Steps on the Road to Component Evolvability ⋆

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

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

Abstract. We have recently developed a calculus for *dynamically evolvable* aggregations of components. The calculus extends CCS with primitives for describing components and their evolvability capabilities. Central to these novel primitives is a restricted form of *higher-order communication* of processes involved in update operations. The origins of our calculus for components can indeed be traced back to our own previous work on expressiveness and decidability results for *core* higher-order process calculi. Here we overview these previous works, and discuss the motivations and design decisions that led us from higher-order process calculi to calculi for component evolvability.

Introduction. The deployment of applications by the *aggregation* of elementary blocks (modules, components, Web services, ...) is a long-standing principle in software engineering. Our interest is in the *correctness* of aggregations of *components* which are subject to *evolvability* and *adaptation* concerns. The term "component" is used here in a broad sense, as it refers to elementary blocks such as Web services in cloud computing scenarios, but also to analogous concepts in different settings, such as services in service-oriented computing or long-running processes in workflow management.

To this end, we have recently defined $\mathcal{E}$, a process calculus equipped with primitives for describing components and their evolvability. Using $\mathcal{E}$ as a basis, we have studied the decidability of verification problems associated to the correctness of aggregations of components [2]. In this short paper, we present $\mathcal{E}$ and discuss the origins and motivations that led to its definition. In particular, we elaborate on the relationship between the notion of component evolvability in $\mathcal{E}$ and *higher-order* process calculi.

Steps Towards Specification Languages. *Higher-order process calculi* are calculi in which processes can be passed around in communications. 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 [8] 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 [7]: 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 [10], Homer [5], and Kell [11].

⋆ Supported by the EU integrated project HATS, the Fondation de Coopération Scientifique Digiteo Triangle de la Physique, and FCT / MCTES (CMU-PT/NGN44-2009-12) - INTERFACES

With the purpose of investigating expressiveness and decidability issues in the higher-order paradigm, a *core* higher-order process calculus, called HOCORE, was introduced [6]. 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:

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

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 $\overline{a}\langle P\rangle$ 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., $\overline{a}\langle P\rangle \parallel a(x).\,Q \rightarrow Q\{^{P}/_{x}\}$ is a sample reduction. See [6,9] for a complete account on the basic 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 influencing the execution of a running (higher-order) process. This is a particularly sensible requirement for the specification of systems featuring forms of evolvability and/or dynamic reconfiguration. In order to deal with those aspects, higher-order process calculi such as Homer and Kell provide primitives that allow to *suspend* running processes. In a nutshell, such primitives rely on named *localities* in which processes can execute and interact with their environment, but also in which their execution can be stopped at any time by interaction with complementary input actions. This way, the suspension of a running process is assimilated to regular process communication. Let us illustrate these intuitions by considering the extension of HOCORE with process suspension. Let $a[P]$ denote the process P inside the so-called *suspension unit* a. Assuming a labelled transition system (LTS) with actions of the form $P \xrightarrow{\alpha} P'$, process suspension is formalized by the following two rules:

$$[\textsc{Trans}]\;\; P \xrightarrow{\alpha} P' \Rightarrow a[P] \xrightarrow{\alpha} a[P'] \qquad\qquad [\textsc{Susp}]\;\; a[P] \xrightarrow{a\langle P\rangle} \mathbf{0}$$

where $a\langle P\rangle$ corresponds to the output action in the LTS of HOCORE (see [6]). As a simple example, process $S = a[P_1] \parallel a(x).\,b[x \parallel x]$ defines a process P_1 running at locality a, in parallel with an input action which may suspend the content of a and relocate two copies of it into locality b. Assuming that P_1 evolves into P_2, and given the above two rules, a possible evolution for S is the process $b[P_2 \parallel P_2]$. Observe how term instantiation plays a prominent rôle in mechanisms for process suspension.

In spite of this simple formulation, we observe that suspension primitives are not entirely satisfactory for describing evolvability as in component systems. The reason is that by assimilating suspension to communication, the evolvability of a running process is *decoupled* into two phases: (i) one in which the state of the process is actually suspended and captured and (ii) one in which the suspended process state is used within a new context. In the previous example: the first phase corresponds to capturing the state at a as P_2, while the second corresponds to substituting P_2 twice inside locality b. By considering that update actions are typically atomic operations in which suspension and

relocation occur at the same time, this decoupling turns out to be not realistic in terms of modeling purposes.

Given the above considerations, in [2] we have defined *Evolvable CCS* (abbreviated $\mathcal{E}$), a variant of CCS without restriction and relabeling, and extended with primitives that allow for process evolvability in a "coupled" style. As in CCS, in $\mathcal{E}$ processes can perform actions or synchronize on them. The grammar of $\mathcal{E}$ extends CCS with *update prefixes* and a primitive notion of *component*, denoted $a[P]$:

$$P ::= \pi.P \;\mid\; a[P] \;\mid\; P \parallel P \;\mid\; !\pi.P \;\mid\; \mathbf{0} \qquad \pi ::= a \;\mid\; \overline{a} \;\mid\; \widetilde{a}\{U\}$$

where the U in the update prefix $\widetilde{a}\{U\}$ represents a context, i.e., a process with some holes $\bullet$. We use a and $\overline{a}$ to denote atomic input and output actions, respectively. The rest of the syntax follows standard lines. Evolution at a is realized by the interaction of component $a[P]$ with the update action $\widetilde{a}\{U\}$, which leads to process $U\{^{P}/_{\bullet}\}$, i.e., the process obtained by replacing every hole in U by P. The previous example would be written in $\mathcal{E}$ as the process $S' = a[P_1] \parallel \widetilde{a}\{b[\bullet \parallel \bullet]\}$, which evolves to $b[P_2 \parallel P_2]$ in a single reduction. This way, evolvability relies on the term instantiation feature of higher-order languages in a more disciplined way, ensuring atomicity in updates.

Steps Towards Decidability of Verification Problems. We are interested in two correctness properties for $\mathcal{E}$ processes. The first one, *k-bounded adaptation* (abbreviated $\mathcal{BA}$) ensures that, given a finite k, at most k errors can arise during the system evolution. The second property, *eventual adaptation* (abbreviated $\mathcal{EA}$), is similar but weaker: it ensures that the system will eventually reach a state from which no other error will arise (that is, only finitely many errors can occur). Both these properties are undecidable for $\mathcal{E}$ processes, as we have shown that $\mathcal{E}$ is a Turing complete model (see [2]). The challenge is then to identify fragments of $\mathcal{E}$ expressive enough so as to represent useful evolvability patterns and for which $\mathcal{BA}$ and/or $\mathcal{EA}$ are still decidable.

A similar scenario was addressed in [3,9] for the case of HOCORE. In spite of its minimality, HOCORE was shown to be Turing complete [6]. As studied in [3,9], central to the expressiveness of HOCORE is the ability of *forwarding* a received process within an *arbitrary context*. We then investigated $\text{HO}^{-\text{f}}$, a fragment of HOCORE in which the kind of processes that can be communicated is limited: in $\text{HO}^{-\text{f}}$, output objects can only correspond to the parallel composition of statically known closed processes (i.e., without free variables) with processes received in previously executed input actions. This limitation to forwarding proved to be effective in terms of verification, as termination for $\text{HO}^{-\text{f}}$ processes was shown to be decidable. From a pragmatic perspective, $\text{HO}^{-\text{f}}$ is able to abstract those scenarios in which objects can be passed around and can only be modified by "appending" to them new objects that admit no inspection. This is the case of, e.g., the mobility of already compiled code, on which it is not possible to apply inverse translations (such as, e.g., reverse engineering).

The study in [3,9] thus suggests that key to the decidability of verification problems for higher-order process calculi is the kind of contexts allowed in higher-order actions. In turn, this is closely related to the term instantiation feature discussed before. Based on this observation, we considered constraints on the ways in which components can be updated in $\mathcal{E}$. As a result, we obtained six variants of $\mathcal{E}$ via two orthogonal characterizations. The first characterization is *structural*, and distinguishes between *static*

and *dynamic* topologies of component aggregations. In a static topology the number of components does not vary along the evolution of the system: components cannot be destroyed nor new components can appear. In contrast, this restriction is not considered in dynamic topologies. The second characterization is *behavioral*, and concerns *update patterns* (i.e., the context U in $\widetilde{a}\{U\}$) which define the behavior of components after an update action. We identified three update patterns, which determine three families of $\mathcal{E}$ calculi—denoted $\mathcal{E}^1$, $\mathcal{E}^2$, and $\mathcal{E}^3$, respectively. The first update pattern admits all kinds of contexts, and so it represents the most expressive form of update. In particular, holes $\bullet$ can appear behind prefixes. The second update pattern forbids such guarded holes in contexts. In the third update pattern we additionally require contexts to have exactly one hole, thus preserving the existing behavior (and possibly adding new behaviors): this is the most restrictive form of update that we consider. These variants of $\mathcal{E}$ capture a fairly ample spectrum of scenarios. They borrow inspiration from existing component models, development frameworks, and programming languages. In [2], we have obtained the following (un)decidability results for $\mathcal{BA}$ and $\mathcal{EA}$ in the different variants of $\mathcal{E}$:

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

The decidability of $\mathcal{EA}$ is shown by resorting to Petri nets, while for $\mathcal{BA}$ we consider results for the theory of *well-structured transition systems* [4,1]. The undecidability results are obtained by resorting to termination problems in Turing complete models.

References

1. P. A. Abdulla, K. Cerans, B. Jonsson, and Y.-K. Tsay. Algorithmic analysis of programs with well quasi-ordered domains. *Inf. Comput.*, 160(1-2):109–127, 2000.
2. M. Bravetti, C. D. Giusto, J. A. Pérez, and G. Zavattaro. Adaptable Processes. Technical report, University of Bologna, 2011. Draft in `www.japerez.phipages.com`.
3. 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.
4. A. Finkel and P. Schnoebelen. Well-structured transition systems everywhere! *Theor. Comput. Sci.*, 256(1-2):63–92, 2001.
5. 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.
6. 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.
7. R. Milner. *Comunication and Concurrency*. Prentice Hall, 1989.
8. R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes, I. *Inf. Comput.*, 100(1):1–40, 1992.
9. J. A. Pérez. *Higher-Order Concurrency: Expressiveness and Decidability Results*. PhD thesis, University of Bologna, 2010. Draft in `www.japerez.phipages.com`.
10. 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.
11. 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.