Denotational prototype semantics for a simple CSP-like language

Eneia Todoran[1] Nikolaos Papaspyrou[2] Kalman Pusztai[1]

[1]Technical University of Cluj-Napoca
Department of Computer Science
Baritiu Street, 28, Cluj-Napoca
Romania
{Eneia.Todoran,Kalman.Pusztai}@cs.utcluj.ro

[2]National Technical University of Athens
Department of Electrical and Computer Engineering
Polytechnioupoly, 15780 Zografou, Athens
Greece
nickie@softlab.ntua.gr

Abstract – **This paper shows that, by using the "continuation semantics for concurrency" (CSC) technique (recently introduced by us), denotational semantics can be used not only as a method for formal specification and design, but also as a method for concurrent languages prototyping. In this new approach, a denotational function uses continuations to produce incrementally a stream of observables, i.e. a single execution trace, rather than an element of some powerdomain construction. By using a random number generator, an arbitrary execution trace is chosen, thus simulating the non-deterministic behavior of a "real" concurrent system. In this paper we employ classic (cpo-based) domains in developing a denotational prototype semantics for a simple concurrent language providing constructs for CSP-like synchronous communication. The CSC technique plays the main role in the design of the denotational model.**

I. INTRODUCTION

In software engineering, a prototype is an initial version of a system which is used to demonstrate concepts, try out design options and, generally, to find out more about the problem and its possible solutions. Ideally, a prototype serves as a mechanism for identifying software requirements. Rapid development of the prototype is essential so that costs are controlled and users can experiment with the prototype early in the software process.

Denotational semantics is a well-known method for formal specification and design of computer languages; its main characteristic is *compositionality*. It is easy to use a functional language and classic denotational techniques to produce rapidly compositional prototypes for various (aspects of) sequential programming languages (we only mention here the early work of Peter Mosses on the use of denotational descriptions in compiler generation [7, 8]. However, to the best of our knowledge, denotational semantics have never been used systematically as a prototyping method for concurrent languages, and all our attempts to get a satisfactory solution to this problem by using only classic compositional techniques have failed.

Our present aim is to show that, by using the "continuation semantics for concurrency" (CSC) technique - recently introduced by us [13, 14] - denotational semantics can be used not only as a method for formal specification and design, but also as a method for compositional prototyping of concurrent programming languages. In this new approach, a denotational function uses continuations to produce incrementally a stream of observables, i.e. a single execution trace, rather than an element of some powerdomain construction[1]. By using a random number generator, an arbitrary execution trace is chosen, thus simulating the non-deterministic behavior of a "real" concurrent system. We call such a denotational model a *denotational prototype*. The immediate implementation of such a denotational model in an appropriate functional language (such as Haskell [10]) is a compositional (prototype) interpreter for the concurrent language under study.

The CSC technique was introduced in [13] using metric semantics [2]. In this paper we employ classic (cpo-based) domains and continuous functions in developing a denotational prototype semantics for a simple concurrent language, providing constructs for parallel composition and CSP-like synchronous communication [4, 5]; the CSC technique plays the main role in the semantic design. We emphasize that, when the CSC technique is used in this mathematical framework no communication attempts or silent steps need to be produced as final yields of a denotational semantics. Throughout this paper, we rely on the mathematical apparatus and notation in [12, 6].

II. SYNTAX AND INFORMAL EXPLANATION

We consider a simple CSP-like language, called L_{CSP}. The syntax of L_{CSP} is given below in BNF. We assume given a set $(v \in)Var$[2] of (numerical) variables, a set $(e \in)Exp$ of numerical expressions, a set $(b \in)BExp$ of boolean expressions, a set $(c \in)Chan$ of communication channels, and a set $(x \in)PVar$ of procedure variables (or procedure identifiers).

Definition 1 *(Syntax of L_{CSP})*
The set $(s \in)Stmt$ of statements in L_{CSP} is given by the following grammar:

[1]As shown in [13], the CSC technique can be used without difficulty to produce elements of appropriate powerdomain constructions, but this is not the subject of the present paper.

[2]In this paper, the notation $(x, y, ... \in)X$ introduces the set X with typical variables $x, y, ...$ Whenever we use a set in a context where a domain is needed, we assume it is equipped with the discrete order.