A CSP model with flexible parallel termination semantics

Paul Howells\(^1\) and Mark d’Inverno\(^2\)

\(^1\) School of Informatics, University of Westminster, 115 New Cavendish St., London, W1W 6RU, UK.
Email: P.Howells@wmin.ac.uk
\(^2\) Department of Computing, Goldsmiths, University of London, London SE14 6NW, UK.
Email: dinverno@gold.ac.uk

Abstract. In the original failure-divergence semantic model for Communicating Sequential Processes (CSP), the incomplete treatment of successful process termination, and in particular parallel termination, permitted unnatural processes to be defined. In response to these problems, a number of different solutions have been proposed by various authors since the original failure-divergence model was developed by Hoare, Brookes and Roscoe. This paper presents an alternative solution to this problem, which is both closer to the original semantic model and provides greater flexibility over the type of parallel termination semantics available in CSP.

Keywords: Concurrency; CSP; Termination

1. Introduction

The aim of this paper is to provide an improved and flexible treatment of successful termination in the language and failure-divergence semantic model of Communicating Sequential Processes (CSP), as presented in [BR85, Hoa85]. By improved we mean solving the problem of the mismatch between the intuitive meaning of the notion of the successful termination of a process and how this notion is represented in the failure-divergence semantic model. And by flexible we mean the introduction of different types of parallel termination semantics that can be used in the language of CSP.

There is a need to improve the semantic treatment of successful termination in the original model as we shall illustrate presently. The need for greater flexibility of the type of parallel termination semantics available in CSP is due to a desire to be able to model and design parallel systems in different environments. For example, in a distributed system it may be impractical or inefficient to synchronize the termination of processes, whereas asynchronous termination would be more appropriate. However, in a multiprocessor machine synchronous termination would be appropriate. In other environments, for example competing information searches on the web, it may be acceptable to only require one of a number of processes to terminate, rather than insist on all processes terminating. We believe that by providing parallel operators with different termination semantics, it will provide a more natural way of modeling and designing these types of systems.

Correspondence and offprint requests to: P. Howells, Email: P.Howells@wmin.ac.uk
We intend to achieve our overall aim of providing an improved and flexible treatment of the successful termination in CSP, in three main stages:

- modify the semantic model for CSP by the addition of a new process axiom, which captures our notion of successful termination of a process; and
- show that the non-parallel features of the language (with minor restrictions) satisfy the new termination axiom; and
- finally define new parallel operators with different termination semantics to replace the existing ones, ensuring that they satisfy the new termination axiom.

We believe a satisfactory solution can be provided which is both closer to the original semantics of CSP and provides more flexibility in the choice of parallel termination semantics available than is provided by the existing solutions.

1.1. Successful termination of sequential processes in CSP

We provide a brief introduction to CSP in the Appendix. Successful termination of sequential processes is modelled in CSP by means of a special event ✓ (pronounced tick). If ✓ occurs at the end of a trace of a process then the process is considered to have successfully terminated. ✓ is used in the definition of the sequential composition of two processes P; Q, where if a ✓ occurs at the end of a trace of P it is used to signal to Q that P has successfully terminated and hence that Q can start to execute. For example, the following condition holds:

\[ s \langle ✓ \rangle \in \text{traces}(P) \land t \in \text{traces}(Q) \Rightarrow s \langle ✓ \rangle t \in \text{traces}(P; Q) \]

The ✓ at the end of the trace \( s \langle ✓ \rangle \) of P signals to Q that it can start to execute, in this case the trace t. Note also that the ✓ at the end of \( s \langle ✓ \rangle \) does not appear in the resultant trace \( s \langle ✓ \rangle t \) of \( P; Q \). The ✓ is hidden from the environment of the sequential process and consequently it does not appear in the trace. ✓ is only generated by the process SKIP, which successfully terminates and then does nothing else, it is defined as follows:

**Definition 1.1**

\( \text{SKIP} \equiv ✓ \rightarrow \text{STOP} \)

This illustrates the difference between the two processes SKIP and STOP, in that SKIP successfully terminates then does nothing, whereas STOP does not successfully terminates but just does nothing. Consequently when SKIP is sequentially composed with another process P, the resultant process is equivalent to P, i.e., SKIP is the (left) identity process for “;”, and is captured in the following law:

\[ \text{SKIP}; P \equiv P \quad (1) \]

Whereas when STOP is sequentially composed with another process P, the resultant process is equivalent to STOP, i.e., STOP is the (left) zero process for “;”, this is represented in the following law:

\[ \text{STOP}; P \equiv \text{STOP} \quad (2) \]

Note that neither of the following equivalences hold in general (see [Hoa85]):

\[ P ; \text{SKIP} \equiv P \quad (3) \]
\[ P ; \text{STOP} \equiv \text{STOP} \quad (4) \]

In the remainder of the paper, we shall make use of the standard set of algebraic laws for CSP, for reasons of brevity we do not include these laws in this paper, they can be found in [Bro83, BR85, Hoa85, How05].