{\text{ph}}$, is inductively defined as follows:

$$\begin{array}{l}
|\bullet|_{\text{ph}} = 0 \qquad |a[U]|_{\text{ph}} = |U|_{\text{ph}} \qquad |U_1 \parallel U_2|_{\text{ph}} = |U_1|_{\text{ph}} + |U_2|_{\text{ph}} \\
|\sum_{i \in I} \pi_i. U_i|_{\text{ph}} = \sum_{i \in I} |U_i|_{\bullet} \qquad |!\pi. U|_{\text{ph}} = |U|_{\bullet}
\end{array}$$

Example 2.15. Let P and U be an \mathcal{E}_s process and an \mathcal{E}_s update pattern defined as

$$P = a[b[Q_1] \parallel P_1] \parallel \tilde{a}\{U\}. P_2 \qquad U = \bar{b}. \tilde{d}\{\bullet \parallel U_1\}. \mathbf{0} \parallel b[a.\bullet \parallel \bullet]$$

Then we have:

- $|P|_{\text{ap}} = 2 + |Q_1|_{\text{ap}} + |P_1|_{\text{ap}} + 0$ and $|U|_{\text{ap}} = 1$
- $|U|_{\bullet} = 2$ and $|U|_{\text{ph}} = 1$.

We are now ready to define an LTS semantics for \mathcal{E}_s and another one for \mathcal{E}_d . Both LTSs are generated by the set of rules in Figure 2; they only differ on a condition associated to update actions. This is the content of the following definition.

Definition 2.16 (LTS for \mathcal{E}_d and \mathcal{E}_s). Given transition labels

$$\alpha ::= a \mid \bar{a} \mid a[P] \mid \tilde{a}\{U\} \mid \tau$$

the LTS for \mathcal{E}_d , denoted $\xrightarrow{\alpha}_d$, is defined by the rules in Figure 2 in which, in rules (TAU3) and (TAU4), we decree $\text{cond}(U, Q) = \text{true}$.

Similarly, the LTS for \mathcal{E}_s , denoted $\xrightarrow{\alpha}_s$, is defined by the rules in Figure 2 in which, in rules (TAU3) and (TAU4), we decree that $\text{cond}(U, Q)$ holds if we have:

- (1) $\text{St}(a[Q]) = \text{St}(a[U'\langle\langle Q \rangle\rangle] \parallel A)$ where $U = a[U'] \parallel A$, for some U', A , and
- (2) $|U|_{\text{ph}} > 0 \Rightarrow |Q|_{\text{ap}} = 0$.

Remark 2.17. The LTS for \mathcal{E}_s and \mathcal{E}_d are finitely branching. The proof proceeds by induction on the syntactic structure of terms; the base cases are $\sum_{i \in I} \pi_i. U_i$ and $!\pi. U$.

We give intuitions on both LTSs. In addition to the standard CCS actions (input, output, τ), we consider two complementary actions for process update: $\tilde{a}\{U\}$ and $a[P]$. The former represents the availability of an update pattern U for the adaptable process at

a ; the latter expresses the fact that the adaptable process at a , with current state P , is ready to update. We often write $\xrightarrow{\alpha}$ instead of $\xrightarrow{\alpha}_{\mathcal{E}_d}$ and $\xrightarrow{\alpha}_s$; the actual LTS used in each case will be clear from the context. Similarly, we define \rightarrow as $\xrightarrow{\tau}$.

In Figure 2, rule (COMP) represents the contribution of a process at a in an update operation; we use \star to denote a unique placeholder. Rule (LOC) formalizes transparency of localities. Rules (SUM), (REPL), (ACT1), and (TAU1) are standard. Rule (TAU3) formalizes process evolvability. To realize the evolution of an adaptable process at a , it requires: (i) a process Q —which represents its current state; (ii) an update action offering an update pattern U for updating the process at a —which is represented in P'_1 by \star (cf. rule (COMP)); (iii) that $\text{cond}(U, Q)$ holds (cf. Definition 2.16). As a result, \star in P'_1 is replaced with process $U\langle\langle Q \rangle\rangle$ (cf. Definition 2.2).

It is useful to elaborate on the definition of $\text{cond}(U, Q)$ —the only point in which the LTS of \mathcal{E}_d and that of \mathcal{E}_s differ. While $\text{cond}(U, Q)$ does not have influence on the update actions of \mathcal{E}_d processes, it does ensure that the syntactic restrictions associated to \mathcal{E}_s processes are preserved along transitions. As specified in Definition 2.16, the condition for \mathcal{E}_s processes is given in two parts. The first part ensures that the current structure of nested adaptable processes—the containment structure denotation from $a[Q]$ —is preserved once Q is substituted into U as a result of the transition. The second part of the condition ensures that no new adaptable processes will appear behind prefixes as a result of the update operation. Recall that by the syntactic restrictions enforced by Definition 2.4, adaptable processes cannot occur behind prefixes. In fact, and using the terminology introduced in that definition, the syntax of \mathcal{E}_s decrees that only processes in process category A (which do not contain adaptable processes) can occur behind prefixes. The second part of the condition ensures precisely this. As a simple example, this part of the condition rules out the synchronization of adaptable process $b[a[\mathbf{0}]]$ with update prefix $\tilde{b}\{b[a[b.\bullet]]\}.Q$, as it would lead to the non static process $b[a[b.a[\mathbf{0}]]] \parallel Q$.

By considering the syntactic restrictions associated to \mathcal{E}_s processes, the following lemma characterizes the conditions under which $\text{cond}(U_0, Q)$ holds for them.

Lemma 2.18. *Let Q and $U_0 = a[U] \parallel A$ be an \mathcal{E}_s process and an \mathcal{E}_s update pattern, respectively. Also, let A be as in Definition 2.4. We have*

$$\text{St}(a[Q]) = \text{St}(a[U\langle\langle Q \rangle\rangle] \parallel A) \wedge (|U|_{\text{ph}} > 0 \Rightarrow |Q|_{\text{ap}} = 0) \quad (2.1)$$

if and only if one of the following holds:

- (0) $|U|_{\bullet} = 0 \wedge \text{St}(Q) = \text{St}(U)$.
- (1) $|U|_{\bullet} = 1 \wedge |U|_{\text{ap}} = 0 \wedge (|U|_{\text{ph}} > 0 \Rightarrow |Q|_{\text{ap}} = 0)$
- (2) $|U|_{\bullet} > 1 \wedge |U|_{\text{ap}} = 0 \wedge |Q|_{\text{ap}} = 0$.

Proof. The “if” direction is straightforward by observing that by definition $|A|_{\text{ap}} = 0$. Therefore, (2.1) reduces to

$$\text{St}(Q) = \text{St}(U\langle\langle Q \rangle\rangle) \wedge (|U|_{\text{ph}} > 0 \Rightarrow |Q|_{\text{ap}} = 0)$$

and the analysis focuses on the structure of U . Hence if $|U|_{\bullet} = 0$ then immediately from (2.1) we have $\text{St}(Q) = \text{St}(U)$. If $|U|_{\bullet} = 1$ then as $\text{St}(Q) = \text{St}(U\langle\langle Q \rangle\rangle)$ we have that $|U|_{\text{ap}} = 0$ and from the second part of (2.1) we conclude $(|U|_{\text{ph}} > 0 \Rightarrow |Q|_{\text{ap}} = 0)$. Finally, if $|U|_{\bullet} > 1$ following from $\text{St}(Q) = \text{St}(U\langle\langle Q \rangle\rangle)$ we conclude $|U|_{\text{ap}} = 0$ and $|Q|_{\text{ap}} = 0$.

As for the “only if” direction, we consider each item separately:

- Item (0): Then Q occurs exactly once only at the left-hand side of the desired equality. Using the first part of the item (i.e., $|U|_{\bullet} = 0$) we infer that $U\langle\langle Q \rangle\rangle = U$. Since by definition $|A|_{\text{ap}} = 0$, we have that the second part of the item (i.e., $\text{St}(Q) = \text{St}(U)$) is enough to obtain $\text{St}(a[Q]) = \text{St}(a[U] \parallel A)$, as wanted.
- Item (1): Then Q occurs exactly once in both sides of the desired equality. The second part of the item (i.e., $|U|_{\text{ap}} = 0$) guarantees that $U\langle\langle Q \rangle\rangle$ does not involve any adaptable processes different from those in Q . The third condition ensures that no adaptable processes occur behind prefixes: if Q has adaptable processes then it should necessarily occur at the top level in $U\langle\langle Q \rangle\rangle$. Hence, the thesis follows.
- Item (2): Then Q occurs exactly once in the right-hand side of the equality, and arbitrarily many times in the left-hand side. The second part of the condition, on the number of adaptable processes in U , follows the same motivations as in the previous case. Given the possibility of arbitrarily many occurrences of Q in the left-hand side, the only option to ensure identical containment structure denotations in both sides is to forbid adaptable processes inside Q , hence the third part of the condition. \square

The following lemma is standard:

Lemma 2.19. *Let P be an \mathcal{E} process. Structural congruence is preserved by reduction: if $P \equiv Q$ and $P \longrightarrow P'$, then also $Q \longrightarrow Q'$ for some $P' \equiv Q'$.* \square

The following lemma states that \mathcal{E}_s processes are closed under reduction. Hence, the operational semantics of \mathcal{E}_s preserves the syntactic conditions of Definition 2.4.

Lemma 2.20 (Static topologies are preserved by reduction). *Let P be an \mathcal{E}_s process. If $P \longrightarrow P'$ then also P' is an \mathcal{E}_s process. Moreover, $\text{St}(P) = \text{St}(P')$.*

Proof. By induction on the derivation of $P \xrightarrow{\tau} P'$. See Appendix A.1, Page 52. \square

2.3. From Static to Dynamic Topologies. We have already remarked that from a syntactic point of view every \mathcal{E}_s process is also an \mathcal{E}_d process. As far as the operational semantics is concerned, an \mathcal{E}_s process could have less possible computations due to the additional constraint $\text{cond}(U, Q)$ of the rules (TAU3) and (TAU4). Nevertheless, in this section we show that it is always possible to translate a process with static topology into a process with dynamic topology which has the same semantics (the two LTSs are isomorphic). More precisely, we will define an encoding $\llbracket \cdot \rrbracket_S^d : \mathcal{E}_s \rightarrow \mathcal{E}_d$ such that the following holds:

$$P \longrightarrow_s P' \text{ if and only if } \llbracket P \rrbracket_S^d \longrightarrow_d \llbracket P' \rrbracket_S^d$$

We start by presenting some auxiliary definitions.

Definition 2.21. Let P be an \mathcal{E}_s process. We define the set

$$\text{Sub}_{\text{St}}(P) = \{\text{St}(a[P']) \mid a[P'] \text{ is a subterm of } P\}$$

Hence, $\text{Sub}_{\text{St}}(P)$ is a set of trees: it contains the containment structure denotations of the adaptable processes occurring in P . Notice that by construction $\text{Sub}_{\text{St}}(P)$ is a set of single-child containment structure denotations.

Example 2.22. Let P be as in Example 2.11. Then, we have

$$\text{Sub}_{\text{St}}(P) = \{\text{St}(a[P_3 \parallel c[S]]), \text{St}(b[P_2]), \text{St}(c[S])\} \cup \text{Sub}_{\text{St}}(S)$$

Convention 2.23. Below P and a stand for an \mathcal{E}_s process and a name, respectively.

- Let S be a set of containment structure denotations. We write $S \downarrow a$ to represent the subset of single-child containment structure denotations of S in which the label of the only child of the root is a .
- We assume an injective function φ that associates containment structure denotations to names in \mathcal{N} . We use κ, κ', \dots to range over the codomain of φ . Moreover, for every P such that $|P|_{\text{ap}} = 0$, we fix $\varphi(\text{St}(a[P])) = \kappa_a$. The definition of φ extends to sets of containment structure denotations as expected; this way, e.g., $\varphi(S \downarrow a)$ stands for the set of names associated to those single-child containment structure denotations in S with label a . With a slight abuse of notation, we sometimes write $\varphi(a[P])$ instead of $\varphi(\text{St}(a[P]))$.

We are now ready to present the definition of $[\![\cdot]\!]_S^d$.

Definition 2.24. Adopting the notations in Convention 2.23, let P and U be an \mathcal{E}_s process and an \mathcal{E}_s update pattern, respectively. Also, let S be a set of containment structure denotations such that $\text{Sub}_{\text{St}}(P) \subseteq S$. Moreover, assume $\text{err} \notin \varphi(S)$. The encoding of P into an \mathcal{E}_d process over \mathcal{N} , denoted $[\![P]\!]_S^d$, is inductively defined in Figure 3, where

- $C1$ stands for $|U|_{\bullet} = 0$;
- $C2$ stands for $(|U|_{\bullet} = 1 \wedge |U|_{\text{ph}} > 0 \wedge |U|_{\text{ap}} = 0) \vee (|U|_{\bullet} > 1 \wedge |U|_{\text{ap}} = 0)$;
- $C3$ stands for $|U|_{\bullet} = 1 \wedge |U|_{\text{ph}} = 0 \wedge |U|_{\text{ap}} = 0$;
- $C4$ stands for $|U|_{\bullet} \neq 0 \wedge |U|_{\text{ap}} \neq 0$.

We now comment on the definition in Figure 3. Unsurprisingly, the encoding only concerns adaptable processes and update prefixes; input and output prefixes are not modified (cf. line (6)), and guarded sum, parallel composition, and holes are treated homomorphically (cf. lines (7), (8), and (9), respectively). Intuitively, the encoding captures correct update actions by renaming every adaptable process and update prefix according to their containment denotation structure. This way, an adaptable process $a[P]$ is translated into an adaptable process on name κ , which depends on its containment structure denotation (cf. line (1)). The intention of this renaming is to allow synchronization only with update prefixes on name κ , that is, with update prefixes having the same containment structure denotation; this way, condition $\text{St}(U) = \text{St}(Q)$ in the LTS of \mathcal{E}_s is enforced via name equality. As for the encoding of a process P at a , it is important to observe that the definition of \mathcal{E}_s ensures that holes syntactically occurring in P do not occur at top level—they can only appear inside an update prefix. As such, they are handled by lines (2)–(7) in recursive applications of the encoding.

Clearly, update prefixes must be modified accordingly; there are four different possibilities, represented by conditions $C1$ – $C4$ of Definition 2.24:

- $C1$ captures the cases in which the update pattern U does not contain holes, i.e., U is a process. Update prefixes with update patterns of this kind are encoded homomorphically, renaming the prefix accordingly (cf. line (2)). Together with the above explained renaming of adaptable processes with respect to the structure of their contents, this condition corresponds to Lemma 2.18(0).
- $C2$ captures the cases in which the update prefix is only meant to interact with adaptable processes whose content have no adaptable processes. As explained before, and by the definition of φ , these are adaptable processes of the form $\kappa_a[P]$ (with $|P|_{\text{ap}} = 0$); this explains the encoding described in line (3). According to Lemma 2.18, this is the case

$$\begin{aligned}
(1) \llbracket a[P] \rrbracket_S^d &= \kappa[\llbracket P \rrbracket_S^d] \quad \text{with } \kappa = \varphi(a[P]) \\
(2) \llbracket \xi \tilde{a}\{a[U] \parallel A\}.U_1 \rrbracket_S^d &= \xi \tilde{\kappa}\{\kappa[\llbracket U \rrbracket_S^d] \parallel \llbracket A \rrbracket_S^d\}. \llbracket U_1 \rrbracket_S^d \\
&\quad \text{with } \kappa = \varphi(a[U]) \quad \text{if } C1 \\
(3) \llbracket \xi \tilde{\kappa}_a\{a[U] \parallel A\}.U_1 \rrbracket_S^d &= \xi \tilde{\kappa}_a\{\kappa_a[\llbracket U \rrbracket_S^d] \parallel \llbracket A \rrbracket_S^d\}. \llbracket U_1 \rrbracket_S^d \quad \text{if } C2 \\
(3a) \llbracket \tilde{a}\{a[U] \parallel A\}.U_1 \rrbracket_S^d &= \sum_{\kappa_i \in \varphi(S \downarrow a)} \tilde{\kappa}_i\{\kappa_i[\llbracket U \rrbracket_S^d] \parallel \llbracket A \rrbracket_S^d\}. \llbracket U_1 \rrbracket_S^d \quad \text{if } C3 \\
(3b) \llbracket !\tilde{a}\{a[U] \parallel A\}.U_1 \rrbracket_S^d &= \prod_{\kappa_i \in \varphi(S \downarrow a)} !\tilde{\kappa}_i\{\kappa_i[\llbracket U \rrbracket_S^d] \parallel \llbracket A \rrbracket_S^d\}. \llbracket U_1 \rrbracket_S^d \quad \text{if } C3 \\
(5) \llbracket \xi \tilde{a}\{a[U] \parallel A\}.U_1 \rrbracket_S^d &= \xi \tilde{\text{err}}\{\mathbf{0}\}. \llbracket U_1 \rrbracket_S^d \quad \text{if } C4 \\
(6) \llbracket \xi \pi.U \rrbracket_S^d &= \xi \pi. \llbracket U \rrbracket_S^d \quad \text{if } \pi = a \text{ or } \pi = \bar{a} \\
(7) \llbracket \sum_{i \in I} \pi_i.U_i \rrbracket_S^d &= \sum_{i \in I} \llbracket \pi_i.U_i \rrbracket_S^d \\
(8) \llbracket U_1 \parallel U_2 \rrbracket_S^d &= \llbracket U_1 \rrbracket_S^d \parallel \llbracket U_2 \rrbracket_S^d \\
(9) \llbracket \bullet \rrbracket_S^d &= \bullet
\end{aligned}$$

Above, $\xi\pi.P$ denotes a possibly replicated prefixed process: $\xi\pi.P$ is either $!\pi.P$ or $\pi.P$, with ξ being the same on both sides of the definition.

Figure 3: The encoding $\llbracket \cdot \rrbracket_S^d : \mathcal{E}_s \rightarrow \mathcal{E}_d$ given in Definition 2.24.

when (i) the update pattern U of the update prefix has exactly one hole that occurs behind a prefix (cf. Lemma 2.18(1) when $|U|_{\text{ph}} > 0$) or (ii) U has more than one hole (cf. Lemma 2.18(2)).

- $C3$ captures the cases in which the update pattern U has exactly one hole which does not occur behind a prefix. These are update prefixes that may synchronize with any adaptable process at name a . In order to account for all the possibilities, each non replicated update prefix at a is encoded as a sum of prefixed processes, each summand corresponding to an update prefix on a name $\kappa_i \in \varphi(S \downarrow a)$. The only difference between the summands is the name of the update prefix; the update pattern within the update prefix and its continuation is the same for all of them (cf. line (3a)). When the update prefix is replicated, rather than the sum of all possible adaptable processes, we consider their product (cf. line (3b)). This condition corresponds to Lemma 2.18(1) when $|U|_{\text{ph}} = 0$.
- $C4$ captures those update patterns that do not adhere to any of the conditions of Lemma 2.18. Hence, interaction with these prefixes may lead to ill-formed \mathcal{E}_s processes. To prevent such undesirable interactions, these update prefixes are renamed into err —a distinguished name signaling error (cf. line (5)).

Before stating the correctness of the encoding, we illustrate it further through a series of examples.

Example 2.25. Below, notice that by virtue of Definition 2.4, $|A_i|_{\text{ap}} = 0$ for every A_i .

(1) Given the \mathcal{E}_s process

$$P_1 = b[c[A_1 \parallel A_2]] \parallel b[d[e.A_3]] \parallel \tilde{b}\{c[A_4]\}.Q_2$$

we have the \mathcal{E}_d process

$$\llbracket P_1 \rrbracket_S^d = \kappa_1 [\llbracket c[A_1 \parallel A_2] \rrbracket_S^d] \parallel \kappa_2 [\llbracket d[e.A_3] \rrbracket_S^d] \parallel \widetilde{\kappa}_1 \{ \llbracket c[A_4] \rrbracket_S^d \}. \llbracket Q_2 \rrbracket_S^d$$

with $\kappa_1 = \varphi(b[c[A_1 \parallel A_2]])$, $\kappa_2 = \varphi(b[d[e.A_3]])$. Notice how the renaming to κ_2 rules out the possibility of an update action for the second adaptable processes on b .

(2) Given the \mathcal{E}_s process

$$P_2 = c[A_1] \parallel c[A_2] \parallel d[A_3] \parallel d[e[A_4]] \parallel \\ \widetilde{c}\{c[\bullet \parallel \bullet] \parallel A_5\}. Q_1 \parallel \widetilde{d}\{d[A_6 \parallel a.\bullet]\}. Q_2$$

we have the \mathcal{E}_d process

$$\llbracket P_2 \rrbracket_S^d = \kappa_c [\llbracket A_1 \rrbracket_S^d] \parallel \kappa_c [\llbracket A_2 \rrbracket_S^d] \parallel \kappa_d [\llbracket A_3 \rrbracket_S^d] \parallel \kappa_1 [\llbracket e[A_4] \rrbracket_S^d] \parallel \\ \widetilde{\kappa}_c \{ \kappa_c [\bullet \parallel \bullet] \parallel \llbracket A_5 \rrbracket_S^d \}. \llbracket Q_1 \rrbracket_S^d \parallel \widetilde{\kappa}_d \{ \kappa_d [\llbracket A_6 \rrbracket_S^d \parallel a.\bullet \rrbracket_S^d \}. \llbracket Q_2 \rrbracket_S^d$$

with $\kappa_1 = \varphi(d[e[A_4]])$. Notice how the renaming to κ_1 rules out the possibility of an update action for the second adaptable processes on d .

(3) Given the \mathcal{E}_s process P_3 defined as:

$$e[f[A_1]] \parallel e[g[h[A_2] \parallel A_3]] \parallel (\widetilde{e}\{e[\bullet \parallel A_4]\}. Q_1 + \widetilde{e}\{e[f[\bullet \parallel \bullet]] \parallel A_5\}. Q_2)$$

we have (assuming S to be minimal) the \mathcal{E}_d process

$$\llbracket P_3 \rrbracket_S^d = \kappa_1 [\llbracket f[A_1] \rrbracket_S^d] \parallel \kappa_2 [\llbracket g[h[A_2] \parallel A_3] \rrbracket_S^d] \parallel \\ (\widetilde{\kappa}_1 \{ \kappa_1 [\bullet \parallel \bullet] \parallel \llbracket A_4 \rrbracket_S^d \}. \llbracket Q_1 \rrbracket_S^d + \widetilde{\kappa}_2 \{ \kappa_2 [\bullet \parallel \bullet] \parallel \llbracket A_4 \rrbracket_S^d \}. \llbracket Q_1 \rrbracket_S^d + \widetilde{\text{err}}\{\mathbf{0}\}. \llbracket Q_2 \rrbracket_S^d)$$

with $\kappa_1 = \varphi(e[f[A_1]])$ and $\kappa_2 = \varphi(e[g[h[A_2] \parallel A_3]])$.

Observe how the first summand in P_3 has been duplicated in $\llbracket P_3 \rrbracket_S^d$, so as to account for the two possible update actions on e .

We are in place to state the promised correspondence between \mathcal{E}_s and \mathcal{E}_d processes:

Theorem 2.26. *Let P be an \mathcal{E}_s process. Also, let S be a set of containment structure denotations, such that $\text{Subst}(P) \subseteq S$. Then we have:*

$$P \longrightarrow_s P' \text{ if and only if } \llbracket P \rrbracket_S^d \longrightarrow_d \llbracket P' \rrbracket_S^d$$

Proof (Sketch). The proof is in two parts, one for the “if” direction and another other for the “only if” direction. In both cases, we proceed by induction on the height of the derivation tree for $P \longrightarrow_s P'$ (resp. $\llbracket P \rrbracket_S^d \longrightarrow_d \llbracket P' \rrbracket_S^d$), with a case analysis on the last applied rule. For the former, we rely on the characterization of reduction for \mathcal{E}_s processes given by Lemma 2.18 so as to show that a reduction in the static side is preserved in the dynamic side. As for the latter, the proof is similar, and exploits the fact that the encoding transforms update prefixes that may lead to incorrect update actions into “error” update prefixes which are unable to participate in reductions. This ensures that for every dynamic reduction there is also a static reduction. See Appendix A.2 in Page 53 for details. \square

3. CORRECTNESS PROPERTIES: BOUNDED AND EVENTUAL ADAPTATION

Here we define the correctness problems that we consider throughout the paper. We would like adaptation properties defined in the most general way possible; this would allow us to analyze models of evolvable systems in different settings. For this purpose, our correctness properties are stated in terms of observability predicates, or *barbs*. The 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 3.1 (Barbs). Let P be an \mathcal{E} process, and let α be an action in $\{a, \bar{a} \mid a \in \mathcal{N}\}$. We write $P \downarrow_\alpha$ if there exists a P' such that $P \xrightarrow{\alpha} P'$. Moreover:

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

Furthermore, we use \nexists_α^k and \nexists_α^ω to denote the negation of \Downarrow_α^k and \Downarrow_α^ω .

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 P with an arbitrary set of processes M representing its possible subsequent modifications. That is, processes in M may contain update actions on the names of the adaptable processes in P , and therefore may potentially lead to its modification (evolution).

Definition 3.2 (Cluster). Let P, P_1, \dots, P_n be \mathcal{E} processes and $M = \{P_1, \dots, P_n\}$. The set of clusters \mathcal{CS}_P^M is defined as:

$$\mathcal{CS}_P^M = \left\{ P \parallel \prod_{i=1}^{m_1} P_1 \parallel \dots \parallel \prod_{i=1}^{m_n} P_n \mid m_1, \dots, m_n \in \mathbb{N} \cup \{0\} \right\}$$

The *adaptation* problems below formalize correctness of clusters with respect to their ability for recovering from errors by means of update actions. More precisely, given a set of clusters \mathcal{CS}_P^M and a barb e (signaling an error), we would like to know if all computations of processes in \mathcal{CS}_P^M

- (1) have *at most* k consecutive states exhibiting e , or
- (2) have a *finite* number of consecutive states exhibiting e .

We thus have the following definition:

Definition 3.3 (Adaptation Problems). Suppose an initial process P , a set of processes M , and a barb e .

- Given $k > 0$, the *bounded adaptation* problem (BA) consists in checking whether for all processes $R \in \mathcal{CS}_P^M$, $R \nexists_e^k$ holds.
- Similarly, the *eventual adaptation* problem (EA) consists in checking whether for all processes $R \in \mathcal{CS}_P^M$, $R \nexists_e^\omega$ holds.

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

Definition 3.4. Let P, P_1, \dots, P_n be \mathcal{E}_s processes such that $M = \{P_1, \dots, P_n\}$. The static cluster set \mathcal{CS}_P^M is transformed 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 $S = \text{Sub}_{\text{St}}(P) \cup \bigcup_{1 \leq i \leq n} \text{Sub}_{\text{St}}(P_i)$.

Theorem 3.5. *Let P, P_1, \dots, P_n be \mathcal{E}_s processes such that $M = \{P_1, \dots, P_n\}$. Then we have $\llbracket \mathcal{CS}_P^M \rrbracket_S^d = \{\llbracket C \rrbracket_S^d \mid C \in \mathcal{CS}_P^M\}$, where $S = \text{SubSt}(P) \cup \bigcup_{1 \leq i \leq n} \text{SubSt}(P_i)$.*

Proof. Immediate by observing that by Definition 2.24, $\llbracket P \rrbracket_S^d$ is an homomorphism with respect to parallel composition, i.e., $\llbracket P \parallel Q \rrbracket_S^d = \llbracket P \rrbracket_S^d \parallel \llbracket Q \rrbracket_S^d$. \square

Notice that, for every cluster C in $\llbracket \mathcal{CS}_P^M \rrbracket_S^d$ by construction we have $\text{SubSt}(C) \subseteq S$. Hence, the operational correspondence given by Theorem 2.26 is individually applicable to each cluster.

4. ADAPTABLE PROCESSES, BY EXAMPLES

Next we present some concrete scenarios of adaptable processes and discuss their representation as \mathcal{E} processes. We also comment on how the adaptation properties proposed in the paper (and their associated decidability results) relate to such scenarios.

4.1. Mode Transfer Operators. In [8], dynamic behavior at the process level is defined by means of two so-called *mode transfer* operators. Given processes P and Q , the *disrupt* operator starts executing P but at any moment it may abandon P and execute Q instead. The *interrupt* operator is similar, but it returns to execute what is left of P once Q emits a termination signal. We can represent similar mechanisms in \mathcal{E} as follows:

$$\text{disrupt}_a(P, Q) \stackrel{\text{def}}{=} a[P] \parallel \tilde{a}\{Q\} \quad \text{interrupt}_a(P, Q) \stackrel{\text{def}}{=} a[P] \parallel \tilde{a}\{Q \parallel t_Q \bullet\}$$

Assuming that P can evolve on its own to P' , the semantics of \mathcal{E} decrees that $\text{disrupt}_a(P, Q)$ may evolve either to $a[P'] \parallel \tilde{a}\{Q\}$ (as locality a is transparent) or to Q (which represents disruption at a). Similarly, by assuming that P was able to evolve into P'' just before being interrupted, process $\text{interrupt}_a(P, Q)$ evolves to $Q \parallel t_Q \bullet P''$. Above, we assume that a is not used in P and Q , and that termination of Q is signaled at the designated name t_Q .

These simple definitions show how defining P as an adaptable process at a is enough to formalize its potential disruption/interruption. It is worth observing that the encoding of $\text{interrupt}_a(P, Q)$ can only be an \mathcal{E}_d^1 process: in the update action at a , there is a hole occurring behind a prefix (hence, it is not a \mathcal{E}^2 process) and the topology of adaptable process is dynamic (since a does not occur in Q , the adaptable process cannot be rebuilt after interruption). In contrast, the encoding of $\text{disrupt}_a(P, Q)$ is both an \mathcal{E}_d^1 and an \mathcal{E}_d^2 process, as in the update pattern there are no holes in the scope of prefixes (in fact, the update pattern does not have any holes).

4.2. Dynamic Update in Workflow Applications. Designing business/enterprise applications in terms of *workflows* is a common practice nowadays. A workflow is a conceptual unit that describes how a number of *activities* coordinate to achieve a particular task. A workflow can be seen as a container of activities; such activities are usually defined in terms of simpler ones, and may be software-based (such as, e.g., “retrieve credit information from the database”) or may depend on human intervention (such as, e.g., “obtain the signed authorization from the credit supervisor”). As such, workflows are typically long-running and have a transactional character. A workflow-based application usually consists of a *workflow runtime engine* that contains a number of workflows running concurrently on top of it; a *workflow base library* on which activities may rely on; and of a number of *runtime services*,

which are application dependent and implement things such as transaction handling and communication with other applications. A simple abstraction of a workflow application is the following \mathcal{E} process:

$$App \stackrel{\text{def}}{=} \text{wfa} \left[\text{we}[\text{WE} \parallel W_1 \parallel \cdots \parallel W_k \parallel \text{wbl}[\text{BL}]] \parallel S_1 \parallel \cdots \parallel S_j \right]$$

where the application is modeled as an adaptable process wfa which contains a workflow engine we and a number of runtime services S_1, \dots, S_j . In turn, the workflow engine contains a number of workflows W_1, \dots, W_k , a process WE (which represents the engine's behavior and is left unspecified), and an adaptable process wbl representing the base library (also left unspecified). As described before, each workflow is composed of a number of activities. We model each W_i as an adaptable process w_i containing a process WL_i —representing the workflow's logic—, and n activities. Each of them is formalized as an adaptable process a_j and an *execution environment* env_j :

$$W_i = w_i \left[\text{WL}_i \parallel \prod_{j=1}^n (\text{env}_j[\text{P}_j] \parallel a_j[!u_j. \widetilde{\text{env}}_j\{\text{env}_j[\bullet \parallel A_j]\}]) \right]$$

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

$$U_1 = !\text{replace}_j. \widetilde{a}_j\{a_j[!u_j. \widetilde{\text{env}}_j\{\text{env}_j[\bullet \parallel A_j^2]\}]\}$$

Hence, given an output at replace_j , process $a_j[!u_j. \widetilde{\text{env}}_j\{\text{env}_j[\bullet \parallel A_j]\}] \parallel U_1$ evolves to $a_j[!u_j. \widetilde{\text{env}}_j\{\text{env}_j[\bullet \parallel A_j^2]\}]$ thus discarding A_j in a *future* evolution of env_j . This kind of *dynamic update* is available in commercial workflow engines, such as the Windows Workflow Foundation (WWF) [44]. Above, for simplicity, we have abstracted from lock mechanisms that keep consistency between concurrent updates on env_j and a_j .

In the above processes, it is worth observing that if processes A_j and A_j^2 contain no adaptable processes, then W_i is an \mathcal{E}_s^3 process. This is because the update action at env_j recreates the adaptable process, and preserves the previous state with a hole that is in parallel to A_j . Otherwise, W_i would be an \mathcal{E}_d^3 process, as the topology of adaptable processes would change as a result of an update action on env_j . For the sake of the example, suppose an emergency activity that executes inside the workflow: process P_j would emit a signal representing an urgent request, and an update action at env_j would represent a response to the emergency, implemented as process A_j . The two adaptation problems are useful to represent the future state of the workflow in which the emergency has been controlled: EA refers to an *undetermined* future state in which the request signal disappears (meaning that the emergency will be eventually controlled); whereas BA refers to a *fixed* future state in which the request signal disappears (meaning that the emergency will be controlled within a certain bound). The topology of A_j is relevant in the light of our decidability results for these two properties: if W_i is given as an \mathcal{E}_s^3 process, then both EA and BA are decidable; otherwise, if W_i is given as an \mathcal{E}_d^3 process, then only BA would be decidable.

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

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

This way, given an output signal at suspend_i , process $w_i[\widetilde{w}_i] \parallel U_3$ evolves to the persistent process $!\text{resume}_i. w_i[\widetilde{w}_i]$ which can be reactivated at a later time. Observe that, in case one considers policies such as U_2 then we would end up with an \mathcal{E}_d^1 process, as the hole and an adaptable process occur guarded behind a prefix.

4.3. Scaling in Cloud Computing Applications. In the emerging cloud computing paradigm, applications are deployed in the infrastructure offered by external providers. Developers act as clients: they only pay for the resources they consume (usually measured as the processor time in remote *instances*) and for associated services (e.g., performance metrics or automated load balancing). Central to the paradigm is the goal of optimizing resources for both clients and provider. An essential feature towards that goal is *scaling*: the capability that cloud applications have for expanding themselves in times of high demand, and for reducing themselves when the demand is low. Scaling can be appreciated in, e.g., the number of running instances supporting the application, and may have important financial effects. Consequently, cloud providers such as Amazon’s Elastic Cloud Computing (EC2) [5] offer libraries and APIs and services for *autoscaling*; also common are external tools which build on available APIs to implement sophisticated scaling policies.

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

$$G_i = g_i [I \parallel \dots \parallel I \parallel S_{dw} \parallel S_{up} \parallel \text{CTRL}_i]$$

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

$$S_{dw} = s_d [!\text{alert}^d. \prod_{j=1}^j \widetilde{\text{mid}}\{0\}] \quad S_{up} = s_u [!\text{alert}^u. \prod_{k=1}^k \widetilde{\text{mid}}\{\text{mid}[\bullet] \parallel \text{mid}[\bullet]\}]$$

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

Observe that both S_{dw} and S_{up} are \mathcal{E}_d^2 processes: since we represent instances as adaptable processes with state, every modification enforced by the scaling policies will result in a different topology of adaptable processes. A correctness guarantee in this setting is that the cloud infrastructure satisfies the scaling requirements of client applications within a fixed bound. More precisely, we would like to ensure that every scaling alert managed by CTRL_i (requesting more instances, for instance) will disappear within a certain bound, meaning that the scaling request is promptly addressed by the cloud provider. This kind of reliability guarantees can be represented in terms of BA, an adaptation problem which is decidable for \mathcal{E}_d^2 processes. Of course, the decidability of correctness guarantees depends much on their actual representations. Above, we have opted for simple, illustrative representations; clearly, different process abstractions may exploit other decidability results.

Autoscaling in EC2 also allows to *suspend* and *resume* the scaling policies themselves. To formalize this capability, we proceed similarly as we did for process U_2 above. This way, for the scale down policy, one can assume that CTRL_i includes a process $U_{dw} = !\text{susp}_{\text{down}} \cdot \tilde{s}_d\{!\text{resume}_{\text{dw}} \cdot s_d[\bullet]\}$ which, provided an output signal on $\text{susp}_{\text{down}}$, captures the current policy, and evolves into a process that allows to resume it later on. Using the same principle, other modifications to the policies are possible. For instance, a natural request is to modify the scaling policies by changing the number of instances involved (i.e., j in S_{dw} and k in S_{up}). As before, if our specification includes the ability of suspending/resuming scaling policies as implemented by U_{dw} , then we would obtain an \mathcal{E}_d^1 process.

5. PRELIMINARIES

We now introduce some background notions on Minsky machines, well-structured transition systems (WSTS), and Petri nets.

5.1. Minsky machines. Our undecidability results will be obtained by encodings of *Minsky machines* [47]. A Minsky machine (MM) 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 three kinds: $\text{INC}(r_j)$ adds 1 to register r_j and proceeds to the next instruction; $\text{DECJ}(r_j, s)$ jumps to instruction s if r_j is zero, otherwise it decreases register r_j by 1 and proceeds to the next instruction; a HALT instruction stops the machine. A MM 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. We assume that instructions are proper, in the sense that there is no program counter that refers to a non-existing instruction. The MM *terminates* whenever the program counter is set to a HALT instruction. A *configuration* of a MM 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 MM, denoted \longrightarrow_M , is defined in Figure 4.

Since MMs are Turing complete, termination is undecidable.

Theorem 5.1 (Minsky [47]). *Minsky machines are Turing complete. Hence, for a MM it is undecidable whether it terminates.* \square

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

Figure 4: Semantics of MMs

We shall exploit encodings into MMs to prove undecidability of EA and BA. In our encodings, we sometimes make the unrestrictive assumption that at the beginning and at the end of the computation the registers (must) contain the value zero.

5.2. Well-Structured Transition Systems. The decidability of BA for \mathcal{E}_d^2 processes will be shown by appealing to the theory of well-structured transition systems [30, 2]. The following results and definitions are from [30], unless differently specified.

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

Definition 5.2 (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 5.3 (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 $Succ(s)$ as the set $\{s' \in S \mid s \rightarrow s'\}$ of immediate *successors* of s . TS is *finitely branching* if, for each $s \in S$, $Succ(s)$ is finite. We also define $Pred(s)$ as the set $\{s' \in S \mid s' \rightarrow s\}$ of *immediate predecessors* of s , while $Pred^*(s)$ and $Pred^+(s)$ denote the sets $\{s \in S \mid s' \rightarrow^* s\}$ and $\{s \in S \mid s' \rightarrow^+ s\}$, respectively, of *predecessors* of s .

Convention 5.4. In the rest of the paper, and with a slight abuse of notation, we will assume the expected point-wise extensions of definitions to sets. For instance, function $Succ$ just defined on states is extended to sets of states as: $Succ(S) = \bigcup_{s \in S} Succ(s)$.

The key tool to the decidability of several properties of computations is the notion of *well-structured transition system* [30, 2]. 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 5.5 (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:

- (1) \leq is a well-quasi-order;
- (2) \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.

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 5.6 (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 5.7 (Effective pred-basis). A well-structured transition system has *effective pred-basis* if there exists an algorithm such that, for any state $s \in S$, it returns the set $pb(s)$ which is a finite basis of $\uparrow Pred(\uparrow s)$.

The following proposition is a special case of Proposition 3.5 in [30].

Proposition 5.8. *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 $Pred^*(I)$ for any upward-closed set I given via a finite basis.* \square

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

Proposition 5.9. *Let S be a finite set. Then the equality is a wqo over S .* \square

We shall also appeal to the following result. In [39], 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 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 $label(n)$ the label of the node n .

Definition 5.10. 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 $label(n) \leq label(f(n))$.

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 5.11 (Kruskal [39]). *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 .* \square

5.3. Petri Nets. We will use Petri nets to prove the decidability of BA for \mathcal{E}_s^3 . More precisely, we will reduce BA for \mathcal{E}_s^3 to a problem on Petri nets, that we call *infinite visit*, which can be easily reduced to place boundedness.

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 transition 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 $m \rightarrow n$. We call *computation* a sequence $m_0 \rightarrow m_1 \rightarrow \dots \rightarrow m_n$. A marking m is *reachable* if there exists a computation with final marking m . A place $p \in S$ is *bounded* if there exists a natural number k such that $m(p) \leq k$ for every reachable marking m . The place boundedness problem is decidable for Petri nets [35].

Definition 5.12 (Infinite visit). Given a Petri net $N = (S, T, m_0)$, a set of places to visit $V \subseteq S$, and a mandatory place $p \in S$, we say that N *infinitely visits* V with mandatory place p , if there exists an infinite sequence $m_0 \rightarrow m_1 \rightarrow m_2 \rightarrow \dots$ and an index i such that for every $j \geq i$ there exists a place $p_j \in V$ such that $m_j(p_j) \geq 1$, and moreover $m_j(p) \geq 1$.

Theorem 5.13. *Given a Petri net $N = (S, T, m_0)$, a set of places $V \subseteq S$, and a mandatory place $p \in S$, it is decidable whether N infinitely visits V with mandatory place p .*

Proof. By reduction to the place boundedness problem. Given a Petri net $N = (S, T, m_0)$ and a set of places $V \subseteq S$, we construct a Petri net $N' = (S \cup \{ph1, ph2, check\}, T', m_0 \cup \{ph1\})$ such that N infinitely visits V with mandatory place p if and only if *check* is not bounded in N' .

The Petri net N' reproduces the computations in N by (possibly) dividing them into two phases: the first phase is witnessed by the presence of one token in the additional place *ph1*, while the second phase by one token in the additional place *ph2*. During the second phase, a transition can be mimicked only if there is at least one token in one of the places in V and one token in the place p . Moreover, during the second phase, each transition puts one token in the additional place *check*.

Formally, we define the set T' of the transitions of N' as follows:

- for each transition $m' \Rightarrow m''$ in T , T' contains the transition $m' \uplus \{ph1\} \Rightarrow m'' \uplus \{ph1\}$;
- T' contains the transition $\{ph1\} \Rightarrow \{ph2\}$;
- for each transition $m' \Rightarrow m''$ in T and for each place $q \in V$, T' contains the transition $m' \uplus \{p, q, ph2\} \Rightarrow m'' \uplus \{p, q, ph2, check\}$.

The first group of transitions governs the first phase of the simulation; the second transition implies the passage from the first to the second phase; while the third group of transitions is for the second phase.

First, assume that N infinitely visits V with mandatory place p . This means that in N there exists an infinite sequence $m_0 \rightarrow m_1 \rightarrow m_2 \rightarrow \dots$ and an index i such that for every $j \geq i$ there exists a place $p_j \in V$ such that $m_j(p_j) \geq 1$ and $m_j(p) \geq 1$. This implies that in N' there is a corresponding computation that mimics the transition $m_0 \rightarrow m_1 \rightarrow m_2 \rightarrow m_{i-1}$ during the first phase, and the transitions $m_i \rightarrow m_{i+1} \rightarrow \dots$ during the second

$$\begin{aligned}
\text{REGISTER } r_j \quad & \llbracket r_j = n \rrbracket_1 = r_j[\langle n \rangle_j] \\
\text{where } \langle n \rangle_j &= \begin{cases} \bar{z}_j & \text{if } n = 0 \\ \bar{u}_j. \langle n - 1 \rangle_j & \text{if } n > 0. \end{cases} \\
\text{INSTRUCTIONS } (i : I_i) & \\
\llbracket (i : \text{INC}(r_j)) \rrbracket_1 &= !p_i. \tilde{r}_j\{r_j[\bar{u}_j. \bullet]\}. \bar{p}_{i+1} \\
\llbracket (i : \text{DECJ}(r_j, s)) \rrbracket_1 &= !p_i. (u_j. \bar{p}_{i+1} + z_j. \tilde{r}_j\{r_j[\bar{z}_j]\}. \bar{p}_s) \\
\llbracket (i : \text{HALT}) \rrbracket_1 &= !p_i. (e + \bar{p}_i)
\end{aligned}$$

Table 2: Encoding of MMs into \mathcal{E}_s^1 .

one. The second phase is infinite, hence *check* is not bounded because each transition in the second phase puts one token in such a place.

Assume now that *check* is unbounded in N' . As tokens are introduced in *check* only during the second phase, this means that there exists no bound to the length of the computations in N' that include the second phase. This implies the existence of at least one infinite computation in N' having both the first and the second phase. Consider now the computation in N composed of the transitions simulated in such an infinite computation of N' . This computation in N has a suffix (the part corresponding to the second phase) in which all the traversed markings have at least one token in one of the places in V as well as one token in p . \square

6. UNDECIDABILITY RESULTS FOR \mathcal{E}^1

We prove that BA and EA are undecidable in both \mathcal{E}_d^1 and \mathcal{E}_s^1 . The result relies on an encoding of MMs into \mathcal{E}_s^1 which satisfies the following: a MM terminates if and only if its encoding into \mathcal{E}_s^1 evolves into a state that starts an infinite computation that traverses states exhibiting a distinguished barb e .

The encoding, denoted $\llbracket \cdot \rrbracket_1$, is given in Table 2. A register j with value m is represented by an adaptable process at r_j that contains the encoding of number m , denoted $\langle m \rangle_j$. In turn, $\langle m \rangle_j$ consists of a sequence of m output prefixes on name u_j , ending with an output action on z_j , which represents zero. Instructions are encoded as replicated processes guarded by p_i , which represents the MM when the program counter $p = i$. Once p_i is consumed, each instruction is ready to interact with the registers. To encode the increment of register r_j , we enlarge the sequence of output prefixes it contains. The adaptable process at r_j is updated with the encoding of the incremented value (which results from putting the value of the register behind some prefixes) and then the next instruction is invoked. The encoding of a decrement of register j consists of an exclusive choice: the left side implements the decrement of the value of a register, while the right one implements the jump to some given instruction. This choice is indeed exclusive: the encoding of numbers as a chain of output prefixes ensures that both an input prefix on u_j and one on z_j are never available at the same time. When the MM reaches the HALT instruction the encoding can either exhibit a barb on e , or set the program counter again to the HALT instruction so as to pass through a state that exhibits e at least $k > 0$ times. The encoding of a MM into \mathcal{E}_s^1 is defined as follows:

Definition 6.1. Let N be a MM, with registers $r_0 = 0$, $r_1 = 0$ and instructions $(1 : I_1) \dots (n : I_n)$. Given the encodings in Table 2, the encoding of N in \mathcal{E} (written $\llbracket N \rrbracket_1$) is defined as $\llbracket r_0 = 0 \rrbracket_1 \parallel \llbracket r_1 = 0 \rrbracket_1 \parallel \prod_{i=1}^n \llbracket (i : I_i) \rrbracket_1 \parallel \overline{p_1}$.

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

Lemma 6.2. *Let N be a MM and $k \geq 1$. N terminates iff $\llbracket N \rrbracket_1 \Downarrow_e^k$.*

Proof. See Appendix B.1, Page 56. □

Theorem 6.3. *BA and EA are undecidable in \mathcal{E}_s^1 .*

Proof (Sketch). The proof proceeds by considering a MM N and its encoding $\llbracket N \rrbracket_1$. Taking the cluster $\mathcal{CS}_{\llbracket N \rrbracket_1}^0 = \{\llbracket N \rrbracket_1\}$, undecidability of BA follows from undecidability of the termination problem in MMs and Lemma 6.2.

Moreover, the number of consecutive barbs on e can be unbounded: once the machine reaches the HALT instruction then a barb e will be continuously available by always choosing to synchronize on $\overline{p_i}$. Hence, there exists a computation where $\llbracket N \rrbracket_1 \Downarrow_e^\omega$ and we can conclude that EA is undecidable. □

Notice that $\llbracket N \rrbracket_1$ is an \mathcal{E}_s^1 process without nested adaptable processes. Hence, even if we consider $\llbracket N \rrbracket_1$ as an \mathcal{E}_d^1 process, update prefixes cannot modify the topology of nested adaptable processes (that is, in the semantics of Figure 2 condition $\text{cond}(U, Q)$ always holds true) and the generated transition system is the same. Formally, this can be verified by using Lemma 2.18. As a consequence, the above undecidability result holds for \mathcal{E}_d^1 processes as well:

Corollary 6.4. *BA and EA are undecidable in \mathcal{E}_d^1 .* □

7. (UN)DECIDABILITY RESULTS FOR \mathcal{E}^2

7.1. Decidability of Bounded Adaptation. Here we prove that despite the previous undecidability result, BA is decidable for \mathcal{E}_d^2 processes. That is, given a process P , a set of processes M , and a barb α , there exists an algorithm to determine whether there exists a process $R \in \mathcal{CS}_P^M$ such that $R \Downarrow_\alpha^k$ holds. The proof appeals to the theory of well-structured transition systems (see Section 5.2). The algorithm consists of five steps:

- (1) We restrict the set of terms under consideration to those reachable by any $R \in \mathcal{CS}_P^M$. We characterize this set by
 - (a) considering the set of *sequential subterms* in \mathcal{CS}_P^M , i.e., the subterms of P and the processes in M that do not have parallel composition or adaptable processes as their topmost operator—see Definition 7.2—and
 - (b) introducing the ordering \preceq over a tree-like representation of the processes in such a set—see Definition 7.7.
- (2) Next, we prove that \preceq is a well-quasi-ordering (cf. Theorem 7.10) which is strongly compatible with respect to \longrightarrow (cf. Theorem 7.14).
- (3) These results enable us to compute a finite basis for the set of processes exhibiting α ; this set is *upward-closed* with respect to \preceq (cf. Theorem 7.20).

- (4) We then show that it is possible to compute the finite basis of the set of processes that expose α at least k consecutive times (Lemma 7.23).
- (5) Finally, we show that it is possible to determine whether or not some process $R \in \mathcal{CS}_P^M$ is included in the set generated by the finite basis (Theorem 7.24).

In what follows, we describe the definitions and results associated to these steps. For the sake of clarity, each of these descriptions is presented separately, in Sections 7.1.1—7.1.5.

Observe that the above strategy requires Kruskal’s theorem (Theorem 5.11) on well-quasi-orderings on trees. Unlike similar previous results exploiting the theory of well-structured transition systems for obtaining decidability results (e.g., [18]), in the case of \mathcal{E}_d^2 it is not possible to find a bound on the “depth” of processes. We illustrate this issue with a small example. Consider the process $R = a[P] \parallel !\tilde{a}\{a[a[\bullet]]\}.\mathbf{0}$. One possible evolution of R is the following:

$$R \longrightarrow a[a[P]] \parallel !\tilde{a}\{a[a[\bullet]]\}.\mathbf{0} \longrightarrow a[a[a[P]]] \parallel !\tilde{a}\{a[a[\bullet]]\}.\mathbf{0} \longrightarrow \dots$$

and thus one obtains a process with an unbounded number of nested adaptable processes. 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 that it has an effective pred-basis. This way, decidability of BA can be shown by following the five steps described above.

7.1.1. *Step (1)*. We start by introducing some auxiliary definitions.

Definition 7.1 (Parallel Processes). Let $P = \prod_{i=1}^m P_i \parallel \prod_{j=1}^n a_j[P'_j]$ be an \mathcal{E} process in normal form. The set of *top-level, parallel processes* of P , is defined as

$$\text{Par}(P) = \{P_i \mid i \in [1..m]\} \cup \{a_j[P'_j] \mid j \in [1..n]\}$$

This definition extends to sets of processes in normal form in the expected way.

Definition 7.2 (Sequential Subprocesses). Let P be an \mathcal{E}_d^2 process. The set of sequential subprocesses of P , denoted $\text{sub}(P)$, is defined inductively as follows:

$$\begin{aligned} \text{sub}(\pi.P) &= \{\pi.P\} \cup \text{sub}(P) \text{ if } \pi = a \text{ or } \pi = \bar{a} \\ \text{sub}(\tilde{a}\{U\}.Q) &= \{\tilde{a}\{U\}.Q\} \cup \text{sub}(U) \cup \text{sub}(Q) \\ \text{sub}(\sum_{i \in I} \pi_i.P_i) &= \{\sum_{i \in I} \pi_i.P_i\} \cup \bigcup_{i \in I} \text{sub}(\pi_i.P_i) \\ \text{sub}(!\pi.P) &= \{!\pi.P\} \cup \text{sub}(P) \\ \text{sub}(P \parallel Q) &= \text{sub}(P) \cup \text{sub}(Q) \\ \text{sub}(a[P]) &= \text{sub}(P) \\ \text{sub}(\bullet) &= \emptyset \end{aligned}$$

Observe that $\text{sub}(\mathbf{0}) = \text{sub}(\sum_{i \in \emptyset} \pi_i.P_i) = \{\mathbf{0}\}$. The definition extends to sets of processes as expected.

Notice that, since we are considering processes $P \in \mathcal{E}_d^2$ (which make use of update patterns that cannot include \bullet in the scope of prefixes), $\text{sub}(P)$ is a set of processes that are not update patterns, that is they cannot have free occurrences of \bullet .

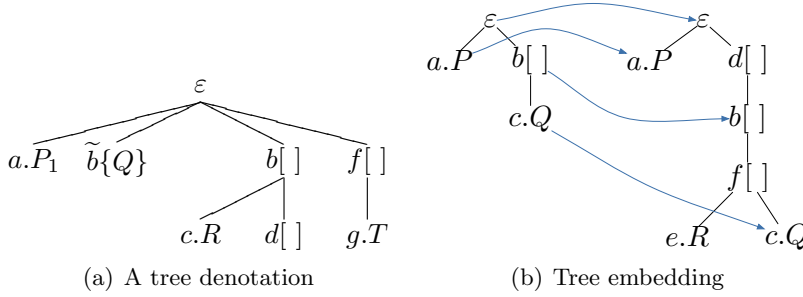


Figure 5: Tree denotations for \mathcal{E}_d^2 processes.

Definition 7.3. Let P be an \mathcal{E}_d^2 process. The set of adaptable processes names occurring in P , denoted $\text{apn}(P)$, is inductively defined (by resorting, in general, to $\text{apn}(U)$ over \mathcal{E}_d^2 update patterns U) as follows:

$$\begin{aligned}
\text{apn}(a[U]) &= \{a\} \cup \text{apn}(U) \\
\text{apn}(\pi.P) &= \text{apn}(P) \text{ if } \pi = a \text{ or } \pi = \bar{a} \\
\text{apn}(\tilde{a}\{U\}.Q) &= \text{apn}(U) \cup \text{apn}(Q) \\
\text{apn}(\sum_{i \in I} \pi_i.U_i) &= \bigcup_{i \in I} \text{apn}(\pi_i.U_i) \\
\text{apn}(!\pi.U) &= \text{apn}(\pi.U) \\
\text{apn}(U_1 \parallel U_2) &= \text{apn}(U_1) \cup \text{apn}(U_2) \\
\text{apn}(\bullet) &= \emptyset
\end{aligned}$$

The definition extends to sets of processes as expected.

Definition 7.4. Given a set of \mathcal{E}_d^2 processes S , we define:

$$\text{lab}(S) = \text{sub}(S) \cup \{a[] \mid a \in \text{apn}(S)\}$$

We are now ready to define the tree denotation of a process.

Definition 7.5 (Tree of a process). Let $P = \prod_{i=1}^m P_i \parallel \prod_{j=1}^n a_j[P'_j]$ be an \mathcal{E}_d^2 process in normal form. The tree denotation of P , denoted $\text{Tr}(P)$, is a tree over $\text{lab}(\{P\}) \cup \{\varepsilon\}$ and it is built as follows. The root is labeled ε , and has $m+n$ children: the former m are leaves 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.

Given a set of \mathcal{E}_d^2 processes S , \mathcal{T}_S denotes the set of trees over $\text{lab}(S) \cup \{\varepsilon\}$.

Example 7.6. Let P be the process $a.P_1 \parallel \tilde{b}\{Q\} \parallel b[c.R \parallel d[]] \parallel f[g.T]$. Given Definition 7.5, $\text{Tr}(P)$ is depicted in Figure 5(a).

We now define the ordering \preceq on processes. It corresponds to the extension of $=$, as described in Definition 5.10, to trees. Notice that when $=$ is extended to trees it is no longer a symmetric relation. More precisely:

Definition 7.7 (Ordering \preceq). Let P and Q be \mathcal{E}_d^2 processes. Also, let $=^{\text{tr}}$ stand for the extension of $=$ as in Definition 5.10. Then we decree: $P \preceq Q$ iff $\text{Tr}(P) =^{\text{tr}} \text{Tr}(Q)$.

In other words, given two processes P and Q such that $\text{Tr}(P) =^{\text{tr}} \text{Tr}(Q)$, one simply checks if all the labels of $\text{Tr}(P)$ occur in $\text{Tr}(Q)$ and respect the ancestor relation.

Example 7.8. Let S and T be the processes defined as

$$\begin{aligned} S &= a.P \parallel b[c.Q] \\ T &= a.P \parallel d[b[f[e.R \parallel c.Q]]] \end{aligned}$$

Then we have $S \preceq T$; tree denotations for both processes (and the injection between them) are depicted in Figure 5(b).

We write $P \longrightarrow_{\succeq} Q$ if there is some P' such that $P \longrightarrow P'$ and $P' \succeq Q$. We now define the set of all derivatives of a given \mathcal{E}_d^2 process and show that \preceq is a wqo over it.

Definition 7.9. Given an \mathcal{E}_d^2 process P , we define $\text{Deriv}(P) = \{Q \mid P \longrightarrow^* Q\}$. This definition is extended to sets of processes in the expected way.

7.1.2. *Step 2.* We start by showing that given a set of processes S , $=^{\text{tr}}$ is a wqo over \mathcal{T}_S .

Theorem 7.10. *Let S be a set of \mathcal{E}_d^2 processes. Then, relation $=^{\text{tr}}$ is a wqo over \mathcal{T}_S .*

Proof. The set $\text{lab}(S)$ is finite by construction. Hence, by Proposition 5.9, equality is a wqo over $\text{lab}(S) \cup \{\varepsilon\}$. Finally, since $=$ is a wqo, using Kruskal's Theorem (Theorem 5.11) we infer that $=^{\text{tr}}$ is a wqo over \mathcal{T}_S . \square

We now prove that the trees constructed from processes contained in the set of all derivatives form a subset of \mathcal{T}_S . The following notion of *monadic and biadic contexts* will be useful in proofs.

Definition 7.11 (Monadic and Biadic Contexts). A *monadic context* is a context with one hole (denoted “.”) and is defined according to the following grammar:

$$C[\cdot] ::= [\cdot] \mid C[\cdot] \parallel P \mid a[C[\cdot]]$$

where P is an \mathcal{E} process. Similarly, a *biadic context* is a context with two holes (denoted “ \cdot_1 ” and “ \cdot_2 ”, respectively) defined according to the following grammar:

$$D[\cdot_1, \cdot_2] ::= C[\cdot_1] \parallel C[\cdot_2] \mid a[D[\cdot_1, \cdot_2]] \parallel P \mid a[D[\cdot_1, \cdot_2]]$$

where P is an \mathcal{E} process and C is a monadic context. As customary, $C[P]$ and $D[R, Q]$ represent the processes obtained by replacing the holes in contexts $C[\cdot]$ and $D[\cdot_1, \cdot_2]$ with processes P and R, Q , respectively.

Lemma 7.12. *Let P be an \mathcal{E}_d^2 process. If $P \longrightarrow Q$ then $\text{Tr}(Q) \in \mathcal{T}_{\{P\}}$.*

Proof. By induction on the height of the derivation tree for $P \longrightarrow Q$, with a case analysis in the last rule used. There are seven cases to check. We recall that $\text{Tr}(Q) \in \mathcal{T}_{\{P\}}$ iff $\text{Tr}(Q)$ is over $\text{lab}(P) \cup \{\varepsilon\}$.

Case (Act1): Then $P = P_1 \parallel P_2$ and $Q = P'_1 \parallel P_2$, with $P_1 \longrightarrow P'_1$. By Definition 7.5 we have $\text{Tr}(P)$ is over $\text{lab}(P_1) \cup \text{lab}(P_2) \cup \{\varepsilon\}$. By inductive hypothesis, we have that $\text{Tr}(P'_1)$ is over $\text{lab}(P_1) \cup \{\varepsilon\}$. Hence we can conclude that $\text{Tr}(Q)$ is over $\text{lab}(P_1) \cup \text{lab}(P_2) \cup \{\varepsilon\}$, thus $\text{Tr}(Q) \in \mathcal{T}_{\{P\}}$.

Case (Act2): Analogous to the case for (ACT1) and omitted.

Case (Loc): hen $P = a[P_1]$ and $Q = a[P'_1]$, with $P_1 \longrightarrow P'_1$. By Definition 7.5 we have $\text{Tr}(P)$ is over $\text{lab}(P_1) \cup \{a[\]\} \cup \{\varepsilon\}$. By inductive hypothesis, we have that $\text{Tr}(P'_1)$ is over $\text{lab}(P_1) \cup \{\varepsilon\}$. Hence we can conclude that $\text{Tr}(Q)$ is over $\text{lab}(P_1) \cup \{a[\]\} \cup \{\varepsilon\}$, thus $\text{Tr}(Q) \in \mathcal{T}_{\{P\}}$.

Cases (Tau1)-(Tau2): Then $P \equiv C_1[A] \parallel C_2[B]$, where C_1 and C_2 are monadic contexts as in Definition 7.11. Moreover, A is either $!b.Q$ or $\sum_{i \in I} \pi_i.Q_i$ with $\pi_l = b$, for some $l \in I$, and B is either $!\bar{b}.R$ or $\sum_{i \in I} \pi_i.R_i$ with $\pi_l = \bar{b}$, for some $l \in I$.

We consider only the case in which $A = \sum_{i \in I} \pi_i.Q_i$ with $\pi_l = b$ and $B = !\bar{b}.R$; the other cases are similar. Then $Q \equiv C_1[Q_l] \parallel C_2[R \parallel !\bar{b}.R]$. We know that $\text{Tr}(P)$ is over $\text{lab}(C_1) \cup \text{lab}(A) \cup \text{lab}(C_2) \cup \text{lab}(B) \cup \{\varepsilon\}$ and by noticing that $\text{lab}(Q_l) \subseteq \text{lab}(A)$ and $\text{lab}(R \parallel !\bar{b}.R) \subseteq \text{lab}(B)$ we can conclude that $\text{Tr}(Q) \in \mathcal{T}_{\{P\}}$.

Cases (Tau3)-(Tau4): Then $P \equiv C_1[A] \parallel C_2[B]$ where:

- C_1 and C_2 are monadic contexts, as in Definition 7.11;
- $A = b[P_1]$, for some P_1 ;
- $B = \sum_{i \in I} \pi_i.R_i$ with $\pi_l = \tilde{b}\{b[U] \parallel P_2\}$ for $l \in I$, or $B = !\tilde{b}\{U\}.R$, for some R .

We consider the case in which $B = !\tilde{b}\{b[U] \parallel P_2\}.R$; the other case is similar. Then $Q \equiv C_1[U \langle P_1 \rangle] \parallel C_2[R \parallel !\tilde{b}\{U\}.R]$. We know that $\text{Tr}(P)$ is over $\text{lab}(C_1) \cup \text{lab}(A) \cup \text{lab}(C_2) \cup \text{lab}(B) \cup \{\varepsilon\}$ and by noticing that $\text{lab}(R \parallel !\tilde{b}\{U\}.R) \subseteq \text{lab}(B)$ and that because of the restrictions on \mathcal{E}_d^2 P_3 cannot occur behind a prefix, then we can conclude that $\text{Tr}(Q) \in \mathcal{T}_{\{P\}}$. \square

Lemma 7.12 can be used to show that $\{\text{Tr}(P) \mid P \in \text{Deriv}(S)\} \subseteq \mathcal{T}_S$, for some set of processes S . Then, using Theorem 7.10 we can conclude that \preceq is a wqo over it:

Corollary 7.13. *Let S be a set of \mathcal{E}_d^2 processes. Then, \preceq is a wqo over $\text{Deriv}(S)$.* \square

The next result states strong compatibility of \preceq with respect to reductions of \mathcal{E}_d^2 .

Theorem 7.14 (Strong Compatibility). *Let P and Q be \mathcal{E}_d^2 processes such that $P \preceq Q$. Then, $P \longrightarrow P'$ implies that there exists Q' such that $Q \longrightarrow Q'$ and $P' \preceq Q'$.*

Proof. By a case analysis on the reduction $P \longrightarrow P'$. It can be the result of either a *input/output synchronization*—through rules (TAU1)/(TAU2)—or an *update synchronization*—through rules (TAU3)/(TAU4)). In both cases, the reduction may be combined with uses of rule (LOC), (ACT1), and (ACT2).

We consider these two kinds of synchronizations separately. Let n be a node with ancestor m , and let $\text{Tr}(P)$ be a tree with root ε . Below, when we say that n is replaced by $\text{Tr}(P)$ we mean that: (i) ε is merged with m ; (ii) all children of ε are added as siblings of n ; and (iii) n itself is removed.

Input/output synchronization: Then we have

$$P \equiv D[A, B]$$

where D is a biadic context as in Definition 7.11, A is either $!a.P_1$ or $\sum_{i \in I} \pi_i.Q_i$ with $\pi_l = a$ and $Q_l = P_1$ for some $l \in I$, and B is either $!\bar{a}.P_2$ or $\sum_{i \in I} \pi_i.R_i$ with $\pi_l = \bar{a}$ and $R_l = P_2$, for some $l \in I$.

Consider the tree $\text{Tr}(P)$, and let m and n be two of its nodes, labeled A and B , respectively.

We first consider the modifications to $\text{Tr}(P)$ when $P \longrightarrow P'$. The tree $\text{Tr}(P')$ is obtained from $\text{Tr}(P)$ in the following way:

- (1) the node labeled A is replaced with $\text{Tr}(P_1)$;
- (2) the node labeled B is replaced with $\text{Tr}(P_2)$.

Since $P \preceq Q$, the definition of \preceq ensures that there exists a mapping f that associates nodes in $\text{Tr}(P)$ to nodes in $\text{Tr}(Q)$. In turn, this ensures the existence of a node $f(m)$ in

$\text{Tr}(Q)$ labeled A . It also ensures the existence of a node $f(n)$ labeled B and which has a common ancestor with $f(m)$. Hence, the reduction can take place in Q as well, and so $Q \longrightarrow Q'$. Now, $\text{Tr}(Q')$ is obtained from $\text{Tr}(Q)$ by applying the same changes described above to the target nodes (of the input and the output) according to f .

The last thing to show is $P' \preceq Q'$, which follows by observing that the mapping between $\text{Tr}(P')$ and $\text{Tr}(Q')$ is necessarily the same mapping f between $\text{Tr}(P)$ and $\text{Tr}(Q)$, for all the nodes that have not been modified by the reduction and that there is a one-to-one correspondence for the other nodes, as the new trees $\text{Tr}(P_1)$ and $\text{Tr}(P_2)$ are added to both $\text{Tr}(P)$ and $\text{Tr}(Q)$. Thus, $\text{Tr}(P') =^{\text{tr}} \text{Tr}(Q')$.

Update synchronization: Then we have

$$P \equiv D[a[P_1], A]$$

where D is a biadic context as in Definition 7.11 and A is either $! \tilde{a}\{P_2\}.R$ or $\sum_{i \in I} \pi_i.Q_i$ with $\pi_i = \tilde{a}\{P_2\}$ and $Q_i = R$ for some $i \in I$. Consider the tree $\text{Tr}(P)$, and let m and n be two of its nodes, labeled A and $a[]$ (with subtree $\text{Tr}(P_1)$), respectively.

We first consider the modifications to $\text{Tr}(P)$ when $P \longrightarrow P'$. The tree $\text{Tr}(P')$ is obtained from $\text{Tr}(P)$ in the following way:

- (1) the node labeled A is replaced with $\text{Tr}(R)$;
- (2) as for the tree rooted in $a[]$ ($\text{Tr}(P_1)$), it is replaced with $\text{Tr}(P_2 \langle\langle P_1 \rangle\rangle)$.

Since $P \preceq Q$, the definition of \preceq ensures that there exists a mapping f that associates nodes in $\text{Tr}(P)$ to nodes in $\text{Tr}(Q)$. In turn, this ensures the existence of a node $f(m)$ in $\text{Tr}(Q)$ labeled A . It also ensures the existence of a node $f(n)$ labeled $a[]$ and which has a common ancestor with $f(m)$. Hence, the update synchronization above can take place in Q as well, and so $Q \longrightarrow Q'$. Now, $\text{Tr}(Q')$ is obtained from $\text{Tr}(Q)$ by applying the same changes described above to the target nodes (of the adaptable process a and of the update in A) according to f .

The last thing to show is that $P' \preceq Q'$, which follows by observing that the mapping between $\text{Tr}(P')$ and $\text{Tr}(Q')$ is the same mapping f between $\text{Tr}(P)$ and $\text{Tr}(Q)$, 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 label in node $f(m)$: all nodes removed in $\text{Tr}(P')$ have been removed in $\text{Tr}(Q')$, hence nodes m and $f(m)$ are still in relation.
- (2) Finally, we consider the two trees rooted in n and $f(n)$, namely $S = \text{Tr}(P_2 \langle\langle P_1 \rangle\rangle)$ and $T = \text{Tr}(P_2 \langle\langle Q_1 \rangle\rangle)$, respectively. S is the same subtree as T apart from some subtrees of P_2 and Q_2 that can be put easily in relation as the subtrees $\text{Tr}(P_1)$ and $\text{Tr}(Q_1)$ are in relation with f .

Thus, $\text{Tr}(P') =^{\text{tr}} \text{Tr}(Q')$. □

7.1.3. *Step (3).* We now move on to characterize the set of predecessors of a given process (cf. Definition 5.3) by means of a finite basis (cf. Definition 5.6). Given a set S of processes, we are only interested in those predecessors whose tree is in \mathcal{T}_S . As it will be clear later on, S is intended to represent all processes in a cluster (cf. Definition 3.2).

Definition 7.15. Let P and S be an \mathcal{E}_d^2 process and a set of \mathcal{E}_d^2 processes, respectively. We define:

$$\text{Pred}_S(P) = \{Q \mid Q \in \text{Pred}(P), \text{Tr}(Q) \in \mathcal{T}_S\}.$$

As we have seen, reductions in \mathcal{E} originate only from synchronizations between input and output prefixes or from synchronizations between an adaptable process and a corresponding update prefix. Our characterization of $\text{Pred}_S(P)$ as a finite basis relies, intuitively, on the formalization of the “parts” of P that might have been involved in a reduction leading to P . We introduce the notion of *syntactic context*: it allows us to reason about the *decompositions* of P , which are useful to describe the subprocesses that have been involved in the reduction to P ; such subprocesses may be contained in P or they can be found in S . In the latter case, we must appeal to *parallel extensions* of the syntactic context defining the given decomposition, as we give next:

Definition 7.16 (Syntactic Contexts, Decompositions, Extensions). *Syntactic contexts*, ranged over K, K', \dots , are defined by the following syntax:

$$K ::= [\cdot] \mid a[K] \mid K \parallel K \mid P$$

where P is as in Definition 2.3 using contexts as in Definition 2.6 (2).

Given a process P , a syntactic context K , and processes \tilde{R} , we say that $K[\tilde{R}]$ is a *decomposition* of P if $P = K[\tilde{R}]$. We assume processes \tilde{R} fill the holes in K preserving the order in which they appear.

A *parallel extension* of K is a syntactic context with exactly two holes obtained in the following way:

$$\text{Ext}(K) = \{K, K \parallel [\cdot], K \parallel [\cdot] \parallel [\cdot]\} \cap SC_2$$

where SC_2 is the set of all syntactic contexts with exactly two holes.

We move on to define the pred-basis function for processes; it is defined with respect to a set of processes S and noted $pb_S(\cdot)$. First, we present some intuitions and auxiliary definitions. Given a process P , the set $pb_S(P)$ represents the basis for the set $\uparrow \text{Pred}_S(\uparrow P)$; in other words, it is a finite representation of those processes that reduce to P , up to \preceq , i.e., a basis for all those Q such that $Q \longrightarrow_{\succeq} P$. To this aim, we consider all the decompositions of P as $K[\tilde{R}]$, for some syntactic context K and processes \tilde{R} , with $|\tilde{R}| \leq 2$. There are finitely many such decompositions. The idea is to characterize a predecessor Q of P by suitably filling in the holes in (possibly an extension of) K so that the Q is such that $\text{Tr}(Q) \in \mathcal{T}_S$. Now, each K can have two, one, or even zero holes (as a process can be a decomposition of itself). In case $|\tilde{R}| < 2$, the syntactic context must be extended so as to contain exactly two holes; this is defined by $\text{Ext}(K)$ above.

Let us analyze the possibilities for such an extended context. As we have seen, reductions in \mathcal{E}_d^2 arise from the synchronization of two complementary prefixes occurring (i) inside two sums, or (ii) one inside a sum and the other in a replicated process; or (iii) both prefixes in two replicated processes. For the sake of readability, and with a little abuse of notation, in the explanation below we use biadic contexts filled in with the interacting prefixes, rather than with the processes in which such prefixes occur. That is, we write $D[\alpha.P, \beta.Q]$ rather than, e.g., $D[\alpha.P + M, !\beta.Q]$. There are six cases. If K has exactly two holes then it means that the reduction is “internal” to process P . That is, the reduction can be traced back by looking at subprocesses of P . Then $P = K[P_1, P_2]$ and no parallel extension is needed. There are two possible cases:

- (1) P is the result of an input/output synchronization and so its predecessors are of the form $Q = K[a.P_1, \bar{a}.P_2]$, for some $a \in \text{apn}(S)$ and where $a.P_1$ and $\bar{a}.P_2$ are processes in $\text{sub}(S)$.

- (2) P is the result of a synchronization between an update prefix and some corresponding adaptable process, and so its predecessors are of the form $Q = K[\tilde{a}\{Q'\}.P_1, a[Q'']]$, where $P_2 = Q' \ll Q''$ and $a \in \text{apn}(S)$. Also, process $\tilde{a}\{Q'\}.P_1$ should belong to $\text{sub}(S)$. Moreover, depending on the number of holes in Q' there are two possible situations: (1) if $|Q'|_\bullet = 0$ then $P_2 = Q'$ and Q'' can be any process in $\text{sub}(S)$; (2) if $|Q'|_\bullet > 0$ then Q'' is taken in such a way that $P_2 = Q' \ll Q''$.

In case K has one hole only then we extend the context with a hole so as to accommodate some process not originally present in P . That is, $P = K[P_1]$ and the reduction to P is characterized by the interaction between a prefix guarding subprocess P_1 and some other subprocess external to P (cases (3) and (4) below). It can also be the case that the reduction is an update synchronization leading to P_1 (case (5)). We thus consider the extended context $D[\cdot, \cdot] \equiv K[\cdot] \parallel [\cdot]$. There are three possible cases:

- (3) P is the result of an input/output synchronization, and so its predecessors are either of the form $Q \equiv D[a.P_1, \bar{a}.Q_2]$ or $Q \equiv D[\bar{a}.P_1, a.Q_2]$, for some $a \in \text{apn}(S)$ and processes $a.P_1$ and $\bar{a}.Q_2$ ($\bar{a}.P_1$ and $a.Q_2$, respectively) belong to $\text{sub}(S)$.
- (4) P is the result of a synchronization between an update prefix guarding P_1 and some corresponding adaptable process. Hence, for some $a \in \text{apn}(S)$, its predecessors are of the form $Q \equiv D[\tilde{a}\{Q'\}.P_1, a[Q'']]$, with processes $\tilde{a}\{Q'\}.P_1$ and Q'' in $\text{sub}(S)$.
- (5) P is the result of a synchronization between an update prefix and some corresponding adaptable process, in such a way that their synchronization leads to P_1 . This way, the predecessors of P are of the form $Q \equiv D[\tilde{a}\{Q'\}.Q_2, a[Q'']]$ or $Q \equiv D[a[Q''], \tilde{a}\{Q'\}.Q_2]$ where $P_1 = Q' \ll Q''$, for some $a \in \text{apn}(S)$. Similarly as in case (2) above, process $\tilde{a}\{Q'\}.Q_2$ should belong to $\text{sub}(S)$. Moreover, depending on the number of holes in Q' there are two possible situations: (1) if $|Q'|_\bullet = 0$ then $P_1 = Q'$ and Q'' can be any process in $\text{sub}(S)$; (2) if $|Q'|_\bullet > 0$ then Q'' is taken in such a way that $P_1 = Q' \ll Q''$.

The last case to consider is when K has no holes, i.e., the trivial decomposition of P as itself. Then $D[\cdot, \cdot] \equiv P \parallel [\cdot] \parallel [\cdot]$ and we have:

- (6) P is the result of a synchronization between the subprocesses in the two added holes. That is, its predecessors are of one of the following: (1) $Q \equiv P \parallel a.R_1 \parallel \bar{a}.R_2$ and (2) $Q \equiv P \parallel \tilde{a}\{Q'\}.R_1 \parallel a[R_2]$. In both cases, $a \in \text{apn}(S)$ and the holes are filled in with processes in $\text{sub}(S)$.

Before giving the definition of $pb_S(Q)$, we introduce an auxiliary notion.

Definition 7.17. Let P be an \mathcal{E}_d^2 process. The set of update patterns occurring in P , denoted $\text{Upd}(P)$, is inductively defined as follows:

$$\begin{aligned}
\text{Upd}(\tilde{a}\{U\}.Q) &= \{U\} \cup \text{Upd}(U) \cup \text{Upd}(Q) \\
\text{Upd}(a[P]) &= \text{Upd}(P) \\
\text{Upd}(\pi.P) &= \text{Upd}(P) \text{ if } \pi = a \text{ or } \pi = \bar{a} \\
\text{Upd}(\sum_{i \in I} \pi_i.U_i) &= \bigcup_{i \in I} \text{Upd}(\pi_i.U_i) \\
\text{Upd}(!\pi.U) &= \text{Upd}(\pi.U) \\
\text{Upd}(U_1 \parallel U_2) &= \text{Upd}(U_1) \cup \text{Upd}(U_2) \\
\text{Upd}(\bullet) &= \emptyset
\end{aligned}$$

This definition extends to sets of processes as expected.

Definition 7.18 (Pred-basis). Let S be a set of \mathcal{E}_d^2 processes and P be an \mathcal{E}_d^2 process such that $\text{Tr}(P) \in \mathcal{T}_S$. Given the set

$$\begin{aligned} \mathcal{G}_{S, \tilde{R}} = & \text{sub}(S) \cup \{a[H] \mid a \in \text{apn}(S), H \in \text{sub}(S)\} \cup \\ & \{a[H] \mid R = U \langle\langle H \rangle\rangle, R \in \tilde{R}, U \in \text{Upd}(S), |U|_{\bullet} \geq 1\} \end{aligned}$$

the *pred-basis* of P with respect to S , denoted $pb_S(P)$, is defined as the set:

$$pb_S(P) = \bigcup_{P=K[\tilde{R}]} \{Q \mid Q \longrightarrow_{\succeq} P, Q = D[\tilde{G}], D \in \text{Ext}(K), \tilde{G} \subseteq \mathcal{G}_{S, \tilde{R}}\}$$

We show that the well structured transition system given above has an effective pred-basis (cf. Definition 5.7).

Theorem 7.19. *Let P and S be a \mathcal{E}_d^2 process and a set of \mathcal{E}_d^2 processes, respectively. We then have that $\uparrow pb_S(P) = \uparrow \text{Pred}_S(\uparrow P)$. Moreover, $pb_S(\cdot)$ is effective.*

Proof. The inclusion $\uparrow pb_S(P) \subseteq \uparrow \text{Pred}_S(\uparrow P)$ follows by construction. We consider the other inclusion, i.e., $\uparrow \text{Pred}_S(\uparrow P) \subseteq \uparrow pb_S(P)$. Given some $R \in \uparrow \text{Pred}_S(\uparrow P)$, then we show that there is a $Q \in pb_S(P)$ such that $Q \preceq R$. As hinted at above, depending on the kind of reduction that can occur to reach process P we should consider six cases. Below, K, K_1 and K_2 are syntactic contexts as in Definition 7.16:

Reduction is “internal” to P . Then we have one of the following cases:

- (1) P is obtained as an input/output synchronization. Then, $R = K_1[A, B]$ (or $R = K_1[B, A]$) where A is either $!a.Q_1$ or $\sum_{i \in I} \pi_i.P_i$ with $\pi_l = a$ and $P_l = Q_1$ for some $l \in I$, and B is either $!\bar{a}.Q_2$ or $\sum_{i \in I} \pi_i.R_i$ with $\pi_l = \bar{a}$ and $R_l = Q_2$, for some $l \in I$. There exists K_2 such that $P = K_2[Q_1, Q_2]$ and $R \longrightarrow_{\succeq} P$. Since $R \in \uparrow \text{Pred}_S(\uparrow P)$ then $A, B \in \text{sub}(S)$ and we can conclude $R \succeq Q = K_2[A, B] \in pb_S(P)$.
- (2) if P is the result of an update of an adaptable process then

$$R = K_1[A, a[Q'']]$$

where A is either $!\tilde{a}\{Q'\}.Q_1$ or $\sum_{i \in I} \pi_i.R_i$ with $\pi_l = \tilde{a}\{Q'\}$ and $R_l = Q_1$ for some $l \in I$, and there exists K_2 such that $P = K_2[Q_1, Q_2]$, $R \longrightarrow_{\succeq} P$ where we have that $Q_2 = Q' \langle\langle Q'' \rangle\rangle$. If $|Q''|_{\bullet} = 0$ then $Q_2 = Q'$ and as $R \in \uparrow \text{Pred}_S(\uparrow P)$ we have $A, Q'' \in \text{sub}(S)$ and therefore $R \succeq Q = K_2[A, a[0]] \in pb_S(P)$. Otherwise if $|Q''|_{\bullet} > 0$ then $A \in \text{sub}(S)$ and we can immediately conclude $R \succeq Q = K_2[A, a[Q'']] \in pb_S(P)$.

Reduction partially present in P . Then we have one of the following cases:

- (3) if $R = K_1[A, B]$ (or $R = K_1[B, A]$) where A is either $!a.Q_1$ or $\sum_{i \in I} \pi_i.P_i$ with $\pi_l = a$ and $P_l = Q_1$ for some $l \in I$, and B is either $!\bar{a}.Q_2$ or $\sum_{i \in I} \pi_i.R_i$ with $\pi_l = \bar{a}$ and $R_l = Q_2$, for some $l \in I$. Then there exists K_2 such that $P = K_2[Q_1]$ and $R \longrightarrow_{\succeq} P$. As $A, B \in \text{sub}(S)$ (respectively $\bar{a}.Q_1, a.Q_2 \in \text{sub}(S)$) we can conclude $R \succeq Q = K_2[A] \parallel B \in pb_S(P)$ (respectively $Q = K_2[B] \parallel A$).
- (4) if $R = K_1[A, a[Q_2]]$ where A is either $!\tilde{a}\{Q'\}.Q_1$ or $\sum_{i \in I} \pi_i.R_i$ with $\pi_l = \tilde{a}\{Q'\}$ and $R_l = Q_1$ for some $l \in I$. Then there exists K_2 such that $P = K_2[Q_1]$ and $R \longrightarrow_{\succeq} P$. As $A \in \text{sub}(S)$ then $R \succeq Q = K_2[A] \parallel a[0] \in pb_S(P)$.

- (5) if $R = K_1[a[Q''], A]$ where A is either $!\tilde{a}\{Q'\}.Q_2$ or $\sum_{i \in I} \pi_i.R_i$ with $\pi_l = \tilde{a}\{Q'\}$ and $R_l = Q_2$ for some $l \in I$. Then there exists K_2 such that $P = K_2[Q_1]$, $R \longrightarrow \succeq P$. If $|Q'|_{\bullet} = 0$ then $Q_1 = Q'$, $A \in \text{sub}(S)$ and we can conclude $R \succeq Q = K_2[a[0]] \parallel A \in \text{pb}_S(P)$. Otherwise if $|Q'|_{\bullet} > 0$ then $Q_1 = Q' \ll \langle Q'' \rangle$ and we can conclude $R \succeq Q = K_2[a[Q'']] \parallel A \in \text{pb}_S(P)$.

Reduction external to P . Then we have:

- (6) $R = K_1[P, A, B]$ or $R = K_1[P, C, a[Q_3]]$ where A is either $!a.Q_1$ or $\sum_{i \in I} \pi_i.P_i$ with $\pi_l = a$ and $P_l = Q_1$ for some $l \in I$, B is either $!\bar{a}.Q_2$ or $\sum_{i \in I} \pi_i.R_i$ with $\pi_l = \bar{a}$ and $R_l = Q_2$, for some $l \in I$ and C is either $!\tilde{a}\{Q_1\}.Q_2$ or $\sum_{i \in I} \pi_i.P_i$ with $\pi_l = \tilde{a}\{Q_1\}$ and $P_l = Q_2$ for some $l \in I$. As all processes A, B, C, Q_3 are taken from $\text{sub}(S)$ we can conclude $R \succeq Q = P \parallel A \parallel B \in \text{pb}_S(P)$ (respectively $Q = P \parallel C \parallel a[Q_3]$).

Moreover, the construction of $\text{pb}_S(Q)$ is effective. In particular, given a syntactic context K , there are finitely many ways of extending it with one or two holes so as to obtain a parallel extension $D \in \text{Ext}(K)$. In Definition 7.18, notice that when filling in the contexts with terms in \tilde{G} , both the set of sequential subprocesses and the ways of constructing an update pattern U are finite. This concludes the proof. \square

Theorem 7.20. *Let S be a set of \mathcal{E}_d^2 processes. $(\text{Deriv}(S), \longrightarrow, \preceq)$ is a finitely branching, well-structured transition system with strong compatibility, decidable \preceq , and effective pred-basis pb_S . Hence, it is possible to compute a finite basis pb^* of $\text{Pred}_S^*(I)$ (and $\text{Pred}_S^+(I)$) for any upward-closed set I which is given via a finite basis.*

Proof. Follows from Proposition 5.8, using Remark 2.17, and Theorems 7.14 and 7.19. \square

7.1.4. *Step (4).* Next, we define the basis of the set of processes that immediately exhibit a barb α .

Definition 7.21. Let S and α be a set of \mathcal{E}_d^2 processes and a name $\alpha \in \{a, \bar{a} \mid a \in \mathcal{N}\}$, respectively. Then, we define:

$$\text{fb}_\alpha(S) = \{R \in \text{sub}(S) \mid R \downarrow_\alpha\}$$

Given an initial process P , a set of processes M , and a barb α , to determine whether BA is decidable, we check if there exists a process $R \in \mathcal{CS}_P^M$ such that $R \downarrow_\alpha^k$. It is sufficient to check if R appears in the set of the predecessors of the processes that can exhibit α at least k consecutive times. Since \preceq imposes a well-quasi order on \mathcal{E}_d^2 processes, it is enough to characterize the set of predecessors by means of its finite basis, as shown by Theorem 7.20. More precisely, if $k = 1$ then it is sufficient to check if R is in the set of predecessors of the processes in $\text{fb}_\alpha(S)$, where $S = M \cup \{P\}$. Otherwise, if $k > 1$ then we need we need to check for the existence of processes R_1, \dots, R_k such that $R \longrightarrow^* R_1 \longrightarrow \dots \longrightarrow R_k$, with $R_i \xrightarrow{\alpha}$ for $i \in [1..k]$. To do this, we proceed backwards. We begin by computing the finite basis $\text{fb}_\alpha(S)$; process R_k should be in its upward closure. Then, we compute a finite basis for the set of processes in $\text{Pred}_S(\text{fb}_\alpha(S))$ which exhibit α immediately; R_{k-1} should be in the upward closure of this finite basis, which is constructed as follows. Notice by virtue of Theorem 7.19, we can rely on the pred-basis given by Definition 7.18, i.e., $\text{pb}_S(\text{fb}_\alpha(S))$, in this case. We consider two classes of elements of $\text{pb}_S(\text{fb}_\alpha(S))$: the first one is composed of those processes that can immediately perform α , while the second contains the rest. The

desired finite basis is obtained by taking the set of processes containing (i) every process in the first class and (ii) every Q in the second class (but with a minimal modification, with respect to the ordering \preceq , in such a way it can exhibit α immediately). The latter is achieved by function $\text{Add}_B(Q)$ (cf. Definition 7.22) which “plugs” into every Q a process in $\text{fb}_\alpha(S)$ either in parallel at the top level or inside an adaptable process. This procedure iterates as expected; each iteration considers the predecessors of the elements of the finite basis obtained in the previous one. In the last step, in order to calculate all the predecessors of process R_1 we apply Theorem 7.20, thus obtaining a finite basis pb^* where it is sufficient to check whether R belongs to its upward closure. More formally:

Definition 7.22. Let S be a set of \mathcal{E}_d^2 processes. Given the following set definitions (with C being a monadic context as in Definition 7.11)

$$\begin{aligned}\text{Add}_B(Q) &= \{Q \parallel R \mid R \in B\} \cup \{C[a[R \parallel Q_i]] \mid Q = C[a[Q_i]], R \in B\} \\ \text{lb}_\alpha(A, B) &= \{Q \in A \mid Q \xrightarrow{\alpha}\} \cup \{\text{Add}_B(Q) \mid Q \in A \text{ and } Q \not\xrightarrow{\alpha}\}\end{aligned}$$

we define the finite basis $\text{FB}_{\alpha,k}(S) = \text{pb}^*(\text{B}_{\alpha,k}(S))$ where $k \geq 1$ and

$$\text{B}_{\alpha,k}(S) = \begin{cases} \text{fb}_\alpha(S) & \text{if } k = 1 \\ \text{lb}_\alpha(\text{pb}_S(\text{B}_{\alpha,k-1}(S)), \text{fb}_\alpha(S)) & \text{otherwise} \end{cases}$$

The effectiveness of $\text{FB}_{\alpha,k}$ will allow us to prove the decidability of BA.

Lemma 7.23. *Let S be a set of \mathcal{E}_d^2 processes, and let $\alpha \in \{a, \bar{a} \mid a \in \mathcal{N}\}$. Then, $\text{FB}_{\alpha,k}(S)$ is effective.*

Proof. The effectiveness of the calculation of the finite basis of $\text{Pred}_S^*(\cdot)$ follows from Theorem 7.20. The set $\text{lb}_\alpha(\cdot, \cdot)$ 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 $\text{fb}_\alpha(S)$, which in turn can immediately exhibit α . \square

7.1.5. *Step (5).* We conclude by showing how to determine whether there exists a process R in \mathcal{CS}_P^M that exhibits α . Recall that $\text{Par}(P)$ is the set of all processes and all adaptable processes in P which are in parallel at top level (see Definition 7.1). We can finally state:

Theorem 7.24. *BA is decidable for \mathcal{E}_d^2 .*

Proof. Let P and $M = \{T_1, \dots, T_n\}$ be an initial process and a set of \mathcal{E}_d^2 processes, respectively. In order to show that BA is decidable, it suffices to check that, given some α and $k \geq 1$, there exists a process $R \in \mathcal{CS}_P^M$ such that $R \Downarrow_\alpha^k$. More precisely, letting $S = \{P\} \cup M$, we have to check if there exists a process $Q \in \text{FB}_{\alpha,k}(S)$ such that $Q \preceq R$. From Lemma 7.23, we know that it is possible to compute the set $\text{FB}_{\alpha,k}(S)$. Then, for each $Q_i \in \text{FB}_{\alpha,k}(S)$ we analyze the processes in $\text{Par}(Q_i)$ (cf. Definition 7.1). Let V be the set of the processes Q'_j in $\text{Par}(Q_i)$ 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 V . At this point, it is enough to check whether $Q_i^* \preceq P$. If this is the case, for at least one $Q_i \in \text{FB}_{\alpha,k}(S)$, then we can conclude that there exists $R \in \mathcal{CS}_P^M$ such that $R \Downarrow_\alpha^k$; otherwise there exists no $R \in \mathcal{CS}_P^M$ such that $R \Downarrow_\alpha^k$. \square

$$\begin{aligned}
\text{CONTROL} &= !a. (\bar{f} \parallel \bar{b} \parallel \bar{a}) \parallel \bar{a}. a. (\bar{p}_1 \parallel e) \parallel !h. (g. \bar{f} \parallel \bar{h}) \\
\text{REGISTER } r_j & \\
\llbracket r_j = m \rrbracket_2 &= \begin{cases} r_j[!inc_j. \bar{u}_j \parallel \bar{z}_j] & \text{if } m = 0 \\ r_j[!inc_j. \bar{u}_j \parallel \prod^m \bar{u}_j \parallel \bar{z}_j] & \text{if } m > 0. \end{cases} \\
\text{INSTRUCTIONS } (i : I_i) & \\
\llbracket (i : \text{INC}(r_j)) \rrbracket_2 &= !p_i. f. (\bar{g} \parallel b. \overline{inc_j}. \bar{p}_{i+1}) \\
\llbracket (i : \text{DECJ}(r_j, s)) \rrbracket_2 &= !p_i. f. (\bar{g} \parallel (u_j. (\bar{b} \parallel \bar{p}_{i+1}) \\
&\quad + z_j. \tilde{r}_j \{ r_j[!inc_j. \bar{u}_j \parallel \bar{z}_j] \}. \bar{p}_s)) \\
\llbracket (i : \text{HALT}) \rrbracket_2 &= !p_i. \bar{h}. h. \tilde{r}_0 \{ r_0[!inc_0. \bar{u}_0 \parallel \bar{z}_0] \}. \tilde{r}_1 \{ r_1[!inc_1. \bar{u}_1 \parallel \bar{z}_1] \}. \bar{p}_1
\end{aligned}$$

Table 3: Encoding of MMs into \mathcal{E}_s^2

Note that the decidability result extends to \mathcal{E}_d^3 , as it is a subcalculus of \mathcal{E}_d^2 . Moreover, by virtue of Theorems 2.26 and 3.5, decidability of BA extends also to \mathcal{E}_s^2 and \mathcal{E}_s^3 . We have:

Corollary 7.25. *BA is decidable for \mathcal{E}_d^3 , \mathcal{E}_s^2 , and \mathcal{E}_s^3 .* \square

7.2. Undecidability of Eventual Adaptation. Here we show that EA is undecidable in \mathcal{E}_s^2 by relating it to termination in MMs; this result carries over to \mathcal{E}_s^1 , \mathcal{E}_d^1 , and \mathcal{E}_d^2 —see Corollary 7.29. This relationship is obtained by defining an encoding tailored to the features of the property. In contrast to the encoding given in Section 6, 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 MM a register is assumed to contain the value zero even if this is not the case). Nevertheless, we are able to define encodings that repeatedly simulate finite computations of the MM, and if the number of repeated simulations is infinite, then we have the guarantee that the number of erroneous steps is finite. Thus infinitely many of the performed simulations are correct. This way, the MM terminates iff its encoding has a non terminating computation. As during its execution the encoding continuously exhibits a barb on e , it then follows that EA is undecidable for \mathcal{E}_s^2 processes.

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

We now discuss the encoding defined in Table 3. We first comment on CONTROL, the process that manages the resources. It is composed of three processes in parallel. The first replicated process is able to produce an unbounded amount of processes \bar{f} and \bar{b} , which

represent the two kinds of resources described above. The second process starts and stops a resource production phase by performing \bar{a} and a , respectively. Then, it starts the MM simulation by emitting the program counter \bar{p}_1 . The third process is used at the end of the simulation to restore some of the consumed resources \bar{f} (that are transformed in \bar{g} , see below).

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

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

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

The full definition of the encoding is as follows.

Definition 7.26. Let N be a MM, with registers r_0, r_1 and instructions $(1 : I_1) \dots (n : I_n)$. Given the CONTROL process and the encodings in Table 3, the encoding of N in \mathcal{E}_s^2 (written $\llbracket N \rrbracket_2$) is defined as $\llbracket r_0 = 0 \rrbracket_2 \parallel \llbracket r_1 = 0 \rrbracket_2 \parallel \prod_{i=1}^n \llbracket (i : I_i) \rrbracket_2 \parallel \text{CONTROL}$.

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

Lemma 7.27. *Let N be a MM. N terminates iff $\llbracket N \rrbracket_2 \Downarrow_e^\omega$.*

Proof. See Appendix C.1, Page 58. □

Exploiting Lemma 7.27 and proceeding exactly as the proof of Theorem 6.3 for \mathcal{E}_s^1 , we can state the following.

Theorem 7.28. *EA is undecidable in \mathcal{E}_s^2 .* □

CONTROL	=	$!a.(\bar{f} \parallel \bar{b} \parallel \bar{a}) \parallel \bar{a}.a.(\bar{p}_1 \parallel e) \parallel !h.(g.\bar{f} \parallel \bar{h})$
REGISTER r_j		$\llbracket r_j = 0 \rrbracket_3 = r_j[Reg_j \parallel c_j[\mathbf{0}]$ with $Reg_j = !inc_j.\tilde{c}_j\{c_j[\bullet]\}.\overline{ack}.u_j.\tilde{c}_j\{c_j[\bullet]\}.\overline{ack}$
INSTRUCTIONS ($i : I_i$)		$\llbracket (i : \text{INC}(r_j)) \rrbracket_3 = !p_i.f.(\bar{g} \parallel b.\overline{inc}_j.ack.\bar{p}_{i+1})$ $\llbracket (i : \text{DECJ}(r_j, s)) \rrbracket_3 = !p_i.f.(\bar{g} \parallel (\bar{u}_j.ack.(\bar{b} \parallel \bar{p}_{i+1}) + \tilde{c}_j\{\bullet\}.\tilde{r}_j\{r_j[Reg_j \parallel c_j[\bullet]]\}.\bar{p}_s))$ $\llbracket (i : \text{HALT}) \rrbracket_3 = !p_i.\bar{h}.h.\tilde{c}_0\{\bullet\}.\tilde{r}_0\{r_0[Reg_0 \parallel c_0[\bullet]]\}.\tilde{c}_1\{\bullet\}.\tilde{r}_1\{r_1[Reg_1 \parallel c_1[\bullet]]\}.\bar{p}_1$

Table 4: Encoding of MMs into \mathcal{E}_d^3 .

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 2.18, since $\llbracket N \rrbracket_M$ is a process in \mathcal{E}_s^2 that does not contain any nested adaptable processes.

Corollary 7.29. *EA is undecidable in \mathcal{E}_s^1 , \mathcal{E}_d^1 , and \mathcal{E}_d^2 .* □

Note that the encoding $\llbracket \cdot \rrbracket_2$ uses processes that do not modify the topology of nested adaptable processes; update prefixes do not remove nor create adaptable processes: they simply remove the processes currently in the updated locations and replace them with the predefined initial content. One may wonder whether the ability to remove processes is necessary for the undecidability result: next we show that this is not the case.

8. (UN)DECIDABILITY RESULTS FOR \mathcal{E}^3

8.1. Undecidability of Eventual Adaptation in \mathcal{E}_d^3 . Here we prove that EA is undecidable for \mathcal{E}_d^3 processes. We obtain this result by means of a non-faithful encoding of MMs similar to the one presented before.

In that encoding, Definition 7.26, process deletion was used to restore the initial state inside the adaptable processes representing the registers. In the absence of process deletion, we use a more involved technique based on the possibility of moving processes to a different context: processes to be removed are guarded by an update prefix $\tilde{c}_j\{c_j[\bullet]\}$ that simply tests for the presence of a parallel adaptable process at c_j ; when a process must be deleted, it is “collected” inside c_j , thus disallowing the possibility to execute such an update prefix.

The encoding is as in Definition 7.26, with registers and instructions as in Table 4:

Definition 8.1. Let N be a MM, with registers r_0, r_1 and instructions $(1 : I_1) \dots (n : I_n)$. Given the CONTROL process and the encodings in Table 4, the encoding of N in \mathcal{E}_d^3 (written $\llbracket N \rrbracket_3$) is defined as $\llbracket r_0 = 0 \rrbracket_3 \parallel \llbracket r_1 = 0 \rrbracket_3 \parallel \prod_{i=1}^n \llbracket (i : I_i) \rrbracket_3 \parallel \text{CONTROL}$.

A register r_j that stores number m is encoded as an adaptable process at r_j that contains m copies of the unit process $u_j.\tilde{c}_j\{c_j[\bullet]\}.\overline{ack}$. It also contains process Reg_j , which creates further copies of the unit process when an increment instruction is invoked, as well as the collector c_j , which is used to store the processes to be removed.

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

This encoding has the same properties of the one discussed for the static case. In fact, in an infinite simulation the collected processes are never involved, otherwise the computation would block.

Lemma 8.2. *Let N be a MM. N terminates iff $\llbracket N \rrbracket_M \Downarrow_e^\omega$.*

Proof. See Appendix D.1, Page 63. □

Lemma 8.2 allows to conclude that EA is undecidable for processes in \mathcal{E}_d^3 . The proof of the following theorem proceeds as the proofs of Theorems 6.3 and 7.28.

Theorem 8.3. *EA is undecidable in \mathcal{E}_d^3 .* □

We can conclude that process deletion is not necessary for proving the undecidability of EA in \mathcal{E}_d^3 . Nevertheless, in the encoding in Table 4 we need to use the possibility to remove and create adaptable processes (namely, the collectors c_j are removed and then reproduced when the registers must be reset). One could therefore wonder whether EA is still undecidable if we remove from \mathcal{E}_d^3 the possibility to remove processes. Next we show that this is not the case.

8.2. Decidability of Eventual Adaptation in \mathcal{E}_s^3 . We prove the decidability of EA in \mathcal{E}_s^3 by resorting to Petri nets. Namely, we reduce the eventual adaptation problem for \mathcal{E}_s^3 to the infinite visit problem (cf. Definition 5.12).

Before formally defining the encoding of \mathcal{E}_s^3 processes into Petri nets, we give some intuitions. The idea is to use the markings of the Petri net to represent the active sequential subprocesses and the available adaptable processes. 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 token corresponding to the process executing the update and another token representing the adaptable process target of the update operations. In order to ensure that update actions take place between processes which are in parallel, we keep track of the adaptable processes in which a process is included: we do so by decorating its place with a list of outer adaptable processes. Intuitively, this list represents the “address” of a single adaptable process within the nested structure of adaptable processes.

We now present some auxiliary notations required by the definition. 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 . It is not restrictive to assume that all

the update actions on a given adaptable process can be executed: even if the static semantics decrees that update actions should satisfy conditions on the nesting structure of adaptable processes, Theorem 2.26 ensures the existence of an \mathcal{E}_d process with the same behavior for which such conditions are always true. 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 location names nestings, i.e. strings composed of names of nested locations, starting from the outermost adaptable process, occurring in one of the processes P, P_1, \dots, P_n . We use σ, θ to range over strings in $\mathcal{A}(P, M)$, and write σa for the string obtained from concatenating σ and a .

Definition 8.4. Let P and $M = \{P_1, \dots, P_n\}$ be \mathcal{E}_s^3 processes. Its associated Petri net is defined as the triple

$$\text{PN}(P, M) = (\text{Places}(P, M), \text{Trans}(P, M), \text{Init}(P))$$

where

- $\text{Places}(P, M) = \{\langle P, \sigma \rangle \mid P \in \mathcal{P}_{seq}(P, M), \sigma \in \mathcal{A}(P, M)\} \cup \mathcal{A}(P, M) \cup \{start, go\}$, with *start* and *go* being two distinguished auxiliary places.
- $\text{Trans}(P, M)$ contains all the instances of the transition schemata reported in Table 5 over the set of places $\text{Places}(P, M)$.
- $\text{Init}(P) = \text{dec}_\varepsilon(P) \uplus \{start\}$, with $\text{dec}_\sigma(P)$ defined inductively as follows:

$$\begin{aligned} \text{dec}_\sigma(a[P]) &= \text{dec}_{\sigma a}(P) \uplus \{\sigma a\} \\ \text{dec}_\sigma(P \parallel P') &= \text{dec}_\sigma(P) \uplus \text{dec}_\sigma(P') \\ \text{dec}_\sigma(P) &= \{\langle P, \sigma \rangle\} \quad \text{otherwise} \end{aligned}$$

where ε corresponds to the empty string and \uplus denotes multiset union.

We now describe the Petri net computation by giving intuitions on the transitions presented in Table 5. The initial marking includes one token in the place *start* plus the tokens corresponding to the active sequential subprocesses of P . The token in *start* allows to generate an arbitrary amount of copies of the processes $P_1, \dots, P_n \in M$ (Transition (1)). 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* (Transition (2)). 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. Transitions (3)–(5) cover the different cases in which an input/output synchronization can arise (namely, interaction between two guarded processes, between a replicated processes and a guarded process, and between two replicated processes), while Transitions (6)–(9) cover the cases in which a synchronization corresponds to an update action. In the latter kind of transitions, we need to check the availability of a target adaptable process, but this adaptable process should not enclose the updating process (as in, e.g., $a[\tilde{a}\{U\} \parallel P]$). More precisely, suppose there is a process Q executing an update action on name a , and let σ be the string of the names of the adaptable processes enclosing Q . The availability of a target adaptable process can be checked by verifying the presence of a token in a place θa which is not a prefix of σ (see Transitions (6) and (8)). If θa is a prefix of σ , then the adaptable process at θa could enclose Q . In such a case, it is sufficient to check that the place θa contains at least two tokens, thus indicating the

- (1) $\{start\} \Rightarrow \{start\} \uplus \text{dec}_\varepsilon(P_i)$ with $P_i \in M$
- (2) $\{start\} \Rightarrow \{go\}$
- (3) $\{go, \langle \sum_{i \in I} \pi_i. A_i, \sigma \rangle, \langle \sum_{j \in J} \rho_j. B_j, \theta \rangle\} \Rightarrow \{go\} \uplus \text{dec}_\sigma(A_l) \uplus \text{dec}_\theta(B_m)$
if $\pi_l = a$ and $\rho_m = \bar{a}$ (for $l \in I, m \in J$)
- (4) $\{go, \langle !\pi. A, \sigma \rangle, \langle \sum_{j \in J} \rho_j. B_j, \theta \rangle\} \Rightarrow$
 $\{go, \langle !\pi. A, \sigma \rangle\} \uplus \text{dec}_\sigma(A) \uplus \text{dec}_\theta(B_m)$
if $\pi = a$ (resp. \bar{a}) and $\rho_m = \bar{a}$ (resp. a) (for $m \in J$)
- (5) $\{go, \langle !\pi. A, \sigma \rangle, \langle !\rho. B, \theta \rangle\} \Rightarrow$
 $\{go, \langle !\pi. A, \sigma \rangle, \langle !\rho. B, \theta \rangle\} \uplus \text{dec}_\sigma(A) \uplus \text{dec}_\theta(B_m)$
if $\pi = a$ (resp. \bar{a}) and $\rho = \bar{a}$ (resp. a)
- (6) $\{go, \langle \sum_{i \in I} \pi_i. A_i, \sigma \rangle, \theta a\} \Rightarrow$
 $\{go\} \uplus \text{dec}_\sigma(A_l) \uplus \text{dec}_\theta(A) \uplus \text{dec}_{\theta a}(U) \uplus \{\theta a\}$
if θa is not a prefix of σ , $\pi_l = \bar{a}\{a[U] \parallel A\}$ (for $l \in I$)
- (7) $\{go, \langle \sum_{i \in I} \pi_i. A_i, \sigma \rangle, \theta a, \theta a\} \Rightarrow$
 $\{go\} \uplus \text{dec}_\sigma(A_l) \uplus \text{dec}_\theta(A) \uplus \text{dec}_{\theta a}(U) \uplus \{\theta a, \theta a\}$
if θa is a prefix of σ , $\pi_l = \bar{a}\{a[U] \parallel A\}$ (for $l \in I$)
- (8) $\{go, \langle !\pi. A', \sigma \rangle, \theta a\} \Rightarrow$
 $\{go, \langle !\pi. A', \sigma \rangle\} \uplus \text{dec}_\sigma(A') \uplus \text{dec}_\theta(A) \uplus \text{dec}_{\theta a}(U) \uplus \{\theta a\}$
if θa is not a prefix of σ , $\pi = \bar{a}\{a[U] \parallel A\}$
- (9) $\{go, \langle !\pi. A', \sigma \rangle, \theta a, \theta a\} \Rightarrow$
 $\{go, \langle !\pi. A', \sigma \rangle\} \uplus \text{dec}_\sigma(A') \uplus \text{dec}_\theta(A) \uplus \text{dec}_{\theta a}(U) \uplus \{\theta a, \theta a\}$
if θa is a prefix of σ , $\pi = \bar{a}\{a[U] \parallel A\}$

Table 5: Transition schemata for the Petri net representation of \mathcal{E}_s^3 processes in Definition 8.4.

existence of a different adaptable process with the same path but that does not enclose Q (see Transitions (7) and (9)).

We now state the correspondence between processes and their associated Petri net.

Lemma 8.5. *Let P be a process of \mathcal{E}_s^3 , and M be the set $\{P_1, \dots, P_n\}$ and $(\text{Places}(P, M), \text{Trans}(P, M), \text{Init}(P))$ be their associated Petri net, as in Definition 8.4. Then, given a marking m , we have $\text{Init}(P) \rightarrow^* \{start\} \uplus m \rightarrow \{go\} \uplus m$ iff $m = \text{dec}_\varepsilon(R)$, for some $R \in \text{CS}_P^M$.*

Proof. Follows by construction of the Petri net. □

Lemma 8.6. *Let P and $(\text{Places}(P, \emptyset), \text{Trans}(P, \emptyset), \text{Init}(P))$ be an \mathcal{E}_s^3 process and its associated Petri net, as in Definition 8.4. Then we have:*

$$P \longrightarrow P' \text{ iff } \text{dec}_\varepsilon(P) \uplus \{go\} \rightarrow \text{dec}_\varepsilon(P') \uplus \{go\}.$$

Proof. See Appendix D.2, Page 68. □

The decidability of EA for \mathcal{E}_s^3 follows from the decidability of the existence of a suffix of an infinite computation composed of markings with at least one token in some given places.

Theorem 8.7. *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 $S = \text{Sub}_{\text{St}}(P) \cup \text{Sub}_{\text{St}}(P_1) \cup \dots \cup \text{Sub}_{\text{St}}(P_n)$, and let $P' = \llbracket P \rrbracket_S^d$ and $M' = \{\llbracket P_1 \rrbracket_S^d, \dots, \llbracket P_n \rrbracket_S^d\}$. Let α be a barb. We have that P and M satisfies EA for the barb α iff the Petri net*

$$(\text{Places}(P', M'), \text{Trans}(P', M'), \text{Init}(P'))$$

has an infinite computation with a suffix composed of markings with one token in go and with at least one token in one of the places $\langle \sum_{i \in I} \pi_i \cdot A_i, \theta \rangle$, with $\pi_l = \alpha$ for some $l \in I$, or $\langle !\alpha \cdot A, \theta \rangle$.

Proof. Suppose that P and M satisfies EA for the barb α then there exists a process $R \in \mathcal{CS}_P^M$ such that $R \Downarrow_\alpha^\omega$. Following from Lemma 8.5 there exists an initial computation of the Petri net that reaches the marking $\text{dec}_\varepsilon(R) \uplus \{go\}$. Then following from Lemma 8.6 there exists an infinite computation with a suffix composed of markings with at least one token in one of the places $\langle \sum_{i \in I} \pi_i \cdot A_i, \theta \rangle$, with $\pi_l = \alpha$ for some $l \in I$, or $\langle !\alpha \cdot A, \theta \rangle$. Notice that in all of these markings, the place *go* contains one token.

Similarly if there exists an infinite computation with a suffix composed of markings with one token in *go* and at least one token in one of the places $\langle \sum_{i \in I} \pi_i \cdot A_i, \theta \rangle$, with $\pi_l = \alpha$ for some $l \in I$, or $\langle !\alpha \cdot A, \theta \rangle$ then for Lemma 8.5 and Lemma 8.6 we know that there exists a process $R \in \mathcal{CS}_P^M$ such that $R \Downarrow_\alpha^\omega$. □

The check of the existence of an infinite computation with a suffix composed of markings with one token in *go* and with at least one token in some given places corresponds to the infinite visit problem (Definition 5.12). Thus since this problem is decidable (Theorem 5.13) it follows that EA is decidable in \mathcal{E}_s^3 .

9. RELATED WORK AND DISCUSSION

We now comment on the origin and motivations for the constructs of \mathcal{E} , review some related works, describe a modeling technique derived from BA and EA, and discuss variants of the adaptation problems considered here.

9.1. On the Constructs for Evolvability. The origins of the \mathcal{E} calculus can be traced back to our own previous work on expressiveness and decidability results for core *higher-order process calculi* (see, e.g., [40, 27, 52]). Below, we overview these previous works, and discuss the motivations that led us from higher-order communication to adaptable processes.

Higher-order (or *process-passing*) concurrency is often presented as an alternative paradigm to the first-order (or *name-passing*) concurrency of the π -calculus for the description of mobile systems. As in the λ -calculus, higher-order process calculi involve *term instantiation*: a computational step results in the instantiation of a variable with a term, which is copied as many times as there are occurrences of the variable. The basic operators of these calculi are usually those of CCS: parallel composition, input and output prefix, and restriction. Replication and recursion can be encoded. Proposals of higher-order process calculi include the higher-order π -calculus [57], Homer [33], and Kell [59].

With the purpose of investigating expressiveness and decidability issues in the higher-order paradigm, a *core* higher-order process calculus, called HOCORE, was introduced [40]. HOCORE is *minimal*, in that only the operators strictly necessary to obtain higher-order communications are retained. Most notably, HOCORE has no restriction operator. Thus all names are global, and dynamic creation of new names is impossible. The grammar of HOCORE processes is:

$$P ::= a(x).P \mid \bar{a}\langle P \rangle \mid P \parallel P \mid x \mid \mathbf{0}$$

An input process $a(x).P$ can receive on name a a process to be substituted in the place of x in the body P ; an output message $\bar{a}\langle P \rangle$ sends the output object P on a ; parallel composition allows processes to interact. As in CCS, in HOCORE processes evolve from the interaction of complementary actions; this way, e.g.,

$$\bar{a}\langle P \rangle \parallel a(x).Q \rightarrow Q\{P/x\}$$

is a sample reduction. (See [40, 52] for complete accounts on the theory of HOCORE.)

While considerably expressive, HOCORE is far from a specification language for settings involving (forms of) higher-order communication. For instance, it lacks primitives for describing the *localities* into which distributed systems are typically abstracted. Similarly, HOCORE also lacks constructs for expressing forms of evolvability and/or dynamic reconfiguration. In order to deal with these aspects, higher-order process calculi such as Homer and Kell provide mechanisms that allow to *suspend* running processes. Such mechanisms rely on a form of named localities for processes, so called *suspension (or passivation) units*. Inside a suspension unit, a process may execute and freely interact with their environment, but it may also be stopped at any time. More precisely, let us consider the extension of HOCORE with process suspension. Let $a[P]$ denote the process P inside the suspension unit a . Assuming an LTS with actions of the form $P \xrightarrow{\alpha} P'$, the semantics of suspension is formalized by the following two rules:

$$\begin{array}{c} \text{(TRANS)} \\ \frac{P \xrightarrow{\alpha} P'}{a[P] \xrightarrow{\alpha} a[P']} \\ \text{(SUSP)} \\ a[P] \xrightarrow{a\langle P \rangle} \mathbf{0} \end{array}$$

where $a\langle P \rangle$ corresponds to the output action in the LTS of HOCORE (see [40]). While rule (TRANS) defines the transparency of suspension units, rule (SUSP) implements suspension: the current state of a located process is “frozen” as an output action, in which it can no longer evolve. Hence, in this semantics input prefixes may interact not only with output actions but also with suspension units; in fact, suspension of a running process is assimilated to regular process communication. As a simple example, consider the following process S :

$$S \triangleq a[P] \parallel a\langle Q \rangle \parallel a(x).R$$

It is easy to see that two possible evolutions for S are $S' \triangleq a[P] \parallel R\{Q/x\}$ and $S'' \triangleq a\langle Q \rangle \parallel R\{P/x\}$. Other evolutions, related to the behavior of P , are also possible. While the semantics for suspension just described allows for a straightforward definition, we observe two potential drawbacks: First, the *dual rôle* of input prefixes induces a form of non determinism that one may regard as unnatural. Consider $a(x).R$ in S above: in the first evolution, it acts as a communication endpoint, whereas in the second it acts as a suspension realizer. Second, such a semantics is only possible for calculi which already feature process passing in

communications. That is, the possibility of suspending/reconfiguring processes at runtime is somehow tied to the calculus being higher-order.

With these drawbacks in mind, in the definition of \mathcal{E} we have opted for a different approach: we do not assume higher-order communication, and rely instead on a restricted form of term instantiation for defining update actions. That is, we exploit a very particular form of higher-order interaction to define process suspension for calculi which may well be first-order. Here, in order to focus on the novel features of adaptable processes, we have considered a variant of CCS. Moreover, as we elaborate below, update in \mathcal{E} can be seen as *objective* rather than as *subjective*: an adaptable process may evolve independently until it is updated by a prefix in its surrounding context. Furthermore, by featuring update prefixes $\tilde{a}\{U\}$ —a dedicated construct for representing the runtime reconfiguration of located processes— \mathcal{E} enforces a separation of concerns, which allows to distinguish interaction/communication from actions of dynamic reconfiguration. We believe these are all reasonable design choices, which allow us to focus on the fundamental aspects of evolvability for concurrent processes. In fact, they could provide a basis for developing new formalisms with adaptation concerns, such as, e.g., an adaptable extension of the π -calculus or a variant of \mathcal{E} with the nested locations of Homer.

9.2. Related Work. We have already discussed related works from the point of view of proof techniques in the Introduction. Below, we comment on some languages/formalisms related to \mathcal{E} .

Loosely related to \mathcal{E} are process calculi for fault tolerance (see, e.g., [9, 50, 56, 31]). These are variants of the π -calculus tailored for describing algorithms on distributed systems; hence, they include explicit notions of sites/locations, network, and failures. A series of extensions to the asynchronous π -calculus so as to model distributed algorithms is proposed in [9]. One such extensions, aimed at representing process failure, is a higher-order operation that defines *savepoints*: process $\text{save}(P).Q$ defines the savepoint P for the current location; if such a location crashes, then it will be restarted with state P . A value-passing calculus to represent and formalize algorithms of distributed consensus is introduced in [50]; it includes a *failure detector* construct $\mathcal{S}(k).P$ which executes P if locality k is *suspected* to have failed. The *partial failure* languages of [56, 31] feature similar constructs; such works aim at developing bisimulation-based proof techniques for distributed algorithms. Crucially, in the constructs for failure proposed in the above works (savepoints, failure detectors), the *post-failure* behavior is defined statically, and does not depend on some runtime behavior. Hence, as discussed in Section 4, these constructs are easily representable in \mathcal{E} . None of the above works addresses adaptation properties related to failures nor studies decidability/expressiveness issues for the languages they work on.

\mathcal{E} relies on transparent localities as a way of structuring communicating processes for update purposes. The hierarchies induced by transparent localities are rather weak; this is in contrast to process hierarchies in calculi such as Ambients [20] or Seal [22]. The ambients in the Ambient calculus represent *administrative domains* and act as containers of concurrent processes. Ambients may be dissolved using the **open** primitive; transparent localities can only be eliminated in \mathcal{E}_d by an explicit synchronization with a suitable update prefix. Movement across the ambient hierarchy is achieved via the **in/out** primitives; it is said to be *subjective* rather than *objective*, as ambients move themselves and are not moved by their context. Adapting this distinction to our setting, it is fair to say that \mathcal{E} features a form of *objective update*, as an adaptable process does not contain information on its future

update actions: it evolves autonomously until it is updated by a suitable update prefix in its context. A fundamental difference of Ambients with respect to higher-order process calculi is that movement is *linear*: it is not possible to duplicate an ambient through its movement. This aspect is one of the main differences between Ambients and Seal, in which process duplication is possible. A main design guideline in Seal is security; in fact, it is intended as a calculus of sealed objects. Within the hierarchy of seals, only parent/child communication is allowed, thus establishing a noticeable difference with respect to the hierarchies of transparent localities in \mathcal{E} .

A suspension-like construct is at the heart of MECo [58], a model for evolvable components. It is defined as a process calculus in which components feature a hierarchical structure, rich input/output interfaces, as well as channel communication. Evolvability in MECo is enforced by a suspension-like construct that stops a component and *extracts* its “skeleton”. Because of its focus on components, adaptation in MECo is mostly concerned about consistent changes in input/output interfaces; in our case, adaptation is defined in terms of some distinguished observables of the system, thus constituting a rather general way of characterizing correctness. COMP [42] is another process calculus for component models. It is intended to be the component model for the ABS modeling language; as such, it aims at providing a unified definition of evolvability for objects, components, and runtime modifications of programs. In COMP, constructs for evolvability are based on the movement primitives of the Ambient calculus rather than on suspension-based constructs, as in \mathcal{E} and MECo. Hence, the semantics of reconfiguration in COMP is quite different from that in \mathcal{E} , which prevents more detailed comparisons.

In a broader setting, related to \mathcal{E} are formalisms for the specification of (dynamic) software architectures. While some of them are based on process calculi, none of them relies on suspension-like constructs to formalize evolution/adaptation. Below we review some of them; we refer the reader to [13, 12, 36, 25] for more extensive reviews.

One of the earliest proposals for formal grounds to dynamic architectures is [4], where a formal system for architectural components which relies on (a fragment of) Hoare’s CSP is introduced. The approach in [4], however, does not consider dynamic architectures. Darwin [43] is an Architecture Description Language (ADL) for distributed systems; it aims at 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 π -calculus model for handling such bindings. Darwin features a mechanism of dynamic instantiation which allows arbitrary changes in the system architecture. Associated techniques for analyzing dynamic change in Darwin have been proposed in [38, 37]. In comparison to \mathcal{E} , the kind of changes possible in Darwin concern the system topology rather than the “state” of the interconnected entities, as in our case. π -ADL [51] is an ADL for dynamic and mobile architectures. Formally defined as a typed variant of the higher-order π -calculus, π -ADL focuses on a combination of structural and behavioral perspectives: while the former describes the architecture in terms of components, connectors, and their configurations, the latter describes it in terms of actions and behaviors. π -ADL is at the heart of ArchWare-ADL [48, 6], a layered ADL for active architectures. ArchWare-ADL complements π -ADL with a style layer that allows the specification of components and connectors, and with an analysis layer which enables the specification of constraints on the styles. In contrast to \mathcal{E} , π -ADL does not offer any construct for supporting system evolvability. In fact, while ArchWare-ADL supports forms of evolution (via mechanisms for stopping running programs and decomposing them into

its main constituents) these are not provided by the formal framework of π -ADL but by technologies on top of it [49]. Pilar [25, 24, 23] is an algebraic, reflective ADL. Reflection in Pilar (defined as the capability of a system to reason and act upon itself) relies on the notion of reification which, roughly speaking, relates between entities in different levels of a specification for defining introspection capabilities. The semantic foundation of Pilar is a first-order, polymorphic typed variant of the π -calculus; no constructs for dynamic update such as those in \mathcal{E} are included in Pilar.

We conclude this review by mentioning other works on formal approaches to dynamic update [32, 11, 62, 61, 19]. They all rely on different approaches from ours.

In [32], an investigation on *on-line software version change* is presented. 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 [32], however, limits to restricted instances of imperative languages. Moreover, the notion of validity says very little about correctness and adaptation. A formal model for adaptation in asynchronous programs in distributed systems is introduced in [11]. Programs are expressed as guarded commands, and represented as automata; adaptation can be then described as transforming one automaton to another automaton. The focus of [11] is the verification of the behavior of system during adaptation, considering the interaction between the new program and the old one. The use of graph rewriting/category theory to formalize software architecture reconfiguration has been studied in [62]. In [10], the *update calculus*, a typed λ -calculus with a primitive operation for updating modules, is proposed. A development of this idea was carried out in [61], where a calculus for *dynamic update* in typed, imperative languages is proposed. There, the focus is on *type-safe* updates—intuitively, the consistent update of type τ with some new type τ' . 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, this is a conceptual difference between *update* (as in works such as [61]) and *dynamic adaptation*, as we have considered it here. A framework for structural component reconfiguration with behavioral adaptation considerations is introduced in [19], where component architectures are given by *nets* of interacting components represented by LTSs. Notice that the concept of “behavioral adaptation” in [19] is different from our notion of adaptation. The former refers to the changes required in component interfaces so as to achieve effective compositions. Instead, our notion of adaptation concerns a higher abstraction level, as we address the evolution of running processes through built-it adaptation mechanisms.

9.3. Applying the Verification Problems. In the examples given in Section 4, both BA and EA were used to check whether a process can reach a state without errors. In general, however, one may be interested in both solving errors and preserving the correct behavior of the system. In particular, one could be interested in checking whether certain states of the systems are still reachable after correcting an error. We now discuss a modeling technique which allows us to express such a property as an *instance* of the BA and EA problems. The key idea is to extend the given system with parallel behaviors, defined in accordance with the observable events associated to errors and adaptation in the system.

We illustrate the technique by considering the particular case of the EA problem; the use of the BA problem is analogous. We consider a system abstracted as a process P , with the following observable actions:

- (i) \bar{a} – which signals that the system has reached the state we are interested in;
- (ii) \bar{e}_s – which is emitted as soon as the system enters in an error phase;
- (iii) \bar{e}_f – which signals that the error has been corrected.

We define a process P^* as an extension of P with parallel behaviors which, roughly speaking, “complement” the above actions. Intuitively, by checking whether for such a P^* and a barb e property EA is satisfied, then we will be able to guarantee that after having corrected an error in P the distinguished state signaled by a is still reachable. Process P^* is defined as follows:

$$P^* \triangleq P \parallel \bar{c} \parallel !c. a. \bar{c} \parallel e_s. (e + c. (e + e_f. (e + a. \bar{c})))$$

Above, we assume that c and e do not occur in P . In P^* , we can identify four parts: the process P which is kept unchanged; process $C \triangleq !c. a. \bar{c}$, which is used to check that the state signaled by a has been reached; process \bar{c} , which is used to spawn the first copy of a ; finally, we have process $R \triangleq e_s. (e + c. (e + e_f. (e + a. \bar{c})))$.

We explain the behavior of P^* . When P enters in an error phase (as signaled by \bar{e}_s), a synchronization takes place and R reaches the process $R_1 \triangleq e + c. (e + e_f. (e + a. \bar{c}))$. This is the first point in which barb e becomes available; the only way to satisfy the EA property is to make e disappear. Then, process R_1 synchronizes with \bar{c} , as this is the only possible evolution, thus obtaining $R_2 \triangleq e + e_f. (e + a. \bar{c})$. Notice that at this point the process P cannot evolve by consuming \bar{a} , as the occurrence of a in the process C is guarded by a prefix c , and no copy of \bar{c} is available. In R_2 , barb e is available again and the process can evolve only when P corrects the error (i.e., when an action \bar{e}_f is observed). As soon as the error phase is completed, P can synchronize on e_f , thus reaching the process $R_3 \triangleq e + a. \bar{c}$. In R_3 , barb e will finally disappear as soon as the system P performs again action \bar{a} .

Clearly, the specific definition of P^* will depend on the features of the given P . Still, the above example is already useful to illustrate how the two verification problems introduced in the paper can provide a suitable basis for reasoning about non-trivial properties of evolvable systems which may depend on the observables of the system under consideration.

9.4. Variants of the Correctness Properties. In this presentation, we have studied correctness of adaptable processes from a rather general perspective; in fact, the definition of BA and EA are based only on minimal observations on the behavior of the system. This allows us to reason about the interplay between correctness and adaptation for diverse classes of concurrent systems. More informative properties (relating correctness and the structure of the system, for instance) can be devised according to the nature of some particular setting.

In this context, it is worth noticing that the technical machinery required for our (un)decidability results can be adapted to handle a slightly different definition of the adaptation problems stated in Definition 3.3. More precisely, such problems can be relaxed so as to consider *non consecutive* error occurrences, rather than consecutive ones. For this purpose, we modify the notion of barbs (cf. Definition 3.1) by admitting an arbitrary number of reductions between the actual error barbs:

Definition 9.1 (Barbs - Alternative Definition). Let P be an \mathcal{E} process, and let α be an action in $\{a, \bar{a} \mid a \in \mathcal{N}\}$.

- Given $k > 0$, we write $P \Downarrow_\alpha^k$ iff there exist Q_1, \dots, Q_k such that $P \longrightarrow^* Q_1 \longrightarrow^* \dots \longrightarrow^* Q_k$ with $Q_i \Downarrow_\alpha$, for every $i \in \{1, \dots, k\}$.

- We write $P \Downarrow_{\alpha}^{\omega}$ iff there exists an infinite computation $P \longrightarrow^* Q_1 \longrightarrow^* Q_2 \longrightarrow^* \dots$ with $Q_i \Downarrow_{\alpha}$ for every $i \in \mathbb{N}$.

Furthermore, we use \Downarrow_{α}^k and $\Downarrow_{\alpha}^{\omega}$ to denote the negation of \Downarrow_{α}^k and $\Downarrow_{\alpha}^{\omega}$, with the expected meaning.

Variants of EA and BA can be then restated considering the new definition above. Thus, given a set of clusters \mathcal{CS}_P^M and a barb e then the BA problems consists in checking whether all computations of processes in \mathcal{CS}_P^M have at most k states exhibiting e . Similarly, EA consists in checking whether there is no computation in which e is observable in infinitely many distinct states. Given these alternative definitions of EA and BA, (un)decidability results can be easily derived from the ones presented here. In fact, Table 1 remains unchanged under the alternative adaptation problems, and straightforwardly all undecidability results hold. As for the decidability results, we should adapt the WSTS construction and the Petri net simulation. In particular, to show decidability of the alternative definition of BA for \mathcal{E}_d^2 processes, it is enough to slightly change the definition of $\text{fb}_{\alpha}(S)$ (Definition 7.22) and substituting the occurrences of pb_S with Pred_S^* whose effectiveness is guaranteed by Theorem 7.20. Concerning the decidability of EA for \mathcal{E}_s^3 , the Petri net semantics presented in Section 8.2 reduces this alternative version of the property to the *repeated coverability* problem. This problem is known to be decidable for Petri nets, see e.g. [28].

10. CONCLUDING REMARKS

We have proposed the concept of *adaptable process* as a way of describing complex evolvability patterns in models of concurrent systems. We have introduced \mathcal{E} , a process calculus of adaptable processes, in which located processes can be updated and relocated at runtime. In our view, this ability improves the kind of reconfiguration that can be expressed in existing (higher-order) process calculi. In the design of \mathcal{E} , we aimed at isolating a small basis for representing reconfiguration of interacting processes: we extended CCS without restriction and relabeling (a non Turing complete model), with transparent localities (arguably the simplest conceivable way of structuring processes into hierarchies) and with update prefixes. The interaction of adaptable processes with update prefixes constitutes a restricted form of higher-order communication that realizes process reconfiguration.

In order to formalize the correctness of evolvable processes, we proposed the *bounded* and *eventual* adaptation problems. We studied the (un)decidability of these problems in several variants of \mathcal{E} , obtained by different evolvability patterns as well as static and dynamic topologies of adaptable processes. Our results shed light on the expressive power of \mathcal{E} as well as on the nature of verification for concurrent processes that may evolve at runtime.

There are a number of practical and technical issues associated to adaptable processes that would be worth investigating in future work.

- We would like to understand how to accommodate (a form of) restriction into \mathcal{E} while preserving our decidability results. This is a delicate issue, as typically adding restriction causes decidability results to break (see, e.g. [18]). Recently, higher-order calculi with *name creation* (which replaces usual name restriction) have been put forward [54]; a creationist treatment of names is claimed to be closer to distributed implementations and is shown to have benefits in the development of associated behavioral theories. Exploring variants of \mathcal{E} with a name creation construct could be therefore insightful.

- In the definition of eventual adaptation we require the absence of computations with infinitely many successive error states. It would be interesting to investigate the impact of fairness on our results, in particular the decidability result for \mathcal{E}_s^3 . In fact, the detected computation with infinitely many successive error state could be unfair, in the sense that a parallel process or thread able to solve the problem is available but never scheduled. In several concurrent models, properties like the existence of an infinite computation turn from decidable to undecidable when restricting to fair computations (see [21] for Petri nets or [64] for CGF, a stochastic CCS-like process calculus for the modeling of chemical systems). As a future work we intend to check whether a similar result applies also to our case.
- It would be interesting to study the behavioral theory of \mathcal{E} processes; recent works on behavioral equivalences for higher-order process calculi with passivation (e.g. [41, 53, 54]) could provide a reasonable starting point. Also, it would be important to devise (logic-based) techniques for enhancing the verification of adaptable processes; in recent work [16], we have studied an alternative for tackling this challenging issue.
- From a practical standpoint, it would be interesting to develop extensions or variants of \mathcal{E} tailored to concrete application settings, to determine how the adaptation problems proposed here fit in such scenarios, and to study how to transfer our decidability results to such richer languages. For instance, it would be interesting to see how our adaptation problems fit in the context of higher-order calculi such as Kell and Homer, which feature rich constructs for structuring processes (kells in Kell, nested locations in Homer).
- Finally, it would be useful to address the complexity of BA and EA. As far as EA is concerned, we have presented its (polynomial) reduction to the Petri net place boundedness problem, for which an EXPSPACE decision procedure exists [55]. Concerning BA, our proof of decidability does not give a precise indication about the complexity, as only the termination of the procedure is guaranteed by the well quasi-ordering we have defined. We plan to investigate the complexity of the problem by comparing BA to the coverability problem for reset Petri nets which is known to be non primitive recursive (see, e.g., [60]). In fact, the possibility of atomically erasing the current contents of an adaptable process is reminiscent of the ability that reset transitions have for removing all the tokens in some given place. Hence, a plausible direction of future work is to investigate suitable abstractions that could help alleviating the state explosion problem.

ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their useful remarks. This work was partially supported by the French projects ANR-2010-SEGI-013 - AEOLUS, ANR-11-INSE-0007 - REVER, by the EU integrated project HATS, and by the Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) through the Carnegie Mellon Portugal Program, grant INTERFACES NGN-44 / 2009.

REFERENCES

- [1] Open Source Erlang. <http://www.erlang.org/>, 2011.
- [2] 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.
- [3] 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.

- [4] R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Trans. Softw. Eng. Methodol.*, 6(3):213–249, 1997.
- [5] Amazon Web Services. Autoscaling. <http://aws.amazon.com/autoscaling/>, 2011.
- [6] ArchWare Consortium. The ArchWare ADL: Definition of the Abstract Syntax and Formal Semantics. Project Deliverable. Available at <http://www-valoria.univ-ubs.fr/ArchLog/ArchWare-IST/ArchWareDocs/D1.1b%20V1.pdf>, 2002.
- [7] J. Armstrong. *Making reliable distributed systems in the presence of software errors*. PhD thesis, SICS, 2003.
- [8] J. Baeten and J. Bergstra. Mode transfer in process algebra. Technical Report Report 00/01, Eindhoven University of Technology, 2000.
- [9] M. Berger and K. Honda. The two-phase commitment protocol in an extended pi-calculus. *ENTCS*, 39(1), 2000.
- [10] G. Bierman, M. Hicks, P. Sewell, and G. Stoye. Formalizing dynamic software updating. In *Proc. of the International Workshop on Unanticipated Software Evolution (USE'03)*, Apr 2003.
- [11] K. N. Biyani and S. S. Kulkarni. Assurance of dynamic adaptation in distributed systems. *Journal of Parallel and Distributed Computing*, 68(8):1097 – 1112, 2008.
- [12] J. S. Bradbury. Organizing definitions and formalisms for dynamic software architectures. Technical Report 2004-477, School of Computing, Queens University, 2004.
- [13] J. S. Bradbury, J. R. Cordy, J. Dingel, and M. Wermelinger. A survey of self-management in dynamic software architecture specifications. In *Proc. of WOSS*, pages 28–33. ACM, 2004.
- [14] M. Bravetti, C. Di Giusto, J. A. Pérez, and G. Zavattaro. Adaptable Processes (Extended Abstract). In *Proc. of FMOODS-FORTE'11*, volume 6722 of *LNCS*, pages 90–105. Springer, 2011.
- [15] M. Bravetti, C. Di Giusto, J. A. Pérez, and G. Zavattaro. Steps on the road to component evolvability. In *Post-proc. of FACS'10*, volume 6921 of *LNCS*, pages 295–299. Springer, 2011.
- [16] M. Bravetti, C. D. Giusto, J. A. Pérez, and G. Zavattaro. Towards the verification of adaptable processes. In T. Margaria and B. Steffen, editors, *ISoLA (1)*, volume 7609 of *Lecture Notes in Computer Science*, pages 269–283. Springer, 2012.
- [17] M. Bravetti and G. Zavattaro. On the expressive power of process interruption and compensation. *Math. Struct. in Comp. Sci.*, 19(3):565–599, 2009.
- [18] 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.
- [19] A. Cansado, C. Canal, G. Salaün, and J. Cubo. A formal framework for structural reconfiguration of components under behavioural adaptation. *ENTCS*, 263:95–110, 2010.
- [20] L. Cardelli and A. D. Gordon. Mobile ambients. *Theor. Comput. Sci.*, 240(1):177–213, 2000.
- [21] H. Carstensen. Decidability Questions for Fairness in Petri Nets. In *Proc. of 4th Annual Symposium on Theoretical Aspects of Computer Science (STACS)*, volume 247 of *LNCS*, pages 396–407, 1987.
- [22] G. Castagna, J. Vitek, and F. Zappa Nardelli. The seal calculus. *Inf. Comput.*, 201(1):1–54, 2005.
- [23] C. E. Cuesta, P. de la Fuente, and M. Barrio-Solórzano. Dynamic coordination architecture through the use of reflection. In *Proc. of the 2001 ACM symposium on Applied computing, SAC '01*, pages 134–140, New York, NY, USA, 2001. ACM.
- [24] C. E. Cuesta, P. de la Fuente, M. Barrio-Solórzano, and M. E. B. Gutiérrez. Coordination in a reflective architecture description language. In *Proc. of COORDINATION'02*, volume 2315 of *LNCS*, pages 141–148. Springer, 2002.
- [25] C. E. Cuesta, P. de la Fuente, M. Barrio-Solórzano, and M. E. B. Gutiérrez. An "abstract process" approach to algebraic dynamic architecture description. *J. Log. Algebr. Program.*, 63(2):177–214, 2005.
- [26] G. Delzanno, A. Sangnier, and G. Zavattaro. Parameterized verification of ad hoc networks. In *Proc. of CONCUR*, volume 6269 of *LNCS*, pages 313–327. Springer, 2010.
- [27] C. Di Giusto, J. A. Pérez, and G. Zavattaro. On the expressiveness of forwarding in higher-order communication. In *ICTAC*, volume 5684 of *LNCS*, pages 155–169. Springer, 2009.
- [28] J. Esparza. Some applications of petri nets to the analysis of parameterised systems, 2003. Talk at WISP'03.
- [29] J. Esparza, A. Finkel, and R. Mayr. On the verification of broadcast protocols. In *Proc. of LICS'99*, pages 352–359, 1999.
- [30] A. Finkel and P. Schnoebelen. Well-structured transition systems everywhere! *Theor. Comput. Sci.*, 256(1-2):63–92, 2001.

- [31] A. Francalanza and M. Hennessy. A fault tolerance bisimulation proof for consensus (extended abstract). In *Proc. of ESOP*, volume 4421 of *LNCS*, pages 395–410. Springer, 2007.
- [32] 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.
- [33] T. Hildebrandt, J. C. Godskesen, and M. Bundgaard. Bisimulation congruences for homer — a calculus of higher order mobile embedded resources. Technical Report TR-2004-52, IT University of Copenhagen, 2004.
- [34] P. Hnetynka and F. Plasil. Dynamic reconfiguration and access to services in hierarchical component models. In *Proc. of CBSE'06*, volume 4063 of *LNCS*, pages 352–359. Springer, 2006.
- [35] R. M. Karp and R. E. Miller. Parallel program schemata. *J. Comput. Syst. Sci.*, 3(2):147–195, 1969.
- [36] S. Kell. A survey of practical software adaptation techniques. *Journal of Universal Computer Science*, 14(13):2110–2157, 2008.
- [37] J. Kramer and J. Magee. The evolving philosophers problem: dynamic change management. *Software Engineering, IEEE Transactions on*, 16(11):1293–1306, nov 1990.
- [38] J. Kramer and J. Magee. Analysing dynamic change in distributed software architectures. *IEE Proc. - Software*, 145(5):146–154, 1998.
- [39] J. B. Kruskal. Well-quasi-ordering, the tree theorem, and vazsonyi’s conjecture. *Transactions of the American Mathematical Society*, 95(2):210–225, 1960.
- [40] I. Lanese, J. A. Pérez, D. Sangiorgi, and A. Schmitt. On the expressiveness and decidability of higher-order process calculi. *Inf. Comput.*, 209(2):198–226, 2011.
- [41] S. Lenglet, A. Schmitt, and J.-B. Stefani. Characterizing contextual equivalence in calculi with passivation. *Inf. Comput.*, 209(11):1390–1433, 2011.
- [42] M. Lienhardt, I. Lanese, M. Bravetti, D. Sangiorgi, G. Zavattaro, Y. Welsch, J. Schfer, and A. Poetzsch-Heffter. A Component Model for the ABS Language. In *Proc. of FMCO 2010*. Springer, 2011. To appear.
- [43] J. Magee and J. Kramer. Dynamic structure in software architectures. In *Proceedings of the 4th ACM SIGSOFT symposium on Foundations of software engineering*, SIGSOFT ’96, pages 3–14, New York, NY, USA, 1996. ACM.
- [44] Microsoft .NET Framework Developer Center. Windows Workflow Foundation. <http://msdn.microsoft.com/en-us/netframework/aa663328.aspx>, 2011.
- [45] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.
- [46] R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes, I. *Inf. Comput.*, 100(1):1–40, 1992.
- [47] M. Minsky. *Computation: Finite and Infinite Machines*. Prentice-Hall, 1967.
- [48] R. Morrison, D. Balasubramaniam, F. Oquendo, B. Warboys, and R. M. Greenwood. An active architecture approach to dynamic systems co-evolution. In *Proc. of ECSA*, volume 4758 of *LNCS*, pages 2–10. Springer, 2007.
- [49] R. Morrison, G. N. C. Kirby, D. Balasubramaniam, K. Mickan, F. Oquendo, S. Cîmpan, B. Warboys, B. Snowdon, and R. M. Greenwood. Support for evolving software architectures in the archware adl. In *4th Working IEEE / IFIP Conference on Software Architecture (WICSA 2004), 12-15 June 2004, Oslo, Norway*, pages 69–78. IEEE Computer Society, 2004.
- [50] U. Nestmann, R. Fuzzati, and M. Merro. Modeling consensus in a process calculus. In *Proc. of CONCUR*, volume 2761 of *LNCS*, pages 393–407. Springer, 2003.
- [51] F. Oquendo. π -ADL: an Architecture Description Language based on the higher-order typed π -calculus for specifying dynamic and mobile software architectures. *SIGSOFT Softw. Eng. Notes*, 29:1–14, May 2004.
- [52] J. A. Pérez. *Higher-Order Concurrency: Expressiveness and Decidability Results*. PhD thesis, University of Bologna, 2010. Draft in <http://www.jorgeaperez.net>.
- [53] A. Piérard and E. Sumii. Sound bisimulations for higher-order distributed process calculus. In *Proc. of FOSSACS*, volume 6604 of *LNCS*, pages 123–137. Springer, 2011.
- [54] A. Piérard and E. Sumii. A higher-order distributed calculus with name creation. In *LICS*, pages 531–540. IEEE, 2012.
- [55] C. Rackoff. The covering and boundedness problems for vector addition systems. *Theor. Comput. Sci.*, 6:223–231, 1978.
- [56] J. Riely and M. Hennessy. Distributed processes and location failures. *Theor. Comput. Sci.*, 266(1-2):693–735, 2001. An extended abstract appeared in Proc. of ICALP’97.

- [57] 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.
- [58] D. Sangiorgi and F. Montesi. A model of evolvable components. In *Proc. of 5th International Symposium on Trustworthy Global Computing (TGC 2010)*, LNCS. Springer, 2010.
- [59] 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.
- [60] P. Schnoebelen. Revisiting ackermann-hardness for lossy counter machines and reset petri nets. In *MFCS*, volume 6281 of LNCS, pages 616–628. Springer, 2010.
- [61] G. Stoye, 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.
- [62] M. Wermelinger and J. L. Fiadeiro. A graph transformation approach to software architecture reconfiguration. *Sci. Comput. Program.*, 44(2):133–155, 2002.
- [63] T. Wies, D. Zufferey, and T. A. Henzinger. Forward analysis of depth-bounded processes. In *FOSSACS*, volume 6014 of LNCS, pages 94–108. Springer, 2010.
- [64] G. Zavattaro and L. Cardelli. Termination problems in chemical kinetics. In *CONCUR*, volume 5201 of *Lecture Notes in Computer Science*, pages 477–491. Springer, 2008.

APPENDIX A. PROOFS FROM SECTION 2

A.1. Proof of Lemma 2.20. We need the following auxiliary definitions.

Definition A.1. Given two \mathcal{E} processes in normal form P and Q , we define $\text{St}(P \parallel Q)$ as follows. The root is labeled ϵ , and has $n + m$ children: the first n sub-children correspond to the children of the root of $\text{St}(P)$, while the rest correspond to the m children of the root of $\text{St}(Q)$,

Proposition A.2 (Syntactic Closure for \mathcal{E}_s processes). *Let P_1, P_2, \dots be \mathcal{E} processes.*

- (1) $P_1, P_2 \in \mathcal{E}_s$ iff $P_1 \parallel P_2 \in \mathcal{E}_s$.
- (2) $P \in \mathcal{E}_s$ iff $a[P] \in \mathcal{E}_s$.
- (3) $P_i \in \mathcal{E}_s$ and $|P_i|_{\text{ap}} = 0$ for $i \in [1..n]$ iff $\sum_{i=1}^n \pi_i. P_i \in \mathcal{E}_s$.
- (4) $P \in \mathcal{E}_s$ and $|P|_{\text{ap}} = 0$ iff $!\pi. P \in \mathcal{E}_s$.

Proof. Immediate from Definitions 2.4 and 2.12. In particular, items (3) and (4) follow by observing that any process P such that $|P|_{\text{ap}} = 0$ belongs to the syntactic category A in the grammar of \mathcal{E}_s processes given in Definition 2.4. □

We repeat the statement in Page 12:

Lemma A.3. *Let P be an \mathcal{E}_s process. If $P \longrightarrow P'$ then also P' is an \mathcal{E}_s process. Moreover, $\text{St}(P) = \text{St}(P')$.*

Proof. The proof proceeds by induction on the height of the derivation tree for $P \longrightarrow P'$, with a case analysis on the last applied rule. There are seven cases to check.

Case (Act1): Then $P = P_1 \parallel P_2$ and $P' = P'_1 \parallel P_2$, with $P_1 \longrightarrow P'_1$. By inductive hypothesis, we have that P'_1 is an \mathcal{E}_s process. By Proposition A.2 we have that $P_2 \in \mathcal{E}_s$, and we can therefore conclude that $P' = P'_1 \parallel P_2$ is an \mathcal{E}_s process.

Moreover, by inductive hypothesis, we have that $\text{St}(P_1) = \text{St}(P'_1)$ and by Definition 2.10 it is easy to see that $\text{St}(P_1 \parallel P_2) = \text{St}(P'_1 \parallel P_2)$ holds.

Case (Act2): Analogous to the case for (ACT1) and omitted.

Case (Loc): Then $P = a[Q]$ and $P' = a[Q']$, with $Q \longrightarrow Q'$. By inductive hypothesis, we have that Q' is an \mathcal{E}_s process. For Proposition A.2 we have that $P' = a[Q']$ is an \mathcal{E}_s process.

Moreover, by inductive hypothesis, we have that $\text{St}(Q) = \text{St}(Q')$. Then, it is immediate to see that by Definition 2.10 $\text{St}(a[Q]) = \text{St}(a[Q'])$.

Cases (Tau1)-(Tau2): Then $P \equiv C_1[A] \parallel C_2[B]$, where C_1, C_2 are monadic contexts as in Definition 7.11. Moreover, A is either $!b.Q$ or $\sum_{i \in I} \pi_i.Q_i$ with $\pi_l = b$, for some $l \in I$, and B is either $!\bar{b}.R$ or $\sum_{i \in I} \pi_i.R_i$ with $\pi_l = \bar{b}$, for some $l \in I$.

We consider only the case in which $A = \sum_{i \in I} \pi_i.Q_i$ and $B = !\bar{b}.R$; the other cases are similar. Then $P' \equiv C_1[Q_l] \parallel C_2[R \parallel !\bar{b}.R]$ and from Proposition A.2 we easily conclude that P' is an \mathcal{E}_s process.

By assumption and by Proposition A.2 we have that A and B are \mathcal{E}_s processes. In turn, this allows us to infer that $\text{St}(\sum_{i \in I} \pi_i.Q_i) = \text{St}(Q_l)$ and $\text{St}(!\bar{b}.R) = \text{St}(R)$, as well-formed \mathcal{E}_s processes do not contain adaptable processes behind prefixes, and therefore their component structure denotations are unaffected by input/output transitions. The thesis then follows by Definition 2.10: $\text{St}(P) = \text{St}(P')$.

Cases (Tau3)-(Tau4): Then $P \equiv C_1[A] \parallel C_2[B]$ where:

- C_1, C_2 are monadic contexts, as in Definition 7.11;
- $A = b[P_1]$, for some P_1 ;
- $B = \sum_{i \in I} \pi_i.R_i$ with $\pi_l = \tilde{b}\{b[U] \parallel P_2\}$ for $l \in I$, or $B = !\tilde{b}\{b[U] \parallel P_2\}.R$, for some P_2, R .

We consider the case in which $B = !\tilde{b}\{b[U] \parallel P_2\}.R$; the other case is similar. Then $P' \equiv C_1[a[U\langle\langle P_1 \rangle\rangle] \parallel P_2] \parallel C_2[R \parallel !\tilde{b}\{b[U] \parallel P_2\}.R]$. For Proposition A.2 we have that $C_2[R \parallel !\tilde{b}\{b[U] \parallel P_2\}.R]$ and P_2 are \mathcal{E}_s processes. We now focus on process $U\langle\langle P_1 \rangle\rangle$, for Proposition A.2 we know that P_1 is an \mathcal{E}_s process, if $|U|_{\text{ph}} = 0$ then it could not occur that an adaptable process in P_1 is prefixed. Otherwise, if $|U|_{\text{ph}} > 0$ then the side condition (2) of rule (TAU3)((TAU4)) ensures that $|P_1|_{\text{ap}} = 0$. As U follows the syntax of \mathcal{E}_s by means of Proposition A.2 we can conclude that $U\langle\langle P_1 \rangle\rangle \in \mathcal{E}_s$.

Moreover, the side condition (1) of rule (TAU3)((TAU4)) implies that

$$\text{St}(b[P_1]) = \text{St}(a[U\langle\langle P_1 \rangle\rangle] \parallel P_2).$$

The thesis then follows by Definition 2.10: $\text{St}(P) = \text{St}(P')$. □

A.2. Proof of Theorem 2.26. We divide the proof into two lemmas. We need some auxiliary results.

Proposition A.4. *Let P_1 and P_2 be \mathcal{E}_s processes. Then $\text{Sub}_{\text{St}}(P_1 \parallel P_2) = \text{Sub}_{\text{St}}(P_1) \cup \text{Sub}_{\text{St}}(P_2)$.*

Proof. Immediate from the definition of $\text{Sub}_{\text{St}}(\cdot)$ (cf. Definition 2.21). □

Lemma A.5. *Let P be an \mathcal{E}_s process. Also, let S be a set of containment structure denotations, such that $\text{Sub}_{\text{St}}(P) \subseteq S$. Given the encoding $\llbracket \cdot \rrbracket_S^d$ in Definition 2.24, if $P \rightarrow_s P'$ then $\llbracket P \rrbracket_S^d \rightarrow_d \llbracket P' \rrbracket_S^d$.*

Proof. By induction on the height of the derivation tree for $P \rightarrow_s P'$, with a case analysis on the last applied rule. There are seven cases to check.

Case (Act1): Then $P = P_1 \parallel P_2$ and $P' = P'_1 \parallel P_2$, with $P_1 \rightarrow_s P'_1$. By inductive hypothesis, we have that $\llbracket P_1 \rrbracket_{S'}^d \rightarrow_d \llbracket P'_1 \rrbracket_{S'}^d$ with $\text{SubSt}(P_1) \subseteq S'$. Now, since $\llbracket \cdot \rrbracket_S^d$ is defined as an homomorphism with respect to parallel composition, and using Proposition A.4, we can immediately infer that $\llbracket P_1 \parallel P_2 \rrbracket_S^d \rightarrow_d \llbracket P'_1 \parallel P_2 \rrbracket_S^d$, with $S' \cup \text{SubSt}(P_2) \subseteq S$, as wanted.

Case (Act2): Analogous to the case for (ACT1) and omitted.

Case (Loc): Then $P = a[Q]$ and $P' = a[Q']$, with $Q \rightarrow Q'$. By inductive hypothesis, we have that $\llbracket Q \rrbracket_{S'}^d \rightarrow_w \llbracket Q' \rrbracket_{S'}^d$, with $\text{SubSt}(Q) \subseteq S'$. From Definitions 2.24 and 2.21 we immediately infer that $\llbracket a[Q] \rrbracket_S^d \rightarrow_d \llbracket a[Q'] \rrbracket_S^d$, with $S' \cup \text{St}(a[Q]) \subseteq S$.

Cases (Tau1)-(Tau2): Then $P \equiv C_1[A] \parallel C_2[B]$, where

- C_1, C_2 are monadic contexts as in Definition 7.11;
- A is either $!b.Q$ or $\sum_{i \in I} \pi_i.Q_i$ with $\pi_l = b$, for some $l \in I$;
- B is either $!\bar{b}.R$ or $\sum_{i \in I} \pi_i.R_i$ with $\pi_l = \bar{b}$, for some $l \in I$.

We consider only the case in which $A = \sum_{i \in I} \pi_i.Q_i$ and $B = !\bar{b}.R$; the other cases are similar. Then, $P' \equiv C_1[Q_l] \parallel C_2[R \parallel !\bar{b}.R]$. Using Definitions 2.24 and 7.11 we verify that the reduction from P is preserved in $\llbracket P \rrbracket_S^d$:

$$\begin{aligned} \llbracket P \rrbracket_S^d &= \llbracket C_1 \left[\sum_{i \in I} \pi_i.Q_i \right] \parallel C_2[!\bar{b}.R] \rrbracket_S^d \quad \text{with } \text{SubSt}(P) \subseteq S \\ &= \llbracket C_1 \rrbracket_S^d \left[\sum_{i \in I} \llbracket \pi_i.Q_i \rrbracket_S^d \right] \parallel \llbracket C_2 \rrbracket_S^d \left[\llbracket !\bar{b}.R \rrbracket_S^d \right] \\ &= \llbracket C_1 \rrbracket_S^d \left[\sum_{i \in I} \pi_i.\llbracket Q_i \rrbracket_S^d \right] \parallel \llbracket C_2 \rrbracket_S^d \left[!\bar{b}.\llbracket R \rrbracket_S^d \right] \end{aligned}$$

At this point, it is immediate to infer a reduction \rightarrow_d on b :

$$\llbracket P \rrbracket_S^d \rightarrow_d \llbracket C_1 \rrbracket_S^d \left[\llbracket Q_l \rrbracket_S^d \right] \parallel \llbracket C_2 \rrbracket_S^d \left[\llbracket R \rrbracket_S^d \parallel !\bar{b}.\llbracket R \rrbracket_S^d \right]$$

which is easily seen to correspond to $\llbracket P' \rrbracket_S^d$, as wanted.

Cases (Tau3)-(Tau4): Then $P \equiv C_1[A] \parallel C_2[B]$ where:

- C_1, C_2 are monadic contexts, as in Definition 7.11;
- $A = b[P_1]$, for some P_1 ;
- $B = \sum_{i \in I} \pi_i.R_i$ with $\pi_l = \tilde{b}\{b[U] \parallel A_2\}$ for $l \in I$, or $B = !\tilde{b}\{b[U] \parallel P_2\}.R$, for some A_2, R .
- $\text{cond}(U, P_1)$ holds

We consider only the case in which $B = !\tilde{b}\{b[U] \parallel A_2\}.R$; the other case is similar. Then, $P' \equiv C_1[a[U \langle\langle P_1 \rangle\rangle] \parallel A_2] \parallel C_2[B \parallel R]$. Since $\text{cond}(U, P_1)$ holds, we rely on Lemma 2.18 to determine the possible cases for U and P_1 : each of them entails a different encoding of $\llbracket P \rrbracket_S^d$. Consequently, we verify that in each case the actions that lead to reduction in P are preserved in $\llbracket P \rrbracket_S^d$.

(a) $|U|_\bullet = 0 \wedge \text{St}(P_1) = \text{St}(U)$. Then, using the definition of $\llbracket \cdot \rrbracket_S^d$, we have

$$\begin{aligned} \llbracket P \rrbracket_S^d &= \llbracket C_1[b[P_1]] \parallel C_2[\tilde{b}\{b[U] \parallel A_2\}.R] \rrbracket_S^d \\ &= \llbracket C_1 \rrbracket_S^d \left[\llbracket b[P_1] \rrbracket_S^d \right] \parallel \llbracket C_2 \rrbracket_S^d \left[\llbracket \tilde{b}\{b[U] \parallel A_2\}.R \rrbracket_S^d \right] \\ &= \llbracket C_1 \rrbracket_S^d \left[\kappa[\llbracket P_1 \rrbracket_S^d] \right] \parallel \llbracket C_2 \rrbracket_S^d \left[!\tilde{\kappa}\{\kappa[\llbracket U \rrbracket_S^d] \parallel \llbracket A_2 \rrbracket_S^d\}.\llbracket R \rrbracket_S^d \right] \end{aligned}$$

At this point, it is immediate to infer a reduction \longrightarrow_d on κ :

$$\begin{aligned} \llbracket P \rrbracket_S^d &\longrightarrow_d \llbracket C_1 \rrbracket_S^d[\star] \{(\kappa[\llbracket U \rrbracket_S^d] \parallel \llbracket A_2 \rrbracket_S^d) \langle \langle P_1 \rangle \rangle / \star\} \parallel \llbracket C_2 \rrbracket_S^d[\llbracket B \rrbracket_S^d \parallel \llbracket R \rrbracket_S^d] \\ &= \llbracket C_1 \rrbracket_S^d[(\kappa[\llbracket U \rrbracket_S^d] \parallel \llbracket A_2 \rrbracket_S^d) \langle \langle P_1 \rangle \rangle] \parallel \llbracket C_2 \rrbracket_S^d[\llbracket B \rrbracket_S^d \parallel \llbracket R \rrbracket_S^d] \\ &= \llbracket C_1 \rrbracket_S^d[\kappa[\llbracket P_1 \rrbracket_S^d] \parallel \llbracket A_2 \rrbracket_S^d] \parallel \llbracket C_2 \rrbracket_S^d[\llbracket B \rrbracket_S^d \parallel \llbracket R \rrbracket_S^d] = P'' \end{aligned}$$

which is easily seen to correspond to $\llbracket P' \rrbracket_S^d$, as desired.

(b) $|U|_{\bullet} = 1 \wedge |U|_{\text{ap}} = 0 \wedge (|U|_{\text{ph}} > 0 \Rightarrow |Q|_{\text{ap}} = 0)$. There are two subcases:

(1) **Case** $|U|_{\text{ph}} > 0$: Then, similarly as in the previous case, using the definition of $\llbracket \cdot \rrbracket_S^d$ we can infer a reduction \longrightarrow_d on name κ_b .

(2) **Case** $|U|_{\text{ph}} = 0$. Then, using the definition of $\llbracket \cdot \rrbracket_S^d$, we have

$$\begin{aligned} \llbracket P \rrbracket_S^d &= \llbracket C_1[b[P_1]] \parallel C_2[\tilde{b}\{b[U] \parallel A_2\}.R] \rrbracket_S^d \\ &= \llbracket C_1 \rrbracket_S^d[\llbracket b[P_1] \rrbracket_S^d] \parallel \llbracket C_2 \rrbracket_S^d[\llbracket \tilde{b}\{b[U] \parallel A_2\}.R \rrbracket_S^d] \\ &= \llbracket C_1 \rrbracket_S^d[\kappa_j[\llbracket P_1 \rrbracket_S^d]] \parallel \llbracket C_2 \rrbracket_S^d\left[\prod_{\kappa_i \in \varphi(S \downarrow b)} \tilde{\kappa}_i\{\kappa_i[\llbracket U \rrbracket_S^d] \parallel \llbracket A_2 \rrbracket_S^d\} \cdot \llbracket R \rrbracket_S^d\right] \end{aligned}$$

with $\kappa_j = \varphi(b[P_1])$. At this point, it is immediate to infer a reduction \longrightarrow_d on κ_j :

$$\begin{aligned} \llbracket P \rrbracket_S^d &\longrightarrow_d \llbracket C_1 \rrbracket_S^d[\star] \{(\kappa_j[\llbracket U \rrbracket_S^d] \parallel \llbracket A_2 \rrbracket_S^d) \langle \langle P_1 \rangle \rangle / \star\} \parallel \llbracket C_2 \rrbracket_S^d[\llbracket B \rrbracket_S^d \parallel \llbracket R \rrbracket_S^d] \\ &= \llbracket C_1 \rrbracket_S^d[(\kappa_j[\llbracket U \rrbracket_S^d] \parallel \llbracket A_2 \rrbracket_S^d) \langle \langle P_1 \rangle \rangle] \parallel \llbracket C_2 \rrbracket_S^d[\llbracket B \rrbracket_S^d \parallel \llbracket R \rrbracket_S^d] \\ &= \llbracket C_1 \rrbracket_S^d[\kappa_j[\llbracket P_1 \rrbracket_S^d]] \parallel \llbracket C_2 \rrbracket_S^d[\llbracket B \rrbracket_S^d \parallel \llbracket R \rrbracket_S^d] = P'' \end{aligned}$$

which is easily seen to correspond to $\llbracket P' \rrbracket_S^d$, as desired.

(c) $|U|_{\bullet} > 1 \wedge |U|_{\text{ap}} = 0 \wedge |Q|_{\text{ap}} = 0$. Then, similarly as in case 1(a), using the definition of $\llbracket \cdot \rrbracket_S^d$ we can infer a reduction \longrightarrow_d on name κ_b . \square

Lemma A.6. *Let P be an \mathcal{E}_s process. Also, let S be a set of containment structure denotations, such that $\text{Sub}_{\text{st}}(P) \subseteq S$. Given the encoding $\llbracket \cdot \rrbracket_S^d$ in Definition 2.24, if $\llbracket P \rrbracket_S^d \longrightarrow_d \llbracket P' \rrbracket_S^d$ then $P \longrightarrow_s P'$.*

Proof. By induction on the height of the derivation tree for $P \longrightarrow_d P'$, with a case analysis on the last applied rule. There are seven cases to check. The analysis of all cases mirrors the one detailed in the proof of Lemma A.5, and we omit it. The crucial point is the fact that the encoding uses the special name err to rename those update prefixes that may lead to incorrect reductions in \mathcal{E}_s . Hence, adaptable processes included in the \mathcal{E}_d process $\llbracket P \rrbracket_S^d$ will be unable to interact with those ‘‘error’’ update prefixes. This ensures that for every reduction \longrightarrow_d there is also a reduction \longrightarrow_s . \square

We repeat the statement in Page 15:

Theorem A.7 (2.26). *Let P be an \mathcal{E}_s process. Also, let S be a set of containment structure denotations, such that $\text{Sub}_{\text{st}}(P) \subseteq S$. Then we have:*

$$P \longrightarrow_s P' \text{ if and only if } \llbracket P \rrbracket_S^d \longrightarrow_d \llbracket P' \rrbracket_S^d$$

Proof. Immediate from Lemmas A.5 and A.6. \square

APPENDIX B. PROOFS FROM SECTION 6

B.1. Proof of Lemma 6.2. The proof relies on two results: completeness (Lemma B.2) and soundness (Lemma B.3). We begin by defining the encoding of MM configuration into \mathcal{E}^1 .

Definition B.1. Let N be a MM with registers r_j ($j \in \{0, 1\}$) and instructions $(1 : I_1), \dots, (n : I_n)$. The encoding of a configuration (i, m_0, m_1) of N , denoted $\llbracket (i, m_0, m_1) \rrbracket_1$, is defined as:

$$\bar{p}_i \parallel \llbracket r_0 = m_0 \rrbracket_1 \parallel \llbracket r_1 = m_1 \rrbracket_1 \parallel \prod_{i=1}^n \llbracket (i : I_i) \rrbracket_1$$

where the encodings $\llbracket r_j = m_j \rrbracket_1$ and $\llbracket (i : I_i) \rrbracket_1, \dots, \llbracket (n : I_n) \rrbracket_1$ are as in Table 2.

Lemma B.2 (Completeness). *Let (i, m_0, m_1) be a configuration of a MM N .*

- (1) *If $(i, m_0, m_1) \rightarrow_M (i', m'_0, m'_1)$ then, for some process P , it holds that $\llbracket (i, m_0, m_1) \rrbracket_1 \rightarrow^* P \equiv \llbracket (i', m'_0, m'_1) \rrbracket_1$.*
- (2) *If $(i, m_0, m_1) \not\rightarrow_M$ then $\llbracket (i, m_0, m_1) \rrbracket_1 \Downarrow_e^k$*

Proof.

- (1) We proceed by a case analysis on the instruction performed by the Minsky machine. Hence, we distinguish three cases corresponding to the behaviors associated to rules M-INC, M-DEC, and M-JMP. Without loss of generality, we restrict our analysis to operations on register r_0 .

Case M-Inc: We have a Minsky configuration (i, m_0, m_1) with $(i : \text{INC}(r_0))$. By Definition B.1, its encoding into \mathcal{E}^1 is as follows:

$$\begin{aligned} \llbracket (i, m_0, m_1) \rrbracket_1 &= \bar{p}_i \parallel \llbracket r_0 = m_0 \rrbracket_1 \parallel \llbracket r_1 = m_1 \rrbracket_1 \parallel \\ &\quad \llbracket (i : \text{INC}(r_0)) \rrbracket_1 \parallel \prod_{l=1..n, l \neq i} \llbracket (l : I_l) \rrbracket_1 \end{aligned}$$

After consuming the program counter p_i we have the following

$$\llbracket (i, m_0, m_1) \rrbracket_1 \longrightarrow r_0[\langle m_0 \rangle_0] \parallel \tilde{r}_0\{r_0[\bar{u}_0 \cdot \bullet]\} \cdot \bar{p}_{i+1} \parallel S = P_1$$

where $S = \llbracket r_1 = m_1 \rrbracket_1 \parallel \prod_{i=1}^n \llbracket (i : I_i) \rrbracket_1$ stands for the rest of the system. The only reduction possible at this point is the synchronization on r_0 , which allows the update of the adaptable process at r_0 :

$$P_1 \longrightarrow r_0[\bar{u}_0 \cdot \langle m_0 \rangle_0] \parallel \bar{p}_{i+1} \parallel S = P_2.$$

By the encoding of numbers, it P_2 can be equivalently written as

$$r_0[\langle m_0 + 1 \rangle_0] \parallel \bar{p}_{i+1} \parallel S$$

and so it is easy to see that $P_2 \equiv \llbracket (i + 1, m_0 + 1, m_1) \rrbracket_1$, as desired.

Case M-Dec: We have a Minsky configuration (i, c, m_1) such that $(i : \text{DEC}(r_0, s))$ and $c > 0$. By Definition B.1, its encoding into \mathcal{E}^1 is as follows:

$$\begin{aligned} \llbracket (i, c, m_1) \rrbracket_1 &= \bar{p}_i \parallel \llbracket r_0 = c \rrbracket_1 \parallel \llbracket r_1 = m_1 \rrbracket_1 \parallel \\ &\quad \llbracket (i : \text{DEC}(r_0, s)) \rrbracket_1 \parallel \prod_{l=1..n, l \neq i} \llbracket (l : I_l) \rrbracket_1 \end{aligned}$$

We begin by consuming the program counter p_i , which leaves the content of $\llbracket(i : \text{DEC}(r_0, s))\rrbracket_1$ exposed. Using the encoding of numbers we have the following:

$$\llbracket(i, c, m_1)\rrbracket_1 \longrightarrow r_0[\overline{u_0} \cdot \langle c - 1 \rangle_0] \parallel (u_0 \cdot \overline{p_{i+1}} + z_0 \cdot \tilde{r}_0\{r_0[\overline{z_0}]\} \cdot \overline{p_s}) \parallel S = P_1$$

where $S = \llbracket r_1 = m_1 \rrbracket_1 \parallel \prod_{i=1}^n \llbracket(i : I_i)\rrbracket_1$ stands for the rest of the system. Notice that only reduction possible at this point is the synchronization on u_0 , which signals the fact we are performing a decrement instruction. After this synchronization we have

$$\begin{aligned} P_1 &\longrightarrow r_0[\langle c - 1 \rangle_0] \parallel \overline{p_{i+1}} \parallel S \\ &\equiv \llbracket(i + 1, c - 1, m_1)\rrbracket_1 \end{aligned}$$

as desired.

Case M-Jmp: We have a Minsky configuration $(i, 0, m_1)$ and $(i : \text{DEC}(r_0, s))$. By Definition B.1, its encoding into \mathcal{E}^1 is as follows:

$$\begin{aligned} \llbracket(i, 0, m_1)\rrbracket_1 &= \overline{p_i} \parallel \llbracket r_0 = 0 \rrbracket_1 \parallel \llbracket r_1 = m_1 \rrbracket_1 \parallel \\ &\quad \llbracket(i : \text{DEC}(r_0, s))\rrbracket_1 \parallel \prod_{l=1..n, l \neq i} \llbracket(l : I_l)\rrbracket_1. \end{aligned}$$

We begin by consuming the program counter p_i , which leaves the content of $\llbracket(i : \text{DEC}(r_0, s))\rrbracket_1$ exposed. Using the encoding of numbers we have the following:

$$\llbracket(i, 0, m_1)\rrbracket_1 \longrightarrow r_0[\overline{z_0}] \parallel (u_0 \cdot \overline{p_{i+1}} + z_0 \cdot \tilde{r}_0\{r_0[\overline{z_0}]\} \cdot \overline{p_s}) \parallel S = P_1$$

where $S = \llbracket r_1 = m_1 \rrbracket_1 \parallel \prod_{i=1}^n \llbracket(i : I_i)\rrbracket_1$ stands for the rest of the system. In P_1 , the only reduction possible is through a synchronization on z_0 , which signals the fact we are performing a jump. Such a synchronization, in turn, enables an update action on r_0 . We then have:

$$\begin{aligned} P_1 &\longrightarrow r_0[\mathbf{0}] \parallel \tilde{r}_0\{r_0[\overline{z_0}]\} \cdot \overline{p_s} \parallel S \\ &\longrightarrow r_0[\overline{z_0}] \parallel \overline{p_s} \parallel S \\ &\equiv \llbracket(s, 0, m_1)\rrbracket_1 \end{aligned}$$

as desired.

(2) We have a Minsky configuration (i, m_0, m_1) with $(i : \text{HALT})$. By Definition B.1, its encoding into \mathcal{E}^1 is as follows:

$$\begin{aligned} \llbracket(i, m_0, m_1)\rrbracket_1 &= p_i \parallel \llbracket r_0 = m_0 \rrbracket_1 \parallel \llbracket r_1 = m_1 \rrbracket_1 \\ &\quad \parallel \llbracket(i : \text{HALT})\rrbracket_1 \parallel \prod_{l=1..n, l \neq i} \llbracket(l : I_l)\rrbracket_1 \\ &\equiv \overline{p_i} \parallel !p_i \cdot (e + \overline{p_i}) \parallel S = P_0 \end{aligned}$$

where $S = \llbracket r_0 = m_0 \rrbracket_1 \parallel \llbracket r_1 = m_1 \rrbracket_1 \parallel \prod_{l=1..n, l \neq i} \llbracket(l : I_l)\rrbracket_1$ stands for the part of the system that is not able to interact. It is easy to see that $P_0 \Downarrow_e^1$. In fact, by synchronizing on p_i and choosing the left-hand side process in the binary sum, we have $P_0 \xrightarrow{e}$. The thesis is easily seen to hold by observing that by releasing new copies of the encoding of $(i : \text{HALT})$, one always reaches a derivative P_j of P_0 such that $P_j \Downarrow_e$. \square

Lemma B.3 (Soundness). *Let (i, m_0, m_1) be a configuration of a MM N . If $\llbracket (i, m_0, m_1) \rrbracket_1 \longrightarrow P_1$ then either:*

(1) *For every computation of P_1 there exists a P_j such that*

$$P_1 \longrightarrow^* P_j = \llbracket (i', m'_0, m'_1) \rrbracket_1$$

and $(i, m_0, m_1) \longrightarrow_M (i', m'_0, m'_1)$; or

(2) *$P_1 \Downarrow_e^k$ and $(i, m_0, m_1) \not\rightarrow_M$.*

Proof. Consider the reduction $\llbracket (i, m_0, m_1) \rrbracket_1 \longrightarrow P_1$. An analysis of the structure of process $\llbracket (i, m_0, m_1) \rrbracket_1$ reveals that, in all cases, the only possibility for the first step corresponds to the consumption of the program counter p_i . This implies that there exists an instruction labeled with i , that can be executed from the configuration (i, m_0, m_1) . We proceed by a case analysis on the possible instruction, considering also the fact that the register on which the instruction acts can hold a value equal or greater than zero.

In the cases in which $(i : \text{INC}(r_j))$ or $(i : \text{DEC}(r_j, s))$, it can be shown that computation evolves deterministically until reaching a process in which a new program counter (that is, some $\overline{p_{i'}}$) appears. The program counter $\overline{p_{i'}}$ is always inside a process that corresponds to $\llbracket (i', m'_0, m'_1) \rrbracket_1$, where $(i, m_0, m_1) \longrightarrow_M (i', m'_0, m'_1)$. That is, for the cases $(i : \text{INC}(r_j))$ and $(i : \text{DEC}(r_j, s))$, we have that Item (1) above holds. The detailed analysis follows the same lines as the one reported for the proof of Lemma B.2, and we omit it.

In the case in which $(i : \text{HALT})$, we have that Item (2) holds. In order to see this, it suffices to observe that if N does not terminate (more precisely: if N does not reach a program counter associated to a **HALT** instruction) then $\llbracket N \rrbracket_1$ does not have a barb on e . In fact, by a simple inspection on the encodings in Table 2 we can deduce that e only appears in the encoding of halt instructions, and does not occur in the encodings of increment and decrement-and-jump instructions. Hence, a barb on e can only be observed when P_1 is the result of triggering a halt instruction. \square

We are now ready to repeat the statement of Lemma 6.2, in Page 25:

Lemma B.4 (6.2). *Let N be a MM and $k \geq 1$. N terminates iff $\llbracket N \rrbracket_1 \Downarrow_e^k$.*

Proof. It follows directly from Lemmas B.2 and B.3. \square

APPENDIX C. PROOFS FROM SECTION 7

C.1. Proof of Lemma 7.27. The proof relies on two auxiliary results: completeness (Lemma C.5) and soundness (Lemma C.6). Completeness relies on the auxiliary Lemma C.3.

We first introduce the notion of encoding of a MM configuration into \mathcal{E}_s^2 . Notice that it in addition to the encodings of registers and instructions, it includes a number of resources \overline{f} and \overline{b} which are always available during the execution of the machine:

Definition C.1. Let N be a MM with registers r_j ($j \in \{0, 1\}$) and instructions $(1 : I_1), \dots, (n : I_n)$. The encoding of a configuration (i, m_0, m_1) of N , denoted $\llbracket (i, m_0, m_1) \rrbracket_2$, is defined as:

$$\overline{p_i} \parallel e \parallel \llbracket r_0 = m_0 \rrbracket_2 \parallel \llbracket r_1 = m_1 \rrbracket_2 \parallel \prod_{i=1}^n \llbracket (i : I_i) \rrbracket_2 \parallel C^{(\alpha, \beta, \gamma)}$$

where

- $C^{\langle\alpha,\beta,\gamma\rangle} \stackrel{\text{def}}{=} \prod^\alpha \bar{f} \parallel \prod^\beta \bar{b} \parallel \prod^\gamma \bar{g} \parallel !a.(\bar{f} \parallel \bar{b} \parallel \bar{a}) \parallel !h.(g.\bar{f} \parallel \bar{h})$, with $\alpha, \beta, \gamma \geq 0$
- the encodings $\llbracket r_j = m_j \rrbracket_2$ and $\llbracket (i : I_i) \rrbracket_2, \dots, \llbracket (n : I_n) \rrbracket_2$ are as in Table 3.

Notice that $C^{\langle\alpha,\beta,\gamma\rangle}$ abstracts the evolution of process CONTROL in Table 3, and the resources that it produces and maintains (namely, α copies of \bar{f} , β copies of \bar{b} , and γ copies of \bar{g}).

Remark C.2. As we have discussed, the presence of copies of \bar{f} is required for the execution of increment and decrement-and-jump instructions. In their absence, the encoding of the MM would reach a deadlocked state. Such outputs are produced at the beginning of the execution of the encoding of a MM, by means of a replicated process. In the proofs below, we assume that the initialization of the encoding always produces enough copies of \bar{f} so as to ensure the existence of a correct simulation of the machine. That is to say, we assume that the absence of copies of \bar{f} is not a possible source of deadlocks.

We prove that given a MM N there exists a computation of process $\llbracket N \rrbracket_2$ which correctly mimics its behavior.

Lemma C.3. *Let (i, m_0, m_1) be a configuration of a MM N .*

(1) *If $(i, m_0, m_1) \rightarrow_M (i', m'_0, m'_1)$ then, for some process P , it holds that*

$$\llbracket (i, m_0, m_1) \rrbracket_2 \rightarrow^* P \equiv \llbracket (i', m'_0, m'_1) \rrbracket_2$$

(2) *If $(i, m_0, m_1) \not\rightarrow_M$ then $\llbracket (i, m_0, m_1) \rrbracket_2 \Downarrow_{PT}^1$.*

Proof.

(1) We proceed by a case analysis on the instruction performed by the Minsky machine. Hence, we distinguish three cases corresponding to the behaviors associated to rules M-INC, M-DEC, and M-JMP. Without loss of generality, we restrict our analysis to operations on register r_0 .

Case M-INC: We have a Minsky configuration (i, m_0, m_1) with $(i : \text{INC}(r_0))$. By Definition C.1, its encoding into \mathcal{E}_s^2 is as follows:

$$\begin{aligned} \llbracket (i, m_0, m_1) \rrbracket_2 &= \bar{p}_i \parallel e \parallel \llbracket r_0 = m_0 \rrbracket_2 \parallel \llbracket r_1 = m_1 \rrbracket_2 \parallel \\ &\quad !p_i.f.(\bar{g} \parallel b.\overline{inc_0.p_{i+1}}) \parallel \prod_{l=1..n, l \neq i} \llbracket (l : I_l) \rrbracket_2 \parallel C^{\langle\alpha,\beta,\gamma\rangle} \end{aligned}$$

We then have:

$$\llbracket (i, m_0, m_1) \rrbracket_2 \rightarrow e \parallel \llbracket r_0 = m_0 \rrbracket_2 \parallel f.(\bar{g} \parallel b.\overline{inc_0.p_{i+1}}) \parallel C^{\langle\alpha,\beta,\gamma\rangle} \parallel S = P$$

where $S = \llbracket r_1 = m_1 \rrbracket_2 \parallel \prod_{l=1}^n \llbracket (l : I_l) \rrbracket_2$ stands for the rest of the system. Starting from P , a possible sequence of reductions is the following:

$$\begin{aligned} P &\rightarrow e \parallel \llbracket r_0 = m_0 \rrbracket_2 \parallel b.\overline{inc_0.p_{i+1}} \parallel C^{\langle\alpha-1,\beta,\gamma+1\rangle} \parallel S \\ &= e \parallel r_0[!inc_0.\bar{u}_0 \parallel \prod_{\bar{u}_0}^{m_0} \bar{u}_0 \parallel \bar{z}_0] \parallel b.\overline{inc_0.p_{i+1}} \parallel C^{\langle\alpha-1,\beta,\gamma+1\rangle} \parallel S \\ &\rightarrow e \parallel r_0[!inc_0.\bar{u}_0 \parallel \prod_{\bar{u}_0}^{m_0} \bar{u}_0 \parallel \bar{z}_0] \parallel \overline{inc_0.p_{i+1}} \parallel C^{\langle\alpha-1,\beta-1,\gamma+1\rangle} \parallel S \\ &\rightarrow \equiv e \parallel r_0[!inc_0.\bar{u}_0 \parallel \prod_{\bar{u}_0}^{m_0+1} \bar{u}_0 \parallel \bar{z}_0] \parallel \overline{p_{i+1}} \parallel C^{\langle\alpha-1,\beta-1,\gamma+1\rangle} \parallel S = P' \end{aligned}$$

It is easy to see that $P' \equiv \llbracket (i+1, m_0+1, m_1) \rrbracket_2$, as desired. Observe how the number of resources changes: in the first reduction, a copy of \bar{f} is consumed, and a copy of \bar{g} is released in its place. Notice that we are assuming that $\beta > 0$, that is, that there is at least one copy of \bar{b} . In fact, since the instruction only takes place after a synchronization on b (i.e., the second reduction above) the presence of at least one copy of \bar{b} in $C^{(\alpha-1, \beta, \gamma+1)}$ is essential to avoid deadlocks.

Case M-Dec: We have a Minsky configuration (i, m_0, m_1) with $m_0 > 0$ and $(i : \text{DEC}(r_0, s))$. By Definition C.1, its encoding into \mathcal{E}_s^2 is as follows:

$$\begin{aligned} \llbracket (i, m_0, m_1) \rrbracket_2 &= \bar{p}_i \parallel e \parallel \llbracket r_0 = m_0 \rrbracket_2 \parallel \llbracket r_1 = m_1 \rrbracket_2 \parallel \\ &\quad !p_i. f. (\bar{g} \parallel (u_0. (\bar{b} \parallel \bar{p}_{i+1}) + z_0. \tilde{r}_0\{r_0[!inc_0. \bar{u}_0 \parallel \bar{z}_0]\}. \bar{p}_s)) \\ &\quad \parallel \prod_{l=1..n, l \neq i} \llbracket (l : I_l) \rrbracket_2 \parallel C^{(\alpha, \beta, \gamma)} \end{aligned}$$

We then have:

$$\begin{aligned} \llbracket (i, m_0, m_1) \rrbracket_2 &\longrightarrow \equiv f. (\bar{g} \parallel (u_0. (\bar{b} \parallel \bar{p}_{i+1}) + z_0. \tilde{r}_0\{r_0[!inc_0. \bar{u}_0 \parallel \bar{z}_0]\}. \bar{p}_s)) \\ &\quad \parallel e \parallel \llbracket r_0 = m_0 \rrbracket_2 \parallel C^{(\alpha, \beta, \gamma)} \parallel S = P \end{aligned}$$

where $S = \llbracket r_1 = m_1 \rrbracket_2 \parallel \prod_{l=1}^n \llbracket (l : I_l) \rrbracket_2$ stands for the rest of the system. Starting from P , a possible sequence of reductions is the following:

$$\begin{aligned} P &\longrightarrow u_0. (\bar{b} \parallel \bar{p}_{i+1}) + z_0. \tilde{r}_0\{r_0[!inc_0. \bar{u}_0 \parallel \bar{z}_0]\}. \bar{p}_s \parallel \\ &\quad e \parallel \llbracket r_0 = m_0 \rrbracket_2 \parallel C^{(\alpha-1, \beta, \gamma+1)} \parallel S \\ &= u_0. (\bar{b} \parallel \bar{p}_{i+1}) + z_0. \tilde{r}_0\{r_0[!inc_0. \bar{u}_0 \parallel \bar{z}_0]\}. \bar{p}_s \parallel \\ &\quad e \parallel r_0[!inc_0. \bar{u}_0 \parallel \prod_{i=0}^{m_0} \bar{u}_0 \parallel \bar{z}_0] \parallel C^{(\alpha-1, \beta, \gamma+1)} \parallel S = P' \\ &\longrightarrow \bar{p}_{i+1} \parallel e \parallel r_0[!inc_0. \bar{u}_0 \parallel \prod_{i=0}^{m_0-1} \bar{u}_0 \parallel \bar{z}_0] \parallel C^{(\alpha-1, \beta+1, \gamma+1)} \parallel S = P'' \end{aligned}$$

It is easy to see that $P' \equiv \llbracket (i+1, m_0-1, m_1) \rrbracket_2$, as desired. Observe how in the last reduction the presence of at least a copy of \bar{u}_0 in r_0 is fundamental for releasing both an extra copy of \bar{b} and the trigger for the next instruction.

Case M-Jmp: We have a Minsky configuration $(i, 0, m_1)$ and $(i : \text{DEC}(r_0, s))$. By Definition C.1, its encoding into \mathcal{E}_s^2 is as follows:

$$\begin{aligned} \llbracket (i, 0, m_1) \rrbracket_2 &= \bar{p}_i \parallel e \parallel \llbracket r_0 = 0 \rrbracket_2 \parallel \llbracket r_1 = m_1 \rrbracket_2 \parallel \\ &\quad !p_i. f. (\bar{g} \parallel (u_0. (\bar{b} \parallel \bar{p}_{i+1}) + z_0. \tilde{r}_0\{r_0[!inc_0. \bar{u}_0 \parallel \bar{z}_0]\}. \bar{p}_s)) \\ &\quad \parallel \prod_{l=1..n, l \neq i} \llbracket (l : I_l) \rrbracket_2 \parallel C^{(\alpha, \beta, \gamma)} \end{aligned}$$

We then have:

$$\begin{aligned} \llbracket (i, 0, m_1) \rrbracket_2 &\longrightarrow \equiv f. (\bar{g} \parallel (u_0. (\bar{b} \parallel \bar{p}_{i+1}) + z_0. \tilde{r}_0\{r_0[!inc_0. \bar{u}_0 \parallel \bar{z}_0]\}. \bar{p}_s)) \\ &\quad \parallel e \parallel \llbracket r_0 = m_0 \rrbracket_2 \parallel C^{(\alpha, \beta, \gamma)} \parallel S = P \end{aligned}$$

where $S = \llbracket r_1 = m_1 \rrbracket_2 \parallel \prod_{l=1}^n \llbracket (l : I_l) \rrbracket_2$ stands for the rest of the system. Starting from P , a possible sequence of reductions is the following:

$$\begin{aligned}
P &\longrightarrow u_0.(\bar{b} \parallel \overline{p_{i+1}}) + z_0.\tilde{r}_0\{r_0[!inc_0.\bar{u}_0 \parallel \bar{z}_0]\}.\bar{p}_s \parallel \\
&\quad e \parallel \llbracket r_0 = 0 \rrbracket_2 \parallel C^{\langle \alpha-1, \beta, \gamma+1 \rangle} \parallel S \\
&= u_0.(\bar{b} \parallel \overline{p_{i+1}}) + z_0.\tilde{r}_0\{r_0[!inc_0.\bar{u}_0 \parallel \bar{z}_0]\}.\bar{p}_s \parallel \\
&\quad e \parallel r_0[!inc_0.\bar{u}_0 \parallel \bar{z}_0] \parallel C^{\langle \alpha-1, \beta, \gamma+1 \rangle} \parallel S \\
&\longrightarrow \tilde{r}_0\{r_0[!inc_0.\bar{u}_0 \parallel \bar{z}_0]\}.\bar{p}_s \parallel e \parallel r_0[!inc_0.\bar{u}_0] \parallel C^{\langle \alpha-1, \beta, \gamma+1 \rangle} \parallel S \\
&\longrightarrow r_0[!inc_0.\bar{u}_0 \parallel \bar{z}_0] \parallel \bar{p}_s \parallel e \parallel C^{\langle \alpha-1, \beta, \gamma+1 \rangle} \parallel S = P'
\end{aligned}$$

It is easy to see that $P' \equiv \llbracket (s, 0, m_1) \rrbracket_2$, as desired. Observe how the number of copies of \bar{b} remains invariant when the MM is correctly simulated.

- (2) If $(i, m_0, m_1) \not\rightarrow_M$ then i corresponds to the **HALT** instruction. Then, by Definition C.1, its encoding into \mathcal{E}_s^2 is as follows:

$$\begin{aligned}
\llbracket (i, m_0, m_1) \rrbracket_2 &= \bar{p}_i \parallel e \parallel \llbracket r_0 = m_0 \rrbracket_2 \parallel \llbracket r_1 = m_1 \rrbracket_2 \parallel \\
&\quad !p_i.\bar{h}.h.\tilde{r}_0\{r_0[!inc_0.\bar{u}_0 \parallel \bar{z}_0]\}.\tilde{r}_1\{r_1[!inc_1.\bar{u}_1 \parallel \bar{z}_1]\}.\bar{p}_1 \parallel \\
&\quad \prod_{l=1..n, l \neq i} \llbracket (l : I_l) \rrbracket_2 \parallel C^{\langle \alpha, \beta, \gamma \rangle}
\end{aligned}$$

We then have:

$$\begin{aligned}
\llbracket (i, m_0, m_1) \rrbracket_2 &\longrightarrow \equiv \bar{h}.h.\tilde{r}_0\{r_0[!inc_0.\bar{u}_0 \parallel \bar{z}_0]\}.\tilde{r}_1\{r_1[!inc_1.\bar{u}_1 \parallel \bar{z}_1]\}.\bar{p}_1 \parallel \\
&\quad e \parallel \llbracket r_0 = m_0 \rrbracket_2 \parallel C^{\langle \alpha, \beta, \gamma \rangle} \parallel S = P
\end{aligned}$$

where $S = \llbracket r_1 = m_1 \rrbracket_2 \parallel \prod_{l=1}^n \llbracket (l : I_l) \rrbracket_2$ stands for the rest of the system. Starting from P , a possible sequence of reductions is the following:

$$\begin{aligned}
P &\longrightarrow^* \tilde{r}_0\{r_0[!inc_0.\bar{u}_0 \parallel \bar{z}_0]\}.\tilde{r}_1\{r_1[!inc_1.\bar{u}_1 \parallel \bar{z}_1]\}.\bar{p}_1 \parallel \\
&\quad e \parallel \llbracket r_0 = m_0 \rrbracket_2 \parallel C^{\langle \alpha+c, \beta, \gamma-c \rangle} \parallel S = P_1
\end{aligned}$$

where the output on h in P interacted with process $C^{\langle \alpha, \beta, \gamma \rangle}$ so as to replace c outputs on g with c outputs on f . After that, a synchronization on h took place between the evolutions of $C^{\langle \alpha, \beta, \gamma \rangle}$ and of P . We now have:

$$\begin{aligned}
P_1 &\longrightarrow \equiv e \parallel \llbracket r_0 = 0 \rrbracket_2 \parallel \llbracket r_1 = m_1 \rrbracket_2 \\
&\quad \tilde{r}_1\{r_1[!inc_1.\bar{u}_1 \parallel \bar{z}_1]\}.\bar{p}_1 \parallel \prod_{l=1}^n \llbracket (l : I_l) \rrbracket_2 \parallel C^{\langle \alpha+c, \beta, \gamma-c \rangle} \\
&\longrightarrow e \parallel \llbracket r_0 = 0 \rrbracket_2 \parallel \llbracket r_1 = 0 \rrbracket_2 \parallel \bar{p}_1 \parallel \prod_{l=1}^n \llbracket (l : I_l) \rrbracket_2 \parallel C^{\langle \alpha+c, \beta, \gamma-c \rangle}
\end{aligned}$$

which corresponds to $\llbracket (1, 0, 0) \rrbracket_2$. In turn, it can be seen that $\llbracket (i, m_0, m_1) \rrbracket_2 \Downarrow_{\bar{p}_1}^1$. \square

Remark C.4. It is instructive to identify the exact point in which an erroneous computation can be made when mimicking the behavior of a decrement-and-jump instruction. Consider again the process P' , as analyzed in the case M-DEC above:

$$\begin{aligned}
P' &= u_0.(\bar{b} \parallel \overline{p_{i+1}}) + z_0.\tilde{r}_0\{r_0[!inc_0.\bar{u}_0 \parallel \bar{z}_0]\}.\bar{p}_s \parallel \\
&\quad e \parallel r_0[!inc_0.\bar{u}_0 \parallel \prod_{l=1}^{m_0} \bar{u}_0 \parallel \bar{z}_0] \parallel C^{\langle \alpha-1, \beta, \gamma+1 \rangle} \parallel S
\end{aligned}$$

where $S = \llbracket r_1 = m_1 \rrbracket_2 \parallel \prod_{l=1}^n \llbracket (l : I_l) \rrbracket_2$ stands for the rest of the system. Above, we analyzed the correct computation from P' , namely a synchronization on u_0 :

$$P' \longrightarrow \overline{p_{i+1}} \parallel e \parallel r_0[!inc_0. \overline{u_0} \parallel \prod_{i=0}^{m_0-1} \overline{u_0} \parallel \overline{z_0}] \parallel C^{\langle \alpha-1, \beta+1, \gamma+1 \rangle} \parallel S = P''$$

with $P'' \equiv \llbracket (i+1, m_0-1, m_1) \rrbracket_2$. The erroneous computation takes place when there is a synchronization on z_0 , rather than on u_0 . We then have:

$$\begin{aligned} P' &\longrightarrow \tilde{r}_0\{r_0[!inc_0. \overline{u_0} \parallel \overline{z_0}]\}. \overline{p_s} \parallel e \parallel \\ &\quad r_0[!inc_0. \overline{u_0} \parallel \prod_{i=0}^{m_0} \overline{u_0}] \parallel C^{\langle \alpha-1, \beta, \gamma+1 \rangle} \parallel S \\ &\longrightarrow \equiv \overline{p_s} \parallel e \parallel r_0[!inc_0. \overline{u_0} \parallel \overline{z_0}] \parallel C^{\langle \alpha-1, \beta, \gamma+1 \rangle} \parallel S = P''' \end{aligned}$$

with $P''' \equiv \llbracket (s, 0, m_1) \rrbracket_2$. The side effect of the above erroneous computation can be seen on the number of copies of \bar{b} that remain after the (erroneous) synchronization on z_0 . In fact, while a correct computation (as P'' above) increases in one the number of such copies, in an incorrect computation (as P''' above) the number of copies of \bar{b} remains invariant. Notice also that copies of \bar{b} can be only produced at the beginning of the execution of the encoding of the MM. This is significant since, as discussed at the end of the case M-INC, the number of copies of \bar{b} has a direct influence on potential deadlocks of the encoding of a MM.

Lemma C.5 (Completeness). *Let N be a MM, if N terminates then $\llbracket N \rrbracket_2 \Downarrow_e^\omega$.*

Proof. Recall that N is said to terminate if there exists a computation

$$(1, 0, 0) \longrightarrow_M^* (h, 0, 0)$$

such that $(h : \text{HALT})$. Lemma C.3 guarantees the existence of a process P such that $\llbracket (1, 0, 0) \rrbracket_2 \longrightarrow^* P \equiv \llbracket (h, 0, 0) \rrbracket_2$, with $P \Downarrow_{p_1}^1$. This ensures that every time that the encoding of N reaches HALT the simulation is restarted. Therefore, termination of N ensures that $\llbracket N \rrbracket_2$ has an infinite computation: since the encoding always exhibits barb e , we can conclude that $\llbracket N \rrbracket_2 \Downarrow_e^\omega$. \square

Lemma C.6 (Soundness). *Let N be a MM. If N does not terminate then $\llbracket N \rrbracket_2 \not\Downarrow_e^\omega$.*

Proof. It is enough to prove that if N does not terminate (that is, if N does not reach a HALT instruction) then all the computations of $\llbracket N \rrbracket_2$ are finite. Since the encoding can mimic the behavior of N both correctly and incorrectly, we have two possible cases:

- (1) In the first case, the simulation of $\llbracket N \rrbracket_2$ is correct and no erroneous steps are introduced. Notice that at every instruction an output on f is consumed permanently: these copies of \bar{f} are only recreated when invoking a HALT instruction, which converts every \bar{g} into a \bar{f} . Since a HALT instruction is never reached, new copies of \bar{f} are never recreated, and the computation of process $\llbracket N \rrbracket_2$ has necessarily to be finite.
- (2) In the second case, the simulation is not correct and one or more wrong guesses occurred in the simulation of a decrement-and-jump instruction. Here, in addition to the possibility of deadlocks described in Item (1) above, erroneous computations constitute another source of deadlocks. In fact, as detailed in Remark C.4, for each one of such wrong guesses a copy of \bar{b} is permanently lost. An arbitrary number of wrong guesses may thus lead to a state in which there are no outputs on b . As discussed at the end of the case of the M-INC in the proof of Lemma C.3, the encoding of an increment instruction reaches a deadlock if a copy of \bar{b} is not available. This means that wrong

guesses in simulating a decrement-and-jump instruction may induce deadlocks when simulating an increment instruction.

Hence, as all the computations of $\llbracket N \rrbracket_2$ are finite, therefore $\llbracket N \rrbracket_2$ barb e cannot be exposed an infinite number of times. \square

We are now ready to repeat the statement of Lemma 7.27, in Page 37:

Lemma C.7 (7.27). *Let N be a MM. N terminates iff $\llbracket N \rrbracket_2 \downarrow_e^\omega$.*

Proof. It follows directly from Lemmas C.5 and C.6. \square

APPENDIX D. PROOFS FROM SECTION 8

D.1. Proof of Lemma 8.2. The proof relies on two results: completeness (Lemma D.3) and soundness (Lemma D.4). The proof is very similar to the one presented for the case of \mathcal{E}_s^2 , and considerations concerning the handling of resources (i.e., process CONTROL) are exactly the same. Hence, Remarks C.2 and C.4 are valid also in this proof.

We first introduce the notion of encoding of a MM configuration into \mathcal{E}_d^3 .

Definition D.1. Let N be a MM with registers r_j ($j \in \{0, 1\}$) and instructions $(1 : I_1), \dots, (n : I_n)$. The encoding of a configuration (i, m_0, m_1) of N , denoted $\llbracket (i, m_0, m_1) \rrbracket_3$, is defined as:

$$\bar{p}_i \parallel e \parallel \llbracket r_0 = m_0 \rrbracket_3 \parallel \llbracket r_1 = m_1 \rrbracket_3 \parallel \prod_{i=1}^n \llbracket (i : I_i) \rrbracket_3 \parallel C^{\langle \alpha, \beta, \gamma \rangle}$$

where

- $C^{\langle \alpha, \beta, \gamma \rangle} \stackrel{\text{def}}{=} \prod^\alpha \bar{f} \parallel \prod^\beta \bar{b} \parallel \prod^\gamma \bar{g} \parallel !a. (\bar{f} \parallel \bar{b} \parallel \bar{a}) \parallel !h. (g. \bar{f} \parallel \bar{h})$, with $\alpha, \beta, \gamma \geq 0$;
- $\llbracket (i : I_i) \rrbracket_3, \dots, \llbracket (n : I_n) \rrbracket_3$ are as in Table 4;
- $\llbracket r_j = m_j \rrbracket_3$ stands for $r_j [\prod^{m_j} U_j \parallel \text{Reg}_j \parallel c_j[G^{(\delta)}]]$ with
 - $\text{Reg}_j = !inc_j. \tilde{c}_j\{c_j[\bullet]\}. \overline{ack}. u_j. \tilde{c}_j\{c_j[\bullet]\}. \overline{ack}$ (as in Table 4)
 - $U_j \stackrel{\text{def}}{=} u_j. \tilde{c}_j\{c_j[\bullet]\}. \overline{ack}$
 - $G_j^{(\delta)} \stackrel{\text{def}}{=} \text{Reg}_j \parallel \prod^\delta U_j$

Similarly as before, in addition to the encodings of registers and instructions, the encoding of a MM configuration includes a number of resources \bar{f} and \bar{b} which are always available during the execution of the machine. These are represented by process $C^{\langle \alpha, \beta, \gamma \rangle}$, which abstracts the evolution of process CONTROL in Table 4, and the resources that it produces and maintains (namely, α copies of \bar{f} , β copies of \bar{b} , and γ copies of \bar{g}). In addition, the encoding of register j in \mathcal{E}_d^3 includes a “garbage” process $G_j^{(\delta)}$ representing residual processes which are accumulated during the execution of the encoding; as we will see, every interaction with such a garbage process will result into a deadlocked process.

We prove that given a MM N there exists a computation of process $\llbracket N \rrbracket_3$ which correctly mimics its behavior. We remind that Remark C.2 applies to this case too.

Lemma D.2. *Let (i, m_0, m_1) be a configuration of a MM N .*

(1) *If $(i, m_0, m_1) \longrightarrow_M (i', m'_0, m'_1)$ then, for some process P , it holds that*

$$\llbracket (i, m_0, m_1) \rrbracket_3 \longrightarrow^* P \equiv \llbracket (i', m'_0, m'_1) \rrbracket_3$$

(2) If $(i, m_0, m_1) \not\rightarrow_M$ then $\llbracket (i, m_0, m_1) \rrbracket_3 \Downarrow_{\overline{p}_1}^1$.

Proof.

(1) We proceed by a case analysis on the instruction performed by the Minsky machine. Hence, we distinguish three cases corresponding to the behaviors associated to rules M-INC, M-DEC, and M-JMP. Without loss of generality, we restrict our analysis to operations on register r_0 .

Case M-INC: We have a Minsky configuration (i, m_0, m_1) with $(i : \text{INC}(r_0))$. By Definition D.1, its encoding into \mathcal{E}_d^3 is as follows:

$$\begin{aligned} \llbracket (i, m_0, m_1) \rrbracket_3 &= \overline{p}_i \parallel e \parallel \llbracket r_0 = m_0 \rrbracket_3 \parallel \llbracket r_1 = m_1 \rrbracket_3 \parallel C^{\langle \alpha, \beta, \gamma \rangle} \parallel \\ &\quad !p_i. f. (\overline{g} \parallel b. \overline{inc}_0. \overline{ack}. \overline{p}_{i+1}) \parallel \prod_{l=1..n, l \neq i} \llbracket (l : I_l) \rrbracket_3 \end{aligned}$$

We then have: $\llbracket (i, m_0, m_1) \rrbracket_3 \longrightarrow R$ where

$$R = \llbracket r_0 = m_0 \rrbracket_3 \parallel f. (\overline{g} \parallel b. \overline{inc}_0. \overline{ack}. \overline{p}_{i+1}) \parallel C^{\langle \alpha, \beta, \gamma \rangle} \parallel S,$$

and $S = e \parallel \llbracket r_1 = m_1 \rrbracket_3 \parallel \prod_{l=1}^n \llbracket (l : I_l) \rrbracket_3$ stands for the rest of the system. Starting from R , a possible sequence of reductions is the following:

$$\begin{aligned} R &\longrightarrow \llbracket r_0 = m_0 \rrbracket_3 \parallel b. \overline{inc}_0. \overline{ack}. \overline{p}_{i+1} \parallel C^{\langle \alpha-1, \beta, \gamma+1 \rangle} \parallel S \\ &= r_0 \left[\prod_{U_0}^{m_0} U_0 \parallel !inc_0. \tilde{c}_0\{c_0[\bullet]\}. \overline{ack}. u_0. \tilde{c}_0\{c_0[\bullet]\}. \overline{ack} \parallel c_0[G_0^{(\delta)}] \right] \parallel \\ &\quad b. \overline{inc}_0. \overline{ack}. \overline{p}_{i+1} \parallel C^{\langle \alpha-1, \beta, \gamma+1 \rangle} \parallel S \\ &\longrightarrow r_0 \left[\prod_{U_0}^{m_0} U_0 \parallel !inc_0. \tilde{c}_0\{c_0[\bullet]\}. \overline{ack}. u_0. \tilde{c}_0\{c_0[\bullet]\}. \overline{ack} \parallel c_0[G_0^{(\delta)}] \right] \parallel \\ &\quad \overline{inc}_0. \overline{ack}. \overline{p}_{i+1} \parallel C^{\langle \alpha-1, \beta, \gamma+1 \rangle} \parallel S = R' \\ &\longrightarrow r_0 \left[\prod_{U_0}^{m_0} U_0 \parallel \tilde{c}_0\{c_0[\bullet]\}. \overline{ack}. u_0. \tilde{c}_0\{c_0[\bullet]\}. \overline{ack} \parallel Reg_0 \parallel c_0[G_0^{(\delta)}] \right] \parallel \\ &\quad \parallel \overline{ack}. \overline{p}_{i+1} \parallel C^{\langle \alpha-1, \beta, \gamma+1 \rangle} \parallel S \\ &\longrightarrow r_0 \left[\prod_{U_0}^{m_0} U_0 \parallel \overline{ack}. u_0. \tilde{c}_0\{c_0[\bullet]\}. \overline{ack} \parallel Reg_0 \parallel c_0[G_0^{(\delta)}] \right] \parallel \\ &\quad \parallel \overline{ack}. \overline{p}_{i+1} \parallel C^{\langle \alpha-1, \beta, \gamma+1 \rangle} \parallel S \\ &\longrightarrow r_0 \left[\prod_{U_0}^{m_0} U_0 \parallel u_0. \tilde{c}_0\{c_0[\bullet]\}. \overline{ack} \parallel Reg_0 \parallel c_0[G_0^{(\delta)}] \right] \parallel \\ &\quad \parallel \overline{p}_{i+1} \parallel C^{\langle \alpha-1, \beta, \gamma+1 \rangle} \parallel S \\ &= r_0 \left[\prod_{U_0}^{m_0+1} U_0 \parallel Reg_0 \parallel c_0[G_0^{(\delta)}] \right] \parallel \overline{p}_{i+1} \parallel C^{\langle \alpha-1, \beta, \gamma+1 \rangle} \parallel S = P \end{aligned}$$

It is easy to see that $P \equiv \llbracket (i+1, m_0+1, m_1) \rrbracket_3$, as desired. Observe how the number of resources changes: in the first reduction, a copy of \overline{f} is consumed, and a copy of \overline{g} is released in its place. Notice that we are assuming that $\beta > 0$, that is, that there is at least one copy of \overline{b} . In fact, since the instruction only takes place after a

synchronization on \bar{b} (i.e., the second reduction above) the presence of at least one copy of \bar{b} in $C^{(\alpha-1, \beta, \gamma+1)}$ is essential to avoid deadlocks. For the same reason, it is interesting to observe that in R' the computation can only evolve if \bar{inc}_0 synchronizes with the replicated input process inc_0 inside r_0 . Had it synchronized with the input on inc_0 inside c_0 , the simulation would have reached a deadlock state, as there are no other adaptable processes at c_0 inside it.

Case M-Dec: We have a Minsky configuration (i, m_0, m_1) with $m_0 > 0$ and $(i : \text{DEC}(r_0, s))$. By Definition D.1, its encoding into \mathcal{E}_d^3 is as follows:

$$\begin{aligned} \llbracket (i, m_0, m_1) \rrbracket_3 &= \bar{p}_i \parallel e \parallel \llbracket r_0 = m_0 \rrbracket_3 \parallel \llbracket r_1 = m_1 \rrbracket_3 \parallel C^{(\alpha, \beta, \gamma)} \parallel \\ &\quad !p_i. f. (\bar{g} \parallel (\bar{u}_0. \text{ack}. (\bar{b} \parallel \bar{p}_{i+1}) + \\ &\quad \tilde{c}_0\{\bullet\}. \tilde{r}_0\{r_0[\text{Reg}_0 \parallel c_0[\bullet]]\}. \bar{p}_s)) \parallel \prod_{l=1..n, l \neq i} \llbracket (l : I_l) \rrbracket_3 \end{aligned}$$

We then have:

$$\begin{aligned} \llbracket (i, m_0, m_1) \rrbracket_3 &\longrightarrow \llbracket r_0 = m_0 \rrbracket_3 \parallel C^{(\alpha, \beta, \gamma)} \parallel \\ &\quad f. (\bar{g} \parallel (\bar{u}_0. \text{ack}. (\bar{b} \parallel \bar{p}_{i+1}) + \\ &\quad \tilde{c}_0\{\bullet\}. \tilde{r}_0\{r_0[\text{Reg}_0 \parallel c_0[\bullet]]\}. \bar{p}_s)) \parallel S = R \end{aligned}$$

where $S = e \parallel \llbracket r_1 = m_1 \rrbracket_3 \parallel \prod_{l=1}^n \llbracket (l : I_l) \rrbracket_3$ stands for the rest of the system. Starting from R , a possible sequence of reductions is the following:

$$\begin{aligned} R &\longrightarrow \llbracket r_0 = m_0 \rrbracket_3 \parallel C^{(\alpha-1, \beta, \gamma+1)} \parallel \\ &\quad ((\bar{u}_0. \text{ack}. (\bar{b} \parallel \bar{p}_{i+1}) + \tilde{c}_0\{\bullet\}. \tilde{r}_0\{r_0[\text{Reg}_0 \parallel c_0[\bullet]]\}. \bar{p}_s)) \parallel S \\ &= r_0 \left[\prod_{U_0}^{m_0-1} U_0 \parallel u_0. \tilde{c}_0\{c_0[\bullet]\}. \overline{\text{ack}} \parallel \text{Reg}_0 \parallel c_0[G_0^{(\delta)}] \right] \parallel C^{(\alpha-1, \beta, \gamma+1)} \\ &\quad \parallel ((\bar{u}_0. \text{ack}. (\bar{b} \parallel \bar{p}_{i+1}) + \tilde{c}_0\{\bullet\}. \tilde{r}_0\{r_0[\text{Reg}_0 \parallel c_0[\bullet]]\}. \bar{p}_s)) \parallel S = R' \\ &\longrightarrow r_0 \left[\prod_{U_0}^{m_0-1} U_0 \parallel \tilde{c}_0\{c_0[\bullet]\}. \overline{\text{ack}} \parallel \text{Reg}_0 \parallel c_0[G_0^{(\delta)}] \right] \parallel C^{(\alpha-1, \beta, \gamma+1)} \parallel \\ &\quad \text{ack}. (\bar{b} \parallel \bar{p}_{i+1}) \parallel S \\ &\longrightarrow r_0 \left[\prod_{U_0}^{m_0-1} U_0 \parallel \overline{\text{ack}} \parallel \text{Reg}_0 \parallel c_0[G_0^{(\delta)}] \right] \parallel C^{(\alpha-1, \beta, \gamma+1)} \parallel \\ &\quad \text{ack}. (\bar{b} \parallel \bar{p}_{i+1}) \parallel S \\ &\longrightarrow r_0 \left[\prod_{U_0}^{m_0-1} U_0 \parallel \text{Reg}_0 \parallel c_0[G_0^{(\delta)}] \right] \parallel C^{(\alpha-1, \beta, \gamma+1)} \parallel \bar{b} \parallel \bar{p}_{i+1} \parallel S \\ &\equiv r_0 \left[\prod_{U_0}^{m_0-1} U_0 \parallel \text{Reg}_0 \parallel c_0[G_0^{(\delta)}] \right] \parallel C^{(\alpha-1, \beta+1, \gamma+1)} \parallel \bar{p}_{i+1} \parallel S = P \end{aligned}$$

It is easy to see that $P \equiv \llbracket (i+1, m_0-1, m_1) \rrbracket_3$, as desired. Observe how in the last reduction the presence of at least a copy of \bar{u}_0 in r_0 is fundamental for releasing both an extra copy of \bar{b} and the trigger for the next instruction. Notice also that if \bar{u}_0 in R' synchronizes with u_0 inside adaptable process c_0 then, as in the case of the increment, the simulation would be deadlocked.

Case M-Jmp: We have a Minsky configuration $(i, 0, m_1)$ and $(i : \text{DEC}(r_0, s))$. By Definition D.1, its encoding into \mathcal{E}_d^3 is as follows:

$$\begin{aligned} \llbracket (i, m_0, m_1) \rrbracket_3 &= \bar{p}_i \parallel e \parallel \llbracket r_0 = 0 \rrbracket_3 \parallel \llbracket r_1 = m_1 \rrbracket_3 \parallel C^{(\alpha, \beta, \gamma)} \parallel \\ &\quad !p_i. f. (\bar{g} \parallel (\bar{u}_0. \text{ack}. (\bar{b} \parallel \bar{p}_{i+1})) + \\ &\quad \quad \tilde{c}_0\{\bullet\}. \tilde{r}_0\{r_0[\text{Reg}_0 \parallel c_0[\bullet]]\}. \bar{p}_s)) \parallel \\ &\quad \prod_{l=1..n, l \neq i} \llbracket (l : I_l) \rrbracket_3 \end{aligned}$$

We then have:

$$\begin{aligned} \llbracket (i, m_0, m_1) \rrbracket_3 &\longrightarrow \llbracket r_0 = 0 \rrbracket_3 \parallel C^{(\alpha, \beta, \gamma)} \parallel f. (\bar{g} \parallel (\bar{u}_0. \text{ack}. (\bar{b} \parallel \bar{p}_{i+1})) + \\ &\quad \tilde{c}_0\{\bullet\}. \tilde{r}_0\{r_0[\text{Reg}_0 \parallel c_0[\bullet]]\}. \bar{p}_s)) \parallel S = R \end{aligned}$$

where $S = e \parallel \llbracket r_1 = m_1 \rrbracket_3 \parallel \prod_{l=1}^n \llbracket (l : I_l) \rrbracket_3$ stands for the rest of the system. Starting from R , a possible sequence of reductions is the following:

$$\begin{aligned} R &\longrightarrow \llbracket r_0 = 0 \rrbracket_3 \parallel C^{(\alpha-1, \beta, \gamma+1)} \parallel \\ &\quad (\bar{u}_0. \text{ack}. (\bar{b} \parallel \bar{p}_{i+1})) + \tilde{c}_0\{\bullet\}. \tilde{r}_0\{r_0[\text{Reg}_0 \parallel c_0[\bullet]]\}. \bar{p}_s) \parallel S \\ &= r_0[\text{Reg}_0 \parallel c_0[G_0^{(\delta)}]] \parallel C^{(\alpha-1, \beta, \gamma+1)} \parallel \\ &\quad (\bar{u}_0. \text{ack}. (\bar{b} \parallel \bar{p}_{i+1})) + \tilde{c}_0\{\bullet\}. \tilde{r}_0\{r_0[\text{Reg}_0 \parallel c_0[\bullet]]\}. \bar{p}_s) \parallel S \\ &\longrightarrow r_0[\text{Reg}_0 \parallel G_0^{(\delta)}] \parallel C^{(\alpha-1, \beta, \gamma+1)} \parallel \tilde{r}_0\{r_0[\text{Reg}_0 \parallel c_0[\bullet]]\}. \bar{p}_s \parallel S \\ &\longrightarrow \equiv r_0[\text{Reg}_0 \parallel c_0[\text{Reg}_0 \parallel G_0^{(\delta)}]] \parallel C^{(\alpha-1, \beta, \gamma+1)} \parallel \bar{p}_s \parallel S \\ &\simeq r_0[\text{Reg}_0 \parallel c_0[G_0^{(\delta)}]] \parallel C^{(\alpha-1, \beta, \gamma+1)} \parallel \bar{p}_s \parallel S = P \end{aligned}$$

It is easy to see that $P \equiv \llbracket (s, 0, m_1) \rrbracket_3$, as desired. Notice that the first reduction results from a synchronization on f . The second reduction arises from an update action on c_0 , which removes that “boundary” for $G_0^{(\delta)}$. Finally, the third reduction is an update action on r_0 . We use \simeq to denote the extension of structural congruence with the axiom $! \pi. P \parallel ! \pi. P = ! \pi. P$.

- (2) If $(i, m_0, m_1) \rightarrow_M$ then i corresponds to the **HALT** instruction. Then, by Definition D.1, its encoding into \mathcal{E}_d^3 is as follows:

$$\begin{aligned} \llbracket (i, m_0, m_1) \rrbracket_3 &= \bar{p}_i \parallel e \parallel \llbracket r_0 = m_0 \rrbracket_3 \parallel \llbracket r_1 = m_1 \rrbracket_3 \parallel C^{(\alpha, \beta, \gamma)} \parallel \\ &\quad !p_i. \bar{h}. h. \tilde{c}_0\{\bullet\}. \tilde{r}_0\{r_0[\text{Reg}_0 \parallel c_0[\bullet]]\}. \\ &\quad \tilde{c}_1\{\bullet\}. \tilde{r}_1\{r_1[\text{Reg}_1 \parallel c_1[\bullet]]\}. \bar{p}_1 \parallel \prod_{l=1..n, l \neq i} \llbracket (l : I_l) \rrbracket_3 \end{aligned}$$

We then have: $\llbracket (i, m_0, m_1) \rrbracket_3 \longrightarrow R$ where

$$\begin{aligned} R &= \llbracket r_0 = m_0 \rrbracket_3 \parallel \llbracket r_1 = m_1 \rrbracket_3 \parallel C^{(\alpha, \beta, \gamma)} \parallel \\ &\quad \bar{h}. h. \tilde{c}_0\{\bullet\}. \tilde{r}_0\{r_0[\text{Reg}_0 \parallel c_0[\bullet]]\}. \tilde{c}_1\{\bullet\}. \tilde{r}_1\{r_1[\text{Reg}_1 \parallel c_1[\bullet]]\}. \bar{p}_1 \parallel S \end{aligned}$$

where $S = e \parallel \prod_{l=1}^n \llbracket (l : I_l) \rrbracket_3$ stands for the rest of the system. Starting from R , a possible sequence of reductions is the following:

$$\begin{aligned} R &\longrightarrow^* \llbracket r_0 = m_0 \rrbracket_3 \parallel \llbracket r_1 = m_1 \rrbracket_3 \parallel C^{\langle \alpha+c, \beta, \gamma-c \rangle} \parallel S \parallel \\ &\quad \tilde{c}_0\{\bullet\}. \tilde{r}_0\{r_0[Reg_0 \parallel c_0[\bullet]]\}. \tilde{c}_1\{\bullet\}. \tilde{r}_1\{r_1[Reg_1 \parallel c_1[\bullet]]\}. \bar{p}_1 = R_1 \end{aligned}$$

where the output on h in R interacted with process $C^{\langle \alpha, \beta, \gamma \rangle}$ so as to replace c outputs on g with c outputs on f . After that, a synchronization on h took place between the evolutions of $C^{\langle \alpha, \beta, \gamma \rangle}$ and of R . We now have:

$$\begin{aligned} R_1 &\longrightarrow r_0 \left[\prod_{i=0}^{m_0} U_0 \parallel Reg_0 \parallel G_0^{(\delta_0)} \right] \parallel r_1 \left[\prod_{i=0}^{m_1} U_1 \parallel Reg_1 \parallel c_1[G_1^{(\delta_1)}] \right] \parallel S \parallel \\ &\quad \tilde{r}_0\{r_0[Reg_0 \parallel c_0[\bullet]]\}. \tilde{c}_1\{\bullet\}. \tilde{r}_1\{r_1[Reg_1 \parallel c_1[\bullet]]\}. \bar{p}_1 \parallel C^{\langle \alpha+c, \beta, \gamma-c \rangle} \\ &\longrightarrow \simeq r_0 \left[Reg_0 \parallel c_0[G_0^{(\delta_0+m_0)}] \right] \parallel r_1 \left[\prod_{i=0}^{m_1} U_1 \parallel Reg_1 \parallel c_1[G_1^{(\delta_1)}] \right] \parallel S \parallel \\ &\quad \tilde{c}_1\{\bullet\}. \tilde{r}_1\{r_1[Reg_1 \parallel c_1[\bullet]]\}. \bar{p}_1 \parallel C^{\langle \alpha+c, \beta, \gamma-c \rangle} \\ &\longrightarrow r_0[Reg_0 \parallel c_0[G_0^{(\delta_0+m_0)}]] \parallel r_1 \left[\prod_{i=0}^{m_1} U_1 \parallel Reg_1 \parallel G_1^{(\delta_1)} \right] \parallel S \parallel \\ &\quad \tilde{r}_1\{r_1[Reg_1 \parallel c_1[\bullet]]\}. \bar{p}_1 \parallel C^{\langle \alpha+c, \beta, \gamma-c \rangle} \\ &\longrightarrow \simeq r_0 \left[Reg_0 \parallel c_0[G_0^{(\delta_0+m_0)}] \right] \parallel r_1 \left[Reg_1 \parallel c_1[G_1^{(\delta_1+m_1)}] \right] \parallel S \parallel \\ &\quad \bar{p}_1 \parallel C^{\langle \alpha+c, \beta, \gamma-c \rangle} \end{aligned}$$

which is easily seen to correspond to $\llbracket (1, 0, 0) \rrbracket_3$, and thus $\llbracket (i, m_0, m_1) \rrbracket_3 \Downarrow_{\bar{p}_1}^1$. \square

It is straightforward to see that Remark C.4 is valid for $\llbracket N \rrbracket_3$ too. Unsurprisingly, the proof concludes following the same lines of the proof of Lemma 7.27.

Lemma D.3 (Completeness). *Let N be a MM. If N terminates then $\llbracket N \rrbracket_3 \Downarrow_e^\omega$.*

Proof. Recall that N is said to terminate if there exists a computation

$$(1, 0, 0) \longrightarrow_M^* (h, 0, 0)$$

such that $(h : \text{HALT})$. Lemma D.2 guarantees the existence of a process P such that $\llbracket (1, 0, 0) \rrbracket_3 \longrightarrow^* P \equiv \llbracket (h, 0, 0) \rrbracket_3$, with $P \Downarrow_{\bar{p}_1}^1$. This ensures that every time that the encoding of N reaches HALT the simulation is restarted. Therefore, termination of N ensures that $\llbracket N \rrbracket_3$ has an infinite computation: since the encoding always exhibits barb e , we can conclude that $\llbracket N \rrbracket_3 \Downarrow_e^\omega$. \square

Lemma D.4 (Soundness). *Let N be a MM. If N does not terminate then $\llbracket N \rrbracket_3 \not\Downarrow_e^\omega$.*

Proof. It is enough to prove that if N does not terminate (that is, if N does not reach a HALT instruction) then all the computations of $\llbracket N \rrbracket_3$ are finite. Since the encoding can mimic the behavior of N both correctly and incorrectly, we have two possible cases:

- (1) In the first case, the simulation of $\llbracket N \rrbracket_3$ is correct and no erroneous steps are introduced. Notice that at every instruction an output on f is consumed permanently: these copies of \bar{f} are only recreated when invoking a HALT instruction, which converts every \bar{g} into a \bar{f} . Since a HALT instruction is never reached, new copies of \bar{f} are never recreated, and the computation of process $\llbracket N \rrbracket_3$ has necessarily to be finite.

(2) In the second case, the simulation is not correct and one or more wrong guesses occurred in the simulation of a decrement-and-jump instruction. Here, in addition to the possibility of deadlocks described in Item (1) above, erroneous computations constitute another source of deadlocks. In fact, as detailed in Remark C.4, for each one of such wrong guesses a copy of \bar{b} is permanently lost. Finally, the last source of error is represented by a wrong synchronization with inc_j (in case of an increment) or u_j (in case of a decrement) inside the adaptable process c_j . As described above, those wrong synchronizations lead to a deadlock. An arbitrary number of wrong guesses may thus lead to a state in which there are no outputs on b . As discussed at the end of the case of the M-INC in the proof of Lemma D.2, the encoding of an increment instruction reaches a deadlock if a copy of \bar{b} is not available. This means that wrong guesses in simulating a decrement-and-jump instruction may induce deadlocks when simulating an increment instruction.

Hence, as all the computations of $\llbracket N \rrbracket_3$ are finite, therefore $\llbracket N \rrbracket_3 \text{ barb } e$ cannot be exposed an infinite number of times. \square

We are now ready to repeat the statement of Lemma 8.2, in Page 39:

Lemma D.5 (8.2). *Let N be a MM. N terminates iff $\llbracket N \rrbracket_3 \Downarrow_e^\omega$.*

Proof. It follows directly from Lemmas D.3 and D.4. \square

D.2. Proof of Lemma 8.6. Here we prove that given an \mathcal{E}_s^3 process P its associated Petri net representation $\text{PN}(P, \emptyset)$ faithfully preserves its behavior. We need some auxiliary propositions and definitions. The following proposition states how to build a Petri net for the parallel composition of two processes starting from the Petri nets of the two processes.

Proposition D.6. *Let P_1 and P_2 be two \mathcal{E}_s^3 processes with associated Petri nets $\text{PN}(P_1, \emptyset)$ and $\text{PN}(P_2, \emptyset)$, as in Definition 8.4. Then, the Petri net $\text{PN}(P_1 \parallel P_2, \emptyset)$ is defined as:*

$$\text{PN}(P_1 \parallel P_2, \emptyset) = (\text{Places}(P_1 \parallel P_2), \text{Trans}(P_1 \parallel P_2), \text{Init}(P_1 \parallel P_2))$$

where

$$\begin{aligned} \text{Places}(P_1 \parallel P_2, \emptyset) &= \text{Places}(P_1, \emptyset) \cup \text{Places}(P_2, \emptyset), \\ \text{Trans}(P_1 \parallel P_2, \emptyset) &= \text{Trans}(P_1, \emptyset) \cup \text{Trans}(P_2, \emptyset) \cup T, \\ \text{Init}(P_1 \parallel P_2) &= \text{Init}(P_1) \uplus \text{Init}(P_2) \end{aligned}$$

with T representing the set of instances of transition schemata in Table 5 that become possible due to the interplay of places in $\text{Places}(P_1, \emptyset)$ and in $\text{Places}(P_2, \emptyset)$.

Proof. Immediate from the definitions. \square

Similarly, the next proposition shows how to obtain the Petri net associated to $a[P]$ starting from the one of P .

Proposition D.7. *Let P be an \mathcal{E}_s^3 process with associated Petri net $\text{PN}(P, \emptyset)$ as in Definition 8.4. Then, the Petri net $\text{PN}(a[P], \emptyset)$ is defined as:*

$$\text{PN}(a[P], \emptyset) = (\text{Places}(a[P]), \text{Trans}(a[P]), \text{Init}(a[P]))$$

where:

- $\text{Places}(a[P], \emptyset) = \{\langle Q, a\sigma \rangle \mid \langle Q, \sigma \rangle \in \text{Places}(P, \emptyset)\} \cup \{a\sigma \mid \sigma \in \text{Places}(P, \emptyset)\} \cup \{a\}$.

- $\text{Trans}(a[P])$ is obtained from $\text{Trans}(P, \emptyset)$ by replacing places in $\text{Places}(P, \emptyset)$ with places in $\text{Places}(a[P], \emptyset)$, as defined above.
- $\text{Init}(a[P])$ is obtained from $\text{Init}(P)$ by (i) replacing places in $\text{Places}(P, \emptyset)$ with places in $\text{Places}(a[P], \emptyset)$, as defined above, and (ii) adding a token in the place for the adaptable process a .

Proof. Immediate from the definitions. \square

Lemma D.8. *Let P and $(\text{Places}(P, \emptyset), \text{Trans}(P, \emptyset), \text{Init}(P))$ be an \mathcal{E}_s^3 process and its associated Petri net, as in Definition 8.4. Then we have:*

- (1) *If $P \longrightarrow P'$ then $\text{dec}_\varepsilon(P) \uplus \{go\} \rightarrow \text{dec}_\varepsilon(P') \uplus \{go\}$.*
- (2) *If $\text{dec}_\varepsilon(P) \uplus \{go\} \rightarrow m \uplus \{go\}$ then, for some P' , $P \longrightarrow P'$ and $\text{dec}_\varepsilon(P') = m$*

Proof. The proof of (1) proceeds by induction on the derivation tree of $P \longrightarrow P'$, with a case analysis on the last applied rule. There are seven cases to check.

Case (Act1): Then we have $P = P_1 \parallel P_2$ and:

$$\frac{P_1 \rightarrow P'_1}{P_1 \parallel P_2 \rightarrow P'_1 \parallel P_2}$$

By inductive hypothesis, we have $\text{dec}_\varepsilon(P_1) \uplus \{go\} \rightarrow \text{dec}_\varepsilon(P'_1) \uplus \{go\}$. By Proposition D.6, $\text{Trans}(P_1, \emptyset) \subseteq \text{Trans}(P_1 \parallel P_2, \emptyset)$. Since by Definition 8.4 $\text{dec}_\varepsilon(P_1 \parallel P_2) = \text{dec}_\varepsilon(P_1) \uplus \text{dec}_\varepsilon(P_2)$, then we can conclude that $\text{dec}_\varepsilon(P) \uplus \{go\} \rightarrow \text{dec}_\varepsilon(P') \uplus \{go\}$ with $\text{dec}_\varepsilon(P') = \text{dec}_\varepsilon(P'_1) \uplus \text{dec}_\varepsilon(P_2)$. This concludes the proof for this case.

Case (Act2): Analogous to the case for (ACT1) and omitted.

Case (Loc): Then we have $P = a[P_1]$ and

$$\frac{P_1 \rightarrow P'_1}{a[P_1] \rightarrow a[P'_1]}$$

By inductive hypothesis, we have $\text{dec}_\varepsilon(P_1) \uplus \{go\} \rightarrow \text{dec}_\varepsilon(P'_1) \uplus \{go\}$. Proposition D.7 states that $\text{Trans}(P, \emptyset)$ is obtained by extending the addresses of places in $\text{Trans}(P_1, \emptyset)$ with name a . Since by Definition 8.4 $\text{dec}_\varepsilon(a[P_1]) = \text{dec}_a(P_1) \uplus \{a\}$ then we can conclude that $\text{dec}_\varepsilon(P) \uplus \{go\} \rightarrow \text{dec}_\varepsilon(P') \uplus \{go\}$ with $\text{dec}_\varepsilon(P') = \text{dec}_a(P'_1) \uplus \{a\}$. This concludes the proof for this case.

Cases (Tau1)-(Tau2): Then $P \equiv C_1[A] \parallel C_2[B]$, where C_1, C_2 are monadic contexts as in Definition 7.11. Moreover, A is either $!b. Q$ or $\sum_{i \in I} \pi_i. Q_i$ with $\pi_l = b$, for some $l \in I$, and B is either $!\bar{b}. R$ or $\sum_{i \in I} \pi_i. R_i$ with $\pi_l = \bar{b}$, for some $l \in I$.

We consider only the case in which $A = \sum_{i \in I} \pi_i. Q_i$ and $B = !\bar{b}. R$; the other cases are similar. Let us denote with σ and θ the address (with respect to the hole) induced by adaptable processes in C_1 and C_2 , respectively. That is, the address of A in C_1 is σ and the address of B in C_2 is θ . Then, by construction of the Petri net, there is a token in the places $\langle \sum_{i \in I} \pi_i. Q_i, \sigma \rangle$ and $\langle !\bar{b}. R, \theta \rangle$. Therefore, transition

$$\{go, \langle \sum_{i \in I} \pi_i. Q_i, \sigma \rangle, \langle !\bar{b}. R, \theta \rangle\} \Rightarrow \{go, \langle !\bar{b}. R, \theta \rangle\} \uplus \text{dec}_\theta(R) \uplus \text{dec}_\sigma(Q_i)$$

(denoted (4) in Table 5) can fire. By definition of dec (cf. Definition 8.4) it is easy to see that this corresponds to $\text{dec}_\varepsilon(P') \uplus \{go\}$. This concludes the proof for this case.

Cases (Tau3)-(Tau4): Then $P \equiv C_1[A] \parallel C_2[B]$ where:

- C_1, C_2 are monadic contexts, as in Definition 7.11;

- $A = b[P_1]$, for some P_1 ;
- $B = \sum_{i \in I} \pi_i. R_i$ with $\pi_l = \tilde{b}\{b[U] \parallel P_2\}$ for $l \in I$, or $B = !\tilde{b}\{b[U] \parallel P_2\}. R$, for some P_2, R .

We consider the case in which $B = !\tilde{b}\{b[U] \parallel P_2\}. R$; the other case is similar. Let us denote with σ and θ the address (with respect to the hole) induced by adaptable processes in C_1 and C_2 , respectively. That is, the address of A in C_1 is σ and the address of B in C_2 is θ . Then, by construction of the Petri net, we have a token in the places $\{\sigma b\}$ and $\langle !\tilde{b}\{b[U] \parallel P_2\}. R, \theta \rangle$. At this point, we should distinguish two cases, depending on whether σb is contained in θ or not. Suppose σb is not contained in θ . That is, there is no process with the nesting structure of C_1 inside C_2 . Then transition

$$\{go, \sigma b, \langle !\tilde{b}\{b[U] \parallel P_2\}. R, \theta \rangle\} \Rightarrow \{go, \langle !\tilde{b}\{b[U] \parallel P_2\}. R, \theta \rangle, \sigma b\} \uplus \text{dec}_\theta(R) \uplus \text{dec}_\sigma(P_2) \uplus \text{dec}_{\sigma b}(U)$$

(denoted (8) in Table 5) can fire. It easy to see that this corresponds to $\text{dec}_\varepsilon(P') \uplus \{go\}$ and we are done.

Similarly, if σb is contained in θ then it means that there exists a process with the same structure of C_1 inside C_2 . Therefore, the place σb is duplicated and a token is present in both places. Then transition

$$\{go, \sigma b, \sigma b, \langle !\tilde{b}\{b[U] \parallel P_2\}. R, \theta \rangle\} \Rightarrow \{go, \sigma b, \sigma b, \langle !\tilde{b}\{b[U] \parallel P_2\}. R, \theta \rangle\} \uplus \text{dec}_\theta(R) \uplus \text{dec}_\sigma(P_2) \uplus \text{dec}_{\sigma b}(U)$$

(denoted (9) in Table 5) can fire. It easy to see that this corresponds to $\text{dec}_\varepsilon(P') \uplus \{go\}$ and this concludes the proof.

We now move on the proof of (2), which proceeds by a case analysis on the transition fired by the Petri net. The transition schemata in Table 5 can be divided into two groups: (1) transitions mimicking a synchronization (i.e., an interaction between an input and an output prefix) and (2) transitions mimicking an update action (i.e., an interaction between an update prefix and an adaptable process). We consider these two groups separately:

- (1) This group comprises transition schemata (3)–(5) in Table 5. For simplicity we concentrate only on transitions of kind (3), as the others are similar. If a transition of this kind can fire, then we have tokens in

$$\{go, \sum_{i \in I} \langle \pi_i. A_i, \alpha \rangle, \sum_{j \in J} \langle \rho_j. B_j, \beta \rangle\}$$

which, by construction of the Petri net, implies that

$$P \equiv D[\sum_{i \in I} \pi_i. A_i, \sum_{j \in J} \rho_j. B_j]$$

where D is a biadic context, as in Definition 7.11. After the fire of the transition, tokens move to $\{go\} \uplus \text{dec}_\alpha(A_l) \uplus \text{dec}_\beta(B_m)$; by construction, this corresponds to a process $P' \equiv D[A_l, A_m]$, and we are done.

- (2) This group comprises transition schemata (6)–(9) in Table 5. For simplicity, we concentrate only on transitions of kind (6), as the others are similar. If a transition of this kind can fire, then we have tokens in

$$\{go, \langle \sum_{i \in I} \pi_i. A_i, \alpha \rangle, \beta\}$$

which, by construction of the Petri net, implies

$$P \equiv D\left[\sum_{i \in I} \pi_i \cdot A_i, a[Q]\right]$$

where D is a biadic context, as in Definition 7.11. After the fire of the transition the tokens move to $\{go\} \uplus \text{dec}_\alpha(A_l) \uplus \text{dec}_\beta(A) \uplus \text{dec}_{\beta a}(U) \uplus \{\beta a\}$; by construction, this corresponds to a process $P' \equiv D[A_l, a[U \langle\langle Q \rangle\rangle] \parallel A]$, and we are done. \square

We can now restate Lemma 8.6, as in Page 41:

Lemma D.9 (8.6). *Let P and $(\text{Places}(P, \emptyset), \text{Trans}(P, \emptyset), \text{Init}(P))$ be an \mathcal{E}_s^3 process and its associated Petri net, as in Definition 8.4. Then we have:*

$$P \longrightarrow P' \text{ iff } \text{dec}_\varepsilon(P) \uplus \{go\} \rightarrow \text{dec}_\varepsilon(P') \uplus \{go\}.$$

Proof. Immediate from Lemma D.8. \square