A Compositional Model for Confluent Dynamic Data-Flow Networks

Frank S. de Boer[1] and Marcello M. Bonsangue[2]

[1] Utrecht University, The Netherlands
frankb@cs.uu.nl

[2] CWI, Amsterdam, The Netherlands
Marcello.Bonsangue@cwi.nl

Abstract. We introduce a state-based language for programming dynamically changing networks which consist of processes that communicate asynchronously. For this language we introduce an operational semantics and a notion of observable which includes both partial correctness and absence of deadlock. Our main result is a compositional characterization of this notion of observable for a confluent sub-language.

1 Introduction

The goal of this paper is to develop a compositional semantics of a confluent subset of the language *MaC* (Mobile asynchronous Channels). *MaC* is an imperative programming language for describing the behavior of dynamic networks of asynchronously communicating processes.

A program in *MaC* consists of a (finite) number of generic process descriptions. Processes can be created dynamically and have an independent activity that proceeds in parallel with all the other processes in the system. They possess some internal data, which they store in *variables*. The value of a variable is either an element of a predefined data type or it is a *reference* to a *channel*. The variables of one process are not accessible to other processes. The processes can interact only by sending and receiving messages *asynchronously* via channels which are (unbounded) FIFO buffers. A message contains exactly one value; this can be a value of some given data type, like integer or a boolean, or it can be a reference to a channel. Channels are created dynamically. In fact, the creation of a process consists of the creation of a channel which connects it with its creator. This channel has a unique identity which is initially known only to the created process and its creator. As with any channel, the identity of this initial channel too can be communicated to other processes via other channels. Thus we see that a system described by a program in the language *MaC* consists of a dynamically evolving network of processes, which are all executing in parallel, and which communicate asynchronously via mobile channels. In particular, this means that the communication structure of the processes, i.e. which processes are connected by which channels, is completely dynamic, without any regular structure imposed on it a priori.

For *MaC* we first introduce a simple operational semantics and the following notion of observable. Let Σ denote the set of (global) states. A global state specifies, for each existing process, the values of its variables, and, for each existing channel, the contents of its buffer. The semantics $\mathcal{O}$ assigns to each program ρ in *MaC* a *partial* function in $\Sigma \rightharpoonup \mathcal{P}(\Sigma)$ such that $\mathcal{O}(\rho)(\sigma)$ collects all final results of successfully terminating computations in σ, if ρ does not have a deadlocking computation starting from σ. Otherwise, $\mathcal{O}(\rho)(\sigma)$ is undefined.

This notion of observable $\mathcal{O}$ provides a semantic basis for the following interpretation of a correctness formula $\{\phi\}\rho\{\psi\}$ in Hoare logic: every execution of program ρ in a state which satisfies the assertion ϕ does *not deadlock* and upon termination the assertion ψ will hold. An axiomatization of this interpretation of correctness formulas thus requires a method for proving absence of deadlock.

In this paper we identify a *confluent* sub-language of *MaC* which allows to abstract from the order between the communications of different processes and the order between the communications on different channels within a process [14, 15]. A necessary condition for obtaining a confluent sub-language is the restriction to local non-determinism and to channels which are uni-directional and one-to-one. In a dynamic network of processes the restriction to such channels implies that at any moment during the execution of a program for each existing channel there are at most two processes whose internal data contain a reference to it; one of these processes only may use this reference for sending values and the other may use this reference only for receiving values.

For confluent *MaC* programs we develop a compositional characterization of the semantics $\mathcal{O}$. It is based on the local semantics of each single process, which includes information about the channels it has created and, for each known channel, information about the sequence of values the process has sent or read. Information about the deadlock behavior of a process is given in terms of a singleton ready set including a channel reference. As such we do not have any information about the order between the communications of a process on different channels and the order between the communications of different processes. In general, this abstraction will in practice simplify reasoning about the correctness of distributed systems.

Comparison with related work: The language *MaC* is a sub-language of the one introduced in [3]. The latter is an abstract core for the Manifold coordination language [4]. The main feature relevant in this context is anonymous communication, in contrast with parallel object-oriented languages and actor languages, as studied, for example, in [5] and [1], where communication between the processes, i.e., objects or actors, is established via their identities.

In contrast to the π-calculus [16] which constitutes a process algebra for mobility, our language *MaC* provides a state-based model for mobility. As such our language provides a framework for the study of the semantic basis of assertional proof methods for mobility. *MaC* can also be seen as a dynamic version of asynchronous CSP [14]. In fact, the language *MaC* is similar to the verification modeling language Promela [12], a tool for analyzing the logical consistency of distributed systems, specifically of data communication protocols. However, the semantic investigations of Promela are performed within the context of temporal logic, whereas *MaC* provides a semantic basis for Hoare logics.

Our main result can also be viewed as a generalization of the compositional semantics of Kahn (data-flow) networks [13] (where the number of processes and the communication structure is fixed). Instead of a function the communication behavior of a process in the language *MaC* is specified in terms of a relation between the sequence of values it inputs and the sequence of values it outputs. This information suffices because of the restriction to confluent programs. Confluence has been studied also in the context of concurrent constraint programming [9] where mobility is modeled in terms of logical variables.

Generalization of Kahn (data-flow) networks for describing dynamically reconfigurable or mobile networks have also been studied in [6] and [11] using the model of stream functions. In this paper we study a different notion of observable which includes partial correctness and absence of deadlock. Furthermore, our language includes both dynamic process and channel creation. On the other hand, we restrict to confluent dynamic networks.

2 Syntax and Operational Semantics

A program in the language *MaC* is a (finite) collection of generic process descriptions. Such a generic process description consists of an association of a unique name P, the so-called *process type*, with a statement describing generically the behavior of its instances.

The statement associated with a process type P is executed by a process, i.e. an instance of that process type. Such a process acts upon some internal data that are stored in *variables.* The variables of a process are *private*, i.e., the data stored in the variables of a process is not accessible by another process, even if both processes are of the same type. We denote by Var, with its typical elements $x, y, \ldots$, the set of variables. The value of a variable can be either an element of a predefined data type, like integer or boolean, or a reference to a channel.

We have the following repertoire of basic actions of a process:

$$x := e \qquad x := \mathsf{new}(P) \qquad x!y \qquad x?y$$

The execution of an assignment $x := e$ by a process consists of assigning the value resulting from evaluation of the expression e to the variable x (we abstract here from the internal structure of e and assume that its evaluation is deterministic and always terminates).

The execution of the statement $x := \mathsf{new}(P)$ by a process consists of the creation of a new process of type P and a new channel which, initially, forms a link between the two (creator and created) processes. A reference to this channel will be stored in the variable x of the creator and to a distinguished variable *chn* of the created process. The newly created process starts executing the statement associated with P in parallel with all the other existing processes.

Processes can interact only by sending and receiving messages via channels. A message contains exactly one value; this can be of any type, including channel references. We restrict in this paper to asynchronous channels that are implemented by (unbounded) FIFO buffers. The execution of the output action $x!y$ sends the value stored in the variable y to the channel referred to by the variable x. The execution of the input action $x?y$ suspends until a value is available through the specified channel. The value read is removed from the channel and then stored in the variable y.

The set of statements, with typical element S, is generated by composing the above basic actions using well-known sequential non-deterministic programming constructs [8]. A program ρ is a finite collection of generic process descriptions of the form $P \Leftarrow S$. The execution of a program $\{P_0 \Leftarrow S_0, \ldots, P_n \Leftarrow S_n\}$ starts with the execution of a root-process of type P_0.

Next we define formally the (operational) semantics of the programming language by means of a transition system. We assume given an infinite set C of channel identities, with typical elements $c, c', \ldots$. The set Val, with typical elements $u, v, \ldots$, includes the set C of channel identities and the value $\perp$ which indicates that a variable is 'uninitialized'.

A *global state* σ of a network of processes specifies the existing channels, that is, the channels that have already been created, and the contents of their buffers. Formally, σ is a partial function in $C \rightharpoonup \mathit{Val}^*$ (here Val^* denotes the set of finite sequences of elements in Val). Its domain $\mathit{dom}(\sigma) \subseteq C$ is a finite set of channel identities, representing those channels which have been created. Moreover, for every existing channel $c \in \mathit{dom}(\sigma)$, the contents of its buffer is specified by $\sigma(c) \in \mathit{Val}^*$. On the other hand, the internal state $s \in \mathit{Var} \rightarrow \mathit{Val}$ of a process simply specifies the values of its variables.

The behavior of a network of processes is described in terms of a transition relation between configurations of the form $\langle X, \sigma \rangle$, where σ is the global state of the existing channels and X is a finite multiset of pairs of the form (S, s), for some internal state s and statement S. A pair of the form (S, s) denotes an active process within the network: its current internal state is given by s and S denotes the statement to be executed. We have the following transitions for the basic actions (we assume given a program ρ). Below the operation of multiset union is denoted by $\uplus$ and by nil we denote the empty statement.

Assignment: $\langle X \uplus \{(x{:}= e, s)\}, \sigma \rangle \rightarrow \langle X \uplus \{(\mathsf{nil}, s[s(e)/x])\}, \sigma \rangle$, where $s(e)$ denotes the value of e in s and $s[v/x]$ denotes the function mapping x to v and otherwise acting as s.

Creation: $\langle X \uplus \{(x{:}= \mathsf{new}(P), s)\}, \sigma \rangle \rightarrow \langle X \uplus \{(\mathsf{nil}, s[c/x]), (S, s_0[c/\mathit{chn}])\}, \sigma' \rangle$, where $P \Leftarrow S$ occurs in ρ and $\sigma' = \sigma[\varepsilon/c]$ for some $c \in C \setminus \mathit{dom}(\sigma)$. Thus σ' extends σ by mapping the new channel c to the empty sequence ε. Moreover, the initial state of the newly created process s_0 satisfies the following: $s_0(x) = \perp$, for every variable $x \in \mathit{Var}$. Note that the new channel c forms a link between the two processes. The statement S is the one associated with the process type P in the program ρ.

Input: Let $s(x) = c(\neq \perp)$. $\langle X \uplus \{(x?y, s)\}, \sigma \rangle \rightarrow \langle X \uplus \{(\mathsf{nil}, s[u/y])\}, \sigma' \rangle$, where $\sigma(c) = w \cdot u$ for some $u \in \mathit{Val}$, and σ'results from σ by removing u from the buffer of c, that is, $\sigma' = \sigma[w/c]$.

Output: Let $s(x) = c(\neq \perp)$ in $\langle X \uplus \{(x!y, s)\}, \sigma\rangle \to \langle X \uplus \{(\mathsf{nil}, s)\}, \sigma'\rangle$, where σ' results from σ by adding the value $s(y)$ to the sequence $\sigma(c)$, that is, $\sigma' = \sigma[s(y) \cdot \sigma(c)/c]$.

The remaining transition rules for compound statements are standard and therefore omitted. By $\to^*$ we denote the reflexive transitive closure of $\to$ and $\langle X, \sigma\rangle \Rightarrow \delta$ indicates the existence of a deadlocking computation starting from $\langle X, \sigma\rangle$, that is, $\langle X, \sigma\rangle \to^* \langle X', \sigma'\rangle$ with X' containing at least one pair (S, s) such that $S \neq \mathsf{nil}$, and from the configuration $\langle X', \sigma'\rangle$ no further transition is possible. Moreover, $\langle X, \sigma\rangle \Rightarrow \langle X', \sigma'\rangle$ indicates a successfully terminating computation with final configuration $\langle X', \sigma'\rangle$, that is, $\langle X, \sigma\rangle \to^* \langle X', \sigma'\rangle$ and X' contains only pairs of the form (nil, s).

We are now in a position to introduce the following notion of observable.

Definition 1. *Let $\rho = \{P_0 \Leftarrow S_0, \ldots, P_n \Leftarrow S_n\}$ be a program. By $\langle X_0, \sigma_0\rangle$ we denote its initial configuration $\langle\{(S_0, s_0)\}, \sigma_0\rangle$, where $s_0(x) = \perp$, for every variable x, and $dom(\sigma_0) = \emptyset$. We define*

$$\mathcal{O}(\rho) = \begin{cases} \delta & \text{if } \langle X_0, \sigma_0\rangle \Rightarrow \delta \\ \{\langle X, \sigma\rangle \mid \langle X_0, \sigma_0\rangle \Rightarrow \langle X, \sigma\rangle\} & \text{otherwise} \end{cases}$$

Note that thus $\mathcal{O}(\rho) = \delta$ indicates that ρ has a deadlocking computation. On the other hand, if ρ does not have a deadlocking computation then $\mathcal{O}(\rho)$ collects all the final configurations of successfully terminating computations.

3 Compositionality

In this section we introduce, for a certain kind of programs, a compositional characterization of the notion of observable defined in the previous section.

First of all we restrict to local non-determinism. Moreover, we assume now a typing of the variables: we have variables of some predefined data types and we assume channel variables to be either of type ι, for input, and o, for output. Let $\bar{C}$, with typical element $\bar{c}$, be a copy of C. A channel variable of type ι always refers to an element of C, whereas, a channel variable of type o always refers to an element of $\bar{C}$. (The set of all possible values thus includes both C and $\bar{C}$.) We restrict to programs which are well-typed. In particular, in an output $x!y$ the variable x is of type o and in an input $x?y$ the variable x is of type ι. An input $x?y$ now also suspends if the value to be read is not of the same type as the variable y. Moreover, we assume that the distinguished variable chn (used for storing the initial link with the creator) is of type o. Consequently, in $x := \mathsf{new}(P)$ the variable x has to be of type ι. In other words, initially, the flow of information along the newly created channel goes from the created to the creator process.

Finally, we assume that an output $x!y$, where y is a channel variable, is immediately followed by an assignment which uninitializes the variable y, i.e. it sets y to $\perp$. But for this latter, we do not allow channel variables (either of type ι or o) to appear in an assignment. As a result, channels are one-to-one and uni-directional.

We extend now the notion of an internal state s to include the following information about the channels. Let $\gamma \notin Val$ and $Val_\gamma = Val \cup \{\gamma\}$. For each channel $c \in C$, $s(c) \in Val_\gamma^*$ denotes, among others, the sequence of values received from channel c, and $s(\bar{c}) \in Val_\gamma^*$, denotes, among others, the sequence of values sent along channel c. More precisely, in a sequence $w_1 \cdot \gamma \cdot w_2 \cdot \gamma \cdots$, the symbol γ indicates that first the sequence of values w_1 has been sent along c (or received from c) and that after control over this channel has been released and subsequently regained again the sequence w_2 has been sent (or received), etc.. Note that a process releases control over a channel only when it outputs that channel and that it subsequently may again regain control over it only by receiving it via some input.

Additionally, we introduce a component $s(\nu) \in (C \cup \{\bot\}) \times \mathcal{P}(C)$. The first element of $s(\nu)$ indicates the channel which initially links the process with its creator (in case of the root-process we have here $\bot$). The second element of $s(\nu)$ indicates the set of channels which have been created by the process itself.

Given this extended notion of an internal state of a process we now present the transitions describing the execution of the basic actions with respect to the internal state of a process (we omit the standard transition for a simple assignment).

Creation: Let $s(\nu) = (u, V)$ and $c \notin V$ in $\langle x\!: = \mathsf{new}(P), s\rangle \rightarrow \langle \mathsf{nil}, s'[c/x]\rangle$. Here s' results from s by adding c, that is, $s'(\nu) = (u, V \cup \{c\})$. The only effect at the local level of the execution of a basic action $x\!: = \mathsf{new}(P)$ is the assignment to x of a channel c which is new with respect to the set of channels already created by the process.

Output 1: If $s(x) = \bar{c}$ and y is not a channel variable, i.e. y is of some given data type like the integers or booleans, then $\langle x!y, s\rangle \rightarrow \langle \mathsf{nil}, s[s(\bar{c}) \cdot s(y)/\bar{c}]\rangle$. The local effect of an output (of a value of some predefined data type) consists of adding the value stored in the variable y to the sequence of values already sent.

Output 2: If $s(x) = \bar{c}$ then $\langle x!y, s\rangle \rightarrow \langle \mathsf{nil}, s[s(\bar{c}) \cdot v/\bar{c}][s(v) \cdot \gamma/v]\rangle$, where $v = s(y)$ and y is a channel variable. So after the output along the channel c of the value v stored in the variable y, first the value v is appended to $s(\bar{c})$, which basically records the sequence of values sent along the channel $\bar{c}$. Finally, the output of channel v (and consequently its release) is recorded as such by γ in the sequence $s(v)$ which records the sequence of values sent along the channel v, in case $v \in \bar{C}$, and received from it, in case $v \in C$. Note that we have to perform the state-changes indicated by $[s(\bar{c}) \cdot v/\bar{c}]$ and $[s(v) \cdot \gamma/v]$ in this order to describe correctly the case that $v = \bar{c}$.

Input: If $s(x) = c(\neq \bot)$ then $\langle x?y, s\rangle \rightarrow \langle \mathsf{nil}, s[v/y, s(c) \cdot v/c]\rangle$, where $v \in Val$ is an *arbitrary* value (of the same type as y). This value is assigned to y and appended to the sequence $s(c)$ of values received so far (along channel c). Note that because channels are one-to-one and unidirectional it cannot be the case that $v = c$.

On the basis of the above transition system (we omit the rules from compound statement since they are standard) we define the operational semantics of statements as follows.

Definition 2. *An (extended)* initial *state s satisfies the following: for some $u \in C \cup \{\bot\}$ we have that $s(chn) = u$, and $s(x) = \bot$, for every other variable, moreover, $s(d) = s(\bar{d}) = \epsilon$, for every channel d, and, finally, $s(\nu) = (u, \emptyset)$. We define $\mathcal{O}(S) = \langle T, R\rangle$, where $T = \{s' \mid \langle S, s\rangle \rightarrow^* \langle \mathsf{nil}, s'\rangle$ for some initial state $s\}$ and $R = \{(s', s(x), t(y)) \mid \langle S, s\rangle \rightarrow^* \langle x?y;S', s'\rangle$, for some initial state $s\}$ (here $t(y)$ denotes the type of y).*

The component T in the semantics $\mathcal{O}(S)$ collects all the final states of successfully terminating (local) computations of S (starting from an initial state). The component R, on the other hand, collects all the intermediate states where control is about to perform an input, plus information about the channel involved and the type of the value to be read. The restriction to local non-determinism implies that when an input $x?y$ is about to be executed, it will always appear in a context of the form $x?y;S$ for some (possibly empty) statement S (no other inputs are offered as an alternative).

The information in R corresponds with the well-known concept of the *ready sets* [17] and will be used for determining whether a program (containing a process type $P \Leftarrow S$) has a deadlocking computation.

Our compositional semantics is based on the *comp*