From Communication Histories to State Transition Machines

Walter Dosch

Institute of Software Technology and Programming Languages
University of Lübeck
Lübeck, Germany

Abstract. The black-box view of an interactive component concentrates on the input/output behaviour based on communication histories. The glass-box view discloses the component's internal state with inputs effecting an update of the state. The black-box view is modelled by a stream processing function, the glass-box view by a state transition machine. We present a formal method for transforming a stream processing function into a state transition machine with input and output. We introduce states as abstractions of the input history and derive the machine's transition functions using history abstractions. The state refinement is illustrated with two applications, viz. iterator components and an interactive stack.

Keywords Interactive component, stream processing, state transition machine, communication history, history abstraction, iterator component, interactive stack

1 Introduction

A distributed system consists of a network of components that communicate asynchronously via unidirectional channels. The communication histories are modelled by sequences of messages, called streams. Streams abstract from discrete or continuous time, since they record only the succession of messages. The input/output behaviour of a communicating component is described by a stream processing function [9, 10] mapping input histories to output histories.

During the development of a component, the software designer employs different points of view. On the specification level, a component is considered as a black box whose behaviour is determined by the relation between input and output histories. The external view is relevant for the service provided to the environment.

On the implementation level, the designer concentrates on the component's internal state where an input is processed by updating the internal state. The internal view, also called glass-box view, is described by a state transition machine with input and output.

A crucial design step amounts to transforming the specified behaviour of a communicating component into a state-based implementation. In our approach,

we conceive machine states as abstractions of the input history. The state stores information about the input history that influences the component's output on future input. In general, there are different abstractions of the input history which lead to state spaces of different granularity.

This paper presents a formal method, called *state refinement*, for transforming stream processing functions into state transition machines. The transformation is grounded on history abstractions which identify subsets of input histories as the states of the machine. The state refinement preserves the component's input/output behaviour, if we impose two requirements. Upon receiving further input, a history abstraction must be compatible with the state transitions and with the generation of the output stream. The formal method supports a top-down design deriving the state-based implementation from a behavioural specification in a safe way.

The paper is organized as follows. In Section 2 we summarize the basic notions for the functional description of interactive components with communication histories. Section 3 introduces state transition machines with input and output. Section 4 presents the systematic construction of a state transition machine that implements a stream processing function in a correct way. History abstractions relate input histories to machine states. With their help, the transition functions of the machine can be derived involving the output extension of the stream processing function. In the subsequent sections, we demonstrate the state refinement for different types of applications. In Section 5 , the transformation of iterator components leads to state transition machines with a trivial state space resulting from the constant history abstraction. Section 6 discusses the state-based implementation of an interactive stack. The history abstraction leading to a standard implementation results from combining a control state and a data state in a suitable way.

2 Streams and Stream Processing Functions

In this section we briefly summarize the basic notions about streams and stream processing functions to render the paper self-contained. The reader is referred to [18] for a survey and to [19] for a comprehensive treatment.

2.1 Finite Streams

Streams model the communication history of a channel which is determined by the sequence of data transferred via a channel. Untimed streams record only the succession of messages and provide no further information about the timing.

Given a non-empty set $\mathcal{A}$ of data, the set $\mathcal{A}^\star$ of finite *communication histories*, for short *streams*, over $\mathcal{A}$ is the least set with respect to subset inclusion defined by the recursion equation $\mathcal{A}^\star = \{\langle\rangle\} \cup \mathcal{A} \times \mathcal{A}^\star$. A stream is either the *empty stream* $\langle\rangle$ or is constructed by the operation $\triangleleft : \mathcal{A} \times \mathcal{A}^\star \to \mathcal{A}^\star$ attaching an element to the front of a stream. We denote streams by capital letters and

elements of streams by small letters. A stream $X = x_1 \lhd x_2 \lhd \ldots \lhd x_n \lhd \langle\rangle$ $(n \geq 0)$ is denoted by $\langle x_1, x_2, \ldots, x_n\rangle$ for short.

The *concatenation* $X\&Y$ of two streams $X = \langle x_1, x_2, \ldots, x_k\rangle$ and $Y = \langle y_1, y_2, \ldots, y_l\rangle$ over the same set $\mathcal{A}$ of data yields the stream $\langle x_1, x_2, \ldots, x_k, y_1, y_2, \ldots, y_l\rangle$ of length $k+l$. The concatenation $X\&\langle x\rangle$ appending an element x at the rear of a stream X is abbreviated as $X \rhd x$.

2.2 Prefix Order

Operational progress in time is modelled by the prefix order. The longer stream forms an extension of the shorter history, and, vice versa, the shorter stream is an initial history of the longer stream.

A stream X is called a *prefix* of a stream Y, denoted $X \sqsubseteq Y$, iff there exists a stream R with $X\&R = Y$. The set of finite streams together with the prefix relation forms a partial order with the empty stream as the least element. Monotonic functions on finite streams possess unique continuous extensions to infinite streams [13].

2.3 Stream Processing Functions

The history of data passing along a communication channel between components and, possibly, their environment is mathematically captured by the notion of a stream. Thus, a deterministic component which continuously processes data from its input ports and emits data at its output ports can be considered as a function mapping input histories to output histories.

A *stream processing function* $f : \mathcal{A}^\star \to \mathcal{B}^\star$ maps an input stream to an output stream. The input type $\mathcal{A}$ and the output type $\mathcal{B}$ determine the syntactic *interface* of the component.

We require that a stream processing function is *monotonic* with respect to the prefix order: $f(X) \sqsubseteq f(X\&Y)$. This property ensures that a prolongation of the input history leads to an extension of the output history. A communicating component cannot change the past output when receiving future input.

A stream processing function describes the *(input/output) behaviour* of a component.

2.4 Output Extension

A stream processing function summarizes the behaviour of a component on entire input streams. A finer view reveals the causal relationship between single elements in the input stream and corresponding segments of the output stream.

The output extension isolates the effect of an input on the output stream after processing a prehistory.

Definition 1 *The* output extension $\varepsilon_f : \mathcal{A}^\star \times \mathcal{A} \to \mathcal{B}^\star$ *of a stream processing function* $f : \mathcal{A}^\star \to \mathcal{B}^\star$ *is defined by*

$$f(X \rhd x) = f(X)\ \&\ \varepsilon_f(X, x)\ . \tag{1}$$

The output extension completely determines the behaviour of a stream processing function apart from its result for the empty input.

3 State Transition Machines with Input and Output

The operational behaviour of distributed systems is often formalized by labelled state transition systems specifying a transition relation between states associated with labels [21]. The transitions denote memory updates, inputs, outputs, or other actions. For the purposes of modelling communicating components, we associate a state transition with receiving an element on the input channel and sending data to the output channel.

3.1 Architecture of the Machine

A state transition machine reacts on input with an update of the internal state generating a sequence of outputs.

Definition 2 *A* state transition machine with input and output*, for short a* state transition machine*,* $M = (\mathcal{Q}, \mathcal{A}, \mathcal{B}, next, out, q_0)$ *consists of a non-empty set* $\mathcal{Q}$ *of* states*, a non-empty set* $\mathcal{A}$ *of* input data*, a non-empty set* $\mathcal{B}$ *of* output data*, a* (single-step) state transition function $next : \mathcal{Q} \times \mathcal{A} \to \mathcal{Q}$*, a* (single-step) output function $out : \mathcal{Q} \times \mathcal{A} \to \mathcal{B}^\star$*, and an* initial state $q_0 \in \mathcal{Q}$*. The types* $\mathcal{A}$ *and* $\mathcal{B}$ *determine the* interface *of the state transition machine.*

Given a current state and an input, the single-step state transition function determines a unique successor state. The single-step output function yields a finite sequence of elements, not just a single element.

The single-step functions can naturally be extended to finite input streams.

Definition 3 *The* multi-step state transition function $next^\star : \mathcal{Q} \to [\mathcal{A}^\star \to \mathcal{Q}]$ *yields the state reached after processing a finite input stream:*

$$next^\star(q)(\langle\rangle) = q \tag{2}$$

$$next^\star(q)(x \lhd X) = next^\star(next(q, x))(X) \tag{3}$$

The multi-step output function $out^\star : \mathcal{Q} \to [\mathcal{A}^\star \to \mathcal{B}^\star]$ *accumulates the output stream for a finite input stream:*

$$out^\star(q)(\langle\rangle) = \langle\rangle \tag{4}$$

$$out^\star(q)(x \lhd X) = out(q, x) \,\&\, out^\star(next(q, x))(X) \tag{5}$$

The multi-step output function describes the (input/output) behaviour *of the state transition machine.*

For each state $q \in \mathcal{Q}$, the multi-step output function $out^\star(q) : \mathcal{A}^\star \to \mathcal{B}^\star$ constitutes a stream processing function. It abstracts from the state transitions and offers a history-based view of the component.

3.2 Output Equivalence

We aim at transforming a state transition machine into a more compact one with a reduced number of states without changing the behaviour. To this end, we are interested in states which induce an equal behaviour when the state transition machine receives further input.

Definition 4 *Two states $p, q \in \mathcal{Q}$ of a state transition machine $M = (\mathcal{Q}, \mathcal{A}, \mathcal{B}, next, out, q_0)$ are called* output equivalent*, denoted $p \approx q$, iff they generate the same output for all input streams: $out^\star(p) = out^\star(q)$.*

An observer of the state transition machine cannot distinguish output equivalent states, as they produce the same output stream for every input stream.

Proposition 1 *Successor states of output equivalent states are also output equivalent.*

3.3 Related models

State transition machines with input and output are closely related to a variety of state-based computing devices used to specify, verify, and analyse the behaviour of distributed systems, among others *generalized sequential machines* [7], *port input/output automaton* [11, 12], *Stream X-machines* [5] and X-machines [7], HAREL's *statecharts* [8], μ-Charts [16] and *UML state diagrams* [15, 14]. A different type of *state transition systems* were used in [1] for specifying the behaviour of components and, in particular, for the verification of safety and liveness properties [3].

4 From Stream Processing Functions to State Transition Machines

In this section, we implement stream processing functions by state transition machines using history abstractions. Given a stream processing function, we construct a state transition machine with the same interface and the same behaviour. The crucial design decision amounts to choosing an appropriate set of states. In our approach, the states of the machine represent sets of input histories that have the same effect on the output for all future input streams.

4.1 History Abstractions

A history abstraction extracts from an input history certain information that influences the component's future behaviour.

Definition 5 *For a stream processing function $f : \mathcal{A}^\star \to \mathcal{B}^\star$ and a set $\mathcal{Q}$ of* states*, a function $\alpha : \mathcal{A}^\star \to \mathcal{Q}$ is called a* history abstraction *for f, if it is* output compatible *(6) and* transition closed *(7):*

$$\alpha(X) = \alpha(Y) \Longrightarrow \varepsilon_f(X, x) = \varepsilon_f(Y, x) \tag{6}$$

$$\alpha(X) = \alpha(Y) \Longrightarrow \alpha(X \triangleright x) = \alpha(Y \triangleright x) \tag{7}$$

The output compatibility guarantees that a history abstraction identifies at most those input histories which have the same effect on future output. The transition closedness ensures that extensions of identified streams are identified as well:

$$\alpha(X) = \alpha(Y) \Longrightarrow \alpha(X\&Z) = \alpha(Y\&Z) \tag{8}$$

The transition closedness constitutes a general requirement, whereas the output compatibility refers to the particular stream processing function.

4.2 Construction of the State Transition Machine

When implementing a stream processing function with a state transition machine, the history abstraction determines the state space, the transition functions, and the initial state.

Definition 6 *Given a stream processing function $f : \mathcal{A}^\star \to \mathcal{B}^\star$ and a surjective history abstraction $\alpha : \mathcal{A}^\star \to \mathcal{Q}$ for f, we construct a state transition machine $M[f, \alpha] = (\mathcal{Q}, \mathcal{A}, \mathcal{B}, next, out, q_0)$ with the same interface as follows:*

$$next(\alpha(X), x) = \alpha(X \triangleright x) \tag{9}$$

$$out(\alpha(X), x) = \varepsilon_f(X, x) \tag{10}$$

$$q_0 = \alpha(\langle\rangle) \tag{11}$$

The state transition function and the output function are well-defined, since the history abstraction is surjective, transition closed, and output compatible.

The following proposition establishes the correctness of the implementation step.

Proposition 2 *Under the assumptions of Def. 6 , the stream processing function and the multi-step output function of the state transition machine agree:*

$$f(X) = f(\langle\rangle)\ \&\ out^\star(q_0)(X) \tag{12}$$

In particular, for a strict stream processing function we have $f = out^\star(q_0)$.

In general, a stream processing function possesses various history abstractions identifying different subsets of input histories as states.

The finest history abstraction is given by the identity function $\alpha(X) = X$ identifying no input histories at all. The associated state transition machine is called the *canonical state transition machine*. Its states correspond to input histories, the state transition function extends the input history input by input, the output function is the output extension.

The coarsest history abstraction $\alpha(X) = [X]_\approx$ maps every input history to the class of output equivalent input histories. The associated state transition machine possesses a minimal state space.

We can generalize the construction of the state transition machine to history abstraction functions that are not surjective. In this case, the state transition functions are uniquely specified only on the subset of reachable states. The transition functions can be defined in an arbitrary way on the subset of unreachable states; this will not influence the input/output behaviour of the machine starting in the initial state.

4.3 State Refinement

Every stream processing function can be transformed into a state transition machine with the same input/output behaviour using a history abstraction.

This universal construction lays the foundations for a formal method for developing a correct state-based implementation of a communicating component from its input/output-oriented specification. We call the formal method *state refinement*, since it transforms a component's communication-oriented black-box description into a state-based glass-box description. The history abstraction documents the essential design decisions for the state space. The state refinement complements other methods of refinement for communicating components, among others interface refinement [2] , property refinement [4], and architecture refinement [17] .

We presented the state refinement transformation $f \mapsto M[f, \alpha]$ for unary stream processing functions f only. The transformation generalizes to stream processing functions with several arguments in a natural way [19] .

5 History Independent Components

This section applies the state refinement transformation to the class of components whose behaviour does not dependent on the previous input history. We uniformly describe the set of history independent stream processing functions by a higher-order function. A constant history abstraction leads to an associated state transition machine with a singleton set as state space.

5.1 Iterator Components

An iterator component repeatedly applies a basic function to all elements of the input stream.

Iterator components are uniformly described by the higher-order function $map : [\mathcal{A} \to \mathcal{B}^\star] \to [\mathcal{A}^\star \to \mathcal{B}^\star]$ with

$$map(g)(\langle\rangle) = \langle\rangle \tag{13}$$

$$map(g)(x \triangleleft X) = g(x) \,\&\, map(g)(X) \ . \tag{14}$$

The higher-order function *map* concatenates the subsequences generated by the single input elements to form the output stream. For every basic function g , the function $map(g)$ distributes over concatenation. Therefore the function $map(g)$ is prefix monotonic. The output extension $\varepsilon_{map(g)}(X, x) = g(x)$ depends only on the current input, but not on the previous input history.

5.2 State Transition Machine of an Iterator Component

The history abstraction of an iterator component need not preserve any information of the previous input history. Thus any transition closed function forms a proper history abstraction, in particular, any constant function.

For constructing the state transition machine $M[map(g), const]$, we choose a singleton state space $\mathcal{Q} = \{q_0\}$ and a constant history abstraction $const(X) = q_0$. The resulting state transition machine is shown in Fig. 1.

$M[map(g), const] = (\{q_0\}, \mathcal{A}, \mathcal{B}, next, out, q_0)$
$next(q_0, x) = q_0$
$out(q_0, x) = g(x)$

Fig. 1. State transition machine of an iterator component

The history independent behaviour of an iterator component is reflected by a "state-free" machine whose singleton state is irrelevant.

Vice versa, any state transition machine $M = (\{q_0\}, \mathcal{A}, \mathcal{B}, next, out, q_0)$ with a singleton state implements the behaviour of an iterator component $map(g)$ where the basic function $g : \mathcal{A} \to \mathcal{B}^\star$ is defined as $g(x) = out(q_0, x)$.

Iterator components are frequently used in various application areas, among others in transmission components, processing units, and control components.

6 Interactive Stack

As a final application we construct the implementation of an interactive stack. The application shows how to combine a control abstraction and a data abstraction into an overall history abstraction.

6.1 Specification

An interactive stack is a communicating component that stores and retrieves data following a last-in/first-out strategy. The component reacts on requests outputting the last datum which has previously been stored, but was not requested yet.

The interactive stack is fault-sensitive: after a pop command to the empty stack, the component breaks and provides no further output whatsoever future input arrives.

Let $\mathcal{D}$ denote the non-empty set of data to be stored in the stack. The component's input $\mathcal{I} = \{pop\} \cup push(\mathcal{D})$ consists of pop commands or push commands along with the datum to be stored.

The component's behaviour forms a stream processing function $stack : \mathcal{I}^\star \to \mathcal{D}^\star$ defined by the following equations ($P \in push(\mathcal{D})^\star$):

$$stack(P) = \langle\rangle \tag{15}$$

$$stack(P \& \langle push(d), pop\rangle \& X) = d \triangleleft stack(P \& X) \tag{16}$$

$$stack(pop \triangleleft X) = \langle\rangle \tag{17}$$

A sequence of push commands generates no output (15). A pop command outputs the datum stored most recently (16). After an erroneous pop command, the interactive stack breaks (17).

The behaviour of the interactive stack leads to the output extension $\varepsilon_{stack} : \mathcal{I}^\star \times \mathcal{I} \to \mathcal{D}^\star$ defined by $(P \in push(\mathcal{D})^\star)$:

$$\varepsilon_{stack}(X, push(d)) = \langle\rangle \tag{18}$$
$$\varepsilon_{stack}(P \triangleright push(d), pop) = \langle d\rangle \tag{19}$$
$$\varepsilon_{stack}(\langle\rangle, pop) = \langle\rangle \tag{20}$$
$$\varepsilon_{stack}(pop \triangleleft X, pop) = \langle\rangle \tag{21}$$
$$\varepsilon_{stack}(P\&\langle push(d), pop\rangle\&X, pop) = \varepsilon_{stack}(P\&X, pop) \tag{22}$$

A push command generates no output after any input history (18). A pop command yields the datum stored most recently which was not requested yet (19) unless the stack contains no datum (20,21).

6.2 Control Abstraction

The future behaviour of a fault-sensitive stack is influenced by the occurrence of an illegal pop command in the preceding input history.

We discriminate between regular and erroneous input histories using a binary control state $Control = \{reg, err\}$. The control abstraction $control : \mathcal{I}^\star \to Control$ classifies input histories as regular or erroneous $(P \in push(\mathcal{D})^\star)$:

$$control(P) = reg \tag{23}$$
$$control(P\&\langle push(d), pop\rangle\&X) = control(P\&X) \tag{24}$$
$$control(pop \triangleleft X) = err \tag{25}$$

A sequence of push commands forms a regular input history (23), whereas a pop command without a preceding push command gives rise to an erroneous input history (25).

The control abstraction is neither transition closed nor output compatible, since it identifies all regular input histories, but forgets the data stored in the component.

6.3 Data Abstraction

The future behaviour of the interactive stack will be influenced by the collection of data stored in the component from the previous input history.

As a second abstraction, we explore the state $Data = \mathcal{D}^\star$ storing a stack of data. The data abstraction $data : \mathcal{I}^\star \to Data$ extracts from the input history the stack of data retained in the component after processing the input stream $(n \geq 0)$:

$$data(\langle push(d_1), \ldots, push(d_n)\rangle) = \langle d_1, \ldots, d_n\rangle \tag{26}$$
$$data(P\&\langle push(d), pop\rangle\&X) = data(P\&X) \tag{27}$$
$$data(pop \triangleleft X) = \langle\rangle \tag{28}$$

The data abstraction is neither output compatible nor transition closed. It identifies regular input histories leading to the empty stack with erroneous input histories resulting in a broken stack.

6.4 History Abstraction

We integrate the control abstraction and the data abstraction into a joint history abstraction.

This design decision leads to a composite state space $\mathcal{Q} = \mathit{Control} \times \mathit{Data}$ combining a control part and a data part. The abstraction function $\alpha : \mathcal{I}^\star \to \mathit{Control} \times \mathit{Data}$ pairs the control and the data abstraction.

The abstraction function : $\alpha(X) = (\mathit{control}(X), \mathit{data}(X))$ keeps all required information from the input history which determines the component's future behaviour. The abstraction function is indeed a history abstraction and supports the transition to a state-based implementation.

6.5 State Transition Machine of an Interactive Stack

The implementation of the interactive stack is derived from the input/output behaviour using the combined history abstraction for control and data states.

The resulting state transition machine is summarized in Fig. 2 . In a regular

$M[\mathit{stack}, \alpha] = (\mathit{Control} \times \mathit{Data}, \mathcal{I}, \mathcal{D}, \mathit{next}, \mathit{out}, (\mathit{reg}, \langle\rangle))$
$\mathit{next}((\mathit{reg}, Q), \mathit{push}(d)) = (\mathit{reg}, Q \triangleright d)$
$\mathit{next}((\mathit{reg}, Q \triangleright q), \mathit{pop}) = (\mathit{reg}, Q)$
$\mathit{next}((\mathit{reg}, \langle\rangle), \mathit{pop}) = (\mathit{err}, \langle\rangle)$
$\mathit{next}((\mathit{err}, \langle\rangle), x) = (\mathit{err}, \langle\rangle)$
$\mathit{out}((\mathit{reg}, Q), \mathit{push}(d)) = \langle\rangle$
$\mathit{out}((\mathit{reg}, Q \triangleright q), \mathit{pop}) = \langle q \rangle$
$\mathit{out}((\mathit{reg}, \langle\rangle), \mathit{pop}) = \langle\rangle$
$\mathit{out}((\mathit{err}, \langle\rangle), x) = \langle\rangle$

Fig. 2. State transition machine of an interactive stack

state, a push command attaches an element to the stack and produces no output. Moreover, a pop command delivers the top of a non-empty stack; for an empty stack it leads to the error state. This state cannot be left any more by further input which produces no output in the error state.

The subset of states reachable from the initial state $(\mathit{reg}, \langle\rangle)$ is isomorphic to the direct sum of the data stack and an error state:

$$\{\mathit{reg}\} \times \mathcal{D}^\star \cup \{(\mathit{err}, \langle\rangle)\} \;\simeq\; \mathcal{D}^\star + \{\mathit{err}\} \tag{29}$$

The transition functions defined on the subset of reachable states can simply be extended to the subset of unreachable states by setting $\mathit{next}((\mathit{err}, Q), x) = \mathit{err}$ and $\mathit{out}((\mathit{err}, Q), x) = \langle\rangle$.

6.6 State Transition Table of an Interactive Stack

For practical purposes, state transition machines are often described by *state transition tables* displaying the different transition rules in a clear way.

Fig. 3 describes the interactive stack by a state transition table. The four *transition rules* relate current states and inputs to new states and outputs. The transition rules tabulate the transition functions *next* and *out* . We use the notational convention that the constituents of the successor state are designated by a prime. For an empty input stream, the state transition table produces no output which agrees with Equation (15) .

Control	Data	Input	Control'	Data'	Output
reg	Q	$push(d)$	*reg*	$Q \triangleright d$	$\langle\rangle$
reg	$Q \triangleright q$	*pop*	*reg*	Q	$\langle q \rangle$
reg	$\langle\rangle$	*pop*	*err*	$\langle\rangle$	$\langle\rangle$
err	$\langle\rangle$	x	*err*	$\langle\rangle$	$\langle\rangle$

Fig. 3. State transition table of an interactive stack

7 Conclusion

Nowadays the specification and the systematic design of communicating components belongs to the central challenges of modern software technology. The software design must safely bridge component descriptions on different levels of abstraction.

The component's specification reflects a communication-oriented view concentrating on input and output histories. History-based specifications raise the abstraction level of initial descriptions. The black-box view provides a functional model of the component important for constructing networks in a compositional way.

The component's implementation decisively depends on the internal state supporting an efficient realization of the transition functions. The glass-box view discloses the component's internal state which is in general composed from various control and data parts.

This paper contributes to a better understanding how to relate communication-oriented and state-based descriptions. We presented a formal method for systematically transforming a stream processing function into a state transition machine. The state refinement employs history abstractions to bridge the gap between input histories and machine states. The transition functions can be derived from the defining equations using the Lübeck Transformation System [6] .

Yet, the crucial design decision consists in discovering a suitable history abstraction which determines the state space. In general, the state of a component

must store at least the information which is needed to process further inputs in a correct way. The particular information depends on the area of application. For example, the state of a counter records the sum of all elements which passed the component; so it depends on the entire prehistory. The state of a memory cell remembers the datum of the last write command which is the only decisive event in the prehistory. The state of a shift register stores a final segment of the input stream that is withheld from the output stream. The state of a transmission component may record the active channel, the successful transmission or the failure of acknowledge.

The state refinement presents a standard transformation from a denotational to an operational description of interactive components [20] . The refinement step can be prepared by calculating the output extension of the stream processing function. This step localizes the component's reaction in response to a single input wrt. a previous input history.

Among the candidates for an implementation, we identified the canonical state transition machine whose state records the complete input history. State transition machines with a reduced state space originate from the canonical machine by identifying states as input histories under history abstractions. By construction, all resulting state transition machines correctly implement the specified behaviour.

The history-oriented and the state-based description of software or hardware components allow complementary insights. Both formalisms shows advantages and shortcomings with respect to compositionality, abstractness, verification, synthesis, and tool support. In long term, proven design methods must flexibly bridge the gap between functional behaviour and internal realization following sound refinement rules.

Origin of this Summary

This summary is an excerpt of the forthcoming paper *Transforming Stream Processing Functions into State Transition Machines* by W. Dosch and A. Stümpel to appear in the *Journal of Computational Methods in Sciences and Engineering*.

References

1. M. Breitling and J. Philipps. Diagrams for dataflow. In J. Grabowski and S. Heymer, editors, *Formale Beschreibungstechniken für verteilte Systeme*, pages 101–110. Shaker Verlag, 2000.
2. M. Broy. (Inter-)action refinement: The easy way. In M. Broy, editor, *Program Design Calculi*, volume 118 of *NATO ASI Series F*, pages 121–158. Springer, 1993.
3. M. Broy. From states to histories: Relating state and history views onto systems. In T. Hoare, M. Broy, and R. Steinbrüggen, editors, *Engineering Theories of Software Construction*, volume 180 of *Series III: Computer and System Sciences*, pages 149–186. IOS Press, 2001.

4. M. Broy and K. Stølen. *Specification and Development of Interactive Systems: Focus on Streams, Interfaces, and Refinement*. Monographs in Computer Science. Springer, 2001.
5. T. Bălănescu, A. J. Cowling, H. Georgescu, M. Gheorghe, M. Holcombe, and C. Vertan. Communicating stream X-machines systems are no more than X-machines. *Journal of Universal Computer Science*, 5(9):494–507, 1999.
6. W. Dosch and S. Magnussen. The Lübeck Transformation System: A transformation system for equational higher order algebraic specifications. In M. Cerioli and G. Reggio, editors, *Recent Trends in Algebraic Development Techniques (WADT 2001)*, number 2267 in Lecture Notes in Computer Science, pages 85–108. Springer, 2002.
7. S. Eilenberg. *Automata, Languages, and Machines*, volume A. Academic Press, 1974.
8. D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.
9. G. Kahn. The semantics of a simple language for parallel programming. In J. Rosenfeld, editor, *Information Processing 74*, pages 471–475. North–Holland, 1974.
10. G. Kahn and D. B. MacQueen. Coroutines and networks of parallel processes. In B. Gilchrist, editor, *Information Processing 77*, pages 993–998. North–Holland, 1977.
11. N. Lynch and M. R. Tuttle. An introduction to input/output automata. *Centrum voor Wiskunde en Informatica, Amsterdam, CWI-Quarterly*, 2(3):219–246, Sept. 1989.
12. N. A. Lynch and E. W. Stark. A proof of the Kahn principle for input/output automata. *Information and Computation*, 82:81–92, 1989.
13. B. Möller. Ideal stream algebra. In B. Möller and J. Tucker, editors, *Prospects for Hardware Foundations*, number 1546 in Lecture Notes in Computer Science, pages 69–116. Springer, 1998.
14. Object Management Group (OMG). *OMG Unified Modeling Language Specification, 3. UML Notation Guide, Part 9: Statechart Diagrams*, Mar. 2003.
15. J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley Object Technology Series. Addison-Wesley, 1998.
16. P. Scholz. Design of reactive systems and their distributed implementation with statecharts. PhD Thesis, TUM-I9821, Technische Universität München, Aug. 1998.
17. G. Ştefănescu. *Network Algebra*. Discrete Mathematics and Theoretical Computer Science. Springer, 2000.
18. R. Stephens. A survey of stream processing. *Acta Informatica*, 34(7):491–541, 1997.
19. A. Stümpel. *Stream Based Design of Distributed Systems through Refinement*. Logos Verlag Berlin, 2003.
20. P. Wegner. Why interaction is more powerful than algorithms. *Communications of the ACM*, 40(5):80–91, May 1997.
21. G. Winskel and M. Nielsen. Models for concurrency. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Semantic Modelling*, volume 4 of *Handbook of Logic in Computer Science*, pages 1–148. Oxford University Press, 1995.