On an Interactive Fault-Sensitive Stack —
From Communication Histories to State Transition Tables

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

Abstract

The black-box view of an interactive component describes the input/output behaviour based on communication histories. The glass-box view discloses the component’s internal state where the inputs effect an update of the state. We formally derive different glass-box views of an interactive fault-sensitive stack from a common black-box view. The transformation uses higher-order functions where the state parameter is introduced as a history compactification. The different glass-box views reflect particular design decisions wrt. the control state and the data state of the component. The transformation uses history compactifications which retain from the input history some information that influences the component’s future behaviour. The case study exemplifies a general method how to formally refine the black-box view of an interactive component into a glass-box view using history compactifications.

Keywords: Interactive stack, communication history, history compactification, control state, data state, black-box view, glass-box view

1 Introduction

The systematic design of correct software components needs sound foundations supporting the step-wise refinement from behavioural specifications to final implementations. A crucial design step amounts to deriving the internal realisation of a component from the external behaviour determined by the communication flow through the component’s interface.

In this paper, we present a case study how to transform a component’s black-box view based on communication histories into different glass-box views employing control and data states. The transformation uses history compactifications which retain from the input history some information that influences the component’s future behaviour.

An interactive stack is a communicating component that stores and retrieves data following a first-in/last-out strategy. A fault-sensitive stack will break upon receiving an illegal command from the environment and not provide any further output whatever input arrives. We formally derive the glass-box views from the same black-view box using different compactifications of the input history.

The specification concentrates on a black-box view describing the input/output relation based on communication histories. The black-box view characterizes the service provided by the component without uncovering its internal structure.

In the black-box view, the stack interacts with the environment in an asynchronous way by receiving input messages and sending output messages. Communication histories, for short streams, record the flow of messages through the component’s interface. The interaction with the environment defines a function from input histories to output histories [8, 9].

During the design process, the extensional view is transformed into a glass-box view disclosing the component’s internal structure. In the glass-box view, an input message effects an update of the internal state and an output depending on the internal state. The properties of the resulting state transition system widely depend on the particular design decisions made for the control state and the data state.

We systematically introduce the control states and
data states of the glass-box view using history compactifications. With a history compactification function, we record some information from the input history that is relevant for the future output history. We elaborate different development lines of the design to explicate the relation between control and data states.

When we abstract control information by discriminating regular and erroneous input histories, we arrive at a glass-box description with two control states. When we abstract data information from the input history, we obtain a glass-box description based on a data space. When we simultaneously abstract control and data information from the input history, we arrive at a simple glass-box description processing the input stream message by message.

The different glass-box descriptions of the interactive stack are uniformly described by state transition systems. The refinement steps are formalized by transformational derivations \[7\]. The emphasis of this paper lies on formally deriving glass-box views of an interactive stack by deductive reasoning. Therefore the paper concentrates on the following contributions:

- As an original technical result, we present three different glass-box views of an interactive fault-sensitive stack uniformly described by state transition systems.
- The different glass-box views are not "invented", but systematically derived from the same external view by transformational reasoning using history compactifications for condensing control and data information.

The design method explicated for this particular data structure generalizes to a large class of interactive components. In this way, the case studies exemplifies a systematic method how to derive glass-box views from black-box views using history compactifications.

The paper is organized as follows. In Section 2, we survey the basic notions about communication streams. In Section 3, we define the black-box view of the interactive fault-sensitive stack as a function from input to output histories. Section 4 studies the transition from the black-box view to a first glass-box view introducing a control state by a control compactification. Section 5 demonstrates the transition from the black-box view to another glass-box view introducing a data state by a data compactification. Finally Section 6 joins the control and data compactifications into a joint compactification function resulting in a glass-box view with the simplest state transition system. The formal derivations are sketched in the appendix.

## 2 Communication Streams

In this section, we survey the basic notions about communication streams and stream processing functions \[11\] to render the paper self-contained. Communication streams model the temporal succession of messages on the unidirectional channels of a network. Given an alphabet \(A\), the set \(A^*\) comprises all (finite) streams \(X = \langle x_1, x_2, \ldots, x_n \rangle\) of length \(|X| = n \geq 0\) with elements \(x_i \in A\) (\(i \in [1, n]\)). Throughout the paper, streams are denoted by capital letters, their elements by small letters. \(A^+\) denotes the set of all nonempty streams over \(A\).

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 \(X \& Y = \langle x_1, x_2, \ldots, x_m, y_1, y_2, \ldots, y_n \rangle\).

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

A stream processing function, for short a stream transformer \(f : \mathcal{A}^* \to \mathcal{B}^*\) maps an input stream to an output stream. It models a (deterministic) component with one input and one output channel. 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 \(f : \mathcal{A}^* \to \mathcal{B}^*\) is called (prefix) monotonic, if \(X \sqsubseteq Y\) implies \(f(X) \sqsubseteq f(Y)\) for all \(X, Y \in \mathcal{A}^*\). Due to a standard theorem, any monotonic function on finite streams has a unique continuous extension to infinite streams \[10\].
3 Black-Box View

An interactive fault-sensitive stack is a communicat-
ing component with one input and one output chan-
nel. The component stores and retrieves an un-
bounded number of data elements following a first-
in/last-out strategy. The stack is fault-sensitive, i.e. it
breaks when receiving an illegal command from the
environment. As time progresses, the interactive
stack consumes an input stream of commands and
produces an output stream of data, compare Fig. 1.

![Diagram of a stack]

Figure 1: External view of an interactive fault-
sensitive stack

3.1 Interface

The interface of the component is determined by the
types of messages on the input and the output channel.

The type $D \neq \emptyset$ of data to be stored in the stack
need not be specified further. The input alphabet

$$\mathcal{I} = \{\text{pop}, \text{top}\} \cup \text{push}(D)$$

consists of $\text{pop}$ and $\text{top}$ commands removing resp. re-
questing the datum stored most recently, and $\text{push}$
commands entering a new datum. The type $\mathcal{O}$ of output
messages simply is $D$.

3.2 Input/Output Behaviour

The input/output behaviour describes a black-box view not revealing the component’s internal structure. We define the history function

$$\text{stack} : \mathcal{I}^* \rightarrow \mathcal{O}^*$$

with the following recursion scheme assuming $P \in \text{push}(D)^*$:

$$\text{stack}(P) = \emptyset$$

A sequence of push commands generates no output
$\emptyset$ . A pop command removes, a top command out-
puts the datum entered most recently — provided it
exists $(5,7)$ . When receiving an unexpected pop or
top command from the environment, the stack breaks
and produces no output whatever further input arrives
$(4,6)$ .

4 Glass-Box View Based on Control
States

The black-box view of the interactive fault-sensitive
stack is now refined to a glass-box view based on
control states. The corresponding history compacti-
fication discriminates between regular and erroneous
input histories.

4.1 Control States

The interactive stack reacts regularly only on a sub-
set of all possible input histories, since a pop or top
request to the empty stack cannot be satisfied.

Therefore we discriminate regular input histories and erroneous input histories which originate from an
illegal pop resp. top command. This design decision
leads to a ternary set of control states:

$$\text{Control} = \{\text{regular}, \text{errorpop}, \text{errortop}\}$$

4.2 Stream Transformer Based on Control
States

Next we derive a stream transformer modelling the
glass-box view of the priority queue based three on
control states.

The control based history function

$$\text{cstack} : \text{Control} \rightarrow [\mathcal{I}^* \rightarrow \mathcal{O}^*]$$

associates with each control state a stream trans-
former:

$$\text{cstack}(\text{regular}) = \text{stack}$$

$$\text{cstack}(\text{errorpop}) = \text{stack} \circ (\langle \text{pop}\rangle \& )$$

$$\text{cstack}(\text{errortop}) = \text{stack} \circ (\langle \text{top}\rangle \& )$$
In the regular state, the control based history function agrees with the original stream transformer. In the erroneous states, the control based history functions behaves as the original stream transformer where an illegal command prefixed to the input stream.

From this specification, we derive a direct recursive version using fold and unfold transformation along with algebraic simplifications, compare Appendix A. Such transformational derivations are well supported by the Lübeck Transformation System [4]. A resulting version reads assuming $P \in \text{push}(D)^*$:

- $cstack(c)(\langle \rangle) = \langle \rangle$ (13)
- $cstack(\text{regular})(\langle \text{push}(d) \rangle \& P) = cstack(\text{regular})(P)$ (14)
- $cstack(\text{regular})(\langle \text{pop} \rangle \& X) = cstack(\text{errorpop})(X)$ (15)
- $cstack(\text{regular})(P \& \langle \text{push}(d), \text{pop} \rangle \& X) = cstack(\text{regular})(P \& X)$ (16)
- $cstack(\text{regular})(\langle \text{top} \rangle \& X) = cstack(\text{errortop})(X)$ (17)
- $cstack(\text{regular})(P \& \langle \text{push}(d), \text{top} \rangle \& X) = \langle d \rangle \& cstack(\text{regular})(P \& \langle \text{push}(d) \rangle \& X)$ (18)
- $cstack(\text{errorpop})(\langle x \rangle \& X) = cstack(\text{errorpop})(X)$ (19)
- $cstack(\text{errortop})(\langle x \rangle \& X) = cstack(\text{errortop})(X)$ (20)

Following equation (3), a sequence of push commands generates no output (14). Following equations (5) and (7), a pop command removes (16), a top command outputs (18) the datum stored most recently. Based on equation (4) resp. (6), an unexpected pop resp. top command changes the control state from okay to errorpop (15) resp. to errortop (17) such that no subsequent output is provided (19)-(20).

### 5 Glass-Box View Based on Data States

In this section, we follow an alternative development line and design a glass-box view based on a data state.

#### 5.1 Data States

The data of a regular input history that have not been requested yet, determine the future behaviour of the interactive priority queue.

Therefore we record the collection of stored data elements and define the data state

$$Data = D^* .$$

#### 5.2 Stream Transformer Based on Data States

In this subsection, we model the glass-box view of the interactive stack based on data states.

The data oriented history function

$$dstack : Data \rightarrow [I^* \rightarrow O^*] \quad (21)$$

associates with each data state a stream transformer:

$$dstack(D)(X) = stack(push^*(D) \& X) \quad (22)$$

The glass-box behaviour of a stack with internal state agrees with the black-box behaviour where the input stream is prefixed with a series of push commands generating the internal stack.
The derivation, compare Appendix B, heads for a direct recursive version:

\[
dstack(D)(\langle push(d)\rangle & X) = \dstack(D&\langle d\rangle)(X)
\]

(23)

\[
dstack(\langle \rangle)(\langle pop\rangle & X) = \langle \rangle
\]

(24)

\[
dstack(D&\langle d\rangle)(\langle pop\rangle & X) = \dstack(D)(X)
\]

(25)

\[
dstack(\langle \rangle)(\langle top\rangle & X) = \langle \rangle
\]

(26)

\[
dstack(D&\langle d\rangle)(\langle top\rangle & X) = (d)&dstack(D&\langle d\rangle)
\]

(27)

A top command outputs the datum stored most recently, viz. the first element of the internal stack (27) — provided it exists. A datum arriving at the input port joins the internal stack without generating output (23). A pop command and a top command in case of the empty internal stack result in an error (24, 26).

An observer can query the internal data state of the interactive stack by inputting a sufficient number of top commands until he gets no more response.

5.3 Transition Table Based on Data States

The data oriented history function of the interactive stack describes a state transition system. Its state transition table is shown in Fig. 3. Each non-stopping transition rule shortens the input by one command.

Regular input streams can be processed command by command until the input stream is exhausted (23, 25, 27). However, erroneous input histories show a nonuniform recursion (24, 26). Therefore we attached a further column to the transition table recording the termination of each rule.

In summary, a data oriented history compactification supports an elementwise processing of regular input histories. For erroneous input histories, however, the data space does not record the occurrence of the first illegal pop or top command.

6 Glass-Box View Based on Joint Control and Data States

In this section we integrate the control compactification and the data compactification of the preceding sections into a joint control and data compactification. This design decision leads to a simple state-based history function inheriting the benefits from the two previous compactifications. The resulting implementation by a state transition system processes the input stream element by element.

6.1 Joining Control and Data States

The central design decision amounts to the proper combination of control and data states.

We integrate the control and data space into the joint state space

\[
state = \{errorpop, errortop\} \cup D^*
\]

(28)
6.2 Stream Transformer Based on Joint Control and Data States

Next we model the glass-box view of the interactive priority queue based on the joint state space.

The joint history function

\[ cdstack : \text{State} \rightarrow \mathbb{I}^* \rightarrow \mathbb{O}^* \]  

integrates the control oriented (10)-(20) and the data oriented history function (22):

\[ cdstack(D) = \text{stack} \circ ((\text{push}^*(D))\&) \]  
\[ cdstack(\text{errorpop}) = \text{stack} \circ ((\text{pop})\&) \]  
\[ cdstack(\text{errortop}) = \text{stack} \circ ((\text{top})\&) \]

Replaying the previous derivations, compare Appendix C, we arrive at a direct recursive algorithm \((D \in \text{Data})\):

\[ cdstack(D)((\text{push}(d))\&X) = cdstack(D\&d)(X) \]  
\[ cdstack(())(\text{pop})\&X = cdstack(\text{errorpop})(X) \]  
\[ cdstack(D\&d)(\text{pop})\&X = cdstack(D)(X) \]  
\[ cdstack(())(\text{top})\&X = cdstack(\text{errortop})(X) \]  
\[ cdstack(D\&d)(\text{top})\&X = cdstack(D\&d)(\text{top})\&X \]

A top command outputs the datum stored most recently; it resides at the top of the internal stack (37). An illegal pop or top command results in the failure state (34,36) which cannot be left with any further input (31,32). A datum from the input is joined the internal stack without generating output (33).

6.3 Transition Table based on Joint Control and Data States

The transition table of the joint history function is depicted in Fig. 4.

The transition table for the joint history function improves the previous transition tables for the control based and the data based history functions. The input can now be processed command by command without any exception.

The implementation needs no look-ahead: the next state and the output are completely determined by the previous state and the current input. This iterative behaviour was achieved by integrating the relevant control and data information from the input history into the joint state space.

7 Conclusion

The stream-based approach to the specification and refinement of communicating components was successfully worked out for many case studies, among others different memory, transmission and control components [3], a memory cell [5], a bounded buffer [6] and an interactive queue [2].

The black-box view of a communicating component provides a functional model relating input to output histories. The black-box view is important for constructing networks of communicating components in a compositional way.

The glass-box view discloses the internal state of a component which is relevant for its efficient implementation with state transition functions. In general, the component’s internal state is composed from various control and data parts.

This paper used the interactive priority queue as an example to present a formal method. The black-box view of the component was transformed into a glass-box view using history compactification functions. These functions extract the control state and/or the data state from the input history. The case study explicited how history compactifications and the corresponding control and data abstractions infer with each other.

The formal design was carried out in a functional framework following a transformational approach. In [1] state-based and history-based specifications are combined with verification techniques in a logical framework.

History-based specifications of communicating components raise the abstraction level of initial descriptions from state-based to functional formalisms. Complementary, in software development we need transformations for safely bridging the gap between history-based specifications and state-based implementations. A forthcoming thesis [12] will provide a comprehensive treatment of formal methods for the state-oriented refinement of stream processing functions.
Figure 4: State transition table of an interactive fault sensitive stack based on a joint control and data space

Acknowledgement

I gratefully acknowledge valuable improvements suggested by A. Stümpel.

References


Appendix A

Equation (13)
\[
cstack(\text{regular})(\langle\rangle) \\
= \text{stack}(\langle\rangle) \\
= \langle\rangle \\
\]
\[
cstack(\text{errorpop})(\langle\rangle) \\
= \text{stack}(\langle\rangle & \langle\rangle) \\
= \langle\rangle \\
\]
\[\text{cstack}(\text{errorpop})(\langle x \rangle)\]
\[= \text{stack}(\langle x \rangle)\]
\[= \langle x \rangle\]

Equation (14)
\[\text{cstack}(\text{regular})(\langle push(d) \rangle)\]
\[= \text{stack}(\langle push(d) \rangle)\]
\[= \langle push(d) \rangle\]

Equation (15)
\[\text{cstack}(\text{regular})(\langle pop \rangle)\]
\[= \text{stack}(\langle pop \rangle)\]
\[= \langle pop \rangle\]

Equation (16)
\[\text{cstack}(\text{regular})(\langle P \rangle)&(\langle push(d) \rangle, \langle pop \rangle)\]
\[= \text{stack}(\langle P \rangle)&(\langle push(d) \rangle, \langle pop \rangle)\]
\[= \langle P \rangle, \text{stack}(\langle push(d) \rangle, \langle pop \rangle)\]
\[= \text{cstack}(\text{regular})(\langle P \rangle)\]

Equation (17)
\[\text{cstack}(\text{regular})(\langle top \rangle)\]
\[= \text{stack}(\langle top \rangle)\]
\[= \langle top \rangle\]

Equation (18)
\[\text{cstack}(\text{regular})(\langle P \rangle)&(\langle push(d) \rangle, \langle top \rangle)\]
\[= \text{stack}(\langle P \rangle)&(\langle push(d) \rangle, \langle top \rangle)\]
\[= \langle P \rangle, \text{stack}(\langle push(d) \rangle, \langle top \rangle)\]
\[= \text{cstack}(\text{regular})(\langle P \rangle)&(\langle push(d) \rangle, \langle top \rangle)\]

Equation (19)
\[\text{cstack}(\text{errorpop})(\langle x \rangle)\]
\[= \text{stack}(\langle x \rangle)\]
\[= \langle x \rangle\]

Equation (20)
\[\text{cstack}(\text{errorpop})(\langle x \rangle)\]
\[= \text{stack}(\langle x \rangle)\]
\[= \langle x \rangle\]

Appendix B

Equation (23)
\[\text{dstack}(\langle push(d) \rangle)\]
\[= \text{stack}(\langle push(d) \rangle)\]
\[= \langle push(d) \rangle\]

Equation (24)
\[\text{dstack}(\langle push(d) \rangle)\]
\[= \text{stack}(\langle push(d) \rangle)\]
\[= \langle push(d) \rangle\]