Modelling operating system structures by timed stream processing functions

MANFRED BROY AND CLAUS DENDORFER
Institut für Informatik, Technische Universität München, Postfach 20 24 20, D-8000 München 2, Germany

Abstract

Some extensions of the basic formalism of stream processing functions are useful to specify complex structures such as operating systems. In this paper we give the foundations of higher order stream processing functions. These are functions which send and accept not only messages representing atomic data, but also complex elements such as functions. Some special notations are introduced for the specification and manipulation of such functions. A representation of time is outlined, which enables us to model time dependent behaviour. Finally, we demonstrate how characteristic operating system structures can be modelled by timed higher order stream processing functions.

Capsule review

The authors propose modelling communicating systems by higher-order functions on streams, where a stream is either a finite or infinite sequence of information. Elements of streams are not restricted to atomic data but can be functions from streams to streams. After giving a number of useful operations on streams, they go on to outline a method for representing time in such a way that models at different granularities can be constructed. The authors also give sufficient conditions to enable one to translate between models at different time scales. Finally, the timed higher-order stream processing functions are illustrated by modelling certain operating system structures.

1 Introduction

In computer science, formal techniques show their full strength when applied to complex system structures. This is because such structures often exhibit peculiar characteristics due to technical and application dependent reasons. For instance, when specifying operating systems, concepts like time, resources, individual processes and their scheduling have to be modelled explicitly. Moreover, programs and processes occur both as data to be handled and as algorithms to be executed.

Higher order agents are communicating entities that do not only exchange atomic data, but also send and receive agents. A number of suggestions have been made to incorporate the passing of processes as messages into the well-known calculi of communicating systems such as CSP and CCS (Astesiano and Reggio, 1987; Hansen

https://doi.org/10.1017/S0956796800000241 Published online by Cambridge University Press

and Chao-Chen, 1990; Milner *et al.*, 1989; Nielson, 1989; Thomsen, 1989). We prefer to follow the concept of functional system models as outlined in Broy (1988), and extend this approach to 'higher order messages'. Higher order functions are also a common concept in functional programming.

Higher order functions can be used to specify communicating agents succinctly, and they provide appropriate models for particular system structures which can be found, for instance, in operating systems. Operating systems show typical characteristics of distributed systems because both consist of several processes, which act in parallel and are coordinated by the exchange of messages. An operating system receives not only first order messages (like signals, data, etc.), but also programs to be executed. This can be modelled by higher order stream processing functions. Since an operating system often exhibits time dependent behaviour, the formalism used must be powerful enough to express timing aspects.

Modelling operating system structures by stream processing functions is certainly not a new idea. See Kahn (1974) for an early example, which anticipates many of the later developments. In the world of functional programming, the advent of lazy evaluators (Friedman and Wise, 1976; Henderson and Morris, 1976) made stream processing and interaction possible. The functional implementation of operating systems is described in Karlsson (1981), Henderson (1982), Jones (1984) and Stoye (1986). The work of Jones and Sinclair (1989) contains an overview of these and other approaches.

We acknowledge that full scale operating systems have already been implemented in functional programming languages (Turner, 1990). In this paper, however, we are rather interested in the *specification* of structures that typically appear in operating systems or other distributed interactive systems. We start by outlining the basic theory of higher order streams and higher order stream processing functions. Then we introduce auxiliary notations to manipulate and specify such functions. Next we deal with the functional modelling of time. Based on the introduced notions we give some small examples of how to model characteristic operating system structures.

2 Streams and stream processing functions

Let M be a set with a partial order $\sqsubseteq$. We assume $\perp \notin M$ and write $M^{\perp}$ for $M \cup \{\perp\}$. Note that $\perp$ represents a pseudo value standing for 'no actual value'. The order $\sqsubseteq$ on M is extended to an order on $M^{\perp}$ by defining for all $m \in M$:

$$\perp \sqsubseteq \perp \wedge \perp \sqsubseteq m \wedge m \not\sqsubseteq \perp.$$

A stream over M is denoted by M^{ω}. It consists of the finite sequences M^{*} and the infinite sequences M^{∞}, which can be understood as mappings from the natural numbers $\mathbb{N}$ to M:

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

We use a few fundamental notations and functions, which are given in Fig. 1. Here, the law $\perp : s = \langle\,\rangle$ should especially be noted. For example, the stream $\langle 1 \rangle$ can be written as $1 : \langle\,\rangle$ and also as $1 : \perp : s$ for arbitrary finite or infinite streams s. Note that this is a property of the function $\_:\_$, which is a constructor in the sense of functional

https://doi.org/10.1017/S0956796800000241 Published online by Cambridge University Press

$\langle\rangle$	empty sequence
$\langle m\rangle$	one element sequence for $m \in M$
$\langle m_1, \ldots, m_n\rangle$	sequence containing n elements, for $n \in \mathbb{N}$ and $m_1, \ldots, m_n \in M$
m^n	sequence $\langle m, \ldots, m\rangle$ of length n, for $m \in M$ and $n \in \mathbb{N}$
$\#s$	length of a stream $s \in M^\omega$, where $\#s \in \mathbb{N} \cup \{\infty\}$
$m : s$	element $m \in M$ prefixed to a stream $s \in M^\omega$, and $\bot : s = \langle\rangle$

Fig. 1. Fundamental notation and functions.

$- \# - :: M^\omega \to M^\omega \to M^\omega$	concatenation of two streams (infix notation)
$\langle\rangle \# x = x$	
$(m : x) \# y = m : (x \# y)$	for $m \in M$
$hd :: M^\omega \to M^\bot$	first element of a stream
$hd.(m : x) = m$	for $m \in M^\bot$ (this implies $hd.\langle\rangle = \bot$ for $m = \bot$)
$tl :: M^\omega \to M^\omega$	rest of a stream without first element
$tl.\langle\rangle = \langle\rangle$	
$tl\,(m : x) = x$	for $m \in M$
$last :: M^\omega \to M^\bot$	last element of a finite non-empty stream
$last.x = \bot$	if $x = \langle\rangle \vee \#x = \infty$
$last.\langle m\rangle = m$	for $m \in M$
$last.(m : x) = last.\ x) = m$	for $m \in M$, if $x \# \langle\rangle \wedge \#x < \infty$
$- \copyright - :: \wp(M) \to M^\omega \to M^\omega$	filter operator (infix notation)
$S\copyright(m : x) = m : (S\copyright x)$	for $m \in S^\bot$ (this implies $S\copyright\langle\rangle = \langle\rangle$ for $m = \bot$)
$S\copyright(m\ .\ x) = S\copyright x$	for $m \in \mathrm{M} \setminus S$

Fig. 2. Standard functions on streams.

programming only with respect to elements from M, and not with respect to $\bot$. The special element $\bot$ must not appear in a stream. This is the reason why we need both finite and infinite streams.

The set M^ω is partially ordered by the *refined prefix order* $\sqsubseteq$, which is defined by the following axiom for all $x, y \in M^\omega$:

$$x \sqsubseteq y \equiv \forall i \in \mathbb{N} : x.i \sqsubseteq y.i.$$

With these definitions, M^ω forms a domain, i.e. a complete partially ordered set with least element $\langle\rangle$, provided that $M^\bot$ with the order $\sqsubseteq$ forms a domain.

We write $M \to N$ to denote the set of all functions from a set M to a set N, and we write $M \twoheadrightarrow N$ to denote the set of continuous functions between two domains M and N. Both arrow operators associate to the right. The set $M \twoheadrightarrow N$ is a domain if we use a partial order $\sqsubseteq$, such that for all $f, g \in M \twoheadrightarrow N$:

$$f \sqsubseteq g \equiv \forall x \in M : f.x \sqsubseteq g.x.$$

Fig. 2 presents a collection of well-known functions on streams, which we will use frequently. The application of a function f to an argument x is denoted by $f.x$. Function application associates to the left, so that $f.g.x$ stands for $(f.g).x$. Function application has a higher binding power than all other infix operators, therefore $f.x \# g.y$ means $(f.x) \# (g.y)$. With the exception of *last* and $\_ \copyright \_$, the functions defined so far are continuous, and therefore also monotonic, with respect to the order

https://doi.org/10.1017/S0956796800000241 Published online by Cambridge University Press

$\sqsubseteq$ on $M^{\perp}$ and M^{ω}, respectively. Concatenation is continuous in its second argument only.

We have seen two domain constructing operations, namely $\_^{\omega}$ and $\_ \twoheadrightarrow \_$, which may be applied repeatedly. Let M and N be domains. A *stream processing function* is an element of $M \twoheadrightarrow N$, such that $\_^{\omega}$ has been used at least once in the construction of both M and N. For a *higher order stream processing function* we have the additional requirement that $\_ \twoheadrightarrow \_$ has been used at least once in the construction of either M or N, or both.

As an example of a higher order stream processing function, consider the function f, which takes a stream of stream processing functions and concatenates their initial outputs, i.e. their respective first outputs for the empty input stream:

$$f :: (M^{\omega} \twoheadrightarrow M^{\omega})^{\omega} \twoheadrightarrow M^{\omega},$$
$$f.(g:x) = hd.(g.\langle\rangle):f.x.$$

Similar to typed and untyped λ-calculus, we distinguish between non-reflexive and reflexive function domains. A non-reflexive function domain is a domain of the form $M \twoheadrightarrow N$ that has been built by finitely many applications of domain constructing operations. The messages of a function from such a domain always have a strictly smaller order of type than the function itself. This is not so for functions from reflexive domains. As an example, consider the reflexive function domain FUN, which is defined as the least solution of the following equation. Note that we used a third domain constructing operation here, $\_ \cup \_$, which is similar to set union:

$$FUN = (M \cup FUN)^{\omega} \twoheadrightarrow (M \cup FUN)^{\omega}.$$

FUN denotes the set of those stream processing functions which take and produce streams that may contain elements from a basic message set M and functions from FUN. For instance, the function f defined above is an element of FUN. Moreover, reflexive domains open the possibility of self-application. The use of reflexive domains, however, increases the complexity of the semantic model. We will restrict ourselves to non-reflexive function domains for the rest of this paper since these seem to be sufficient for all practical applications.

3 Operations on stream processing functions

In this section we introduce a number of useful operations on functions. We start by describing how functions can generally be defined. Then we introduce notations to specify stream processing functions. Finally, we treat the class of those stream processing functions which operate on labelled streams.

3.1 General operations on functions

We view a function f as a mathematical object for which the two sets $DOMAIN.f$ (the *domain* of f) and $RANGE.f$ (the *range* of f) are defined, such that for every $m \in DOMAIN.f$ the function application $f.m$ yields an element of $RANGE.f$. Note that here we use the word 'domain' in a different sense than in the last section.

https://doi.org/10.1017/S0956796800000241 Published online by Cambridge University Press

The simplest possible function is the unique *empty function* ν described by:

$$\nu :: \varnothing \to \varnothing.$$

Another simple function which we may find useful is the mapping from a singleton set into another one. For all a and b, the function $a \mapsto b$ is defined as follows:

$$a \mapsto b :: \{a\} \to \{b\},$$
$$(a \mapsto b).a = b.$$

Let M and N be arbitrary sets. For the rest of this section we will assume that a function f from M to N is given

$$f :: M \to N.$$

Two functions with disjoint domains can be joined 'in parallel'. Let $f' :: M' \to N'$ such that $M \cap M' = \varnothing$ (or, more liberally, such that $f.m = f'.m$ for all $m \in M \cap M'$). Then we write $f \cup f'$ to denote the *join* of f and f'

$$f \cup f' :: (M \cup M') \to (N \cup N'),$$
$$(f \cup f').m = f.m \quad \text{for } m \in M,$$
$$(f \cup f').m = f'.m \quad \text{for } m \in M'.$$

Another operation which combines two functions is functional (sequential) composition: given a function g, where $RANGE.f = DOMAIN.g$

$$g :: N \to R.$$

Then we write $f;g$ for the *functional (sequential) composition* of f and g

$$f;g :: M \to R,$$
$$(f;g).m = g.(f.m).$$

Note the order of application. Functional composition is associative.

Given a set $M' \subseteq M$, we write $f|_{M'}$ to denote the *restriction* of f to M', i.e. the function which has domain M' and, within this domain, behaves like f

$$f|_{M'} :: M' \to N,$$
$$(f|_{M'}).m = f.m \quad \text{for } m \in M'.$$

Up to now we have seen three operations on functions, namely join, functional composition, and restriction. These operations take one or more functions and construct another function as a result. Here we consider it essential that the domain and range of the resulting function only depend on the domains and ranges of the argument functions, and not on their graphs.

We now present an example of an operation which changes only the graph of a function but leaves its domain and range invariant. Let $m \in M, n \in N$. Then the function resulting from a pointwise change of f is defined by

$$f[m \mapsto n] :: M \to N,$$
$$(f[m \mapsto n]).x = f.x \quad \text{if } x \neq m,$$
$$(f[m \mapsto n]).m = n.$$

https://doi.org/10.1017/S0956796800000241 Published online by Cambridge University Press

Note that we could also have defined $f[m \mapsto n]$ in terms of the operators given so far

$$f[m \mapsto n] = f|_{M \setminus \{m\}} \cup (m \mapsto n).$$

3.2 *Operations on simple stream processing functions*

The operations introduced in the last section applied to all kinds of functions. Now, we treat concepts tailored to stream processing functions.

Consider arbitrary domains $M^\perp$ and $N^\perp$, and let f be a continuous stream processing function

$$f :: M^\omega \to\!\!\!\!\to N^\omega.$$

For all $m \in M^\perp$ the function $f \ll m$ ('f after m') behaves like f after consuming element m

$$f \ll m :: M^\omega \to\!\!\!\!\to N^\omega,$$
$$(f \ll m).x = f.(m:x).$$

For all $n \in N^\perp$ the function $n \ll f$ ('n then f') outputs n and then behaves like f

$$n \ll f :: M^\omega \to\!\!\!\!\to N^\omega,$$
$$(n \ll f).x = n:f.x.$$

As a further overloading, $s \ll f$ will be defined for all streams $s \in N^\omega$. Note that in this case the function $\_ \ll \_$ is not monotonic in its first argument

$$s \ll f :: M^\omega \to\!\!\!\!\to N^\omega,$$
$$(s \ll f).x = s +\!\!+ f.x.$$

With these operations we can specify a number of well-known functions in a convenient way:

- The *map*-function, which performs pointwise application of a function to the elements of a stream. The middle line of the declaration below can be read: 'The function *map*.f, after taking in a message m, outputs $f.m$ and then behaves like *map*.f again.'

$$map :: (M \to M) \to M^\omega \to M^\omega$$
$$map.f \ll m = f.m \ll map.f \quad \text{for } m \in M$$
$$map.f \ll \perp = \perp \ll map.f.$$

- A simple interactive store, which always records the last data item received. The special symbol ? denotes a query operation.

$$f :: D \to (D \cup \{?\})^\omega \to D^\omega$$
$$f.d \ll ? = d \ll f.d$$
$$f.d \ll d' = f.d' \quad \text{for } d' \in D$$
$$f.d \ll \perp = \perp \ll f.d.$$

A non-deterministic interactive system is specified as a set of deterministic systems, i.e. as a predicate on stream processing functions. Every choice of a function that

https://doi.org/10.1017/S0956796800000241 Published online by Cambridge University Press

fulfils this predicate represents a possible behaviour of the system. The well-known merge anomaly (Brock and Ackermann, 1981), which arises if non-deterministic systems are specified by predicates on their input and output histories, is avoided in our approach. This has been shown in Broy (1988).

The notations presented above are useful in expressing specifications. Consider, for example, a system that inserts an arbitrary but finite number of special elements ✓ into a stream. This system can be characterized by the most liberal predicate which fulfils the following equation. Note that this equation has a uniquely defined weakest solution. One can read the specification as follows: 'A function represents a possible behaviour of the system if it takes an input m, then outputs some elements ✓, then outputs m, and then represents another possible behaviour of the system.'

$$P :: (\dot{M}^{\omega} \twoheadrightarrow (\dot{M} \cup \{\surd\})^{\omega}) \to \mathbb{B},$$
$$P.f \equiv \exists n \in \mathbb{N}, g : P.g \wedge (\forall m \in M^{\perp} : f \ll m = \surd^n \ll m \ll g).$$

3.3 Operations on labelled stream processing functions

Often it is convenient to consider stream processing functions with labelled input and output streams. The labels can be understood as channel or port names. Let $M^{\perp}$ be a domain, which is assumed to be fixed for the rest of this section. Then a *labelled stream processing function* is an element of the domain

$$(I \to M^{\omega}) \twoheadrightarrow (O \to M^{\omega}),$$

where I and O are two arbitrary and not necessarily disjoint sets of labels for the input and output streams, respectively. Since the arrow associates to the right we also write $(I \to M^{\omega}) \twoheadrightarrow O \to M^{\omega}$ for the above domain. Let f be a labelled stream processing function. We write

$$IN.f \quad \text{for } I,$$
$$OUT.f \quad \text{for } O.$$

Moreover, we use the ternary operators $\_ \ll \_?\_$ and $\_!\_ \ll \_$, which are defined as follows for all input labels $i \in IN.f$, output labels $o \in OUT.f$, and data items $m \in M^{\perp}$

$$f \ll i?m \quad \text{is the function defined by } (f \ll i?m).s = f.(s[i \mapsto (m : s.i)]),$$
$$o!m \ll f \quad \text{is the function defined by } (o!m \ll f).s = (f.s)[o \mapsto (m : f.s.o)].$$

These simple operations on labelled stream processing functions considerably support the process of writing specifications. For example, an agent *zip* which zips two streams together can be defined as follows:

$$zip :: (\{x, y\} \to M^{\omega}) \twoheadrightarrow \{r\} \to M^{\omega},$$
$$zip \ll x?m \ll y?m' = r!m \ll r!m' \ll zip.$$

Note that an expression like $f \ll x?m \ll y?m'$ does *not* impose any order on the processing of messages which arrive at different input ports. For any labelled function f with $\{x, y\} \subseteq IN.f$ and $x \neq y$, we have

$$f \ll x?m \ll y?m' = f \ll y?m' \ll x?m.$$

https://doi.org/10.1017/S0956796800000241 Published online by Cambridge University Press

Proof
Let $s \in (IN.f \to M^{\omega})$ be an arbitrary input channel state and $\{x, y\} \subseteq IN.f$ with $x \neq y$. We calculate

$$
\begin{aligned}
&(f \ll x?m \ll y?m').s \\
&\quad = (f \ll x?m).(s[y \mapsto (m' : s.y)]) && \text{[definition of } \_ \ll \_?\_\text{]} \\
&\quad = f.((s[y \mapsto (m' : s.y)])[x \mapsto (m : (s[y \mapsto (m' : s.y)]).x)]) && \text{[definition of } \_ \ll \_?\_\text{]} \\
&\quad = f.((s[y \mapsto (m' : s.y)])[x \mapsto (m : s.x)]) && \text{[since } x \neq y\text{]} \\
&\quad = f.((s[x \mapsto (m : s.x)])[y \mapsto (m' : s.y)]) && \text{[since } x \neq y\text{]} \\
&\quad = (f \ll y?m' \ll x?m).s && \text{[calculation above reversed]}
\end{aligned}
$$

Therefore the above theorem holds by extensionality. □

Again, non-deterministic behaviour can be expressed by giving a predicate on deterministic labelled stream processing functions. Consider, for instance, the predicate *MERGE*, which characterizes the set of deterministic merge functions. *MERGE* is the weakest predicate that fulfils the following equation

$$MERGE :: ((\{x, y\} \to M^{\omega}) \twoheadrightarrow \{r\} \to M^{\omega}) \to \mathbb{B},$$

$$MERGE.f \equiv \exists g : MERGE.g \wedge (\forall m \in M^{\perp} : f \ll x?m = r!m \ll g \vee f \ll y?m = r!m \ll g).$$

As an example of a function which fulfils the predicate *MERGE*, consider the function *zip* defined above.

A basic operation on labelled stream processing functions is the renaming of port labels. Let f be a labelled stream processing function. For $i \in IN.f$ and $i' \notin IN.f$ we define $[i'/i]f$ as the function which acts like f, but with input label i renamed to i'

$$[i'/i]f :: (((IN.f \setminus \{i\}) \cup \{i'\}) \to M^{\omega}) \twoheadrightarrow OUT.f \to M^{\omega},$$

$$([i'/i]f).x = f.(x|_{IN.f \setminus \{i\}} \cup (i \mapsto x.i')).$$

We use a symmetric notation to rename output labels, where $o \in OUT.f$ and $o' \notin OUT.f$

$$f[o'/o] :: (IN.f \to M^{\omega}) \twoheadrightarrow ((OUT.f \setminus \{o\}) \cup \{o'\}) \to M^{\omega},$$

$$(f[o'/o]).x = (f.x)|_{OUT.f \setminus \{o\}} \cup (o' \mapsto f.x.o).$$

Again, let f be a labelled stream processing function, and choose $I' \subseteq IN.f$ and $O' \subseteq OUT.f$. We assume that the result of f does not depend on any input channel from $IN.f \setminus I'$, i.e. that $f.x = f.y$ if only $x|_{I'} = y|_{I'}$. Then $f|^{I'}_{O'}$ is the restriction of f to input ports from I' and output ports from O'

$$f|^{I'}_{O'} :: (I' \to M^{\omega}) \twoheadrightarrow O' \to M^{\omega},$$

$$(f|^{I'}_{O'}).x = (f.x')|_{O'} \quad \text{where } x' :: IN.f \to M^{\omega} \quad \text{such that } x'|_{I'} = x.$$

Port names are not only helpful for the specification of individual agents, they can also be used to describe the connections in a whole network. For functions with port names, a uniform composition operator $\_ \parallel \_$ (called 'connect') can be defined. Our intuitive view of this operation is that ports with identical names should be connected. We require that there are not two output ports with the same name. Input ports which are connected to an output port can no longer be influenced by messages arriving

https://doi.org/10.1017/S0956796800000241 Published online by Cambridge University Press

from the environment. Assume that two labelled stream processing functions f and g are given, such that $OUT.f \cap OUT.g = \varnothing$. We write IN for $IN.f \cup IN.g$, and OUT for $OUT.f \cup OUT.g$. Then $f \| g$ is defined as the least solution of the equations

$$f \| g :: (IN \to M^{\omega}) \twoheadrightarrow OUT \to M^{\omega},$$

$$(f \| g).x = f.(x'|_{IN.f}) \cup g.(x'|_{IN.g}) \quad \text{where } x' = x|_{IN \setminus OUT} \cup ((f \| g).x)|_{IN \cap OUT}.$$

Let us consider a few special cases to obtain a clearer understanding of the connect operator. First we assume that f and g are independent and feedback free, i.e. that all four sets $IN.f$, $IN.g$, $OUT.f$, and $OUT.g$ are pairwise disjoint. Then $f \| g$ is simply the parallel composition of f and g, which can be expressed by a non-recursive equation

$$(f \| g).x = f.(x|_{IN.f}) \cup g.(x|_{IN.g}).$$

Next we consider the case that (some of) the output channels of f are connected to g, but that there are no feedback loops. Let $IN.f$, $IN.g$, $OUT.f$, and $OUT.g$ be pairwise disjoint, with the exception that $OUT.f \cap IN.g \neq \varnothing$. Then $f \| g$ can be seen as the sequential composition of the agents f and g. Again, a non-recursive equation, which is derived from the definition of $\_ \| \_$ by unfolding, is sufficient to express this

$$(f \| g).x = f.(x|_{IN.f}) \cup g.((f.(x|_{IN.f}))|_{IN.g \cap OUT.f} \cup x|_{IN.g \setminus OUT.f}).$$

If we furthermore require $OUT.f = IN.g$, then we even obtain

$$(f \| g)|^{IN.f}_{OUT.g} = f;g.$$

Finally, the special case of a simple (one-channel) feedback loop is treated. Let $IN.f \cap OUT.f = \varnothing$, and take $i \in IN.f$, $o \in OUT.f$. We write $f[i \leftarrow o]$ to denote the agent obtained from f by connecting output port o to input port i. The definition of $f[i \leftarrow o]$ uses the empty function ν as a dummy argument for the binary connect operator

$$f[i \leftarrow o] :: ((IN.f \setminus \{i\}) \to M^{\omega}) \twoheadrightarrow OUT.f \to M^{\omega},$$

$$f[i \leftarrow o] = ([o/i]f \| \nu)|^{IN.f \setminus \{i\}}_{OUT.f}.$$

As an alternative, $f[i \leftarrow o]$ can also be described as the least function which fulfils the following system of equations

$$(f[i \leftarrow o]).x = y \quad \text{where } y = f.(x \cup (i \mapsto y.o)).$$

An example of a network of labelled stream processing functions will be given in section 5.

4 Timed stream processing functions

The modelling of time in an interactive system requires a number of design decisions on the particular aspects which should be expressed in the model. Our approach is very simple, but it is nevertheless powerful enough to support the description of a number of interesting time dependent interactive systems.

4.1 Basic definitions

In our model we assume a global discrete time, where in every time interval at most one message can be sent or received. This may be understood as a restriction of the maximum data transfer speed of the channels. Each element of a timed stream

https://doi.org/10.1017/S0956796800000241 Published online by Cambridge University Press

represents the (single) communication event on a channel during one time interval. If the ith element is a message $m \in M$, then m has been sent onto the channel during the ith time interval. If the ith element is the special symbol $\surd$ (called 'tick'), then no proper message has been sent. We require $\surd \notin M$ and write $M^{\surd}$ for $M \cup \{\surd\}$. A *timed stream* is an element of $(M^{\surd})^{\omega}$, for which we write M^{τ}. Such a stream represents the (finite or infinite) timed history of the communication on some channel.

For the rest of this section we will assume that the partial order $\sqsubseteq$ on $M^{\surd}$ is the identity relation, i.e. that there are not two elements in $M^{\surd}$ which are non-equal but comparable. This implies that $(M^{\surd})^{\perp}$ is a flat domain, and that the refined prefix order $\sqsubseteq$ on M^{τ} actually is the 'standard' prefix relation.

Since our model reflects a global notion of time we require that the output history is fixed for a time interval of length n as soon as the input history is known for this interval. Hence a *timed* stream processing function is a continuous function f which operates on timed streams and has the *time progress property*

$$f :: M^{\tau} \to\!\!\!\!\!\to N^{\tau},$$
$$\forall x \in M^{\tau} : \#f.x \geqslant \#x.$$

We call f a *synchronous function* if in every time interval the output for exactly this interval is determined, i.e. if input and output always have the same length

$$\forall x \in M^{\tau} : \#f.x = \#x.$$

Similarly, f is called an *output delayed function* if at least a bit of its future behaviour is known in advance for all finite input streams

$$\forall x \in M^{\tau} : \#f.x > \#x \vee \#f.x = \infty.$$

Our intuitive understanding is that output delayed functions model systems which have a certain delay or propagation time.

The above concepts will also be applied to *timed labelled* stream processing functions. For this class of functions, the time progress property has the following form. Note that infinite output must be produced by a timed function f with $IN.f = \varnothing$. This is an immediate consequence of the definition below if we state that $\min \varnothing = \infty$

$$\forall x \in (IN.f \to M^{\tau}), o \in OUT.f : f.x.o \geqslant \min\{\#x.i \mid i \in IN.f\}.$$

A timed labelled stream processing function is synchronous if each output stream has exactly the length of the shortest input stream

$$\forall x \in (IN.f \to M^{\tau}), o \in OUT.f : \#f.x.o = \min\{\#x.i \mid i \in IN.f\}.$$

Finally, a timed labelled stream processing function is output delayed if each output stream is strictly longer than the shortest input stream, or infinite

$$\forall x \in (IN.f \to M^{\tau}), o \in OUT.f : \#f.x.o > \min\{\#x.i \mid i \in IN.f\} \vee \#f.x.o = \infty.$$

When specifying timed systems, we have to be careful to establish the time progress property. For instance, the standard function *tl* is not a timed function since it

https://doi.org/10.1017/S0956796800000241 Published online by Cambridge University Press

decreases the length of its input stream. A timed version of *tl* would replace the input stream's first element with a time tick. Our intuitive understanding is that the timed function *ttl* is idle during the first time interval, and from then on copies its input to the output channel

$$ttl :: M^{\surd} \twoheadrightarrow M^{\surd},$$
$$ttl = \surd \lll tl.$$

Sometimes we want to write a specification in which some output is produced step by step. Then the following operations $\Im$ (for 'initial') and $\Re$ (for 'remainder') can be employed

$$\Im.f = hd.(f.\langle\rangle),$$
$$\Re.f = f; tl.$$

Note that $\Im.f \lll \Re.f = f$ for all output delayed stream processing functions f. See section 5 for an example of the use of these operations.

4.2 Connecting timed functions

Shifting our attention from single functions to networks composed of functions, we now investigate the various composition operations. It is easy to see that sequential composition of synchronous and output delayed functions yields functions of the same respective class. Parallel composition preserves both the time progress and the output delay property, but not synchronicity, in general. The feedback of an output delayed function is output delayed again. This is an important observation, since it guarantees that a whole network of functions is output delayed if only the component functions are. Hence output delayed networks can easily be specified in a modular way.

On the other hand, the standard feedback operator applied to a synchronous function yields as result a function which does never produce any output, so that it certainly does not fulfil the time progress property. This problem can be remedied by using explicitly 'delayed' operators. For example, consider an alternative feedback operator which 'initializes' the feedback loop with a time tick. We define $f[i \overset{\surd}{\leftarrow} o]$ for $IN.f \cap OUT.f = \varnothing$, $i \in IN.f$, $o \in OUT.f$ as the least function which solves the following equations

$$f[i \overset{\surd}{\leftarrow} o] :: ((IN.f \setminus \{i\}) \to M^{\surd}) \twoheadrightarrow OUT.f \to M^{\surd},$$
$$(f[i \overset{\surd}{\leftarrow} o]).x = y \quad \text{where } y = f.(x \cup (i \mapsto (\surd : y.o))).$$

Let the function g be defined by $g.y = f.(x \cup (i \mapsto (\surd : y.o)))$. The delayed feedback operator has some interesting properties

(1) $f[i \overset{\surd}{\leftarrow} o]$ is synchronous for any synchronous function f.
(2) $f[i \overset{\surd}{\leftarrow} o]$ is output delayed for any output delayed function f.
(3) $f[i \overset{\surd}{\leftarrow} o]$ is timed (has the time progress property) for any timed function f.
(4) The function g has a unique fixed point for any synchronous function f.

https://doi.org/10.1017/S0956796800000241 Published online by Cambridge University Press

Proof of (1)
Let y' be an arbitrary fixed point of g. For all $o' \in OUT.f$ we calculate

$$\begin{aligned}
\#y'.o' &= \#(f.(x \cup (i \mapsto (\surd : y'.o)))).o' && \text{[definition of } g\text{]} \\
&= \min\{\#(x \cup (i \mapsto (\surd : y'.o))).i' \mid i' \in IN.f\} && [f \text{ is synchronous}] \\
&= \min(\{\#x.i' \mid i' \in IN.f \setminus \{i\}\} \cup \{1 + \#y'.o\}) && \text{[range splitting]} \\
&= \min\{\#x.i' \mid i' \in IN.f \setminus \{i\}\} && \text{[see below]}.
\end{aligned}$$

The last step is valid because the assumption $1 + \#y'.o < \min\{\#x.i' \mid i' \in IN.f \setminus \{i\}\}$ leads to a contradiction: let $o' = o$. Then $\#y'.o = 1 + \#y'.o$ by the first three steps of the calculation above and the assumption. This implies $1 + \#y'.o = \infty$, which is a contradiction because ∞ can not be strictly smaller than any value from the range of *min*. Thus we have shown that all output streams have the same length as the shortest input stream, which implies (1) by the definition of synchronicity. Note that we took y' to be *any* fixed point of g, not necessarily the least one. This will be important in the proof of (4). □

Proof of (2) and (3)
Analog to the above. □

Proof of (4)
Note that the functions used in the definition of g are continuous, hence a least fixed point y exists. By the proof of (1) we know that $\#y.o' = \#y'.o'$ for all fixed points y' of g and all $o' \in OUT.f$. If we had $y \neq y'$ then y and y' would not be comparable, so that y could not be the least fixed point. Hence $y = y'$ must hold. It should be noted that proposition (4) is only valid because we assumed that $(M^{\surd})^{\perp}$ is a flat domain. □

A property corresponding to (4) does not hold for output delayed functions (and therefore neither for general timed functions). Consider, as a counterexample, the output delayed function f defined as follows:

$$\begin{aligned}
f &:: (\{i, i'\} \to M^{\tau}) \twoheadrightarrow \{o\} \to M^{\tau}, \\
f.x &= (o \mapsto \langle\surd\rangle) && \text{if } \#x.i \leqslant 2 \wedge \#x.i' = 0, \\
f.x &= (o \mapsto (\surd : x.i)) && \text{otherwise.}
\end{aligned}$$

Now we have that $(f[i \overset{\surd}{\leftarrow} o]).(i' \mapsto \langle\rangle) = (o \mapsto \langle\surd\rangle)$, but the infinite channel state $(o \mapsto \langle\surd, \surd, \surd, \ldots\rangle)$ is a fixed point of g, too.

The general connect operator $\_ \| \_$ can also be redefined to insert explicit time ticks both into sequential composition and feedback. It depends on the physical characteristics of the modelled system whether or not the insertion of a time tick between two sequentially composed functions is appropriate. Such an operator, however, does not maintain synchronicity, i.e. it will generally turn synchronous functions into non-synchronous ones.

The exclusive use of output delayed functions seems to be a convenient way to avoid these specially defined operators. If the time scale is chosen fine enough, the causality between input and output is modelled appropriately, and more information

https://doi.org/10.1017/S0956796800000241 Published online by Cambridge University Press

is maintained. On the other hand, one could argue that the use of output delayed functions introduces unnecessary details (namely, explicit delay times) into a specification. This is a valid point. We will investigate how one can abstract from detailed timing information in the next section.

There is another problem that arises when composing timed stream processing functions: our basic assumption was that every element of a timed stream represented a certain fixed time interval. However, it is well possible that two timed functions have been designed to work with 'different speeds', i.e. on different time scales. (Note that this does not necessarily mean that the length of an interval is given in, say, microseconds). It is obvious that we can not just connect these functions without first converting them to a uniform time scale.

We will first tackle the problem of adapting single streams to different time scales. A timed stream is easily converted to a finer time scale by inserting a certain number of ticks before each element. For every $i \in \mathbb{N}$, the function $expand.i$ performs this scaling by a factor of $i+1$

$$expand :: \mathbb{N} \to M^{\surd} \twoheadrightarrow M^{\surd},$$
$$expand.i \ll m = \surd^i \ll m \ll expand.i.$$

The converse operation is a bit more complicated: for every time interval represented by $i+1$ elements on the finer time scale, any one message which occurred during this interval should be produced, if at least one such message exists. We make an arbitrary decision by defining that *compress* takes the *first* proper message from every time interval, or delivers a time tick if no such message occurred in a complete interval

$$compress :: \mathbb{N} \to M^{\surd} \twoheadrightarrow M^{\surd}$$
$$compress.i.x = \langle\,\rangle \qquad \text{if } \#x \leqslant i \wedge M \mathbin{©} x = \langle\,\rangle$$
$$compress.i.x = \langle hd.(M \mathbin{©} x)\rangle \qquad \text{if } \#x \leqslant i \wedge M \mathbin{©} x \neq \langle\,\rangle$$
$$compress.i.(\langle m_0, \ldots, m_i\rangle \mathbin{+\!\!+} x) = hd.((M \mathbin{©} \langle m_0, \ldots, m_i\rangle) \mathbin{+\!\!+} \langle\surd\rangle) : compress.i.x.$$

The following proposition can easily be proved by induction on the structure of x:

$$\forall i \in \mathbb{N}, x \in M^{\surd} : (expand.i ; compress.i).x = x. \qquad (*)$$

It is clear that compressing a stream, i.e. adapting it to a coarser time scale, may cause many data elements to be lost if there is not enough 'time' between the individual messages. A sufficient (and more than necessary) condition that compressing a stream by a factor of $i+1$ will not lose any messages is that the minimal space between two messages is at least i. We define the function *minspace* such that for a timed stream $x \in M^{\surd}$, $minspace.x$ is the shortest interval of ticks between two messages in x

$$minspace :: M^{\surd} \to \mathbb{N} \cup \{\infty\},$$
$$minspace.x = \min\{i \in \mathbb{N} \mid \exists y \in (M^{\surd})^*, m, m' \in M : y \mathbin{+\!\!+} \langle m\rangle \mathbin{+\!\!+} \surd^i \mathbin{+\!\!+} \langle m'\rangle \sqsubseteq \hat{x}\}.$$

In the definition above we used the refined prefix order $\sqsubseteq$ just as a convenient substitute for the standard prefix relation. No data will be lost by compressing a sufficiently sparse stream

$$\forall i \in \mathbb{N}, x \in M^{\surd} : minspace.x \geqslant i \Rightarrow M \mathbin{©} compress.i.x = M \mathbin{©} x.$$

Having defined how streams are converted between different time scales, it is now

https://doi.org/10.1017/S0956796800000241 Published online by Cambridge University Press

easy to state operations which perform the corresponding adjustment on functions. Here, the operator $\_\uparrow\_$ (read 'scale up by') adapts a function to an environment which works on a finer time scale, and $\_\downarrow\_$ (read 'scale down by') is the converse operation

$$f\uparrow i = compress.i; f; expand.i,$$
$$f\downarrow i = expand.i; f; compress.i.$$

With the up- and downscaling operations we may combine timed functions which are based on different time scales by speeding them up or slowing them down in an appropriate way. As an immediate consequence of (*), we obtain the proposition

$$\forall i \in \mathbb{N}, x \in M^{\tau}: (f\uparrow i)\downarrow i = f. \qquad (**)$$

Note that the converse of (**), namely $(f\downarrow i)\uparrow i = f$, does not hold in general. This problem will be addressed again in the next section.

Obviously, both the upscaling and the downscaling operations preserve the time progress property. Moreover, an output delayed function remains output delayed under upscaling, and a synchronous function remains synchronous under downscaling. Note that upscaling in general does not preserve synchronicity, and downscaling does not preserve the output delay property.

4.3 Time scale conversion as a means of abstraction

An important feature of a good specification formalism is that it allows us to view a system at different levels of abstraction. One method of abstraction is to convert the behaviour of a timed function to a coarser time scale. Hence the downscaling operator $\_\downarrow\_$ can also be seen as an abstraction operator.

When applying the abstraction operator to some function f, we have to be careful that no essential information is lost. In the following we will state conditions on f, which ensure that f is a proper candidate for time abstraction.

Remember that *compress.i* only takes the first message from an interval of length $i+1$, and discards subsequent messages. Since we do not want to lose any messages, the first informal requirement is

(1) If supplied with a sufficiently sparse input stream, f produces a sufficiently sparse result stream.

Now assume that the abstracted version of f receives some message m as an input. On the finer time scale, we really should consider *all* of the $i+1$ intervals (of length $i+1$) which contain i ticks and the message m, thereby obtaining a possibly non-deterministic abstraction. However, our deterministic downscaling operator just selects one of these $i+1$ intervals, namely the one which starts with m. Thus we would again lose information if this arbitrary selection actually had any effect observable in the coarser time modelling or, in other words

(2) If the input stream is sufficiently sparse, then the behaviour of f, as seen on the coarser time scale, only depends on those properties of the input which can also be observed on the coarser time scale.

We start to formalize these requirements by defining an equivalence relation $\sim_t$

https://doi.org/10.1017/S0956796800000241 Published online by Cambridge University Press

(called '*i*-space equivalence'). Two streams $x, y \in M^\tau$ are *i*-space equivalent iff they are equal on a time scale which is coarser by a factor of $i+1$

$$x \sim_i y \equiv compress.i.x = compress.i.y.$$

A function f is called *i-stable* iff the following formal translation of the requirements (1) and (2) holds for all streams $x, y \in M^\tau$

$$minspace.x \geqslant i \Rightarrow minspace.(f.x) \geqslant i \wedge \tag{1}$$

$$(minspace.x \geqslant i \wedge minspace.y \geqslant i \wedge x \sim_i y) \Rightarrow f.x \sim_i f.y. \tag{2}$$

It is easy to see that, in the context of time abstraction, f does indeed always receive sufficiently sparse input streams due to the properties of *expand*. Thus, for an *i*-stable function f, the downscaling $f \downarrow i$ can be understood as a proper deterministic time abstraction.

The notions introduced in this section to support time abstraction can also be used to express converse versions of last section's propositions (*) and (**). Let f be an *i*-stable function. Then, for all $x \in M^\tau$ and $i \in \mathbb{N}$, these implications hold

$$minspace.x \geqslant i \Rightarrow (compress.i; expand.i).x \sim_i x,$$

$$minspace.x \geqslant i \Rightarrow ((f \downarrow i) \uparrow i).x \sim_i f.x.$$

The possibility of switching back and forth between different time scales is vital both for the composition of timed functions and for the abstraction of lower level details. As we already mentioned in the last section, downscaling/abstraction need not preserve the output delay property, so that the already mentioned problems with feedback arise again. It is an interesting open question to find an abstraction method which guarantees that abstracted functions maintain the time progress property even under feedback.

4.4 The stability of functions

In the last two sections we saw that agents operating at different (but constant) speeds could easily be connected, and we introduced the notion of stability to denote a function which showed a similar reaction to a whole class of possible input streams. We will now merge the two concepts to characterize agents that are 'well-behaved' even for changing input speeds, if only certain timing constraints are met.

Consider, for example, an agent receiving keyboard input from a human operator. This agent will normally just echo its input. However, there should be some idle time after each arriving message to mask key bounces. Also a special timeout message m_t should be sent to, say, a screen saver if no proper input has arrived for some time. The function $f.1$ defined below represents such an agent with an idle time of one unit and a timeout threshold of nine units

$$\begin{aligned}
&f :: \mathbb{N} \to M^\tau \twoheadrightarrow M^\tau \\
&f.n \ll \surd = \surd \ll f.(n+1) && \text{if } n \neq 9 \\
&f.n \ll \surd = m_t \ll f.(n+1) && \text{if } n = 9 \\
&f.n \ll m = \surd \ll f.1 && \text{for } m \in M, \quad \text{if } n \neq 1 \\
&f.n \ll m = m \ll f.1 && \text{for } m \in M, \quad \text{if } n = 1 \\
&f.n \ll \bot = \bot \ll f.n
\end{aligned}$$

https://doi.org/10.1017/S0956796800000241 Published online by Cambridge University Press

What does it mean that a function behaves 'similarly' for two input streams? The first requirement (1) is that there are the same (proper) messages in both result streams. However, this is not enough because we want to distinguish, for instance, a function that immediately outputs a message from a function which outputs the same message only as a reaction to some input. Therefore, the two input streams will be partitioned into time intervals, such that every time interval (except the first one) starts with a message, and from then on contains ticks only. We state as a second requirement (2) that the same messages are produced during corresponding time intervals. Given a function f, two streams $x, y \in M^\infty$ are *f-equivalent* (written $x \sim_f y$) iff the two conditions are satisfied

$$x \sim_f y \equiv M \copyright f.x = M \copyright f.y \wedge \tag{1}$$

$$\forall x', y' \in (M^{\surd})^*, m, m' \in M : (\#(M \copyright x') = \#(M \copyright y') \wedge$$

$$x' +\!\!+ \langle m \rangle \sqsubseteq x \wedge y' +\!\!+ \langle m' \rangle \sqsubseteq y) \Rightarrow M \copyright f.x' = M \copyright f.y'. \tag{2}$$

We want to describe functions that show a similar reaction to input streams which have at least a spacing of i and at most a spacing of k. Here, i may be seen as the minimum required distance between two messages, and k as the maximum allowed distance. The auxiliary function *maxspace* returns the maximum number of consecutive ticks in a finite stream

$$maxspace :: (M^{\surd})^* \to \mathbb{N},$$

$$maxspace.x = \max\{i \in \mathbb{N} \mid \exists y \in (M^{\surd})^* : y +\!\!+ \surd^i \sqsubseteq x\}.$$

A function f is called *i, k-stable* for natural numbers i and k iff all finite streams which carry the same messages and fulfil the timing requirements are f-equivalent, i.e. iff the following implication holds for all $x, y \in (M^{\surd})^*$. Here we assume that m is an arbitrary element of M. The use of m is merely a coding trick to ensure that *minspace* will also consider initial and trailing sequences of ticks

$$(M \copyright x = M \copyright y \wedge minspace.(\langle m \rangle +\!\!+ x +\!\!+ \langle m \rangle) \geqslant i \wedge maxspace.x \leqslant k \wedge$$

$$minspace.(\langle m \rangle +\!\!+ y +\!\!+ \langle m \rangle) \geqslant i \wedge maxspace.y \leqslant k) \Rightarrow x \sim_f y.$$

Note that we defined i, k-stability using only finite streams to avoid some technical problems. This is sufficient as long as we restrict ourselves to continuous functions since such a function is 'i, ∞-stable' iff it is i, k-stable for all $k \in \mathbb{N}$. There are some obvious consequences of the above definition

(1) Every function is i, i-stable for all $i \in \mathbb{N}$.
(2) An i, k-stable function is also i', k'-stable for $i \leqslant i'$ and $k' \leqslant k$.

We can characterize a function f by giving its *stability regions*, i.e. a partition of the natural numbers such that f is i, k-stable (for $i \leqslant k$) iff i and k are in the same stability region. For example, the (single) stability region of the identity function is $\mathbb{N}$. The keyboard input agent $f.1$ defined above has the three stability regions $\{0\}$, $\{1, \ldots, 8\}$ and $\{9, \ldots\}$, corresponding to its three kinds of behaviour (deletion of input during idle time, normal operation, timeout). If we changed the agent such that repeated timeouts were produced, we would get extra stability regions representing those input

https://doi.org/10.1017/S0956796800000241 Published online by Cambridge University Press

streams for which one, two, three, etc., timeouts occur after every keyboard input. So the concept of stability allows us to investigate the time dependent properties of a function.

5 Modelling operating system structures

The modelling of operating system structures is a real test for a specification method. This is because many 'low level' concepts like time and shared resources have to be described. As a first example, we give the definition of a simple processor. This is an interactive system that takes a stream consisting of processes (modelled by functions) and input data. An arriving data element is fed into the current process, and one element of output is produced. An arriving new task replaces the current one. We will now define a function f, which represents such a multiplexing processor. Note that we expect f's first argument and all incoming functions to be output delayed, otherwise f may just deliver a partial result

$$
\begin{aligned}
&f :: (M^{\tau} \twoheadrightarrow M^{\tau}) \to ((M^{\tau} \twoheadrightarrow M^{\tau}) \cup M^{\surd})^{\omega} \twoheadrightarrow M^{\tau},\\
&f.g \lll d = \Im.g \lll f.(\Re.g \lll d) \quad \text{for } d \in M^{\surd},\\
&f.g \lll h = \surd \lll f.h \quad \text{for } h \in M^{\tau} \twoheadrightarrow M^{\tau},\\
&f.g \lll \bot = \bot \lll f.g.
\end{aligned}
$$

The arrival of a new task can be viewed as an interrupt. When this happens, both the old process and any pending outputs are lost. Since this is not a desirable behaviour, we will enhance our simple processor with a waiting queue for tasks. A newly arriving process is just added to the queue. Upon termination of the current task the next waiting process is started.

We will model the termination of a process by a predicate *finished* on tasks, which is true iff a task is completed

$$finished :: (M^{\tau} \twoheadrightarrow M^{\tau}) \to \mathbb{B}.$$

For instance, we could require that every task sent a special message m_c after completing its job, and define

$$finished.g \equiv \Im.g = m_c.$$

Then a processor with a queue of tasks is described by the function f, where

$$
\begin{aligned}
&f :: (M^{\tau} \twoheadrightarrow M^{\tau})^{*} \to ((M^{\tau} \twoheadrightarrow M^{\tau}) \cup M^{\surd})^{\omega} \twoheadrightarrow M^{\tau},\\
&f.(g:q) = f.q \quad \text{if } finished.g,\\
&f.(g:q) \lll d = \Im.g \lll f.((\Re.g \lll d):q) \quad \text{for } d \in M^{\surd}, \text{ if } \neg finished.g,\\
&f.q \lll h = \surd \lll f.(q +\!\!+ \langle h \rangle) \quad \text{for } h \in M^{\tau} \twoheadrightarrow M^{\tau},\\
&f.\langle\rangle \lll d = \surd \lll f.\langle\rangle \quad \text{for } d \in M^{\surd},\\
&f.q \lll \bot = \bot \lll f.q.
\end{aligned}
$$

Up to this point we have restricted our attention to stream processing functions that were small enough to be defined and understood without a descriptive formal specification of their behaviour. We will now show how a function can be specified

https://doi.org/10.1017/S0956796800000241 Published online by Cambridge University Press

by a predicate on its input and output streams. Such a descriptive specification is useful because it allows us to talk about the behaviour of an agent without anticipating its implementation. Thus we separate the two concerns of *what* an agent should do versus *how* this is achieved.

Again we consider a multiplexing processing unit, which is now separated into the processor proper and a scheduler. For simplicity, we assume that a set T of tasks is given, which carry all input and output data with them, so that there is no need for additional data items. We require that $(T^{\surd})^{\perp}$ is a flat domain. Furthermore, a predicate *finished* is needed, which tells us whether or not a task is completely evaluated

$$finished :: T \to \mathbb{B}.$$

The different priority levels of tasks are modelled by a total pre-order $\succcurlyeq$, such that for $t, t' \in T$, $t \succcurlyeq t'$ holds iff t has at least the same priority level as t'. This pre-order is extended onto $T^{\surd}$ by defining $t \succcurlyeq \surd$ and $\surd \not\succcurlyeq t$ for all $t \in T$. This means that the special message $\surd$ has a strictly lower priority level than any task

$$\_ \succcurlyeq \_ :: T^{\surd} \times T^{\surd} \to \mathbb{B}.$$

The whole processing unit is a timed labelled stream processing function, which is defined as a network of two other functions, namely the scheduler f and the processor g (see Fig. 3)

$$unit :: (\{new\} \to T^{\tau}) \twoheadrightarrow \{result\} \to T^{\tau},$$
$$unit = (f \parallel g)|^{\{new\}}_{\{result\}}.$$

We will not describe the processor g in detail. Our intuition is that g is an output delayed agent which accepts unfinished tasks, manipulates them in some way, and produces either finished results or tasks that require some more processing

$$g :: (\{process\} \to T^{\tau}) \twoheadrightarrow \{old\} \to T^{\tau}.$$

Our aim is to give a descriptive specification of the scheduling function f

$$f :: (\{new, old\} \to T^{\tau}) \twoheadrightarrow \{result, process\} \to T^{\tau}.$$

The scheduler is a synchronous agent which receives tasks on its input channels, stores the tasks if necessary, and sends the finished ones to the result channel and the unfinished ones back to the processor. Tasks with higher priority levels are sent first. The following list describes these requirements in a more exact, but still informal way. The requirements (4) and (5) would have to be strengthened if non-synchronous scheduler functions were permitted

(1) Only those tasks are produced as output which have been received as input.
(2) No finished tasks appear on the process channel.
(3) No unfinished tasks appear on the result channel.
(4) If there are pending finished tasks (i.e. finished tasks on the input channels which have not been output so far), then the last output on the result channel is a task which has at least the same priority level as these.
(5) If there are pending unfinished tasks, then the last output on the process channel is a task which has at least the same priority level as these.

https://doi.org/10.1017/S0956796800000241 Published online by Cambridge University Press

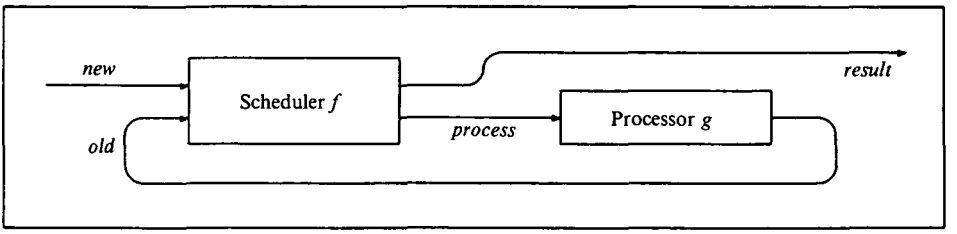

Fig. 3. A processing unit.

Note that a specification does not have to cover all possible input cases explicitly. For example, when thinking about the scheduler, we find it convenient to restrict our attention to finite input streams with equal length. We will later see that, as soon as the scheduler's behaviour is fixed for these inputs, its behaviour for *all* input streams is uniquely defined because of monotonicity and continuity (*).

A possible translation of the informal rules given above into a formal specification is that a scheduler is a synchronous function f such that the following implication holds for all $x \in (\{new, old\} \to T^{\tau})$, and for all $t \in T$. Generally, we do *not* require that there is only one function which meets a given specification. In fact, there exist many scheduler functions because our specification does not restrict the order in which two pending finished (or unfinished) tasks with the same priority level appear on the output channels

$$\#x.new = \#x.old \wedge \#x.new < \infty \wedge \#x.old < \infty$$

$$\Rightarrow$$

$$\#(\{t\} \copyright x.new) + \#(\{t\} \copyright x.old) \geqslant \#(\{t\} \copyright f.x.result) + \#(\{t\} \copyright f.x.process) \wedge \quad (1)$$

$$finished.t \Rightarrow \{t\} \copyright f.x.process = \langle\,\rangle \wedge \quad (2)$$

$$\neg finished.t \Rightarrow \{t\} \copyright f.x.result = \langle\,\rangle \wedge \quad (3)$$

$$(pending.t \wedge finished.t) \Rightarrow last.(f.x.result) \geqslant t \wedge \quad (4)$$

$$(pending.t \wedge \neg finished.t) \Rightarrow last.(f.x.process) \geqslant t. \quad (5)$$

Here we used $pending.t$ as an abbreviation of the formula

$$\#(\{t\} \copyright x.new) + \#(\{t\} \copyright x.old) > \#(\{t\} \copyright f.x.result) + \#(\{t\} \copyright f.x.process).$$

Now the following formal version of proposition (*) will be shown: let f and g be two scheduler functions such that $\#x'.new = \#x'.old < \infty$ implies $f.x' = g.x'$ for all input channel states x'. Then $f = g$.

Proof

Take an arbitrary input channel state $x \in (\{new, old\} \to T^{\tau})$, and let

$$n = \min\{\#x.new, \#x.old\}.$$

First we will consider the case $n < \infty$. Let $x' = (new \mapsto y) \cup (old \mapsto z)$ such that $y \subseteq x.new$, $\#y = n$ and $z \subseteq x.old$, $\#z = n$.

https://doi.org/10.1017/S0956796800000241 Published online by Cambridge University Press

Note that $\#x'.new = \#x'.old < \infty$, so $f.x' = g.x'$ holds. It only remains to show that $f.x' = f.x$ (and, symmetrically, $g.x' = g.x$)

$$\#f.x'.new = n = \#f.x.new \qquad [f \text{ is synchronous}]$$

$$\#f.x'.old = n = \#f.x.old \qquad [f \text{ is synchronous}]$$

$$f.x' \sqsubseteq f.x \qquad [f \text{ is monotonic, and } x' \sqsubseteq x]$$

$$f.x' = f.x \qquad [\text{from lines 1–3, since } (T^{\surd})^{\perp} \text{ is a flat domain}].$$

Thus we have shown $f.x = g.x$ for all finite input channel states x. Since f and g are continuous functions, their behaviour for the case $n = \infty$ only depends on their behaviour for finite inputs. Hence $f = g$ holds by extensionality. □

It is an interesting observation that, because of the explicit modelling of time, all progress requirements have already been captured by rules for the finite computations, together with continuity. Since liveness conditions on infinite behaviours are notoriously hard to deal with, this is quite a positive effect.

These examples demonstrate that typical concepts of operating systems can be modelled by higher order stream processing functions. Also, the proof techniques developed for functional system specification can be applied.

6 Concluding remarks

Higher order stream processing functions are a powerful and flexible instrument for the specification and modelling of all kinds of systems including operating systems. In some cases, the explicit inclusion of time is not only necessary, but can even lead to simpler specifications.

The functional treatment of operating system structures provides a solid foundation for their specification, modelling, simulation, and verification. In a first case study, the inter-process communication system of the *Multiprocessor Multitasking Kernel* (*MMK*) developed at the Technische Universität München has been specified with the techniques described in this paper (Dendorfer, 1991). Larger, more complex case studies should be the next steps to obtain additional evidence about the practicability of these techniques.

Acknowledgement

This work was supported by the Sonderforschungsbereich 342 Werkzeuge und Methoden für die Nutzung paralleler Architekturen.

References

Astesiano, E. and Reggio, G. 1987. SMoLCS-driven concurrent calculi. In H. Ehrig *et al.* (Editors), *Proc. Conference on Theory and Practice of Software Development. Lecture Notes in Computer Science 249*, pp. 169–201. Springer-Verlag.

Brock, J. D. and Ackermann, W. B. 1981. Scenarios: A model of non-determinate computation. In J. Díaz and I. Ramos (Editors), *Formalization of Programming Concepts. Lecture Notes in Computer Science 107*, pp. 252–259. Springer-Verlag.

Broy, M. 1988. Nondeterministic data flow programs: How to avoid the merge anomaly. *Science of Computer Programming* **10**: 65–85.

Broy, M. 1990. Functional specification of time sensitive communicating systems. In J. W. de Bakker *et al.* (Editors), *Stepwise Refinement of Distributed Systems. Lecture Notes in Computer Science 430*, pp. 153–179. Springer-Verlag.

Dendorfer, C. 1991. *Funktionale Modellierung eines Postsystems.* SFB-Bericht Nr. 342/28/91A. Technische Universität München.

Friedman, D. P. and Wise, D. S. 1976. Cons should not evaluate its arguments. In *Proc. 3rd Colloquium on Automata, Languages and Programming*, pp. 257–284, Edinburgh University Press.

Hansen, M. R. and Chao-Chen, Z. 1990. Specification and verification of higher order processes. In B. Rovan (Editor), *Mathematical Foundations of Computer Science. Lecture Notes in Computer Science 452*, pp. 322–327. Springer-Verlag.

Henderson, P. 1982. Purely functional operating systems. In Darlington *et al.* (Editors), *Functional Programming and its Applications.* Cambridge University Press.

Henderson, P. and Morris, J. H. 1976. A lazy evaluator. In *Proc. 3rd Conference on the Principles of Programming Languages*, pp. 95–103. ACM.

Jones, S. B. 1984. *A range of operating systems written in a purely functional style.* Technical Monograph PRG-42. Programming Research Group, Oxford University.

Jones, S. B. and Sinclair, A. F. 1989. Functional programming and operating systems. *The Computer Journal* **32**: 162–174.

Kahn, G. 1974. The semantics of a simple language for parallel programming. In J. L. Rosenfeld (Editor), *Information Processing 74*, pp. 471–475. Elsevier.

Karlsson, K. 1981. *Nebula—A functional operating system.* Laboratory for Programming Methodology, Chalmers University.

Milner, R., Parrow, J. and Walker, D. 1989. *A Calculus of Mobile Processes, Parts I and II. LFCS Reports 89-85 and 89-86.* Edinburgh University.

Nielson, F. 1989. The typed λ-calculus with first-class processes. In E. Odijk *et al.* (Editors), *Parallel Architectures and Languages Europe. Lecture Notes in Computer Science 366*, pp. 357–373. Springer-Verlag.

Stoye, W. 1986. Message-based functional operating systems. *Science of Computer Programming* **6**: 291–311.

Thomsen, B. 1989. A calculus of higher order communicating systems. In *Proc. 16th Symposium on Principles of Programming Languages*, pp. 143–154. ACM.

Turner, D. 1990. An approach to functional operating systems. In D. Turner (Editor), *Research Topics in Functional Programming*, pp. 199–217. Addison-Wesley.