Equations for Describing Dynamic Nets of Communicating Systems[1)]

Manfred Broy

Institut für Informatik, Technische Universität München
80290 München, Germany

Abstract. We give a notation and a logical calculus for the description and deductive manipulation of dynamic networks of communicating components. We represent such nets by hierarchical systems of recursive equations for streams. We give logical rules that describe the communication within a net and the dynamic creation of components, channels and rearrangement of the net structure. Such net transformations are based on a calculus of declarations of identifiers for data elements and especially for streams and equational logic. We demonstrate the modelling of interactive systems that correspond to dynamically changing net structures as obtained in systems with dynamic process creation (such as in object oriented approaches) within a framework of classical equational logic.

1. Introduction

It is one of the central issues of computing science to model distributed interactive information processing systems in terms of their static and dynamic properties. The description of systems by formal texts or graphics is static in nature, of course. The dynamic aspects of a system are also called its behaviour. For its description we need special techniques that allow to capture dynamic properties by static description techniques.

There are two options for describing the behaviour of networks of communicating agents:

- mathematical structures ("data structures") for the representation of the behaviour of systems (such as streams, traces, event structures, computation sequences) in terms of their histories,
- rules that reflect the changes (the computation steps) in a system (such as state machines, transition systems, operational semantics).

---

1) This work was partially sponsored by the Sonderforschungsbereich 342 "Werkzeuge und Methoden für die Nutzung paralleler Rechnerarchitekturen" and by the EC BRA Working Group COMPASS.

Typically some aspects of a system are of static nature (do not change over the life time of the system) while others are of dynamic nature.

So far we used the terms static and dynamic in a rather informal way. A formalization of these notions is hardly possible without choosing a formal model or a formal description technique for systems. But even then a characterisation of the notion of dynamic and static aspects is not obvious. Is a system invariant that asserts that the reachable system states always show a specific property considered as a static or a dynamic property?

Nevertheless, in spite of our difficulties to formally define the notions static and dynamic it is considered an important characteristic of a system which of its properties are of a static and which of a dynamic nature. Therefore it might be helpful to classify some basic system concepts. We may distinguish between external and internal properties of a system. External properties of a system are:

- the syntactic interface given by its communication channels to the outside world with their sorts.

Internal properties of a distributed interactive system are among others:

- the number of its subcomponents,
- the connections of its subcomponents (channels),
- the state shape (attributes and sorts).

In the following we give an approach based on logic for the description of interacting distributed systems of communicating components. We describe such systems by nets. We describe nets by hierarchical sets of equations which are logical formulas of a specific syntactic form.

The paper is structured as follows. In section 2 we introduce a very simple notation for data flow nets. Using this notation we describe nets by defining equations for streams. More precisely, we use a notation that allows to write hierarchical systems of equations for streams. With each such system of equations we associate a data flow graph that visualises the components and the communication structure of the described system model. This is worked out in section 3. In section 4 we give logical inference rules for the hierarchical systems of equations. These rules allow us to transform the syntactic structure of the equations without changing their logical meaning. The rules allow us in particular to change the structure of the network associated with the formulas. This allows to model the dynamic creation and deletion of channels and components. In section 5 we show how the rules can be generalised for changing also the syntactic interface of components to model the dynamic change of interfaces.

We treat mobile communication as a more elaborate example in section 6. In section 7 we show how to extend the approach to nondeterministic systems. In an appendix the most important notions for streams are given.

2. Representation of Nets

In this section we introduce a logical notation that allows us to describe data flow nets. We use specific logical formulas for the description of nets. A net description is given by a set of declarations. Declarations are equations for data elements, in our case equations for streams in a specific syntactic form. The equations contain free identifiers. The structure of the equations allows us to isolate a subset of these identifiers which we call *declared* or *output channels* while we call the others *input channels*.

The declarations on one hand define semantically a family of streams (one for each output channel) for any given assignment of streams to its input channels. The streams associated with the output channels denote the output histories of the described system.

The syntactic structure of the set of equations determines a data flow net which represents a communication structure of a system in terms of its components and communication channels.

2.1 Nets as Sets of Equations

We study equations formed by identifiers and terms built by the function symbols of a given signature Σ. A signature $\Sigma = (S, F)$ consists of a set S of sorts (including the specific sort Stream) and a set F of function symbols for which functionalities are specified.

Given a signature $\Sigma = (S, F)$ and a set X of S-sorted identifiers we can form terms over the function symbols in Σ and the identifiers in X. Every term has an individual sort.

Given terms t_1 and t_2 of the same sort we can form equations $t_1 = t_2$. An equation $x = t$ where x is an identifier and t is a term is called *explicit*.

Let $Z \subseteq X$ be a set of identifiers. For every identifier $z \in Z$ let t_z be a term of the same sort as z. Let

$$E = \{z = t_z : z \in Z\}$$

denote a set of explicit equations. The set E contains exactly one equation for every identifier in the set Z. Let the set

$$\text{Free}(E)$$

denote the set of identifiers obtained by the union of set Z and the set of free variables in the terms t_z. The set Z denotes the set of identifiers declared in E. Formally, we define this set of declared identifiers of the set of explicit equations E as follows:

$$\text{Dec}(E) = Z.$$

A set of explicit equations E where its set of declared identifiers Z are of sort Stream can be understood as the description of an interactive system. Each identifier $z \in Z$ stands for a communication channel, each equation stands for a computation node in the network. Every channel has exactly one source which is the equation in which it occurs on the left hand side and it has all nodes as targets which are the equations in which it occurs on the right hand side. This way a set of explicit equations can be understood as a network. Such a network can be represented more explicitly by a directed graph. The net can be understood as an interactive system that produces the streams assigned as the values of the declared identifiers as output. We formally associate a net to sets of equations in section 3.

However, often not all values of declared identifiers of a set of equations should be exported as output. Let $O \subseteq \text{Dec}(E)$ be the set of identifiers that we want to export as output. We denote a component with the channels in the set O of exported identifiers by the following expression:

O: E

O is called the set of *output channels* of the component O: E (where E is a set of explicit equations). The channels in the set

Dec(E)\O

are called *hidden*. The identifiers in the set Dec(E)\O are also called the *internal channels* of the net represented by E.

The set In(O: E) of *input channels* of the component described by O: E is defined by the following equations:

In(O: E) = Free(E) \ Dec(E)

Logically the effect of hiding channels can be expressed by existential quantification over the identifiers that are to be hidden. Let $\{x_1, ..., x_n\}$ be the set of hidden identifiers in the net O: E. Logically the term O: E is syntactic sugar. It represents the formula:

$$\exists\ x_1, ..., x_n: E$$

We define the set of externally visible declared identifiers for the term O: E as follows:

Dec(O: E) = O

If the set O, that is used to hide channels, is omitted, then this is equivalent to a component denotation where O = Dec(E).

2.2 Composition of Nets

We are not only interested in data flow nets with a flat communication structure as induced by the sets of explicit equations, but also in hierarchical nets that consist of components that again consist of networks of interacting components. Such *hierarchical nets* are obtained by putting nets together by parallel composition.

Let N_1 and N_2 be formulas denoting nets with disjoint sets of output identifiers for N_1 and N_2 , in mathematical terms $Dec(N_1) \cap Dec(N_2) = \emptyset$. We write

$$N_1 \parallel N_2$$

for the parallel composition of the nets N_1 and N_2.

The parallel composition operator leads to a simple formal notation for describing systems by hierarchical nets. With the possibilities to denote nets by sets of equations as introduced so far we obtain the following syntax for expressions denoting nets:

```
‹Net expression› ::=     ‹ID-Set›: ‹Net expression›  |
                         ‹EQ-Set› |
                         ‹Net expression› || ‹Net expression›

‹EQ-Set› ::=  { ‹explicit equation› [, ‹explicit equation›]* }
```

‹ID-Set› :: = { ‹id› [, ‹id›]* }

‹explicit equation› ::= ‹id› = ‹term›

We even allow the recursive declaration of such nets. For this purpose, we introduce identifiers for nets and write explicit (recursive) equations for nets. Given a net identifier f and a net expression N we write the equation

$f = N_0$

to associate a recursively declared net to the net identifier f. We, in addition, allow parameters in such equations for nets. For recursive nets we obtain the following syntax:

‹Net expression› ::= ‹ID-Set›: ‹Net expression› |
‹EQ-Set› |
‹Net expression› || ‹Net expression› |
‹Net-ID› {(‹term› {, ‹term› }*)}

where the nets associated to net identifiers are defined by net equations:

‹Net equation› ::= ‹Net-ID› { (‹id› {, ‹id› }*) } = ‹Net expression›

For a recursively defined net described by the net equation

$f = N$

we define the set of declared identifiers (along the lines of least fixpoint concepts) by the inclusion least set D for which the following formula holds:

$\mathrm{Dec}(f) = D \Rightarrow D = \mathrm{Dec}(N)$

The generalisation of this definition to parameterized net declarations is straightforward. Our notation for nets uses nothing than equational logic plus some syntactic sugar for hiding channels which logically can be replaced by existential quantification.

3. Nets as Hierarchical Data Flow Graphs

A net has a topological structure that can be represented by a directed graph where each arc has one component as its source and a set of components as its targets (a stream can be shared and consumed by several components). A special source of an arc (then we speak of an *input arc*) and a special target of an arc (then we speak of an *output arc*) is the environment of the net. Arcs are also called channels and logical represented by identifiers for data elements and in particular for streams.

With every net expression we associate a hierarchical data flow graph. This graph is defined as follows.

(1) With the net expression O: E formed of a set of equations E with output channels O we associate a node with the set O as output arcs and the set $(O \cup \mathrm{Free}(E))\backslash\mathrm{Dec}(E)$ as input arcs.

(2) With every expression O: (C_1 || ... || C_n) we associate a node with a data flow graph that has n nodes and the set

$$\cup (\{Dec(C_i): 1 \le i \le n \} \cup \{In(C_i): 1 \le i \le n \})$$

as its arcs (its channels).

By this definition we associate a hierarchical data flow graph with every net expression. However, a net expression can not only be seen as a net but also be understood as a logical formula. Every logical inference rule that is applied to transform the formula leads to a new formula and this way also to a new data flow net. To explain how nets are associated with systems of explicit equations we consider a simple example.

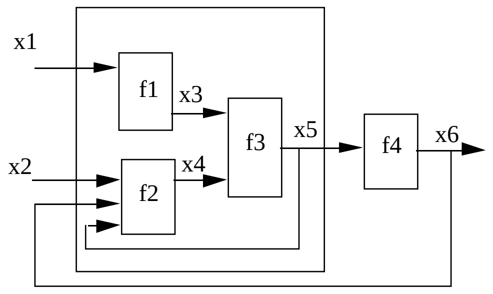

Fig. 1 A hierarchical data flow net

The hierarchical data flow net given in Fig. 1 corresponds to the following net expression:

{x6}: (({x5}: {x3 = f1(x1), x4 = f2(x2, x6, x5), x5 = f3(x3, x4)}) || {x6 = f4(x5)})

This example demonstrates how systems of explicit equations can be understood as a representation of a hierarchical data flow nets of interactive components.

4. A Calculus for Dynamic Nets

A set of explicit equations can be understood logically as a formula consisting the equations composed by logical conjunction. The notation

$$O: \{x_1 = t_1, ..., x_n = t_n\}$$

is understood as syntactic sugar for the logical formula

$$\exists\, z_1, ..., z_k: x_1 = t_1 \wedge ... \wedge x_n = t_n$$

where the set $\{z_1, ..., z_k\} = \{x_1, ..., x_n\} \setminus O$ denotes the set of internal channels.

Parallel composition of two networks corresponds to logical composition by logical conjunction of the respective equations.

Since our notation for nets can be understood as formulas in equational logic, we can use the calculus of equational logic to manipulate net expressions. We are in particular interested in logical inference rules that correspond to well-known transformations modelling the dynamic behaviour of communicating systems.

Given a net in terms of hierarchical sets of equations we apply substitutions to manipulate these equations. These substitutions can be used to mimic communication as well as the transformation of the net structure.

(1) *Rule of renaming of internal channels* (logical principle: α-renaming, renaming of bound identifiers): The following rule allows to rename an internal channel z to the identifier x provided x is a fresh identifier.

$$\frac{z \in \text{Dec}(E) \setminus O \qquad x \notin O \cup \text{free}(E)}{O\text{: } E = O\text{: } E[x/z]}$$

(2) *Rule of internal communication* (logical principle: redefining the stream x declared by the equation x = a & R to x = R and replacing simultaneously all occurrences of the identifier x on the right hand side of equations by the term a & x): The following rule allows to communicate the first message in a stream to all consumers of that stream.

$$\frac{(x = a \,\&\, R) \in E \qquad x \notin O \qquad x \notin \text{Free}(a)}{O\text{: } E = O\text{: } (E \setminus \{x = a \,\&\, R\})[(a \,\&\, x)/x] \cup \{x = R[(a \,\&\, x)/x]\}}$$

Note that a may be an arbitrary expression as long as the identifier x does not occur free in a.

(3) *Scope sharing* (logical principle: fusion of existential quantifiers): The following rule allows to share two disjoint scopes. By this rule a hierarchical data flow graph can be turned into a flat one.

$$\frac{(O_1 \cup \text{Dec}(E_1)) \cap (O_2 \cup \text{Dec}(E_2)) = \varnothing}{(O_1\text{: } E_1) \parallel (O_2\text{: } E_2) = (O_1 \cup O_2)\text{: } (E_1 \cup E_2)}$$

(4) *Nesting scopes* (logical principle: elimination of existentially quantified variables that do not occur free): This is a very simple rule that allows to simplify nested hiding.

$$O_1\text{: } O_2\text{: } E = O_1 \cap O_2\text{: } E$$

(5) *Local channel elimination/introduction* (logical principle: fold/unfold, replacing equal by equal): The following rule allows to introduce a channel or to eliminate a channel by introducing/eliminating a declaration.

$$\frac{(z = t) \in E \qquad z \notin O \cup \text{Free}(t)}{O\text{: } E = O\text{: } (E \setminus \{z = t\})\,[t/z]}$$

This rule can be applied to declarations for arbitrary elements including streams.

(6) *Recursion* (logical principle: unfold rule): Recursively defined networks can be unfolded according to the following rule.

$$\frac{f(x) = N}{O\text{: } (E \parallel f(t)) = O\text{: } (E \parallel N[t/x])}$$

These rules induce an equivalence relation on nets. These rules can be used to change the data flow graphs associated with a net.

(7) *Elimination of unused channels* (logical principle: elimination of conjunctions of existential quantified formulas that evaluate to true): The following rule allows to eliminate parts of a net (this may change the input alphabet).

$$\frac{\text{Dec}(E_2) \cap (O \cup \text{Free}(E_1)) = \emptyset}{O\text{: } E_1 \cup E_2 = O\text{: } E_1}$$

This rule allows in particular to eliminate net fragments that do not contribute to the output channels.

Our set of rules can be extended by rules that state least fixpoint principles for explicit equations (that can be viewed as declarations) and recursive equations for nets. We do not present such an extension here, since for our goals the simple set of rules given above is sufficient.

Especially interesting are the rules of communication and the rule of internal channel elimination and introduction. The rule of communication is demonstrated by the following simple example.

Example: Demonstration of the rule of communication. Let the function

succ*: Stream Nat $\to$ Stream Nat

be specified by the axiom;

succ*(n & s) = (n+1) & succ*(s)

The net expression

{y}: { x = 1 & succ*(x)} || { y = succ*(ft(x) & y) }

can be transformed by the rule of internal communication to the following logically equivalent expression:

{y}: { x = succ*(1 & x) } || { y = succ*(ft(1 & x) & y) }

which can be reduced by the axioms given for the function succ* and the function ft (see appendix) to the expression

{y}: { x = 2 & succ*(x) } || { y = 2 & succ*(y) } □

Let E1, E2 and E3 be sets of explicit equations. The application of the rule of internal channel elimination and introduction is illustrated by Fig. 2.

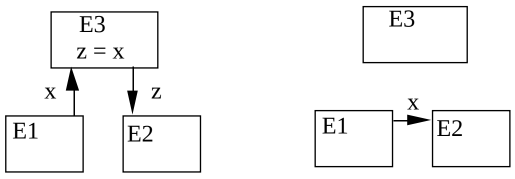

Fig. 2 Application of the rule of local channel elimination; we assume that z and x do not occur in E3

To illustrate our approach we use as a simple and well known example the sieve of Erathostenes.

Example: Sieve of Eratosthenes. Let the following function

$$\text{filter: Nat} \to (\text{Stream Nat} \to \text{Stream Nat})$$

be given. We specify it by the following axioms:

$$m \ \mathbf{mod}\ n = 0 \Rightarrow \text{filter}(n).(m \ \& \ s) = \text{filter}(n).s$$

$$m \ \mathbf{mod}\ n > 0 \Rightarrow \text{filter}(n).(m \ \& \ s) = m \ \& \ \text{filter}(n).s$$

We can use the function sieve to define a recursive net:

$$\text{sieve}(x, y) \equiv \{y\}: \{y = \text{ft}.x \ \& \ r, z = \text{filter}(\text{ft}.x).x\} \parallel \text{sieve}(z, r)$$

We obtain the sieve of Eratosthenes by the following net expression:

$$\{p\}: \ x = 2 \ \& \ \text{succ}^*(x)\} \parallel \text{sieve}(x, p) \ .$$

This net generates as output on its output channel p the stream of prime numbers in ascending order. This is not difficult to prove.

We demonstrate how we mimic the execution of the concurrent program associated with the net by our logical inference rules. We introduce an auxiliary identifier y for p such that p can accumulate the result stream:

{p}: {p = y, x = 2 & succ*(x)} || sieve(x, y)

≡ {*unfolding sieve*}

{p}: {p = y, x = 2 & succ*(x)} || {y}:{y = ft.x & r, z = filter(ft.x).x} || sieve(z, r)

≡ {*simplifying the result*}

{p}: {p = y, x = 2 & succ*(x), y = ft.x & r, z = filter(ft.x).x} || sieve(z, r)

≡ {*communication over x and simplifying the result*}

{p}: {p = y, x = succ*(2 & x), y = 2 & r , z = filter(2).x} || sieve(z, r)

≡ {*communication over y and simplifying the result*}

{p}: {p = 2 & y, x = succ*(2 & x), y = r, z = filter(2).x} || sieve(z, r)

≡ {*renaming r to y, eliminating z, unfolding succ**}

{p}: {p = 2 & y, x = 3 & succ*(x)} || sieve(filter(2).x, y)

≡ {*communication over x, unfolding filter, simplifying the result*}

{p}: {p = 2 & y, x = succ*(3 & x)} || sieve(3 & filter(2).x, y)

≡ {*simplifying the result, unfolding sieve*}

{p}: {p = 2&y, x = succ*(3&x), y = 3&r, z = filter(3).(3&filter(2).x)} || sieve(z, r)

≡ {*unfolding filter, communication over y, eliminating r*}

{p}: {p = 2 & 3 & y, x = succ*(3 & x), z = filter(3).filter(2).x} || sieve(z, y)

≡ …

These rewriting technique can easily be used as the basis for an inductive proof that shows that the output stream p is the stream of all prime numbers. □

The example demonstrates that an appropriate application of the logical inference rules allows to mimic the computation of the net.

5. Components with Dynamic Interfaces

So far we have given logical inference rules that correspond to equivalence transformations of hierarchical systems of equations that change the internal structure of the net associated with the formula but do not change its syntactic interface. In this section we show how to deal with dynamic interfaces in our setting.

The syntactic interface of a component represented by a set of equations is determined by its set of free identifiers, their sorts, and the information whether an identifier represents an output channel (is declared) or an input channel (is free, but not declared). As well understood in logic, we cannot replace a logical formula by a logically equivalent one which has a different set of free identifiers[1]. Therefore we cannot change the syntactic interface of nets by logical rules. However, if we have additional logical propositions we can change a formula. Consider the formula

(*) $x = a + b$

If we know in addition to (*) that

(**) $y = x - b + c$

holds, then under this assumption the formula (*) is equivalent to the formula

$x = a + b \wedge y = a + c$

This formula contains the free identifiers y and c not contained in (∗). We can obviously replace a formula by a logical equivalent one, containing a different set of free identifiers if we have additional propositions as assumptions. This idea will be used in the following to manipulate components with dynamic changing interfaces in given contexts.

5.1 Streams of Streams

The access interface of a system is determined by its set of input and output channels. For describing systems that are dynamically changing their access interface, we have to be able to change these sets in communication steps.

To increase the number of input channels of a component is not a problem, if we allow to transmit input channels (represented by identifiers) as values. Assume x has the sort Stream Stream M and y has the sort Stream M. Then we may allow defining equations for the stream ft.x of the form

[1] We may at most eliminate or introduce identifiers for which the formula does not contain any specific logical proposition. For instance, we can always add or eliminate tautologies such as x = x as long as we assume that all domains are nonempty.

ft.x = ...

When allowing these equations we can no longer guarantee the consistency of the logical formulas with such simple syntactic means as used so far. We may, however, use more sophisticated syntactical conditions to ensure consistency. We do not go into the discussion of such conditions, but rather give rules for dealing with systems of equation in the context of equations.

(1) *Rule of input channel renaming* (logical principle: equal for equal): If y is an input channel and from the context it is known that y = x holds then we can use x instead of y wherever we want.

$$\frac{y = x \qquad y \notin O \qquad x \notin \text{Free}(E)}{O\text{: } E = O\text{: } E[x/y]}$$

(2) *Rule of output channel renaming* (logical principle: equal for equal): An output channel y can be replaced by the channel x if x = y is ensured in the context.

$$\frac{x = y \qquad y \in O \qquad x \notin \text{Free}(E)}{O\text{: } E = (O\backslash\{y\}) \cup \{x\}\text{: } E[x/y]}$$

(3) *Rule of input channel introduction by communication* (logical principle: equal for equal): If x is a stream of streams and y is its first element then we can replace y by ft.x, if y is not an output channel.

$$\frac{x = y \,\&\, R \qquad y \notin O \qquad x \notin \text{Free}(E)}{O\text{: } E[\text{ft}.x/y] = O\text{: } E}$$

(4) *Rule of output channel introduction by communication* (logical principle: equal for equal): If y is the first element of stream x which is a stream of streams and x is an output channel then y can be added to the output channels.

$$\frac{y = \text{ft}.x \qquad x \in O \qquad y \notin \text{Dec}(E)\backslash O}{O\text{: } E \cup \{\text{ft}.x = R'\} = O \cup \{y\}\text{: } E \cup \{y = R'\}}$$

These rules allow to change the syntactic interface of components under certain context constraints that are again expressed by logical conditions.

5.2 Channels as Data Elements

In a more radical approach to systems with the behaviour of dynamic networks we may introduce a specific sort of channel names. Then channel names are data elements like other data, too. This has the disadvantage, however, that all the rules from equational logic, that are immediately available in the case of equational presentations of systems, now have to be redefined.

Similar to the π-calculus of Robin Milner (see [Milner et al. 92]) we now consider channels no longer as identifiers in logic but as elements of a set of channel names that can be both used as values and as identifiers. These elements then can be in particular communicated like values. For instance we write

$\exists$ x: {x = c & D, ft.x = D'}

The rule of communication allows us to change this formula to

$$\exists\ x: \{x = D[c \mathbin{\&} x / x],\ c = D'[c \mathbin{\&} x / x]\}$$

We may even recursively introduce an unbounded number of channels

$$\exists\ x: \{x = c \mathbin{\&} D\} \parallel X \qquad \textbf{where} \qquad X = \{ft.x = D',\ X[rt.x/x]\}$$

A further possibility to model changing syntactic interfaces is to allow equations (nets) be transferred as messages. We may write then

$$\exists\ x: \{x = \{y = D'\} \mathbin{\&} D,\ ft.x\ \}$$

and obtain by the rule of communication

$$\exists\ x: \{x = D,\ y = D'\}$$

However, we pay a high prize by introducing an explicit sort of channel names, since now all the logical rules of channels which above are introduced just as special cases of inference rules of equational logic have to be stated axiomatically. This would lead to a calculus with some similarity to π–calculus.

6. Example: Mobile Communication

In this section we give an extended example. We treat a system that consists of a number of mobile communication units (mcu), a number of transmission stations (ts), and of a transmission centre (tc). The system can be described by the net expression

$$tc \parallel ts_1 \parallel \ldots \parallel ts_n \parallel mcu_1 \parallel \ldots \parallel mcu_m$$

We describe a mobile communication unit by the function car:

$$car(x, y) = \{y\}: y = f^*(x)$$

where the function f* mimics the response of cars to input. For simplicity let us assume the following equation for f* (let f be some given function):

$$f^*(d \mathbin{\&} x) = f(d) \mathbin{\&} f^*(x)$$

Fig. 3 shows the structure of our network for mobile communication.

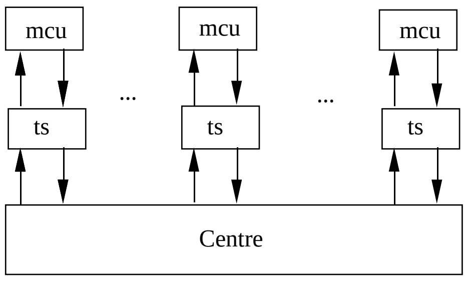

Fig. 3 Mobile Communication

We describe the transmission stations with the help of the function station. In the net expression

$$\text{station}(r, x, y, z)$$

the channels r and x serve as input channels and y and z serve as output channels. The channel r is supposed to carry the input from the transmission centre.

The input channel r of a station can carry messages of the following forms

connect(x, y)

disconnect

message(d)

The message connect(x, y) advises the station to work from now on with a car with input stream y and output stream x. In other words y is declared by the station and x is used. The message disconnect advises the station to drop its current connection and to send its current connection channels back to the centre via channel z using the message connect(x, y). The data message (d) is transmitted to the car it is connected to.

The defining equations for the stations are given in the following:

Establish car connection:

$$\text{station}(\text{connect}(x, y) \,\&\, r, x', y', z) \equiv \text{station}(r, x, y, z)$$

Disconnect car connection (x' and y' are internal channels):

$$\text{station}(\text{disconnect} \,\&\, r, x, y, z) \equiv$$

$$\{y, z\}\colon \{z = \text{connect}(x, y) \,\&\, z'\} \parallel \text{station}(r, x', y', z)$$

Transmit message to car and from car

$$\text{station}(\text{message}(d) \,\&\, r, x, y, z) \equiv$$

$$\{y, z\}\colon \{y = d \,\&\, y',\ z = \text{ft}.x \,\&\, z'\} \parallel \text{station}(r, \text{rt}.x, y', z')$$

We define

$$\text{mcu}_i = \text{car}(x_i, y_i)$$

$$\text{ts}_i = \text{station}(x_i, z_i, r_i, y_i)$$

Now the structure of the network can be changed individually according the messages send by the centre. We just show the interaction of the station with a mobile communication unit with the help of the following subsystem:

$$\{\text{out}\}\colon \{ r = \text{connect}(x, y) \,\&\, \text{message}(d) \,\&\, \text{disconnect} \,\&\, r,\ \text{out} = z\} \parallel \text{station}(r, x', y', z) \parallel \text{car}(y, x)$$

$\equiv$ *{communication on r}*

$$\{\text{out}\}\colon \{ r = \text{message}(d) \,\&\, \text{disconnect} \,\&\, r',\ \text{out} = z\} \parallel \text{station}(\text{connect}(x, y) \,\&\, r, x', y', z) \parallel \text{car}(y, x)$$

$\equiv$ *{unfold station}*

$$\{\text{out}\}\colon \{ r = \text{message}(d) \,\&\, \text{disconnect} \,\&\, r',\ \text{out} = z\} \parallel \text{station}(r, x, y, z) \parallel \text{car}(y, x)$$

≡ *{communication on r}*

{out}: { r = disconnect & r', out = z} || station(message(d) & r, x, y, z) || car(y, x)

≡ *{unfold station}*

{out}: { r = disconnect & r', out = z} ||
{y, z}: {y = d & y', z = ft.x & z'} || station(r, rt.x, y', z') || car(y, x)

≡ *{simplification, communication on y, elimination of x'}*

{out}: { r = disconnect & r', out = z, z = ft.x & z'} ||
station(r, rt.x, y, z) || car(d & y, x)

≡ *{unfold car, simplification}*

{out}: { r = disconnect & r', out = z, z = ft.x & z', x = f*(d & y)} ||
station(r, rt.x, y, z)

≡ *{unfold f*, communication over x, simplification}*

{out}: { r = disconnect & r', out = z, z = f(d) & z', x = f*(y)} || station(r, x, y, z)

≡ *{communication over z, elimination of z', simplification}*

{out}: { r = disconnect & r', out = f(d) & z, y = f*(x)} || station(r, x, y, z)

≡ *{communication over r}*

{out}: { r = r', out = f(d) & z, y = f*(x)} || station(disconnect & r, x, y, z)

≡ *{unfold station}*

{out}: { r = r', out = f(d) & z, y = f*(x)} ||
{y, z}: {z = connect(x, y) & z'} || station(x, z', r, y')

≡ *{communication over z, simplification}*

{out}: { r = r', out = f(d) & connect(x, y) & z', y = f*(x)} || station(x, z', r, y')

≡ ...

This example shows how the creation of new channels is treated in our calculus of declarations. All we have used here are rules of equational logic.

7. Extension to Nondeterministic Systems

Following [Broy 87a] we understand and specify nondeterministic components by sets of deterministic components. A deterministic component is represented by an equation

$$x = f(x_1, ..., x_n)$$

A nondeterministic component is represented by the following logical formula

$$\exists f: x = f(x_1, ..., x_n) \wedge Q.f$$

along the lines of [Broy 87a] where the predicate Q specifies a set of functions. The calculus introduced so far carries over to nondeterministic components in a straightforward way since logically all the rules for manipulating equations can also be applied if the respective equation is inside of existentially quantified formulas of the form given above.

Logically a nondeterministic component does not uniquely define its output streams. Therefore some of the steps of inference are not equivalence transformations but implication steps. Given a "nondeterministic system" represented by

$$x = E_1 \vee x = E_2$$

in a decision step we may replace it by

$$x = E_1$$

or by

$$x = E_2$$

as well. This shows that also from an operational point of view nondeterminism is adequately modelled that way.

8. Conclusion

It is interesting to observe that the logical notions of scope and equations can be made the basis for the dynamics of communicating systems. This is not surprising, since scoping and name spaces are essential for mobile communication and also an integral part of predicate logic.

Logic is very flexible. If the equation

$$x = t$$

holds in a particular scope, we can use x instead of t and vice versa everywhere in this scope without changing the meaning of the terms. This holds of course also, when the term t contains free identifiers as long as there are no scope conflicts.

Of course using logical rules in a flexible way does not lead to an operational semantics. Only if we add a strategy that expresses which rule is to be used in which situation a concrete operational semantics is obtained. A concrete operational semantics is given by the concurrent constraint logic approaches (cf. [Saraswat 89] and [Smolka 94]).

The relationship between logical inference rules and execution steps for interactive systems as well as the relation between logical expressions and networks gives an interesting way to model dynamic networks of communicating systems.

Software engineers have a very specific intuitive understanding of the dynamic behaviour of interactive systems when they talk about the creation and deletion of components. Obviously, they think about dynamically changing networks and with respect to components with dynamic changing interfaces. They like to think about situations, where suddenly there is a new channel created that they can use to communicate and also that a channel is deleted. This fits very well with the operational understanding of systems, where the system changes step by step its state. And the state is determined by the network of currently active and existing components.

It is quite clear that there are many ways to model such systems mathematically. For instance, such systems can of course be simply modelled by state machines where there is one state component which denotes the set of existing components.

What is the best way to represent the behaviour of dynamically changing systems is an open question. It is completely unclear whether it is better to find just very straightforward mathematical representations in terms of well-known structures where the dynamic behaviour is represented quite implicitly or if it is much better to have special calculi like Milner's π–calculus to represent the behaviour of such systems.

Acknowledgement

I thank Radu Grosu, Ketil Stølen, Klaus Dendorfer and Cliff Jones for stimulating discussions.

Appendix: Streams

Streams are used to denote histories of communications on channels. Given a set M of messages a stream over M is a finite or infinite sequence of elements from M. Let

$$(M \cup \{\perp\}, \sqsubseteq)$$

be a partially ordered set with least element $\perp$ which is complete (every directed set has a least upper bound).

By M^* we denote the finite sequences over M. M^* includes the empty sequence which is denoted by ‹›.

By M^∞ we denote the infinite sequences over the set M. M^∞ can be understood to be represented by the total mappings from the natural numbers $\mathbb{N}$ into M. We denote the set of streams over the set M by M^ω. Formally we have

$$M^\omega =_{def} M^* \cup M^\infty.$$

We introduce a number of functions on streams that are useful in system descriptions.

A classical operation on streams is the *concatenation* of two streams which we denote by ˆ. The concatenation is a function that takes two streams (say s and t) and produces a stream as result starting with s and continuing with t. Formally the concatenation has the following functionality:

$$.\hat{}.: M^\omega \times M^\omega \to M^\omega$$

If s is infinite, then concatenating s with t yields s again:

$$s \in M^\infty \Rightarrow s\hat{}t = s$$

Concatenation is associative and has the empty stream ‹› as its neutral element:

$$r\hat{}(s\hat{}t) = (r\hat{}s)\hat{}t, \qquad \langle\rangle\hat{}s = s = s\hat{}\langle\rangle,$$

For $m \in M$ we denote by ‹m› the one element stream. We write m & s for ‹m›ˆs.

On the set M^ω of streams we define a *prefix ordering* $\sqsubseteq$. We write $s \sqsubseteq t$ for streams s and t to express that s is a *prefix* of t. Formally we have for streams s and t:

$$s \sqsubseteq t \quad \text{iff} \quad \exists\, r, s' \in M^\omega: s'\hat{}r = t \wedge \forall\, i \in \{1,..., \#s\}: s.i \sqsubseteq s'.i.$$

The prefix ordering defines a partial ordering on the set M^{ω} of streams. If $s \sqsubseteq t$, then we also say that s is an *approximation* of t. The set of streams ordered by $\sqsubseteq$ is complete in the sense that every directed set $S \subseteq M^{\omega}$ of streams has a *least upper bound* denoted by lub S. A nonempty subset S of a partially ordered set is called *directed*, if

$$\forall\, x, y \in S: \exists\, z \in S: x \sqsubseteq z \wedge y \sqsubseteq z .$$

By least upper bounds of directed sets of finite streams we may describe infinite streams. Infinite streams are also of interest as (and can also be described by) fixpoints of prefix monotonic functions. The streams associated with feedback loops in interactive systems correspond to such fixpoints.

A *stream processing function* is a function

$$f: M^{\omega} \to M^{\omega}$$

that is *prefix monotonic* and *continuous.* The function f is called prefix *monotonic*, if for all streams s and t we have

$$s \sqsubseteq t \Rightarrow f.s \sqsubseteq f.t .$$

For better readability we often write for the function application f.x instead of f(x). The function f is called *continuous,* if for all directed sets $S \subseteq M^{\omega}$ of streams we have

$$\text{lub}\, \{f.s: s \in S\} = f.\text{lub}\, S .$$

If a function is continuous, then its results for infinite input can be already predicted from its results on all finite approximations of the input.

By $\perp$ we denote the pseudo element which represents the result of diverging computations. We write $M^{\perp}$ for $M \cup \{\perp\}$. Here we assume that $\perp$ is not an element of M. On $M^{\perp}$ we define also a simple partial ordering by:

$$x \sqsubseteq y \text{ iff } x = y \vee x = \perp$$

We use the following functions on streams

$$ft: M^{\omega} \to M^{\perp},$$

$$rt: M^{\omega} \to M^{\omega}.$$

They are defined as follows: the function ft selects the first element of a stream, if the stream is not empty. The function rt deletes the first element of a stream, if the stream is not empty. The properties of the functions can be expressed by the following equations that can also be used as defining axioms for them (let $m \in M$, $s \in M^{\omega}$):

$$ft.\langle\rangle = \perp, \qquad rt.\langle\rangle = \langle\rangle, \qquad ft(m\hat{\ }s) = m, \qquad rt(m\hat{\ }s) = s.$$

The concept of sequences is essential for modelling the stepwise proceeding of computations. When modelling a system component by a state machine or a transition system we obtain finite or infinite computations in the form of sequences of states.

References

[Broy 85]
M. Broy: Specification and top down design of distributed systems. In: H. Ehrig et al. (eds.): Formal Methods and Software Development. Lecture Notes in Computer Science 186, Springer 1985, 4-28, Revised version in JCSS 34:2/3, 1987, 236-264

[Broy 86]
M. Broy: A theory for nondeterminism, parallelism, communication and concurrency. Habilitation, Fakultät für Mathematik und Informatik der Technischen Universität München, 1982, Revised version in: Theoretical Computer Science 45 (1986) 1-61

[Broy 87a]
M. Broy: Semantics of finite or infinite networks of communicating agents. Distributed Computing 2 (1987), 13-31

[Broy 87b]
M. Broy: Predicative specification for functional programs describing communicating networks. Information Processing Letters 25 (1987) 93-101

[Dybier, Sander 88]
P. Dybier, H. Sander: A functional programming approach to the specification and verification of concurrent systems. Chalmers University of Technology and University of Göteborg, Department of Computer Sciences 1988

[Grosu 94]
R. Grosu: A formal foundation for concurrent object oriented programming. Ph. D. Thesis, Technische Universität München, Fakultät für Informatik, submitted 1994

[Kahn, MacQueen 77]
G. Kahn, D. MacQueen: Coroutines and networks of processes, Proc. IFIP World Congress 1977, 993-998

[Milner et al. 92]
R. Milner, J. Parrow, D. Walker: A calculus of mobile processes. Part i + ii, Information and Computation, 100:1 (1992) 1-40, 41-77

[Saraswat 89]
V.A. Saraswat: Concurrent constraint programming languages. Ph. D. Thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, CA, 1989

[Smolka 94]
G. Smolka: A calculus for higher order concurrent constraint programming with deep guards. DFKI Research Report RR-94-03