Centrum voor Wiskunde en Informatica
REPORT*RAPPORT*

A Compositional Model for Confluent Dynamic Data-Flow Networks

F.S. de Boer, M.M. Bonsangue

Software Engineering (SEN)

SEN-R0021 July 31, 2000

Report SEN-R0021
ISSN 1386-369X

CWI
P.O. Box 94079
1090 GB Amsterdam
The Netherlands

CWI is the National Research Institute for Mathematics and Computer Science. CWI is part of the Stichting Mathematisch Centrum (SMC), the Dutch foundation for promotion of mathematics and computer science and their applications.
SMC is sponsored by the Netherlands Organization for Scientific Research (NWO). CWI is a member of ERCIM, the European Research Consortium for Informatics and Mathematics.

Copyright © Stichting Mathematisch Centrum
P.O. Box 94079, 1090 GB Amsterdam (NL)
Kruislaan 413, 1098 SJ Amsterdam (NL)
Telephone +31 20 592 9333
Telefax +31 20 592 4199

A Compositional Model for Confluent Dynamic Data-Flow Networks

Frank S. de Boer
Utrecht University
P.O. Box 80.089, 3508 TB Utrecht, The Netherlands
E-mail: frankb@cs.uu.nl

Marcello M. Bonsangue
CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
E-mail: marcello@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.

2000 Mathematics Subject Classification: 68N15, 68N99, 68Q10, 68Q55, 68Q60
1998 ACM Computing Classification System: D.1.3, D.1.5, D.3.3, F.1.1, F.3.2
Keywords and Phrases: Kahn network, asynchronous communication, dynamic process creation, dynamic interconnection structure, class, object, mobility
Note: Work carried out under the project SEN 3.1 "Formal Methods for Coordination Languages"

Table of Contents

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 [17, 18]. 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 [4]. 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 [5]. 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 [6] and [1], where communication between the processes,

i.e., objects or actors, is established via their identities.

In contrast to the π-calculus [19] 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 [17]. In fact, the language *MaC* is similar to the verification modeling language Promela [15], 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 [16] (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 [11] 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 [8] and [13] 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

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 [10], like sequential composi-

tion ';', non-deterministic choice '+', repetition '$\mathsf{do} - \mathsf{od}$' and guards '$g \rightarrow$', where the guard '$g$' is either an input statement '$x?y$', a boolean condition 'b', or a boolean condition followed by an input, like in CSP [14]. 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.

As a simple example we present the following program consisting of three process descriptions: a sender P_2, a receiver P_1 and a root P_0 which establishes a communication link between the two:

$$\begin{array}{lcl} P_0 & \Leftarrow & x1{:=}\,\mathsf{new}(P_1)\ ;\ x2{:=}\,\mathsf{new}(P_2)\ ;\ x2!x1 \\ P_1 & \Leftarrow & \mathsf{do}\ \mathit{chn}?x\ ;\ \cdots\ \mathsf{od} \\ P_2 & \Leftarrow & y{:=}\,0\ ;\ \mathit{chn}?x\ ;\mathsf{do}\ x!y\ ;\ y{:=}\,y+1\ \mathsf{od} \end{array}$$

The sender receives through its initial connection a reference to a channel through which it sends the values it produces. The receiver uses its initial connection for receiving values. The connection between the sender and the receiver is established by an instance of the P_0 process. Note that the communication between the sender and the receiver is anonymous.

3. Operational semantics

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.

Process and channel creation: Let $P \Leftarrow S$ occurs in ρ.

$$\langle X \uplus \{(x{:=}\,\mathsf{new}(P), s)\}, \sigma \rangle \rightarrow \langle X \uplus \{(\mathsf{nil}, s[c/x]), (S, s_0[c/\mathit{chn}])\}, \sigma' \rangle,$$

where $\sigma' = \sigma[\varepsilon/c]$ 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 $s(x) = c(\neq \perp)$. If $\sigma(c) = w \cdot u$ for some $u \in \mathit{Val}$ then

$$\langle X \uplus \{(x?y, s)\}, \sigma\rangle \to \langle X \uplus \{(\mathsf{nil}, s[u/y])\}, \sigma'\rangle,$$

where 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 3.1 *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.

As already discussed in the introduction, this notion of observable provides a semantic basis for a generalization of the usual notion of partial correctness in Hoare logic which includes the requirement of absence of deadlock. More precisely, this notion of observable provides the following interpretation of a Hoare triple $\{\phi\}\rho\{\psi\}$: If the initial configuration $\langle X_0, \sigma_0\rangle$ satisfies ϕ then no computation of ρ deadlocks and every terminating computation of ρ results in a final configuration that satisfies the postcondition ψ. An axiomatization of this notion of program correctness thus will require a method for proving absence of deadlock.

4. 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. Our abstract semantics, decoupling the inherent ordering of the transmission and reception of values through different channels, is not compositional in the general case. In [4] examples are given where compositionality breaks down because the environment is allowed to influence the nondeterministic behavior of a component. Theere are three reasons why this may happen: (1) the presence of input actions in a nondeterministic choice; (2) the reception of a value from a non private channel; and (3) the emission of a value on a non-private channel.

We rule out the first kind of external non-determinism by restricting to local non-determinism. To avoid the interferences caused by non-private cahnnels, 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.

4.1 Local Semantics

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 $s(\nu) \in (C \cup \{\perp\}) \times \mathcal{P}(C)$. 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 $\perp$). 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.

Assignment:

$$\langle x{:=}\, e, s\rangle \to \langle \mathsf{nil}, s[s(e)/x]\rangle,$$

where $s(e)$ denotes the value of e

Process and channel creation: Let $s(\nu) = (u, V)$ and $c \notin V$ in

$$\langle x{:=}\,\mathsf{new}(P), s\rangle \to \langle \mathsf{nil}, s'[c/x]\rangle.$$

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

$$\langle x!y, s\rangle \to \langle \mathsf{nil}, s[s(\bar{c}) \cdot s(y)/\bar{c}]\rangle.$$

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

$$\langle x!y, s\rangle \to \langle \mathsf{nil}, s[s(\bar{c}) \cdot v/\bar{c}][s(v) \cdot \gamma/v]\rangle,$$

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 $s(x) = c(\neq \perp)$ then

$$\langle x?y, s\rangle \to \langle \mathsf{nil}, s[v/y, s(c) \cdot v/c]\rangle,$$

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 4.1 *An (extended)* initial *state s satisfies the following: for some $u \in C \cup \{\perp\}$ we have that $s(chn) = u$, and $s(x) = \perp$, 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

- $T = \{s' \mid \langle S, s\rangle \to^* \langle \mathsf{nil}, s'\rangle$ *for some initial state* $s\}$, *and*
- $R = \{(s', s(x), t(y)) \mid \langle S, s\rangle \to^* \langle x?y;S', s'\rangle$, *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* [20] and will be used for determining whether a program (containing a process type $P \Leftarrow S$) has a deadlocking computation.

4.2 Compatible Internal States

Our compositional semantics is based on the *compatibility* of a set of internal states (without loss of generality we may indeed restrict to sets rather than multisets of *extended* internal states s because of the additional information $s(\nu)$). In order to define this notion we use the set $C_\perp = C \cup \{\perp\}$, ranged over by $\alpha, \beta, \ldots$, to identify processes. The idea is that the channel which initially links the created process with its creator will be used to identify the created process itself ($\perp$ will be used to identify the root-process). We use these process identifiers in finite sequences of labeled inputs $(\alpha, c?v)$ and outputs $(\alpha, c!v)$ to indicate the process involved in the communication. Given such a sequence h and a channel $c \in C$ we denote by $sent(h, c)$ the sequence of values in Val sent to the channel c. It is defined by induction on the length of h:

$$sent(\varepsilon, c) = \varepsilon \qquad sent((\alpha, c!v) \cdot h, c) = v \cdot sent(h, c)$$

In all other cases the leftmost (labeled) communication is discarded. Similarly, we denote by $rec(h, c)$ the sequence of values in Val received from the channel c:

$$rec(\varepsilon, c) = \varepsilon \qquad rec((\alpha, c?v) \cdot h, c) = v \cdot rec(h, c)$$

In all other cases the leftmost (labeled) communication is discarded.

A *history* h is a (finite) sequence of labeled inputs $(\alpha, c?v)$ and outputs $(\alpha, c!v)$ which satisfies the following.

Prefix invariance: For every prefix h' of h and channel c we have that the sequence $rec(h', c)$ of values delivered by c is a prefix of the sequence $sent(h', c)$ of values received by the channel c.

Input ownership: For every prefix $h_0 \cdot (\alpha, c?v)$ of h, either the process α owns the input of the channel c in h_0, or $h_0 = h_1 \cdot (\alpha, d?c) \cdot h_2$ for some channel d distinct from c and α owns the input of the channel c in h_2. A process α is said to be the owner of the input of a channel c in a sequence h if, for any channel e, there is no occurrence in h of an output $(\alpha, e!c)$, and for every occurrence in h of an input $(\beta, c?w)$ we have $\alpha = \beta$.

Output ownership: For every prefix $h_0 \cdot (\alpha, c!v)$ of h, either the process α owns the output of the channel c in h_0, or $h_0 = h_1 \cdot (\alpha, d?c) \cdot h_2$ for some channel d (*not* necessarily distinct from c) and α owns the output of the channel c in h_2. A process α is said to be the owner of the output of a channel c in a sequence h if for any channel e there is no occurrence in h of an output $(\alpha, e!c)$, and for every occurrence in h of an output $(\beta, c!w)$ we have $\alpha = \beta$.

Input/output ownership essentially states that a process can communicate along a channel only if either it is the first user of that channel or it has received that channel via a preceding communication. Moreover, exclusive control over a channel is released only when that channel is outputted.

For a given history h, a channel $c \in C$, and a process identifier α, we next define the sequence $in(h, \alpha, c)$ of values in Val_γ which consists of the stream of values received from the channel c by the process α. Occurrences of γ in those sequences will denote release of control of the channel c by the process α. Thus $in(h, \alpha, c)$ denotes the sequence of values received from c the *first* time α has gained its control, followed by the stream of values received by α from c the *second* time it has gained control over c, and so on. We define $in(h, \alpha, c)$ so by induction on the length of h:

$$in(\varepsilon, \alpha, c) = \varepsilon$$
$$in((\alpha, c?v) \cdot h, \alpha, c) = v \cdot in(h, \alpha, c) \qquad in((\alpha, d!c) \cdot h, \alpha, c) = \gamma \cdot in(h, \alpha, c)$$

In all other cases the leftmost (labeled) communication is discarded. Similarly, we define the sequence $out(h, \alpha, c)$ which consists of the stream of values semt by the process α along the channel c the *first* time it has gained control over c, followed by the stream of values send by process α along channel c the *second* time it has gained control over c, and so on.

$$out(\varepsilon, \alpha, c) = \varepsilon \qquad out((\alpha, c!v) \cdot h, \alpha, c) = v \cdot out(h, \alpha, c)$$
$$out((\alpha, c!\bar{c}) \cdot h, \alpha, c) = \bar{c} \cdot \gamma \cdot out(h, \alpha, c) \qquad out((\alpha, d!\bar{c}) \cdot h, \alpha, c) = \gamma \cdot out(h, \alpha, c)$$

where $v \neq \bar{c}$ and $d \neq c$. In all other cases the first (labeled) communication is discarded. Note that the case of outputting the value $\bar{c}$ along channel c itself requires a special treatment.

We can obtain the local information of a process from a given history as follows. For a history h, an internal state s, we write $s \simeq h$ if $s(\nu) = (\alpha, V)$ implies, for every channel c, both $s(c) = in(h, \alpha, c)$ and $s(\bar{c}) = out(h, \alpha, c)$. Thus $s \simeq h$ basically states that the information about the communication behavior in the internal state s is *compatible* with the information given by the history h. The compatibility of h with respect to a set of internal states X is defined below.

Definition 4.2 *Let h be a history and X be a finite set of internal states. We say that h is compatible with X if the following two conditions hold:*

1. *for every $s \in X$, $s \simeq h$;*
2. *there exists a finite tree (the tree of creation) with X as nodes such that*
   - *if s is the root of the tree then $s(\nu) = (\perp, V)$, for some $V \subseteq C$;*
   - *if $s \in X$ with $s(\nu) = (u, V)$ then $u = v$ for all $s' \in X$ with $s' \neq s$ and $s'(\nu) = (v, W)$;*
   - *if $s \in X$ with $s(\nu) = (u, V)$ then for all $v \in V$ there exists a direct descendent node $s' \in X$ with $s'(\nu) = (v, W)$, for some $W \subseteq C$.*

The existence of a tree of creation ensures the uniqueness of the name of the created channels. It is worthwhile to observe that it is not sufficient to require disjointness of the names used by any two distinct existing processes, as this does not exclude cycles in the creation ordering (for example, two processes creating each other).

Let h be a history compatible with a finite set of (internal) states X. For each channel c which appears in X, we denote by $own(h, c)$ the sequence of processes who had the ownership of the reference for inputting from the channel c. It is defined by nduction on the length of h. We list the main cases (in all other cases the rightmost communication is simply discarded).

$$own(\varepsilon, c) = c \qquad own(h \cdot (\beta, d?c), c) = own(h, c) \cdot \beta$$

Initially c is owned by the process in X which is identified by c itself. Similarly, we define the sequence $own(h, \bar{c})$ of processes who had the ownership of the reference for outputting to c, by induction on the length of h. We list the main cases (in all other cases the rightmost communication is simply discarded).

$$own(\varepsilon, \bar{c}) = \alpha \qquad own(h \cdot (\beta, d?\bar{c}), \bar{c}) = own(h, \bar{c}) \cdot \beta$$

where α is the process that created the channel c. Formally, we take α such that for some $s \in X$ with $s(\nu) = (\alpha, V)$ and $c \in V$. The existence and uniqueness of such process identifier is guaranteed by the fact that the given set of states X can be organized as a tree.

For a given set of (internal) states X there may be several histories, each of them compatible with X. The next theorem specifies the relevant information recorded in a history.

Theorem 4.3 *Let X be a finite set of (internal) states, and h_1 and h_2 be two histories compatible with X. For all process id's α and channels c the following holds:*

1. $in(h_1, \alpha, c) = in(h_2, \alpha, c)$ *and* $out(h_1, \alpha, c) = out(h_2, \alpha, c)$;
2. $sent(h_1, c) = sent(h_2, c)$ *and* $rec(h_1, c) = rec(h_2, c)$;
3. $own(h_1, c) = own(h_2, c)$ *and* $own(h_1, \bar{c}) = own(h_2, \bar{c})$.

We give a sketch of the proof. The first item holds because both $s \simeq h_1$ and $s \simeq h_2$ for every $s \in X$. Thus, for every channel c, $in(h_1, \alpha, c) = s(c) = in(h_2, \alpha, c)$ and $out(h_1, \alpha, c) = s(\bar{c}) = out(h_2, \alpha, c)$, where $s(\nu) = (\alpha, V)$, for some V.

We prove the second and third items together by showing that for each prefix h of h_1 and for each channel c it holds that $sent(h, c)$ is a prefix of $sent(h_2, c)$, $rec(h, c)$ is a prefix of $rec(h_2, c)$, $own(h, c)$ is a prefix of $own(h_2, c)$ and $own(h, \bar{c})$ is a prefix of $own(h_2, \bar{c})$. We proceed by induction on the length of h. The base case holds trivially: $sent(\varepsilon, c) = rec(\varepsilon, c) = \varepsilon$, while $own(\varepsilon, c) = c$ and by definition $own(h_2, c)$ also starts with c, on the other hand, both $own(\varepsilon, \bar{c})$ and $own(h_2, \bar{c})$ start with the (unique) process α such that for some $s \in X$, $s(\nu) = (\alpha, V)$, with $c \in C$.

So let us assume h is a proper prefix of h_1 such that the above holds. We proceed by case analysis of the leftmost communication in h_1 after h. The main case is that of an input, say, $(\alpha, c?v)$. We first assume that $v \notin C \cup \bar{C}$. It follows that $sent(h \cdot (\alpha, c?v), d) = sent(h, d)$ and $own(h \cdot (\alpha, c?v), d) = own(h, d)$, for every channel d. Thus we have only to prove that $rec(h \cdot (\alpha, c?v), c) = rec(h, c) \cdot v$ is a prefix of $rec(h_2, c)$. To this end, assume $own(h, c) = \alpha_1 \cdots \alpha_n$. By the channel ownership property of the history h, we have that $\alpha_n = \alpha$. Furthermore, by the induction hypothesis, $\alpha_1 \cdots \alpha_n$ is a prefix of $own(h_2, c)$. By construction we can partition the string $rec(h, c)$ in n substrings $w_1 \cdots w_n$, where w_i denotes the sequence of values channel c has delivered to the process α_i, as recorded in the state $s_i(c)$ of α_i (note that the control character γ is used to take into account multiple occurrences of the same process identifier in the string $\alpha_1 \cdots \alpha_n$). By the compatibility of h_1 (with respect to the given set X) and because $\alpha_n = \alpha$, v must be the value following the string w_n as recorded in the state $s_n(c)$ of the process α_n. Since also h_2 is compatible (with respect to the given set X), it follows that the

leftmost communication involving an input from c after those recorded in $rec(h, c)$ must be $(\alpha, c?v)$. Indeed, if it would have been $(\beta, c?w)$ for some β different from α, then this input should have been preceded by an output $(\alpha, d!c)$ (this follows from the input ownership and prefix invariance of h_2). But then this output $(\alpha, d!c)$ would have been recorded by a γ after w_n in the internal state of α by compatibility of h_2, contradicting the fact that w_n is in fact followed by v (as argued above). We conclude $rec(h, c) \cdot v$ is a prefix of $rec(h_2, c)$.

In case of an input $(\alpha, c?d)$, with $d \in C$ we have to prove that $rec(h \cdot (\alpha, c?d), c) = rec(h, c) \cdot v$ is a prefix of $rec(h_2, c)$ and, additionally, that $own(h \cdot (\alpha, c?d), d) = own(h, d) \cdot \alpha$ is a prefix of $own(X, h_2, d)$. This can be proved in a similar way as above.

This theorem states that the compatibility relation abstracts from the order of communication between different channels in a global history. For example, even the ordering between inputs and outputs on different channels is irrelevant. This contrasts with the usual models of asynchronous communicating non-deterministic processes [17, 18]. This abstraction is made possible because of the restriction to confluent programs.

In order to formulate the main theorem of this paper we still need some more definitions. We say that a set X of extended internal states is *consistent* if there exists a history h compatible with X. Given a consistent set X of extended internal states s, we denote by $conf(X)$, the corresponding (final) configuration $\langle \tilde{X}, \sigma \rangle$. That is, $\tilde{X}$ consists of those pairs $(\mathsf{nil}, \tilde{s})$ for which there exists $s \in X$ such that $\tilde{s}$ is obtained from s by removing the additional information about the communicated values and the created channels. The global state σ derives from a history h compatible with X in the obvious way (i.e. by mapping every channel c such that $s(\nu) = (c, V)$ for some $s \in X$ and $V \subseteq C$, to the sequence obtained by deleting the prefix $rec(h, c)$ from $sent(h, c)$). Note that the above Theorem 4.3 guarantees that σ is indeed well-defined.

Definition 4.4 *We assume given T_i and R_i, for $i = 1, \ldots, n$, with T_i a set of (extended) internal states and R_i a set of triples of the form (s, c, t), where s is an extended internal state, c is a channel and t is a type (of the value to be read from c in the state s).*

We denote by $\bigsqcup_i T_i$ the set of final configurations $conf(X)$ such that the set X of (extended) internal states is consistent and every state s in X belongs to some T_i. Additionally, for some state $s \in T_0$ we have $s(\nu) = \langle \bot, V \rangle$, for some $V \subseteq C$.

Analogously, by $\bigsqcup_i \langle T_i, R_i \rangle$ we denote the set of final configurations $conf(X)$ such that X is consistent, and there exists a state s in X that does not belong to any T_i, and, finally, every state s in X either belongs to some T_i or there exists a triple $(s, c, t) \in R_i$ such that either $\sigma(c) = \epsilon$ or the first value of $\sigma(c)$ is not of type t.

Abstracting from the control information, the set of configurations $\bigsqcup_i \langle T_i, R_i \rangle$ in fact describes all possible deadlock configurations, whereas $\bigsqcup_i T_i$ describes all the final configurations of successfully terminating computations of the given program. Finally, we are in a position to formulate the main theorem of this paper.

Theorem 4.5 *Let $\rho = \{P_0 \Leftarrow S_0, \ldots, P_n \Leftarrow S_n\}$ and $\mathcal{O}(S_i) = \langle T_i, R_i \rangle$, $i = 0, \ldots, n$. We have that*

$$\mathcal{O}(\rho) = \begin{cases} \bigsqcup_i T_i & \text{if } \bigsqcup_i \langle T_i, R_i \rangle = \emptyset \\ \delta & \text{otherwise.} \end{cases}$$

We give a sketch of the proof. In the proof below we assume that, given a set X of (extended) internal states, for each $s \in X$ we can identify the T_i (or R_i) to which it belongs.

We first argue that $\bigsqcup_i \langle T_i, R_i \rangle \neq \emptyset$ implies the existence of a deadlocking computation of the given program ρ. Let X be a consistent set of internal states such that $conf(X) = \langle \tilde{X}, \sigma \rangle \in \bigsqcup_i \langle T_i, R_i \rangle$, and let h be an history compatible with X. By definition, for every $s \in X$, there exists a local computation

$$\langle S_i, s_0 \rangle \rightarrow^* \langle S', s \rangle$$

where s_0 is an initial state, and either $S' = \mathsf{nil}$ and $s \in T_i$ or $S' = x?y;S''$ and $\langle s, s(x), t(y)\rangle \in R_i$. Thus either the computation terminates in a state s or it blocks when executing the input $x?y$ (because $\sigma(c) = \epsilon$ or the rightmost element of $\sigma(c)$ has a different type than that of y). Note that, by definition of $\bigsqcup_i \langle T_i, R_i\rangle$, there is at least one such local computation that blocks when executing an input.

We show that $\langle X_0, \sigma_0\rangle \Rightarrow \delta$, where $X_0 = \{(S_0, s_0)\}$ and $\sigma_0(\nu) = \emptyset$ (here s_0 denotes the initial state of the root-process, that is $s_0(\nu) = (\bot, \emptyset)$). We do so by constructing a deadlocking computation of the program ρ out of the local computations such that for every intermediate configuration $\langle X', \sigma'\rangle$ of this global computation and corresponding history h' we have, for all channels c,

1. $sent(h', c)$ is a prefix of $sent(h, c)$ and $rec(h', c)$ is a prefix of $rec(h, c)$;
2. $own(h', c)$ is a prefix of $own(h, c)$and $own(h', \bar{c})$ is a prefix of $own(h, \bar{c})$.

We proceed by induction on the length of the computation. The initial configuration $\langle X_0, \sigma_0\rangle$ clearly satisfies the above requirements. Suppose we have already constructed a computation

$$\langle X_0, \sigma_0\rangle \rightarrow^* \langle X', \sigma'\rangle$$

such that the corresponding history h' satisfies the above properties 1. and 2.. Suppose there exists a process in X' which is enabled (if this is not the case then we are done). We consider first the main case of an input: we have $(x?y;S, s) \in X'$ such that $\sigma'(c) = w \cdot u$ and u is of the right type, where $c = s(x)$. We have to show that in the local computation of this process at this point the same value u has been 'guessed'. This in fact can be proved along the same lines as the analogous case above in the proof of Theorem 4.3. In case the enabled process is executing an action $x := \mathsf{new}(P_i)$ in an internal state s, we may assume without loss of generality that the new channel created corresponds with the new channel as recorded by the local computation of this process, since X can be organized as a tree as described in definition 4.2. Let s_0 be the initial state of the newly created process, we then have to add (S_i, s_0) to the resulting configuration. A more detailed proof of these cases and of the other ones can be found in the full version of the paper.

Conversely, it is clear that $\bigsqcup_i \langle T_i, R_i\rangle = \emptyset$ implies absence of deadlock (of the given program ρ).

Let us now assume that $\bigsqcup_i \langle T_i, R_i\rangle = \emptyset$ and that $conf(X) = \langle \tilde{X}, \sigma\rangle$ is a final configuration, where X is a consistent set of (extended) internal states such that every state s in X belongs to some T_i. Thus, for every $s \in X$ there exists a terminating computation

$$\langle S_i, s_0\rangle \rightarrow^* \langle \mathsf{nil}, s\rangle,$$

where, as before, s_0 denotes an initial state. As above we can construct a (global) computation

$$\langle X_0, \sigma_0\rangle \rightarrow \langle X', \sigma'\rangle$$

of ρ out of these (local) computations such that $X' \subseteq \tilde{X}$. We do know that we can reach a terminating configuration $\langle X', \sigma'\rangle$ because of the absence of a deadlocking computation. However, we do not know yet that *all* the given local computations are present in this interleaving. So we still have to prove that $\langle X', \sigma'\rangle = \langle \tilde{X}, \sigma\rangle$. This follows by the assumption that X can be organized as a tree (as described in Definition 4.2). More precisely, we can prove by induction on the *depth* of a state $s \in X$ in this tree that $\tilde{s} \in X'$: the root-process clearly exists in X'. So let s in X occur at depth $n + 1$. By definition its creator occurs at depth n and thus, by induction, exists in X'. By construction of the global computation we conclude that $\tilde{s}$ also exists in X'. Thus $\tilde{X} = X'$, and, a fortiori (by theorem 4.3), $\sigma = \sigma'$.

Thus the observable behavior of a confluent *MaC* program can be obtained in a compositional manner from the local semantics of the statements of each process description of the program. The information of the ready sets of each local semantics is used to determine if the program deadlocks.

5. Conclusion and Future Work

To the best of the authors knowledge, we have presented a first state-based semantics for a confluent language for mobile data-flow networks which is compositional with respect to the abstract notion of observable considered in this paper. This notion of observable is more abstract than the bisimulation-based semantics for most action-based calculi for mobility [19, 12, 9], and the trace-based semantics for state-based languages [15, 22, 21]. We still have to investigate the full abstractness of our compositional semantics.

The proposed semantics will be used for defining a compositional Hoare logic for confluent *MaC* programs along the lines of [6]. Given appropriate *assertion* languages for describing properties of internal and global states, respectively, the correctness of a statement S will be specified by a formula $I{:}\{p\}S\{q\}$. Here p and q are local assertions describing properties of the initial and final states of executions of S, and I is a set of blocking invariants, namely one $I_t(z)$ for each data type t (here z is a logical variable ranging over channel references). Such a correctness formula will be interpreted, for $\mathcal{O}(S) = \langle T, R\rangle$, as follows: If every initial state satisfies p then every $s \in T$ satisfies q and for every $(s, c, t) \in R$ we have that s satisfies $I_t(c)$. The set of invariants I thus describes the internal states where control is about to execute an input. The proof rules (and the method for proving absence of deadlock) then can be obtained in a relatively straightforward manner from the compositional semantics. The fact that the order between the communications between different processes and the communication on different channels within a process is semantically irrelevant will in general simplify the correctness proofs.

Currently we are investigating the additional information needed to enlarge the class of program of *MaC* maintaining a compositional semantics with respect to our notion of observable. For example, allowing external boolean guards (which evaluate on the basis of the content of a channel) and non-destructive input operations would not require any major adjustment to the structure of our extended states, whereas allowing external non-determinism and both synchronous and asynchronous channels would require additional local information about the order between the communications of processes on different channels.

In the context of synchronous communication, another line of research concerns a study of an 'existential' notion of deadlock, as studied in process algebras or as expressed for example in the failure semantics of CSP [7]. Even if failure semantics is traditionally studied in the context of action-based calculi, according to our opinion for mobile languages it will be easier to define the concept of failure in a state-based language, like our restriction-free *MaC* language with synchronous channels.

Acknowledgements We like to thank the Amsterdam Coordination Group, especially Jaco de Bakker, Frahad Arbab and Jan Rutten for discussions and suggestions about the contents of this paper. Thank also to Erika A'braham-Mumm for her helpful comments.

References

1. G. Agha, I. Mason, S. Smith, and C. Talcott. A foundation for actor computation *Journal of Functional Programming*, 1(1):1-69, 1993.
2. R. Amadio, I. Castellani, and D. Sangiorgi. On Bisimulations for the Asynchronous π-calculus. *Theoretical Computer Science*, 195:291–324, 1998.
3. F. Arbab, F.S. de Boer, and M.M. Bonsangue. A coordination language for mobile components. In *Proc. of SAC 2000*, pp. 166–173, ACM press, 2000.
4. F. Arbab, F.S. de Boer, and M.M. Bonsangue. A logical interface description language for componensts. In *Proc. of Coordination 2000*, LNCS, 2000.
5. F. Arbab, I. Herman, and P. Spilling. An overview of Manifold and its implementation. *Concurrency: Practice and Experience*, 5(1):23–70, 1993.
6. F.S. de Boer. Reasoning about asynchronous communication in dynamically evolving object structures. To appear in *Theoretical Computer Science*, 2000.
7. S.D. Brookes, C.A.R. Hoare, and W. Roscoe. A theory of communicating sequential processes. *Jour. Assoc. Comput. Mach.* 31:560–590, 1984.
8. M. Broy. Equations for describing dynamic nets of communicating systems. In *Proc. 5th COMPASS workshop*, vol. 906 of LNCS, pp. 170–187, 1995.
9. L. Cardelli and A.D. Gordon. Mobile ambients. In *Proc. of Foundation of Software Science and Computational Structures*, vol. 1378 of LNCS, pp. 140-155, 1998.
10. E.W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.
11. M. Falaschi, M. Gabbrielli, K. Marriot, and C. Palamidessi. Confluence in concurrent constraint programming. In *Theoretical Computer Science*, 183(2), 1997.
12. C. Fournet and G. Gonthier. The reflexive chemical abstract machine and the join calculus. In *Proc. POPL'96*, pp. 372–385, 1996.
13. R. Grosu and K. Stølen. A model for mobile point-to-point data-flow networks without channel sharing. In *Proc. AMAST'96*, LNCS, 1996.
14. C.A.R. Hoare. Communicating sequential processes. *Communication of the ACM*, 21(8):666-677, 1978.
15. G.J. Holzmann. The model checker SPIN *IEEE Transactions on Software Engineering* 23:5, 1997.

16. G. Kahn. The semantics of a simple language for parallel programming. In IFIP74 Congress, North Holland, Amsterdam, 1974.
17. He Jifeng, M.B. Josephs, and C.A.R. Hoare. A theory of synchrony and asynchrony. In *Proc. IFIP Conf. on Programming Concepts and Methods*, 1990.
18. B. Jonsson. A fully abstract trace model for dataflow and asynchronous networks. *Distributed Computing*, 7:197–212, 1994.
19. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, parts I and II. *Information and Computation* 100(1):1–77, 1992.
20. E.-R. Olderog and C.A.R. Hoare. Specification-oriented semantics for communicating processes. *Acta Informatica* 23:9–66, 1986.
21. L. Petre and K. Sere. Coordination among mobile objects. In *Proc. of Coordination'99*, vol. 1594 of LNCS, 1999.
22. G.-C. Roman, P.J. McCann, and J.Y. Plun. Mobile UNITY: Reasoning and specification in mobile computing. *ACM TOSEM* 6(3):250–282, 1997.