Designing an Interactive Stack

WALTER DOSCH
Institute of Software Technology and Programming Languages
Medical University of Lübeck
Ratzeburger Allee 160, D–23538 Lübeck
Germany

Abstract: We design an interactive stack in the setting of stream processing functions. The design passes through a series of abstraction levels relating different views of the stack component. The external view describes the component's input/output behaviour, the transition view captures the stepwise transitions, and the state-based view discloses the internal structure. The different descriptions can systematically be derived from the input/output behaviour following sound transformation rules. The case study exemplifies general methods for the specification and refinement of interactive components.

Key-Words: Interactive stack, stream processing, communication history, component: interface, specification, behaviour and refinement

1 Introduction

The description of classical data structures like lists, trees and arrays was originally based on their storage representations. Lateron the theory of abstract data types allowed the representation independent specification of data strutures based on the characteristic properties of their operations. The algebraic approach also influenced object-oriented software technology by providing succinct notions for interface and encapsulation.

Many components in distributed systems encapsulate some abstract data type. Here the static data structure is generalized to an interactive component which communicates with its environment. The communication histories are modelled by streams, that are sequences of messages. Streams abstract from a notion of time and record only the flow of messages on the channels of the distributed system. The input/output behaviour of an interactive component is described by a stream processing function [7, 8] mapping input histories to output histories.

In this paper, we systematically design an interactive stack in the setting of stream processing functions. An interactive stack is a communicating component that stores and returns data upon request following a last-in first-out strategy. The design passes through a series of abstraction levels relating different views of interactive stacks. In the external view, the stack component is considered as a black box without referring to its internal structure. The stack interacts with the environment by receiving input commands and sending output messages. The interaction with the environment defines a function between input and output histories. The transition view describes the reaction of the interactive stack on a single input command after having processed a previous input history. The transition view models the causal relationship between single input commands and the corresponding segments of the output history. The internal view discloses the control structure and the state of a component; it prepares the implementation by a state transition system.

The more refined descriptions of the interactive stack component can systematically be derived from the input/output behaviour within a uniform framework. The refinement steps can be formalized by transformation rules [6] which expose the design decisions underlying the component's implementation.

Section 2 surveys the basic notions about communication streams. In Section 3 , we define the interface of the interactive stack preparing the external view. Hereby we carefully discriminate between regular and

erroneous input histories. In Section 4, we validate the specification by examining characteristic properties. In Section 5, we derive the transition behaviour from the external behaviour. The state-based description in Section 6 discloses the component's internal structure; it describes the effect of an input command by an update of the local state. In Section 7, we pesent an implementation by a state transition machine introducing a control state for handling erroneous situations. Section 8 shows an equivalent description by an infinite state transition system. The conclusion revisits the approach and outlines further design steps.

2 Streams and Stream Processing Functions

In this section, we survey the basic notions about streams and stream processing functions [10] to render the paper self-contained.

2.1 Streams

Streams model the temporal succession of messages on the unidirectional channels of a network. Given an alphabet $\mathcal{A}$, the equation

$$\mathcal{A}^\star \;=\; \{\langle\rangle\} \,\cup\, \mathcal{A} \times \mathcal{A}^\star \qquad (1)$$

defines the set $\mathcal{A}^\star$ of *finite streams* $X = \langle x_1, x_2, \ldots, x_n\rangle$ of length $||X|| = n \geq 0$ with elements $x_i \in \mathcal{A}$ $(i \in [1, n])$. Throughout the paper, streams are denoted by capital letters, their elements by small letters.

Finite streams are generated from the *empty stream* $\langle\rangle$ by repeatedly attaching an element to the front of the stream: $\langle x_1, x_2, \ldots, x_n\rangle = x_1 \lhd x_2 \cdots \lhd x_n \lhd \langle\rangle$. The *concatenation* of two streams $X = \langle x_1, x_2, \ldots, x_m\rangle$ and $Y = \langle y_1, y_2, \ldots, y_n\rangle$ yields the stream $X\&Y = \langle x_1, x_2, \ldots, x_m, y_1, y_2, \ldots, y_n\rangle$. Appending an element to the rear of a stream is denoted by $X \rhd x = X\&\langle x\rangle$.

The number of occurrences of elements of a set $\mathcal{B}$ in a stream X is denoted by $||X||_{\mathcal{B}}$.

The set $\mathcal{A}^\star$ of finite streams over $\mathcal{A}$ forms a *partial order* under the *prefix relation*. Here a stream X *approximates* a stream Y, denoted by $X \sqsubseteq Y$, iff $X\&R = Y$ holds for some stream $R \in \mathcal{A}^\star$. The prefix relation models operational progress in time: the shorter stream forms an initial part of the communication history. The empty stream is the least element which can be extended to every history.

A function $f : \mathcal{A} \to \mathcal{B}$ on elements is extended to a function $f^\star : \mathcal{A}^\star \to \mathcal{B}^\star$ on streams by setting $f^\star(\langle x_1, \ldots, x_n\rangle) = \langle f(x_1), \ldots, f(x_n)\rangle$. Moreover, all functions $f : \mathcal{A} \to \mathcal{B}$ are naturally extended to (equally denoted) functions on subsets $M \subseteq \mathcal{A}$ by setting $f(M) = \{f(a) \mid a \in M\}$.

2.2 Stream Processing Functions

A *stream processing function*, for short a *stream transformer* $f : \mathcal{A}^\star \to \mathcal{B}^\star$ maps an input stream to an output stream. It models a *deterministic component* with one input and one output channel; we will not enter the area of nondeterminism here. The types $\mathcal{A}$ and $\mathcal{B}$ determine the *syntactic interface* of the component.

In the sequel, we concentrate on monotonic functions where further input leads to further output. A stream transformer $f : \mathcal{A}^\star \to \mathcal{B}^\star$ is called (*prefix*) *monotonic*, if $X \sqsubseteq Y$ implies $f(X) \sqsubseteq f(Y)$ for all $X, Y \in \mathcal{A}^\star$. Due to a standard theorem, any monotonic function on finite streams has a unique continuous extension to infinite streams [9].

3 Specification

An *interactive stack* is a communicating component with one input and one output channel. The component can store an unbounded number of data elements. The input consists of push commands entering a datum, and pop commands requesting the datum stored most recently. When time progresses, the interactive stack consumes an input stream of commands and produces an output stream of data, compare Fig. 1.

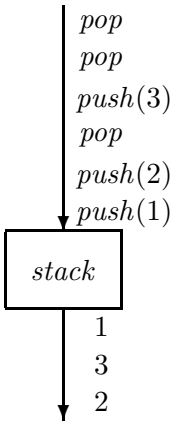

Figure 1: External view of an interactive stack of natural numbers

3.1 Interface

First we concentrate on the *syntactic interface* of the component and define the types of messages on the input and the output channel. The type $\mathcal{D} \neq \emptyset$ of data to be stored in the interactive stack need not to be specified further.

The component receives as input either a push command together with the datum to be stored or a pop command requesting the datum stored most recently. Hence we define the type $\mathcal{I}$ of input messages as

$$\mathcal{I} \quad = \quad \{pop\} \cup push(\mathcal{D}) \tag{2}$$

The type $\mathcal{O}$ of output messages simply is $\mathcal{D}$.

3.2 Specifying the Input/Output Behaviour

Then we specify the *behaviour* of the interactive stack by a stream processing function

$$stack : \mathcal{I}^\star \to \mathcal{O}^\star$$

mapping input histories to output histories. The input/output behaviour describes an external or black-box view of the stack component not revealing its internal structure.

3.3 Domain of the History Function

In general, a component is not designed for an arbitrary environment; so it will react regularly only on a subset of all possible input histories. We document the assumptions on the environment and define the set of regular input histories.

When an interactive stack receives a pop command, it outputs the datum which has been stored most recently and has not yet been output. Therefore an empty stack cannot respond regularly to a pop command.

Thus we partition the set $\mathcal{I}^\star$ of all input histories into the disjoint sets $\mathcal{R}$ of regular and $\mathcal{E}$ of erroneous input histories:

$$\mathcal{I}^\star \quad = \quad \mathcal{R} \,\dot\cup\, \mathcal{E} \tag{3}$$

An input stream represents a *regular input history* iff for each prefix the number of pop commands does not exceed the number of push commands:

$$X \in \mathcal{R} \quad \text{iff} \quad \forall Y \sqsubseteq X : ||Y||_{\{pop\}} \leq ||Y||_{push(\mathcal{D})} \tag{4}$$

The sets of regular and erroneous input histories show characteristic closure properties. For example, initial segments of regular input histories are regular as well:

$$X \& Y \in \mathcal{R} \quad \Rightarrow \quad X \in \mathcal{R} \tag{5}$$

Complementary, prolongations of erroneous input histories are erroneous as well:

$$X \in \mathcal{E} \quad \Rightarrow \quad X \& Y \in \mathcal{E} \tag{6}$$

For the set $\mathcal{R}$ of regular input histories, we make use of an equivalent inductive definition to ease function definitions by pattern matching:

(1) $push(\mathcal{D})^\star \subseteq \mathcal{R}$

(2) If $P \in push(\mathcal{D})^\star$ and $P \& X \in \mathcal{R}$,
then $P \& \langle push(d), pop \rangle \& X \in \mathcal{R}$.

3.4 Regular Behaviour

The input/output behaviour of the stack relates the input and output histories passing through the component's interface.

The regular behaviour $stack : \mathcal{R} \to \mathcal{O}^\star$, viz. the behaviour on the subset of regular input histories, is defined with the above recursion scheme ($P \in push(\mathcal{D})^\star$):

$$stack(P) = \langle\rangle \tag{7}$$

$$stack(P \& \langle push(d), pop \rangle \& X) = d \triangleleft stack(P \& X) \tag{8}$$

A sequence of push command generates no output (7). A pop command outputs the most recent datum which has not yet been requested (8).

The regular behaviour is well-defined, since the recursive equation (8) decreases the number of pop commands in each step which eventually leads to equation (7).

3.5 Irregular Behaviour

In a loose approach to system modelling, the reaction of an interactive stack upon receiving erroneous input need not be specified. With such an *underspecification* the designer expresses the willingness to accept every behaviour on irregular input.

In subsequent refinement steps, the underspecification can be resolved by adding further design decisions. As an important constraint, we must take into account the monotonicity of the stream transformer.

3.5.1 Fault Tolerant Behaviour

A fault tolerant stack ignores an illegal pop command from the environment and continues to perform its service on future input commands. The fault tolerant behaviour is specified by adding the equation

$$stack(pop \triangleleft X) \quad = \quad stack(X)\,. \tag{9}$$

As a consequence, an initial sequences of pop commands will not influence the input/output behaviour.

3.5.2 Fault Sensitive Behaviour

When receiving an unexpected pop command from the environment, the interactive stack may break and not provide any output whatever further input commands arrive:

$$stack(pop \triangleleft X) \quad = \quad \langle\rangle\,. \tag{10}$$

The output stems from the longest regular prefix of the irregular input history.

For the remainder of this paper, we will further elaborate the fault sensitive behaviour.

4 Validation

The validation of a component's specification provides valuable feedback whether the specified behaviour meets the informal requirements.

In this section we validate the specification of the interactive stack in two respects. First we check the soundness of the stream transformer wrt. formal criteria. Then we explore the behaviour of the interactive stack for special input and output histories.

4.1 Monotonicity

The stack component is described by a monotonic stream transformer:

$$stack(X) \quad \sqsubseteq \quad stack(X \& Y) \tag{11}$$

The length of the output stream is limited by the number of pop commands:

$$||\, stack(X)\,|| \quad = \quad ||X||_{\{pop\}} \quad \text{if } X \in \mathcal{R} \tag{12}$$
$$||\, stack(X)\,|| \quad < \quad ||X||_{\{pop\}} \quad \text{if } X \in \mathcal{E} \tag{13}$$

4.2 Decomposition Properties

The output of the interactive stack for a composite input stream can be inferred from the output histories of the substreams:

$$stack(X \& Y) \;=\; stack(X) \,\&\, stack(red(X) \& Y) \tag{14}$$

The auxiliary function $red : \mathcal{I}^\star \to \mathcal{I}^\star$ reduces an input stream by removing all executable commands ($P \in push(\mathcal{D})^\star$):

$$red(P) \quad = \quad P \tag{15}$$
$$red(D \& \langle push(d), pop\rangle \& X) \quad = \quad red(P \& X) \tag{16}$$
$$red(pop \triangleleft X) \quad = \quad pop \triangleleft X \tag{17}$$

The auxiliary functions abstracts from the initial substream the information needed for processing the final substream. If the initial substream X is erroneous, the final substream Y must not provides an output.

4.3 Inversion

The behaviour of the interactive stack is a function associating an output stream with every input stream. The inverse function $stack^{-1} : \mathcal{O}^\star \to \mathcal{P}(\mathcal{I}^\star)$ determines the set of all input histories generating a given output stream.

A regular input history causes the empty output iff it contains no requests:

$$stack^{-1}(\langle\rangle) \cap \mathcal{R} \quad = \quad push(\mathcal{D})^\star \tag{18}$$

An erroneous input history generates the empty output iff it starts with a pop command:

$$stack^{-1}(\langle\rangle) \;\cap\; \mathcal{E} \quad = \quad \{pop\} \triangleleft \mathcal{I}^\star \tag{19}$$

Altogether the validation increases the confidence in the specified behaviour of the interactive stack which serves as starting point for the subsequent refinement.

5 Transition Behaviour

The external behaviour is based on a summary description that refers to entire communication histories; it abstracts from the transitions effected by the single commands of the input stream.

In this section, we model the component in greater detail when processing the input stream element by element. This description reveals basic operational traits, since it records which segment of the output history is caused by which command from the input history.

The *transition behaviour* of the interactive stack

$$trans : \mathcal{I}^\star \to [\mathcal{I} \to \mathcal{O}^\star]$$

yields that segment of the output stream caused by a single input command after having processed a previous input history:

$$stack(X \triangleright x) \quad = \quad stack(X) \,\&\, trans(X)(x) \tag{20}$$

The transition behaviour reflects the causality of a component. The implicit specification (20) makes only sense for monotonic stream transformers where a prolongation of the input history effects a prolongation of the output history as well.

We derive an explicit definition by fold and unfold transformations along with algebraic simplifications ($P \in push^\star(\mathcal{D})$):

$$trans(X)(push(d)) = \langle\rangle \quad (21)$$
$$trans(X)(pop) = trans(red(X))(pop) \quad (22)$$
$$trans(\langle\rangle)(pop) = \langle\rangle \quad (23)$$
$$trans(pop \triangleleft X)(pop) = \langle\rangle \quad (24)$$
$$trans(P \triangleright push(d))(pop) = \langle d\rangle \quad (25)$$

A push command causes no output after any input history (21). The effect of a pop command depends on the input history (22). If the reduced input history is empty or erroneous, a pop command effects no output (23, 24). If the reduced input history is regular and nonempty, the pop command outputs the most recent datum that has not yet been requested (25).

6 State-Transition Behaviour

The input/output behaviour of the interactive stack was specified without referring to the internal structure of the component. A refined view discloses the internal or local state of the component and describes the effect of an input command by an update of the internal state. With the disclosure of the internal state space, we arrive at a *glass-box view* of the stack component.

The state of a component abstracts from the input history the information that determines the component's behaviour on future input. We record the data that have not yet been requested as the internal state and choose the data type $\mathcal{D}^\star$ as state space. The state transition behaviour

$$stb : \mathcal{D}^\star \to [\mathcal{I}^\star \to \mathcal{O}^\star]$$

of the interactive stack with an internal state agrees with the external behaviour where the input stream is prefixed with the push commands generating the internal stack:

$$stb(D)(X) \quad = \quad stack(push^\star(D)\&X) \quad (26)$$

A direct recursive version of the higher order stream transformer stb can easily be derived from this specification:

$$stb(D)(\langle\rangle) \quad = \quad \langle\rangle \quad (27)$$
$$stb(D)(push(d) \triangleleft X) \quad = \quad stb(D \triangleright d)(X) \quad (28)$$
$$stb(\langle\rangle)(pop \triangleleft X) \quad = \quad \langle\rangle \quad (29)$$
$$stb(D \triangleright d)(pop \triangleleft X) \quad = \quad d \triangleleft stb(D)(X) \quad (30)$$

A datum from the input is pushed onto the internal stack without generating output (28). The attempt to pop an empty stack results in an error. (29). A pop command to a nonempty stack outputs the datum stored most recently (30).

An observer can query the internal state of the interactive stack by inputting a sufficient number of pop commands until he gets no more response. So the internal state is transparent to the environment.

7 State Transition Machine

The state transition behaviour of the stack can equivalently be described as a state transition machine with input and output [5]. In the seuqel, we settle the components of the machine and implement the interactive stack.

A *state transition machine with input and output*

$$M \quad = \quad (\mathcal{S}, \mathcal{I}, \mathcal{O}, \delta, \varphi)$$

consists of a set $\mathcal{S}$ of *states*, an *input alphabet* $\mathcal{I}$, an *output alphabet* $\mathcal{O}$, a *one-step state transition function* $\delta : \mathcal{S} \times \mathcal{I} \to \mathcal{S}$ and a *one-step output function* $\varphi : \mathcal{S} \times \mathcal{I} \to \mathcal{O}^\star$.

The *multi-step output function* $\varphi^\star : \mathcal{S} \to [\mathcal{I}^\star \to \mathcal{O}^\star]$ of the state transition machine M with

$$\varphi^\star(q)(\langle\rangle) \quad = \quad \langle\rangle \quad (31)$$
$$\varphi^\star(q)(x \triangleleft X) \quad = \quad \varphi(q,x) \,\&\, \varphi^\star(\delta(q,x))(X) \quad (32)$$

forms a prefix monotonic stream transformer.

For the interactive stack, the set of states

$$\mathcal{S} \quad = \quad \{fail\} \cup \mathcal{D}^\star \quad (33)$$

combines the control state $fail$ recording an illegal pop command with the data space holding the stored elements. The input set $\mathcal{I} = \{pop\} \cup push(\mathcal{D})$ and the output set $\mathcal{O} = \mathcal{D}$ are obvious.

The control state indicates an illegal pop command in the previous history. The introduction of a control state supports transforming the equations (27)–

(29) into the regular form used in a state transition machine:

$$\begin{aligned}
\delta(fail, x) &= fail && (34)\\
\delta(D, push(d)) &= D \triangleright d && (35)\\
\delta(\langle\rangle, pop) &= fail && (36)\\
\delta(D \triangleright d, pop) &= D && (37)
\end{aligned}$$

The output function is given by

$$\begin{aligned}
\varphi(fail)(x) &= \langle\rangle && (38)\\
\varphi(D, push(d)) &= \langle\rangle && (39)\\
\varphi(D \triangleright d, pop) &= \langle d\rangle && (40)\\
\varphi(\langle\rangle, pop) &= \langle\rangle\,. && (41)
\end{aligned}$$

The correctness of the implementation

$$\varphi^\star = stb \qquad (42)$$

follows from the fact that the multi-step output function coincides with the external behaviour of the interactive stack.

Note that classical Mealy machines are restricted to a finite input and a finite state space. Moreover, they provide exactly one output for each input.

8 State Transition Diagram

A state transition machine with input and output may be visualized by a *state transition diagram* having the set of states as vertices. Each directed edge from one state to a successor state is labelled by a corresponding pair naming input and output.

For the interactive stack, we obtain a state transition diagram with the set of nodes $\{fail\} \cup \mathcal{D}^\star$ and four types of arcs:

$$\begin{aligned}
&\{fail \xrightarrow{(x,\langle\rangle)} fail \mid x \in \mathcal{I}\} \quad \cup\\
&\{D \xrightarrow{(push(d),\langle\rangle)} D \triangleright d \mid D \in \mathcal{D}^\star\} \quad \cup\\
&\{D \triangleright d \xrightarrow{(pop,\langle d\rangle)} D \mid D \triangleright d \in \mathcal{D}^\star\} \quad \cup\\
&\{\langle\rangle \xrightarrow{(pop,\langle\rangle)} fail\}
\end{aligned}$$

The state transition diagram of the interactive stack is infinite, but quite regular. Fig. 2 shows an initial part the state transition diagram for a binary data type $\mathcal{D} = \{0, 1\}$.

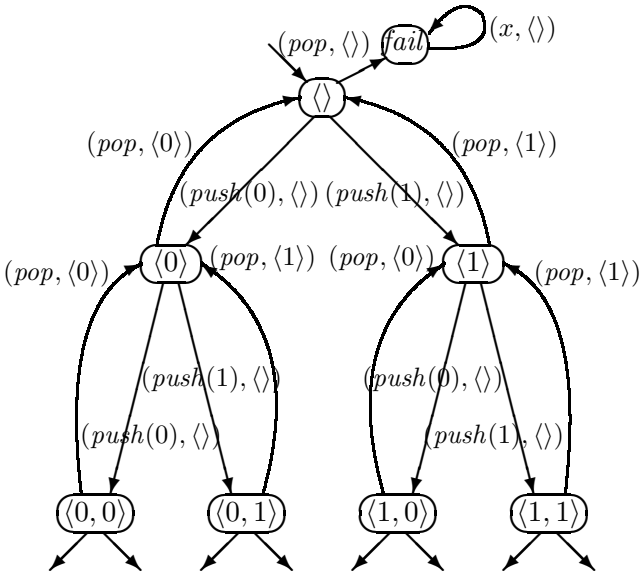

Figure 2: Initial part of the infinite state transition diagram of an interactive stack of binary values

9 Conclusion and Related work

For getting an insight into the advantages and shortcomings of design techniques for the components of distributed systems, it is necessary to apply the approaches to case studies.

The stream-based approach was successfully worked out for various case studies, among others a merge component [4], a buffer of length one [2] and different memory, transmission and control components [3]. In [1] several versions of an interactive queue are modelled which differ in the reaction to unexpected or erroneous input.

Many design studies concentrate on a single view of the communicating component — usually the state-oriented glass-box view. In the software development, however, we need various points of view to bridge the levels of abstraction between a problem-oriented specification and the final implementation.

In this paper we demonstrated that the stream-based approach offers a sound framework for uniformly describing the behaviour of an interactive stack on different levels of abstraction. The refined views can systematically be derived from the external behaviour following sound transformation rules.

The state-based description from the previous sections is not the final version of the design. Further design steps will refine the output interface by introducing an acknowledge channel. Also the input interface can be refined separating pop and push com-

mands. An essential design step then consists in introducing a timing behaviour for the component. In the framework of timed streams, the input and corresponding output events are related in a discrete time frame. A quite different development line concerns the transition to an interactive stack of bounded size. After this design step, the internal state can be further refined by introducing a memory management on an array of fixed length.

References

[1] Manfred Broy. Views of queues. *Science of Computer Programming*, 11:65–86, 1988.

[2] Manfred Broy. Specification and refinement of a buffer of length one. In Manfred Broy, editor, *Deductive Program Design*, volume 152 of *NATO ASI Series F*, pages 273–304. Springer, 1996.

[3] Manfred Broy and Gheorghe Ştefănescu. The algebra of stream processing functions. *Theoretical Computer Science*, 258:99–129, 2001.

[4] Walter Dosch and Annette Stümpel. Merging ordered streams. In S. Y. Shin, editor, *Proceedings of the 15th International Conference on Computers and their Applications*, pages 377–382. International Society for Computers and their Applications, 2000.

[5] Walter Dosch and Annette Stümpel. From stream transformers to state transition systems with input and output. In N. Ishii, T. Mizano, and R. Lee, editors, *International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD'01)*, pages 1033–1040. International Association for Computer and Information Science, 2001.

[6] Martin S. Feather. A survey and classification of some program transformation approaches and techniques. In L.G.L.T. Meertens, editor, *Program Specification and Transformation*, pages 165–195. IFIP, Elsevier Science Publishers, 1987.

[7] Gilles Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Information Processing 74*, pages 471–475. North–Holland, Amsterdam, 1974.

[8] Gilles Kahn and David B. MacQueen. Coroutines and networks of parallel processes. In Bruce Gilchrist, editor, *Information Processing 77*, pages 993–998. North–Holland, Amsterdam, 1977.

[9] Bernhard Möller. Algebraic structures for program calculation. In Manfred Broy and Ralf Steinbrüggen, editors, *Calculational System Design*, volume 173 of *NATO ASI Series F*, pages 25–97. IOS Press, 1999.

[10] Robert Stephens. A survey of stream processing. *Acta Informatica*, 34(7):491–541, 1997.