Journal of Logical and Algebraic Methods in Programming 84 (2015) 742–762

Contents lists available at ScienceDirect

Journal of Logical and Algebraic Methods in Programming

www.elsevier.com/locate/jlamp

Revisiting sequential composition in process calculi

Hubert Garavel [a,b,c,d,*]

[a] INRIA, Grenoble, France
[b] Univ. Grenoble Alpes, LIG, F-38000 Grenoble, France
[c] CNRS, LIG, F-38000 Grenoble, France
[d] Saarland University, Saarbrücken, Germany

ARTICLE INFO

Article history:
Received 22 November 2014
Received in revised form 4 May 2015
Accepted 1 August 2015
Available online 1 September 2015

Keywords:
Concurrency theory
Formal specification
Formal semantics
Process algebra
Process calculus
Sequential composition

ABSTRACT

The article reviews the various ways sequential composition is defined in traditional process calculi, and shows that such definitions are not optimal, thus limiting the dissemination of concurrency theory ideas among computer scientists. An alternative approach is proposed, based on a symmetric binary operator and write-many variables. This approach, which generalizes traditional process calculi, has been used to define the new LNT language implemented in the CADP toolbox. Feedback gained from university lectures and real-life case studies shows a high acceptance by computer-science students and industry engineers.

1. Introduction

Process calculi (or process algebras) are a cornerstone of concurrency theory and exist under many variants, among which are CCS [1,2], CSP [3–5], MEIJE [6], ACP[1] [7–9], LOTOS [10,11], PSF [12–14], μCRL [15,16], and mCRL2 [17], to name only a few. From a theoretical point of view, process calculi are *mathematical models* for the study of concurrency. To this aim, they are expected to be of manageable complexity, i.e., have a minimal number of constructs or, at least, a small set of core constructs to which other constructs can be translated.

In practice, however, this definition is too narrow and process calculi clearly have another role. They are often used as *modelling languages* to formally describe the behavior of complex concurrent systems, including telecommunication protocols, distributed software, and hardware circuits. The intent of applying process calculi to real-life problems can be traced back at least to the mid-80s, when LOTOS was defined as an international standard to formally describe OSI protocols and services. Moreover, many compilers and software verification tools have been developed to implement process calculi, e.g., CWB [18] and CWB-NB [19] for CCS, FDR [20,21] and PAT [22,23] for CSP, CADP [24] for LOTOS, the mCRL tools [25,26] for μCRL and mCRL2, etc. All in one, this is a clear indication that more is expected from process calculi than merely providing a theoretical framework for abstractly reasoning about concurrency.

* Correspondence to: INRIA Grenoble, 655 avenue de l'Europe, 38330 Montbonnot St Martin, France.
E-mail address: hubert.garavel@inria.fr.
URL: http://convecs.inria.fr.

[1] For simplicity we will systematically use the name ACP rather than its sub-algebra acronyms, BPA (*Basic Process Algebra*) and PA (*Process Algebra*), even if parallel composition and interprocess communication are not the prime focus of the present article.

http://dx.doi.org/10.1016/j.jlamp.2015.08.001
2352-2208/

Whether a given process calculus can conveniently play these two roles is very much an open question. For instance, Milner believed that CCS was meant for theory only and he proposed, to describe more concrete systems, another language named $\mathcal{M}$ [2, Ch. 8] based on shared variables and algorithmic programming constructs, together with a translation algorithm from $\mathcal{M}$ to CCS. Although there have been follow-ups to this idea, e.g., [27,28], [5, Ch. 18], one may wonder if having two languages instead of one is a true benefit. Moreover, most of the aforementioned process calculi claim to be modelling languages for concurrent systems as well as formalisms supporting mathematical reasoning, therefore indicating a general trend towards a single-language approach.

To describe and analyze real systems, process calculi have genuine advantages: they provide built-in parallel composition operators, they are equipped with formal semantics, and they have compositionality properties (e.g., congruence results) for scaling to large component-based systems. Despite a growing number of success stories, the practical impact of process calculi is not as high as it should be: formal approaches are not widely used in industry and few recent languages for modelling or programming take inspiration from concurrency theory.

There are several reasons for this situation [29]: fragmentation (multiple, similar yet incompatible process calculi), lack of expressiveness, and lack of user-friendliness, all resulting in a steep learning curve that discourages potential users. The aforementioned dichotomy between mathematical models and modelling languages also contributed to creating a gap: the simplicity and elegance of CCS, reinforced by the algebraic approach of ACP, shifted the focus towards mathematical models, weakening the links with mainstream programming languages.

We believe it is important to disseminate concurrency theory results to a wider audience, and that process calculi can be an essential vector for this. It is high time to reconsider the dichotomy between process calculi and programming languages, and to come up with enhanced process calculi acceptable by professionals. As part of this agenda, we review the way sequential composition is handled in process calculi. This topic is often seen as a matter of secondary importance with respect to concurrency and mobility, but the present article reopens the case by considering sequential composition as a major subject that almost entirely shapes a process calculus and determines its compatibility with mainstream programming languages. The core of the issue is not so much *expressiveness* — most process calculi provide the same expressiveness whatever the approach chosen for sequential composition — but *conciseness* (objective) and *convenience* (subjective).

The present article is organized as follows. Section 2 sets a few definitions that will be used throughout the article. Section 3 reviews and gives a critical evaluation of the sequential composition operators available in traditional process calculi. Section 4 proposes an enhanced definition of sequential composition, which has been integrated in LNT [30], the most recent specification language supported by the CADP toolbox [24]. Section 5 discusses the expressiveness of LNT and proposes encodings that map large fragments of former process calculi (CCS, ACP, LOTOS, and CSP) to LNT. Section 6 illustrates the advantages of LNT on several design patterns that cannot be concisely expressed using traditional process calculi. Finally, Section 7 gives concluding remarks and perspectives for future work.

2. Preliminary definitions

The present article assumes that the reader has some basic familiarity with the general concepts of process calculi, but no extensive knowledge is required. As the various process calculi use different vocabulary and notations to define their syntax and semantics, we introduce here common terminology and notations that will be used for all languages throughout the article.

In the following, terminal symbols (i.e., identifiers or constants) are noted in lower case, while non-terminal symbols (defined by BNF rules) are noted in upper case. If y (resp. Y) is a symbol of a given category, then y', y'', y_0, y_1, y_2, etc. (resp. Y', Y'', Y_0, Y_1, Y_2, etc.) are also symbols of that same category.

We note t a *data type*, x a *variable*, and v a *constant*. Variables and constants are typed and we note "$v \in t$" the fact that v is a constant of type t. We note V an *expression* (also: *data expression*, *value expression*, or *value term*), i.e., a syntactic term which is built using variables, constants, and function symbols and that computes a typed value.

We note a an *action* (also: *atomic action* or *atom*), i.e., a communication event proposed by the system under study or its environment. Any action a can be decomposed as a tuple $a = g\ o_1\ \ldots\ o_n$, where g is a *gate* (also: *channel* or *communication port*) and $o_1\ \ldots\ o_n$ a possibly empty list of *offers* (also: *experiment offers* or *action parameters*). Each offer is either an emission (noted "$!\,V$") of some expression V, or a reception (noted "$?x{:}t$", or simply "$?x$") of some constant value to be stored in variable x of type t. We note *gate*(a) the gate of action a. There exists an *internal* (also: *invisible* or *hidden*) gate noted either "τ" or "**i**", which must be used without offer (and is thus also considered to be an action).

We note B a *behavior* (also: *behavior expression* or *process term*), i.e., a syntactic term built using actions and the various operators of the process calculi being considered. Most of our attention focuses on sequential composition operators, so that other kinds of operators (e.g., parallel composition, disruption, etc.) will only be mentioned when needed. We note "$[v_1/x_1 \ldots v_n/x_n]B$" the behavior obtained from B by replacing all free occurrences of variables $x_1, \ldots, x_n$ by constants $v_1, \ldots, v_n$, respectively. We note p a *process identifier* (also: *agent name*), i.e., a symbolic name that can be associated to a behavior (this is needed to define recursive behaviors).