Steps on the Road to Component Evolvability*

Mario Bravetti\textsuperscript{1}, Cinzia Di Giusto\textsuperscript{2}, Jorge A. Pérez\textsuperscript{3}, and Gianluigi Zavattaro\textsuperscript{1}

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

Abstract. We have recently developed a calculus for \textit{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 \textit{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 \textit{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 \textit{aggregation} of elementary blocks (modules, components, Web services, ...) is a long-standing principle in software engineering. Our interest is in the \textit{correctness} of aggregations of \textit{components} which are subject to \textit{evolvability} and \textit{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 \cite{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 \textit{higher-order} process calculi.

Steps towards Specification Languages. \textit{Higher-order process calculi} are calculi in which processes can be passed around in communications. Higher-order (or \textit{process-passing}) concurrency is often presented as an alternative paradigm to the first-order (or \textit{name-passing}) concurrency of the \(\pi\)-calculus \cite{8} for the description of mobile systems. As in the \(\lambda\)-calculus, higher-order process calculi involve \textit{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 \cite{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 \(\pi\)-calculus \cite{10}, Homer \cite{5}, and Kell \cite{11}.

\* Supported by the EU integrated project HATS, the Fondation de Coopération Scientifique Digitéo 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:

\[ P ::= a(x).P \mid \pi(P) \mid P \mid P \mid x \mid 0 \]

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 \(\pi(P)\) 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., \(\pi(P) \parallel a(x).Q \rightarrow Q\{P/x\}\) 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 \(P \alpha \rightarrow P'\), process suspension is formalized by the following two rules:

\[(\text{TRANS})\quad P \alpha \rightarrow P' \Rightarrow a[P] \alpha \rightarrow a[P']\quad \quad (\text{SUSP})\quad a[P] \xrightarrow{a(P)} 0\]

where \(a(P)\) 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