SPECIFICATION OF DYNAMIC RECONFIGURATION IN THE CONTEXT OF INPUT/OUTPUT RELATIONS

Ketil Stølen

Institute for Energy Technology
Postbox 173, N-1751 Halden, Norway
Ketil.Stoelen@hrp.no

Abstract: Recent advances in telecommunication and software technology have motivated the study of components with dynamically changing syntactic interfaces. Formal development methods are traditionally directed towards components with static interfaces. We investigate this short-coming of formal development methods and outline how it can be overcome.

We start by presenting a semantic model for interactive components communicating asynchronously by message passing. On the top of this model we build a simple specification language directed towards components with static interfaces. Then we generalise this language to handle components with dynamic interfaces. We introduce operators for composition and hiding.

Appeared in: Proc. FMOODS'99, pages 259-272, Kluwer Academic, 1999.

INTRODUCTION

Since the late 80ies much research within theoretical computer science has been directed towards dynamically reconfigurable networks. The emphasis has mainly been on semantic issues; in particular, how should dynamically reconfigurable networks be represented faithfully and fully abstractly. This has, for example, lead to the development of the Pi-calculus [15], and to new refinements of the Actor model [4]. Most of the early proposals have an operational flavour. Recent denotational approaches [11], [18] are rather technical, and in most cases, directed towards the Pi-calculus.

The above mentioned research attempts to find mathematical models suited to describe the behaviour of already completed systems. A formalism well-suited for describing an already completed system is not necessarily ideal as a specification language to be used in a process of step-wise system development, or as a notation for

formal reasoning and verification. On the contrary, formal development methods like VDM [12], B [3], TLA [14] and Unity [7] support system descriptions that combine abstract properties formulated in a purely descriptive (non-algorithmic) manner with more programming like notations. Abstraction is achieved by various means. Well-known techniques to achieve abstraction comprise:

- unbounded nondeterminism in the form of under-specification;
- fairness requirements and general liveness properties;
- unbounded memories and communication buffers;
- abstraction from causality;
- time abstraction including instantaneous communication and computation.

Formal development methods are mainly directed towards system components with static interfaces. This is a serious drawback when trying to tackle dynamically reconfigurable networks. We study this problem of formal development methods in the context of input/output-relations (I/O-relations) on streams. The use of I/O-relations to specify computerised components has a long tradition. For example, VDM and B are both based on this approach: A specification of a sequential component characterises the relationship between its initial and final states. Interactive components can be specified accordingly. Focus [6] is, for instance, based on I/O-relations: A specification of an interactive component C characterises the relationship between its tuples of input and output streams. A tuple of input streams represents histories of input messages sent by C's environment along C's input ports. A tuple of output streams represents histories of output messages sent by C itself along C's output ports. The syntactic interface of such a specification is static: Its input/output ports remain the same throughout the execution. There are of course various ways to generalise this specification paradigm to tackle dynamically changing interfaces. We advocate a minimalistic approach. As argued in [1]:

> "A new class of systems is often viewed as an opportunity to invent new semantics. A number of years ago, the new class was distributed systems. More recently, it has been real-time systems. The proliferation of new semantics may be fun for semanticists, but developing a practical method for reasoning formally about systems is a lot of work. It would be unfortunate if every new class of systems required inventing new semantics, along with proof rules, languages, and tools."

The approach outlined in this paper allows dynamically reconfigurable networks to be described at their level of abstraction with only minor modifications to the stream-based specification paradigm outlined above. As a consequence, the specification techniques, refinement principles and verification calculi already developed for I/O-relations on streams carry over straightforwardly.

The remainder of the paper is divided into six sections: Section 2 introduces some basic terminology; Section 3 outlines a semantic model for interactive components; Section 4 introduces a specification paradigm for components with static interfaces — referred to as static components in the sequel; Section 5 generalises this paradigm to tackle components with dynamic interfaces — referred to as dynamic components in the sequel; Section 6 motivates and defines several operators for hiding; Section 7 draws some conclusions.

BASICS

Streams, in their traditional form, are finite or infinite sequences of messages. They represent the communication histories of channels. The representation of communication histories is also the purpose of streams in this paper. However, inspired by [16, 13, 5], we work with the so-called timed streams — finite or infinite sequences that, in addition to ordinary messages, contain ticks represented by the symbol $\surd$. A timed stream ends with a tick if it is finite and contains infinitely many ticks, otherwise. The interval between two consecutive ticks represents a fixed least unit of time. For example, the timed stream

$$\langle m_1, m_2, \surd, m_3, \surd, \surd, m_4, \surd, \surd, \surd, m_5, m_6, m_7, \surd, \ \ldots \rangle$$

models a communication history whose messages are transmitted in the listed order and scheduled in accordance with the timing requirements imposed by the ticks. This implies that the first two messages are sent during the first time unit, the third message is sent during the second time unit, no message is sent during the third time unit, and so on. Since each infinite timed stream is required to contain infinitely many ticks, it follows that time never halts. In the sequel, whenever we refer to streams, we mean timed streams unless anything else is stated explicitly.

Let M denote the set of all messages and H the set of all infinite streams over M. $\surd^\infty$ denotes the stream consisting of $\surd$'s only. If x and y are streams then $x^\frown y$ denotes their concatenation, namely x if x is infinite, and the result of prefixing y with x, otherwise. We also need an operator that extracts the finite sequence of messages transmitted during a time unit. For any stream $x \in H$, $x[j]$ denotes the finite sequence of messages occurring before the first tick in x if $j = 1$, and the finite sequence of messages occurring between the $(j-1)$st and the jth tick in x if $j > 1$.

By N we denote the set of all channel names. We use the notation $a \mapsto x$ to assign a stream x to a single channel name a. We refer to $a \mapsto x$ as a maplet. H_A denotes the function domain $A \to H$; each element represents a named stream tuple. A named stream tuple $h \in H_A$ assigns a communication history $h(a)$ in the form of an infinite stream to each channel name $a \in A$. We may think of named stream tuples as sets of maplets. Consequently, if $A = \{a_1, a_2, \ldots, a_n\}$ and $h_1, h_2, \ldots, h_n \in H$ then

$$\{a_1 \mapsto h_1, a_2 \mapsto h_2, \ldots, a_n \mapsto h_n\} \in H_A$$

For any $h \in H_A$ and $j \in Nat$, $h{\downarrow_j}$ denotes the result of truncating each stream in h immediately after the jth tick if $j \geq 1$, and denotes

$$\{a \mapsto \langle\rangle \mid a \in A\}$$

where $\langle\rangle$ represents the empty stream, otherwise. We also need a projection operator. For any $h \in H_A$ and set of channel names B, by $h|_B$ we denote the element of $H_{A \cap B}$ such that

$$a \in A \cap B \Rightarrow h|_B(a) = h(a)$$

SEMANTIC MODEL

Consider a system component communicating asynchronously by message passing via unidirectional channels. Its syntactic interface is static in the sense that it has a fixed

number of input ports and a fixed number of output ports. Assume that the channels connected to these ports are named by sets of identifiers I and O, respectively. Hence, H_I is the set of possible histories for the input ports, and H_O is the set of possible histories for the output ports. We represent the I/O-relation of such a component by a set-valued function

$$\kappa \in H_I \to \wp(H_O)$$

By $\wp$ we denote the power-set operator. For each input history, κ yields a set of output histories. In this paper such functions are referred to as interaction functions. We consider many-to-many communication: Several components may have output ports for the same channel, and several components may have input ports for the same channel. A component may have both an input and an output port for the same channel. If $a \in I \cap O$ and $s \in \kappa(r)$ then $r(a)$ is the history of messages sent by the environment via the component's input port for a, and $s(a)$ is the history of messages that the component itself sends along its output port for a.

If we restrict ourselves to computerised systems, interaction functions are certainly too expressive. For example, the interaction function

$$\kappa \in H_{\{i\}} \to \wp(H_{\{o\}})$$

where

$$\kappa(\{i \mapsto \langle\surd\rangle^\frown h\}) = \{\{o \mapsto h\}\}$$

outputs the input received during time unit $j+1$ already during time unit j.

Hence, κ is non-causal in the sense that it describes an entity capable of predicting the future. We say that an interaction function κ is causal if its output until time $j+1$ depends only on input received until time j. Formally:

$$\forall r, s \in H_I; j \in Nat : r{\downarrow_j} = s{\downarrow_j} \Rightarrow \{t{\downarrow_{j+1}} \mid t \in \kappa(r)\} = \{t{\downarrow_{j+1}} \mid t \in \kappa(s)\}$$

κ is total if

$$\forall s \in H_I : \kappa(s) \neq \{\}$$

and deterministic if

$$\forall s \in H_I : \#\kappa(s) = 1$$

Any computerised component is total. After all, it will always react in some way or another to an input history — for example, by doing nothing or by breaking. A computerised component is also causal if the least time unit is chosen small enough. Hence, if the aim of this paper had been to come up with a model for computerised components only, it would have been natural to restrict our attention to functions that are both total and causal. However, the aim of this paper is to present a specification technique and its semantics. The main objective of a specification is to distinguish those computerised components that are acceptable as implementations from those that are not. Since all computerised components are causal, imposing causality does not reduce the set of possible implementations. It may, however, result in a more complicated specification. We therefore do not require specifications to be causal. In fact, we do not even insist on totality. A requirements specification will often contain conflicting requirements that may easily result in inconsistencies. Such inconsistencies

typically require careful analysis before they are eliminated. If we insist on totality such inconsistent specifications cannot be modelled.

Ideally, a specification should be total and contain the behaviours of at least one causal interaction function. This requires, of course, a notion of refinement allowing non-causal behaviours to be eliminated on the way to the final implementation. This is, however, not a problem given the concepts of refinement found in formal development methods like VDM, B, TLA, Unity and Focus: A specification C is a refinement of a specification A if $C \Rightarrow A$. Since $C \Rightarrow A$ if C describes the sub-set of causal behaviours allowed by A, it follows that non-causal behaviours can be refined away as any other aspect of under-specification. We may of course write a specification whose behaviours are completely captured by a causal interaction function when this is desirable. We say that such a specification is fully realizable. However, we do not restrict ourselves to fully realizable specifications.

We represent the composition of components by a specially designed operator $\otimes$. In order to define this operator, we first introduce a powerful merge function. For arbitrary sets of channel names A and B, we define

$$merge \in H_A \times H_B \to \wp(H_{A \cup B})$$

to be the function that yields a set of named stream tuples of which each element is a merge of the named stream tuples given as arguments. If A and B are disjoint then $merge(r, s)$ contains exactly one element — namely the result of gluing r and s together. If A and B are not disjoint, then the result is still deterministic for each channel name in $(A \setminus B) \cup (B \setminus A)$. For any other name $a \in A \cap B$, merge performs an instantaneous and nondeterministic merge of the streams $r(a)$ and $s(a)$. Instantaneous in the sense that the messages received during time unit j, namely $r(a)[j]$ and $s(a)[j]$ are output during time unit j. Nondeterministic in the sense that each interleaving of the finite streams $r(a)[j]$ and $s(a)[j]$ is a possible output.

$$\begin{array}{lcll} merge(r,s) & \equiv & \{\, t \in H_{A\cup B} \mid \forall a \in A \cup B : & \\ \qquad t(a) & = & r(a) & \text{if } a \in A \setminus B \\ \qquad t(a) & = & s(a) & \text{if } a \in B \setminus A \\ \qquad t(a) & \in & \{c \in H \mid \forall j \in N : c[j] \in mrg(\, r(a)[j], s(a)[j]\,)\,\} & \text{otherwise}\} \end{array}$$

$mrg(x, y)$ denotes the set of finite streams obtained by merging the two finite (untimed) streams x and y; thus, $mrg(x, y)$ is equal to the set of all possible interleavings of x and y.

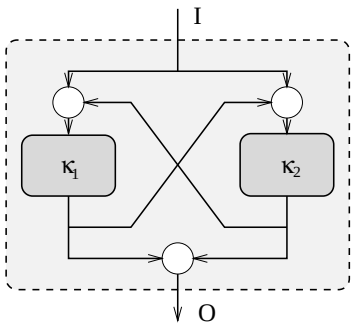

Figure 1 Parallel Composition

For arbitrary sets of channel names I, I_1, I_2, O, O_1 and O_2 such that $I = I_1 \cup I_2$ and $O = O_1 \cup O_2$, we define

$$\otimes \in (H_{I_1} \to \wp(H_{O_1})) \times (H_{I_2} \to \wp(H_{O_2})) \to (H_I \to \wp(H_O))$$

to be the operator such that

$$\begin{aligned}
\kappa_1 \otimes \kappa_2(r) \quad \equiv \quad \{ \quad s \in H_O \quad | \quad & \exists r_1 \in H_{I_1}; r_2 \in H_{I_2}; s_1 \in H_{O_1}; s_2 \in H_{O_2} : \\
& r_1 \quad \in \quad merge(r, s_2)|_{I_1} \\
& r_2 \quad \in \quad merge(r, s_1)|_{I_2} \\
& s_1 \quad \in \quad \kappa_1(r_1) \\
& s_2 \quad \in \quad \kappa_2(r_2) \\
& s \quad \in \quad merge(s_1, s_2) \qquad \}
\end{aligned}$$

The first instance of $merge$ merges the overall input with the output of κ_2; the second does the same for the output of κ_1; the third merges the outputs of κ_1 and κ_2. Figure 1 represents the merge functions as white balls. If both κ_1 and κ_2 contain the behaviours of at least one total, deterministic and causal interaction function then $\kappa_1 \otimes \kappa_2$ contains the behaviours of at least one total, deterministic and causal interaction function. This is a consequence of Banach's fixpoint theorem since causality in the deterministic case corresponds to contractivity with respect to the Baire metric [8].

Note that $\otimes$ does not feed the output of κ_1 (κ_2) back as input for κ_1 (κ_2). We may of course also define an operator that supports this kind of feedback.

SPECIFICATION OF STATIC COMPONENTS

So far we have represented I/O-relations by interaction functions. This is mathematically elegant, but not very practical when writing specifications. In a formal development method based on I/O-relations, a specification is typically a pair (Int, F) of a syntactic interface Int declaring the input/output observables and a formula F in predicate logic in which these input/output observables occur as free variables.

In this paper the input/output observables are channel ports. By $?a$ we denote an input port for the channel a, and by $!a$ we denote an output port for the channel a. Note that this is just a convention to simplify the presentation: A component in a dynamic network cannot exploit this convention to deduce the name of a port from the name of its complement port. This convention is overloaded to a set of channel names A in the obvious manner

$$?A \equiv \{\, ?a \mid a \in A\}, \quad !A \equiv \{\, !a \mid a \in A\}, \quad ?!A \equiv ?A \cup !A$$

This means that the set of all ports P is characterised by

$$P = ?!N = \{\, ?n, !n \mid n \in N\}$$

Hence, a specification of a static component is a pair

$$(Int, F)$$

where $Int \subseteq P$ and F is a formula whose free variables are contained in Int.

For example, the specification

$$(\{\, ?i_1,\, ?i_2, !o\}, !o = ?i_1)$$

describes a component with two input ports $?i_1$, $?i_2$ and one output port $!o$ that outputs along $!o$ what it receives via $?i_1$ without delay and ignores the input received on $?i_2$.

To define the meaning of such specifications in terms of the semantic model introduced above, we introduce a mapping $[\![\ ]\!]_{sta}$ which when applied to a specification $(?I \cup !O, F)$ yields an interaction function

$$[\![\, (?I \cup !O, F)\,]\!]_{sta} \in H_I \to \wp(H_O)$$

such that for all $r \in H_I$

$$s \in [\![\, (?I \cup !O, F)\,]\!]_{sta}(r) \Leftrightarrow F \wedge (\bigwedge_{i \in I} ?i = r(i)) \wedge (\bigwedge_{o \in O} !o = s(o))$$

We refer to $[\![\, S\,]\!]_{sta}$ as the denotation of the specification S.

The operator for the composition of interaction functions is lifted to specifications in the obvious manner

$$[\![\, S_1 \otimes S_2\,]\!]_{sta} \equiv [\![\, S_1\,]\!]_{sta} \otimes [\![\, S_2\,]\!]_{sta}$$

Of course, the specification paradigm outlined above is very simplified. For example, the ports are untyped, the constructs used to express F has not been introduced, and there is no syntactic sugar. For our rather theoretical discussions, this is not a problem. However, as demonstrated in [19, 20, 6], this paradigm can easily be extended to tackle problems of non-trivial size.

SPECIFICATION OF DYNAMIC COMPONENTS

A dynamic component differs from a static one in that it may dynamically gain and grant access to new ports by sending and receiving ports as messages. A dynamic component D has a set of initial ports $Init$; these are the ports D knows initially. Additionally, D may recursively gain access to new ports: The ports received via the input ports in $Init$, via input ports received via the input ports in $Init$, via input ports received via the input ports received via the input ports in $Init$, and so on.

We redefine H to be the set of all infinite timed streams over $M \cup P$. Assume $Init$ contains at least one input port. Since D in principle may receive any port via its initial input ports, and thereby gain send and receive access to any channel, the behaviour of D can, in general, be captured by an interaction function

$$\kappa \in H_I \to \wp(H_O)$$

if and only if $I = O = N$.

Assume we want to specify D. Since, contrary to earlier, ports can be transmitted as messages, it seems natural to make a clear distinction between the name of a port and the stream of messages sent or received via a port. We therefore introduce two specialised variables

$$in \in ?N \to H, \quad out \in !N \to H$$

to be used in a specification to look up the stream of a port. A specification is then a pair

$$(Init, F)$$

where $Init \subseteq P$ and F is a formula whose free variables are contained in $Init \cup \{in, out\}$. Since the streams for the input ports may contain new ports, it follows that F may depend on input ports and constrain output ports that are not in $Init$. To make sure that $(Init, F)$ gains access to new ports only via the streams for its initial input ports (in the recursive manner outlined above), we require that F is syntactically constrained from gaining access to new ports by other means. This means for example that F may refer to constants like $?N$, $!N$ and P only in a very restrictive manner. Since a detailed syntactic definition of the language used to express F is outside the scope of this paper, we do not list these constraints here.

Let $[\![\ ]\!]$ be the mapping which when applied to a specification $(Init, F)$ yields an interaction function

$$[\![\ (Init, F)\ ]\!] \in H_N \to \wp(H_N)$$

such that for all $r \in H_N$

$$s \in [\![\ (Init, F)\ ]\!](r) \Leftrightarrow F \wedge (\bigwedge_{n \in N} in(?n) = r(n) \wedge out(!n) = s(n))$$

Unfortunately, the interpretation $[\![\ (Init, F)\ ]\!]$ is too liberal since it allows arbitrary behaviour for those output ports that $(Init, F)$ does not gain access to: If there is some output port $!n \notin Init$ which $(Init, F)$ does not gain access to through $r \in H_N$, then $[\![\ (Init, F)\ ]\!](r)$ allows arbitrary behaviour for $!n$, and not just $\surd^\infty$ as we would have liked.

If F allows arbitrary output via the port $!n$ with respect to the input history r, we say that F is chaotic for n with respect to r. Formally:

$$chaotic(F, r, n) \equiv$$
$$\forall s \in H_N : s \in [\![(Init, F)]\!](r) \Rightarrow \forall h \in H : (s \dagger \{n \mapsto h\}) \in [\![(Init, F)]\!](r)$$

By $s \dagger \{n \mapsto h\}$ we denote the function obtained from s by redefining s to yield h when applied to n. Based on this predicate, we define the denotation of a specification $(Init, F)$ to be the interaction function

$$[\![(Init, F)]\!]_{dyn} \in H_N \to \wp(H_N)$$

such that for all $r \in H_N$

$$[\![(Init, F)]\!]_{dyn}(r) \equiv$$
$$\{s \in [\![(Init, F)]\!](r) \mid \forall n \in N : chaotic(F, r, n) \Rightarrow s(n) = \surd^{\infty} \vee\, !n \in Init\}$$

Hence, if F is chaotic for n with respect to r then this is interpreted to mean that no message (different from $\surd$) is sent along $!n$, unless $!n$ is an element of the initial interface. Note that with this interpretation we cannot specify a component that allows arbitrary behaviour along an output port that is not in its initial interface. However, from a pragmatic point of view, this is hardly a problem. Composition is redefined straightforwardly:

$$[\![S_1 \otimes S_2]\!]_{dyn} \equiv [\![S_1]\!]_{dyn} \otimes [\![S_2]\!]_{dyn}$$

Why do we put so much emphasis on restricting $(Init, F)$ from influencing ports it should not know? After all, in an earlier section, we argued against imposing causality, and the phenomenon discussed above may seem as just another instance of the same problem. Well, there is a fundamental difference: What we are concerned about here is hiding. Hiding, in contrast to causality, is of no importance for the actual computation of a system, but is essential for its comprehensibility. Carefully defined notions of hiding are prerequisites for clear interfaces and tight control with interference — issues of great concern to any designer of a specification language.

Before we study the issue of hiding in more detail, we have a look at a small example. A dynamically reconfigurable network differs from a static one in that direct channel connections can be built up between components that are without such connections initially. Of course, a direct connection can be built up between two components only if there already is some kind of indirect connection. In order to build up such a connection n from a component B to a component A there has to be some indirect connection, or at least a connection to some common source that informs A and B about n's existence. The network pictured in Figure 2 illustrates such a situation.

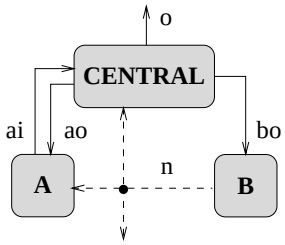

Figure 2 Dynamically Reconfigurable Network

We claim that this network captures the essence of dynamic reconfiguration (in the sense of the Pi-calculus). It consists of three components A, B and Central. The unbroken arrows describe the initial configuration; the broken arrows the connections that should be built up dynamically. Initially, there is no direct connection from A to B, but there is an indirect one via Central; from B to A there is no connection at all. The network is supposed to dynamically build up direct communication links from B to A, and also to the Central and the overall environment. For each connection request, represented by an arbitrary message, that A sends along $!ai$, the Central selects a new channel n, whose name is taken from an infinite set of channel names C; it sends the corresponding input port $?n$ along $!ao$ and $!o$, and the corresponding output port $!n$ along $!bo$. B may then send messages along $!n$ — messages that are received by the holders of the input port $?n$, namely A, the Central and the overall environment.

As explained above, we specify these components by pairs

$$(Init_{\mathrm{A}}, F_{\mathrm{A}}), \quad (Init_{\mathrm{B}}, F_{\mathrm{B}}), \quad (Init_{\mathrm{Central}}, F_{\mathrm{Central}})$$

where the first element describes the initial syntactic interface and the second element the actual behaviour. Clearly, we have that

$$Init_{\mathrm{A}} \equiv \{!ai, ?ao\}, \quad Init_{\mathrm{B}} \equiv \{?bo\}, \quad Init_{\mathrm{Central}} \equiv \{?ai, !ao, !bo, !o\} \cup ?!C$$

What A does with the messages it receives from B is of no importance for this paper; we therefore leave F_{A} unspecified. We just assume that A sends an arbitrary number of messages taken from M along $!ai$. What exactly B sends along the dynamically created channels is also of little interest here. We require only that it sends some arbitrary number of packages taken from a set $Package \subseteq M$. The behaviour of B is then formalised as follows:

$$F_{\mathrm{B}} \quad \equiv \quad \forall t \in !P : t \in \mathsf{rng}(\overline{in}(?bo)) \Rightarrow \exists p \in Package^{\omega} : \overline{out}(t) = p$$

$\overline{in}$ denotes the result of removing all $\surd$'s in the streams returned by in; $\overline{out}$ is defined accordingly. Hence, we abstract away the time information and leave the actual timing of the messages open. This is just a matter of convenience; if this component had been time-dependent, we could have used the ticks to impose time constraints. U^{ω} denotes the set of all untimed (finite and infinite) streams over U, and $\mathsf{rng}(s)$ yields the set of messages in the untimed stream s. Consequently, F_{B} states that an arbitrary number

of packages is sent along each output port received via $?bo$. Note that $[\![\ ]\!]_{dyn}$ implies that nothing (except infinitely many ticks) is sent along the output ports not received via $?bo$.

The behaviour of the Central is specified as follows:

$$
\begin{aligned}
F_{Central} \quad \equiv \quad & \exists prop \in C^{\infty} : \\
& \forall i, j \in Nat_{+} : i \neq j \Rightarrow prop.i \neq prop.j \\
& \overline{out}(!o) = \overline{out}(!ao) \\
& f(\overline{in}(?ai), prop) = (\overline{out}(!ao), \overline{out}(!bo)) \\
& \mathsf{where}\ \forall m \in M; p \in C; ir \in M^{\omega}; or \in C^{\infty} : \\
& \quad f(m \,\&\, ir, p \,\&\, or) = (?p, !p) \,\&\, f(ir, or) \\
& \quad f(\langle\rangle, or) = (\langle\rangle, \langle\rangle)
\end{aligned}
$$

The existentially quantified variable $prop$ is an oracle: It represents an infinite stream of channel names (element of C^{∞}) characterising the order in which channel names are selected from C. The first conjunct makes sure that $prop$ is without repetitions. Nat_{+} is the positive natural numbers. $s.j$ denotes the jth element of the infinite stream s. A line break without indentation represents conjunction; indentation captures scoping. The second conjunct makes sure that the stream of messages sent via $!o$ and $!ao$ is the same. The third conjunct requires that for each message received via $?ai$, an input and an output port for a new channel is sent along $!ao$ and $!bo$, respectively. $a \,\&\, s$ denotes the result of appending the message a to the head of s. The append operator is overloaded to pairs in the obvious manner. As already mentioned, $\langle\rangle$ denotes the empty stream.

HIDING

As already argued, hiding is an important issue for any designer of a specification language. If we compose the three specifications of the previous section using only $\otimes$, as below

$$((Init_{\mathrm{A}}, F_{\mathrm{A}}) \otimes (Init_{\mathrm{B}}, F_{\mathrm{B}})) \otimes (Init_{\mathrm{Central}}, F_{\mathrm{Central}})$$

we clearly have a hiding problem: The environment may not only observe what is sent along ai, ao and bo, it may also send messages along these channels itself. Of course, this is not only a problem in the dynamic case, also in a static network we would like to hide certain channels from the environment. To deal with this problem we introduce two hiding operators $(!n)$ and $(?n)$. For $n \in N$, we define

$$[\![(?n) : (Init, F)]\!]_{dyn}, [\![(!n) : (Init, F)]\!]_{dyn} \in H_N \to \wp(H_N)$$

to be the functions such that for all $r \in H_N$

$$
\begin{aligned}
[\![(?n) : (Init, F)]\!]_{dyn}(r) \quad & \equiv \quad \{s \mid s \in [\![(Init, F)]\!]_{dyn}(r \dagger \{n \mapsto \surd^{\infty}\})\} \\
[\![(!n) : (Init, F)]\!]_{dyn}(r) \quad & \equiv \quad \{s \dagger \{n \mapsto \surd^{\infty}\} \mid s \in [\![(Init, F)]\!]_{dyn}(r)\}
\end{aligned}
$$

We use $(a) : S$ as a short-hand for $(?a) : (!a) : S$ and $(a, b) : S$ as a short-hand for $(a) : (b) : S$. Both operators are generalised straightforwardly to tackle sets of channel names. We may then respecify our network as follows (under the assumption that $\otimes$ binds stronger):

$$(?C) : (ai, ao, bo) : ((Init_{\mathrm{A}}, F_{\mathrm{A}}) \otimes (Init_{\mathrm{B}}, F_{\mathrm{B}})) \otimes (Init_{\mathrm{Central}}, F_{\mathrm{Central}})$$

Clearly, the initial interface of the resulting network is $\{!o\} \cup !C$.

Although the channels ai, ao, bo and the ports $?C$ have been hidden, it may seem that we still have a hiding problem, because we do not really want the environment to observe what is sent along $!C$ unless it is informed about the existence of these ports via $!o$. But is this a problem? After all, if the environment is a component in our sense, which seems to be a reasonable assumption, and if $?C$ is not included in the initial interface of the environment then there is no way the environment can observe what is sent along $!C$ before it learns about these ports via $!o$.

To state explicitly in a specification that a set of ports should not be included in the initial interface of any component that is to be connected to the specified component, we may represent specifications by triples

$$(Ext_{Init}, Int_{Init}, F)$$

where the initial interface $Init$ has been split into two disjoint sets, namely an external interface Ext_{Init} and an internal interface $Init_{Init}$. If we then impose the syntactic requirement that two specifications cannot be composed by $\otimes$ if there is a port in the internal interface of one of these, whose complement port ($?n$ and $!n$ are complements) is in the internal or external interface of the other, then the problem has disappeared. It may seem strange to include information about internal names in the external interface. However, when working with truly distributed systems, where there is no central manager taking care of naming clashes, this seems unavoidable.

CONCLUSIONS

Formal developments methods are traditionally directed towards systems with static interfaces. This paper generalises a specification paradigm based on I/O-relations on streams to tackle the problem of specifying components with dynamic interfaces.

The proposed specification paradigm for dynamic components is a straightforward generalisation of the paradigm for static components. In contrast to [9, 10], the semantic mapping does not depend on recursive definitions of the domain and range of the corresponding interaction function. Moreover, the operators for composition and hiding have been simplified, and any dependency upon causality has been removed. The proposed approach is fully compositional and combines well with step-wise system development along the lines of Focus [6]: In fact the principles for property, interface and conditional refinement carry over straightforwardly.

There is an interesting relationship between our approach and the Actor model [4]. The Actor model is based on asynchronous message passing. Each actor has a unique address. An actor system maintains two sets of actor addresses: A set of addresses identifying the local actors known by the environment of the actor system, and a set

of addresses identifying the environment actors known by the actor system itself. The external interface of an actor system is equal to this pair of sets. Since messages may contain actor addresses, the sizes of these sets may increase at run-time. The Actor model is basically a programming notation extending call-by-value Lambda-calculus with three primitives for actor manipulation. If we interpret channel names as actor addresses, we may use our specification language to describe programs written in the Actor model. Our approach is also related to the work of Saraswat [17]. As pointed out in [2], there is also a relationship between our work and that of Abramsky.

This paper considers only many-to-many communication. We may describe various forms of point-to-point communication in a similar manner.

Acknowledgments

The author has benefited from discussions with Manfred Broy and Radu Grosu. In particular, the close collaboration with Radu Grosu has been very important. Demissie Aredo and Olaf Owe read a draft version and provided helpful feedback. The research presented in this paper has been carried out within the ADAPT-FT project. ADAPT-FT is financed through DITS — a strategic research program run by the Norwegian Research Council.

References

[1] M. Abadi and L. Lamport. An old-fashioned recipe for real-time. *ACM Transactions on Programming Languages and Systems*, 16:1543–1571, 1994.

[2] S. Abramsky. Retracing some paths in process algebra. In *Proc. CONCUR'96, LNCS 1119*, pages 1-17, 1996.

[3] J.R. Abrial. *The B Book: Assigning Programs to Meaning*. Cambridge University Press, 1996.

[4] G. Agha, I. A. Mason, S. Smith, and C. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7:1–72, 1997.

[5] M. Broy. Functional specification of time sensitive communication systems. *ACM Transaction on Software Engineering and Methodology*, 2:1–46, 1993.

[6] M. Broy and K. Stølen. Focus on system development. Book manuscript, June 1998.

[7] K. M. Chandy and J. Misra. *Parallel Program Design, A Foundation*. Addison-Wesley, 1988.

[8] R. Engelking. *General Topology*. PWN — Polish Scientific Publishers, 1977.

[9] R. Grosu and K. Stølen. A model for mobile point-to-point data-flow networks without channel sharing. In *Proc. AMAST'96, LNCS 1101*, pages 504–519, 1996.

[10] R. Grosu and K. Stølen. Specification of dynamic networks. In *Proc. NWPT'96*, pages 67–76. Universitetet i Oslo, 1997.

[11] L. J. Jagadeesan and R. Jagadeesan. Causality and true concurrency: A data-flow analysis of the pi-calculus. In *Proc. AMAST'95, LNCS 936*, pages 277–291, 1995.

[12] C. B. Jones. *Systematic Software Development Using VDM, Second Edition*. Prentice-Hall, 1990.

[13] J. N. Kok. A fully abstract semantics for data flow nets. In *Proc. PARLE'87, LNCS 259*, pages 351–368, 1987.

[14] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16:872–923, 1994.

[15] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, part I and II. *Information and Computation*, 100:1–77, 1992.

[16] D. Park. The "fairness" problem and nondeterministic computing networks. In *Proc. 4th Foundations of Computer Science, Mathematical Centre Tracts 159*, pages 133–161. Mathematisch Centrum Amsterdam, 1983.

[17] V. A. Saraswat, M. Rinard, and P Panangaden. Semantic foundations of concurrent constraint programming. In *Proc. POPL'91*, pages 333–352, 1991.

[18] I. Stark. A fully abstract domain model for the π-calculus. In *Proc. LICS'96*, pages 36–42. IEEE Computer Society Press, 1996.

[19] K. Stølen. Using relations on streams to solve the RPC-memory specification problem. In *Formal Systems Specification, The RPC-Memory Specification Case Study, LNCS 1169*, pages 477–520. 1996.

[20] K. Stølen and M. Fuchs. An exercise in conditional refinement. To appear in Prospects for Hardware Foundations, Springer, 1998.