Step by Step to Histories*

Max Breitling and Jan Philipps

Institut für Informatik
Technische Universität München
80290 München
Germany
{max.breitling|jan.philipps}@in.tum.de

Abstract. The behavior of reactive systems is typically specified by state machines. This results in an operational description of how a system produces its output. An alternative and more abstract approach is to just specify the relation between the input and output histories of a system. In this work, we propose a way to combine state-based and history-based specifications: Abstract communication history properties of system components can be derived from temporal logic properties of state machines. The history properties can then be used to deduce global properties of a complete system.

1 Introduction

To allow precise reasoning about a hard- or software system, a mathematical foundation for both systems and properties is a prerequisite. For some classes of systems —in particular, clocked hardware— temporal logics have been used successfully to formalize and to reason about their properties.

Temporal logic and model checking are less successful, however, when the dataflow between loosely coupled components that communicate asynchronously via communication channels is examined. For such systems, a black box view which just relates input and output is more useful than the state-based glass box view of a component. Black box properties of dataflow components and systems can be concisely formulated as relations over the communication history of components [7, 8]; such properties are inherently modular and allow easy reasoning about the global system behavior.

For individual data flow components, however, a state-based glass box view is helpful. State machines are good design documents for a component's implementation. Moreover, they provide an operational intuition that can aid in structuring proofs: Safety properties, for example, are typically shown using induction over the machine transitions.

In this paper we show —based on the ideas of Broy's verification of the Alternating Bit Protocol [6]— how specifications of the black box view of a

* This work is supported by the DFG within the Sonderforschungsbereich 342.

system or system component can be systematically derived from state machine specifications of the components. Thus we bridge the gap between techniques for easy verification of dataflow properties and more operational descriptions that are close to efficient implementations of a system.

The paper is structured as follows: In the next section we introduce some mathematical concepts and notations. § 2 and § 3 describe history specifications for the black box view, and state machines for the glass box view of a component, respectively. In § 4 we present verification rules for temporal logic properties that are used in § 5 to relate the black box and glass box views of a component. In § 6 we demonstrate how the black box views support compositional reasoning about a system. The conclusion in § 7 gives an outlook on future work.

2 History Relations

A dataflow system is a network of components. Each component has input and output ports. Ports of different components are connected by directed channels. Communication over these channels is asynchronous, message buffers are assumed to be unbounded. The black box view of a dataflow system regards only the communication between components and abstracts from the internal workings inside the components.

Systems in the black box view are modeled as relations over communication histories. The relations are expressed using formulas in predicate logic where the formula's free variables range over *streams*. Each free variable represents the communication history over one of the component's input or output ports.

There is a rich mathematical basis for this system model [7, 8]; this section contains only a short overview over the concepts used in the rest of the paper.

2.1 Streams.

The communication history between components is modeled by *streams*. A stream is a finite or infinite sequences of messages. Finite streams can be enumerated, for example: $\langle 1, 2, 3, \ldots 10 \rangle$; the empty stream is denoted by $\langle\,\rangle$. For a set of messages Msg, the set of finite streams over Msg is denoted by Msg^*, that of infinite streams by Msg^∞. By Msg^ω we denote $\mathsf{Msg}^* \cup \mathsf{Msg}^\infty$. Given two streams s, t and $j \in \mathbb{N}$, $\#s$ denotes the length of s. If s is finite, $\#s$ is the number of elements in s; if s is infinite, $\#s = \infty$. We write $s \frown t$ for the concatenation of s and t. If s is infinite, $s \frown t = s$. We write $s \sqsubseteq t$, if s is a prefix of t, i.e. if $\exists\, u \in \mathsf{Msg}^\omega \bullet s \frown u = t$. The j-th element of s is denoted by $s.j$, if $1 \leq j \leq \#s$; it is undefined otherwise. $\mathsf{ft}.s$ denotes the first element of a stream, i.e. $\mathsf{ft}.s = s.1$, if $s \neq \langle\,\rangle$.

The prefix relation $\sqsubseteq$ is a partial order. The set of streams Msg^ω together with $\sqsubseteq$ forms a complete partial order (CPO); the empty stream $\langle\,\rangle$ is the least element in this CPO. This means that for every chain $\{\, s_i \mid i \in \mathbb{N} \,\}$ of streams, where for each i: $s_i \sqsubseteq s_{i+1}$, there is a unique least upper bound $\bigsqcup\{\, s_i \mid i \in \mathbb{N}\}$. A predicate Φ where the free variables range over streams M^ω is *admissible*, if it holds for the limit of a chain of valuations for its variables, provided that it holds for each element of the chain. We then write $\mathsf{adm}\ \Phi$. Syntactical criteria for admissibility can be found in [12].

Stream concatenation and the prefix order can be extended pointwise to tuples of streams; continuity of functions and admissibility of prefix can also be defined for stream tuples.

2.2 Component Specification

Figure 1 shows the system structure of a bounded transmission system with three components: a sender, a receiver, and a buffer with a capacity for $N \geq 2$ data messages. For now, we just examine the sender.

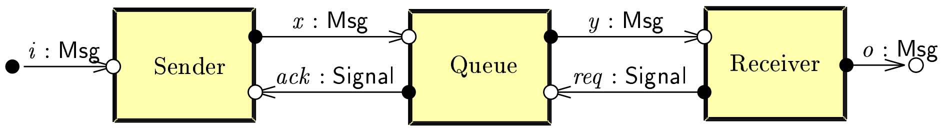

Fig. 1. Bounded Buffer

The black box view of the sender is specified by giving a set of input channel identifiers I and a set of output channel identifiers O (where $I \cap O = \varnothing$) to define its interface. The behavior is specified by a predicate with free variables from I and O. Each channel identifier has an assigned type that describes the set of messages allowed on that channel. Typically, we write the specification in the following style:

Sender

in i : Msg, ack : Signal
out x : Msg

$x \sqsubseteq i$
$\#x = min(\#i, 1 + \#ack)$

Intuitively, the sender behaves as follows: On channel x it forwards the messages it receives on channel i, in the same order, but possibly not all of them. This safety property is denoted by the first assertion. The second assertion contains both a safety and liveness part: For liveness, it demands the sender to send *at least* the number of messages it receives on i; but only as long as each message is acknowledged; the safety part asserts that *at most* this number is received.

The specification pattern of the sender is typical for history specifications: The specification is a conjunction of prefix expressions which restrict the data values on the output channels, and (in-)equalities, which specify the length of the output histories in terms of the length of the input histories.

2.3 Component Composition

The history relation of a composed system can be derived from the history relations of its components. Components may share input channels, but each output channel must be controlled by only one single component. This is captured in the definition of compatibility: Two components $\mathcal{S}_1$ and $\mathcal{S}_2$ are *compatible* if they do not share output channels: $O_{\mathcal{S}_1} \cap O_{\mathcal{S}_2} = \varnothing$.

The result of the composition, noted as $\mathcal{S}_1 \otimes \mathcal{S}_2$, is again a system specification. Channels with identical names are connected, the output of the composition is the union of the two component's output channels, and the input of the composition consists of those input channels that remain unconnected.

$$I_{\mathcal{S}_1 \otimes \mathcal{S}_2} \stackrel{\text{df}}{=} (I_{\mathcal{S}_1} \cup I_{\mathcal{S}_2}) \setminus (O_{\mathcal{S}_1} \cup O_{\mathcal{S}_2}), \quad O_{\mathcal{S}_1 \otimes \mathcal{S}_2} \stackrel{\text{df}}{=} O_{\mathcal{S}_1} \cup O_{\mathcal{S}_2}$$

The behavior of the composed system is defined as the conjunction of the component behavior predicates.

3 State Machines

State machines are a more operational way to specify dataflow components than history relations. We use the term *state machine* both for the abstract syntax (state transition systems, § 3.2) and for the concrete graphical representation (state transition diagrams, § 3.4). The executions of state transition systems are defined in § 3.3.

First we give a formal definition of variable valuations for an assertion. Variable valuations allow us to talk about the validity of assertions in the different states of a state machine execution.

3.1 Variable Valuations

We assume an (infinite) set Var of variable names. A valuation α is a function that assigns to each variable in Var a value from the variable's type. By $\mathsf{free}(\Phi)$ we denote the set of free variables in a logical formula Φ. If an assertion Φ evaluates to true when each variable $v \in \mathsf{free}(\Phi)$ is replaced by $\alpha(v)$, we write $\alpha \models \Phi$.

Variable names can be *primed*: For example, v' is a new variable name that results from putting a prime behind v. We extend priming to sets $V' \stackrel{\text{df}}{=} \{ v' \mid v \in V \}$ and to valuations: Given a valuation α of variables in Var, α' is a valuation of variables in V' with $\alpha'(v') = \alpha(v) \quad$ for all variables $v \in \mathsf{Var}$. Priming can also be extended to predicates, functions and other expressions: If Ψ is an assertion with $\mathsf{free}(\Psi) \subseteq V$, then Ψ' is the assertion that results from priming all free variables.

Note that an unprimed valuation α assigns values to all *unprimed* variables, while a primed valuation β' only assigns values to all *primed* variables. If an assertion Φ contains both primed and unprimed variables, we need two valuations to determine its truth. If Φ evaluates to true when each unprimed variable $v \in \mathsf{free}(\Phi)$ is replaced by $\alpha(v)$ and each primed variable $v' \in \mathsf{free}(\Phi)$ is replaced by $\beta'(v)$, we write $\alpha, \beta' \models \Phi$. Two valuations *coincide* on a subset $V \subseteq \mathsf{Var}$ if $\forall\, v \in V \bullet \alpha(v) = \beta(v)$. We then write $\alpha \stackrel{V}{=} \beta$.

3.2 State Transition Systems

A state transition system is a tuple $\mathcal{S} = (I, O, A, \mathcal{I}, \mathcal{T})$, where I, O, A are sets of variables. A state of our system is described by a valuation α, that assigns values to all variables in $V \stackrel{\text{df}}{=} I \cup O \cup A$. $\mathcal{I}$ is an assertion with $\mathsf{free}(\mathcal{I}) \subseteq V$ that characterizes the initial states of the state transition system. $\mathcal{T}$ is a finite set of transitions; each transition $\tau \in \mathcal{T}$ is an assertion with $\mathsf{free}(\mathcal{I}) \subseteq V \cup V'$. The tuple elements have to obey the following restrictions.

The sets I and O, with $I \cap O = \varnothing$, contain the input and output channel variables. The variables range over finite streams which represent the communication history to and from the component. The set A contains local state attributes, as e.g. a variable σ for a control state and variables for data states. Additionally, A contains for every $i \in I$ a variable i°. These variables hold the part of the external input stream i that has already been processed by $\mathcal{S}$. The restrictions on the initialization and transition assertions defined below ensure that $i^\circ \sqsubseteq i$ always holds. We can therefore define i^+ as the part of the message history that has not yet been processed by $i = i^\circ \frown i^+$.

The assertion $\mathcal{I}$ characterizes the initial states of the system. We require $\mathcal{I}$ to be satisfiable for arbitrary input streams

$$\exists\, \alpha \bullet \alpha \models \mathcal{I} \quad \wedge \quad \left(\forall\, \beta \bullet \beta \stackrel{O \cup A}{=} \alpha \quad \Rightarrow \quad \beta \models \mathcal{I}\right)$$

and to assert that initially no input has been processed and no output has yet been produced:

$$\mathcal{I} \Rightarrow \bigwedge_{i \in I} i^\circ = \langle\,\rangle \wedge \bigwedge_{o \in O} o = \langle\,\rangle$$

The set $\mathcal{T}$ contains the allowed transitions of $\mathcal{S}$. Every transition $\tau \in \mathcal{T}$ is an assertion over $V \cup V'$ and relates states with their successor states. Unprimed variables in τ are valuated in the current state, while primed variables are valuated in the successor state. All transitions must guarantee that the system does not take back messages it already sent, that it can not undo the processing of input messages, that it can only read messages that have been sent to the component and that it does not change the variables for input streams, since these are controlled by the environment:

$$\tau \Rightarrow \bigwedge_{o \in O} o \sqsubseteq o' \wedge \bigwedge_{i \in I} i^\circ \sqsubseteq i^{\circ\prime} \wedge \bigwedge_{i \in I} i^{\circ\prime} \sqsubseteq i \wedge \bigwedge_{i \in I} i = i'$$

In addition to the transitions in $\mathcal{T}$, there is an implicit *environment transition* τ_ϵ. This transition is defined to allow the environment to extend the input, while it leaves the controlled variables $v \in O \cup A$ unchanged:

$$\tau_\epsilon \Leftrightarrow \bigwedge_{v \in O \cup A} v = v' \wedge \bigwedge_{i \in I} i \sqsubseteq i'$$

A transition is *enabled* in a state α, written as $\alpha \models \mathsf{En}(\tau)$, iff there is a state β such that $\alpha, \beta' \models \tau$.

3.3 Executions

An *execution* of a STS $\mathcal{S}$ is an infinite stream ξ of valuations that satisfies the following three requirements:

1. The first valuation in ξ satisfies the initialization assertion:

   $$\xi.1 \models \mathcal{I}$$

2. Each pair of subsequent valuations $\xi.k$ and $\xi.(k+1)$ in ξ are related either by a transition in $\mathcal{T}$ or by the environment transition τ_ϵ:

   $$\xi.k, \xi'.(k+1) \models \tau_\epsilon \vee \bigvee_{\tau \in \mathcal{T}} \tau$$

3. Each transition $\tau \in \mathcal{T}$ of the STS is taken infinitely often in an execution, unless it is disabled infinitely often (weak fairness):

   $$\big(\forall\, k \bullet \exists\, l \geq k \bullet \xi.l \models \neg\, \mathsf{En}(\tau)\big) \vee \big(\forall\, k \bullet \exists\, l \geq k \bullet \xi.l, \xi'.(l+1) \models \tau\big)$$

By $\langle\!\langle \mathcal{S} \rangle\!\rangle$ we denote the set of all executions of a system $\mathcal{S}$.

3.4 State Transition Diagrams

Typically, state transition systems are specified by *state transition diagrams* (STDs). We use a subset of the STD syntax from the CASE tool AutoFocus [9]. STDs are directed graphs where the vertices represent (control) states and the edges represent transitions between states. One vertex is designated as *initial state*; graphically this vertex is marked by an opaque circle in its left half. Edges are labeled; each label consists of four parts, represented by the following schema:

$$\{\mathit{Precondition}\}\ \mathit{Inputs} \triangleright \mathit{Outputs}\ \{\mathit{Postcondition}\}$$

Inputs and *Outputs* stand for lists of expressions of the form $i?x$ and $o!exp$ ($i \in I, o \in O$) respectively, where x is a constant value or a (transition-local) variable of the type of i, and *exp* is an expression of the type of o. The *Precondition* is a boolean formula containing data state variables and transition-local variables as free variables, while *Postcondition* and *exp* may also contain primed variables. The distinction between pre- and postconditions does not increase the expressiveness, but improves readability. If the pre- or postconditions are equivalent to true, they can be omitted.

The informal meaning of a transition is as follows: If the available messages on the input channels can be matched with *Inputs*, the precondition is true and the postcondition can be made true by assigning proper values to the primed

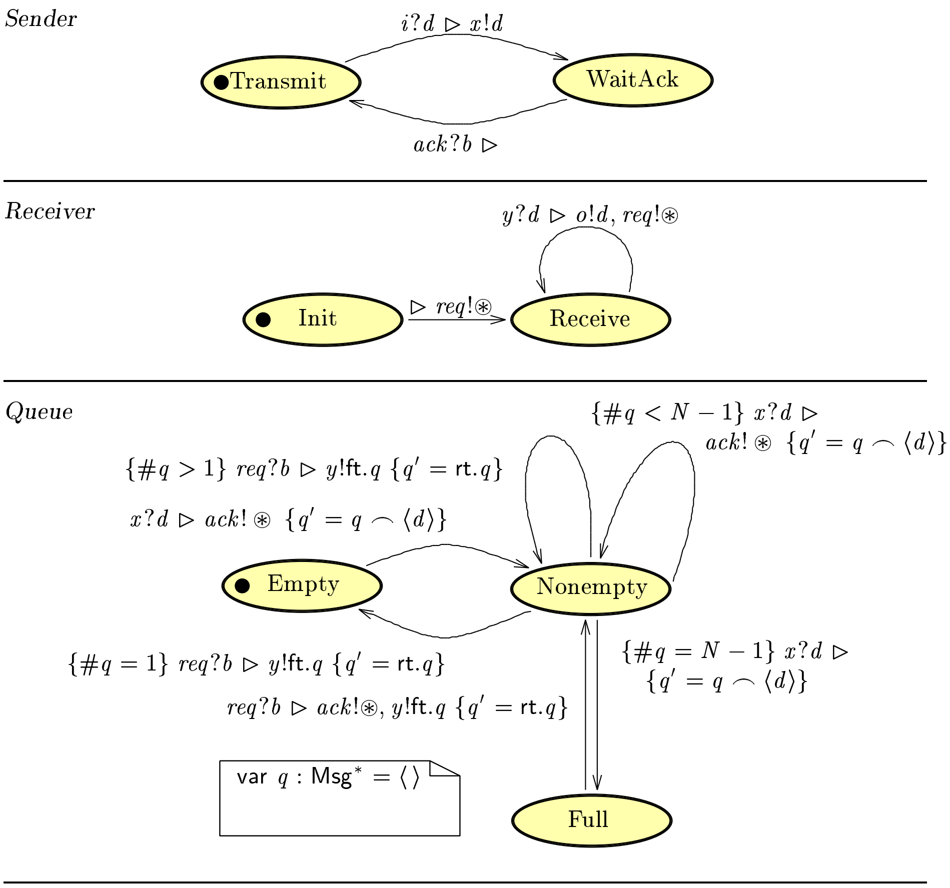

Fig. 2. Sender, Receiver and Queue STDs

variables, then the transition is enabled. If the transition is executed, the inputs are read, the outputs are written and the postcondition is made true.

Figure 2 shows the STDs of sender, queue and receiver of the transmission system (see Fig. 1). Again, we focus on the sender component: If the sender receives some data d on channel i, this message is immediately forwarded on x, and the system starts waiting for an acknowledgment message on channel ack. When the acknowledgment is received, the sender is ready to receive the next message from i.

State transition diagrams can be encoded schematically as state transition systems. For the sender component, the variable sets are defined as follows: $I = \{i, ack\}$, $O = \{x\}$ (see Fig. 1), $A = \{i^\circ, ack^\circ, \sigma\}$. The state attributes consist of the processed message stream for each of the two input channels, and a variable σ to hold the current control state.

The initial assertion $\mathcal{I}$ of the sender is defined as:

$$\sigma = \mathit{Transmit} \wedge i^\circ = \langle\,\rangle \wedge \mathit{ack}^\circ = \langle\,\rangle \wedge x = \langle\,\rangle$$

The transition τ_1 from the state *Transmit* to the state *WaitAck* in the sender STD is encoded as the following assertion:

$$\begin{array}{lll}
\exists\, d\,. & \sigma = \mathit{Transmit} & \textsf{\textit{We move from the source state}} \\
& \wedge\ \sigma' = \mathit{WaitAck} & \textsf{\textit{to the target state.}} \\
& \wedge\ \#i^\circ < \#i & \textsf{\textit{There are unread messages in channel }} i. \\
& \wedge\ \mathit{ft}.i^+ = d & \textsf{\textit{Let }} d \textsf{\textit{ be the first of them,}} \\
& \wedge\ i^{\circ\prime} = i^\circ \frown \langle d\rangle & \textsf{\textit{which we consume}} \\
& \wedge\ x' = x \frown \langle d\rangle & \textsf{\textit{and send on channel }} x, \\
& \wedge\ \mathit{ack}^{\circ\prime} = \mathit{ack}^\circ & \textsf{\textit{whereas we don't read from channel }} \mathit{ack}, \\
& \wedge\ i = i' \wedge \mathit{ack} = \mathit{ack}' & \textsf{\textit{and leave the input channels unchanged.}}
\end{array}$$

The second transition τ_2 of the sender can be encoded similarly. Note that the initialization and transition assertion obey the restrictions from § 3.2.

The queue and receiver components lead to similar transition assertions. In case of the queue component, there is an additional variable q in A. Initially, $q = \langle\,\rangle$; the transitions change q according to the queue STD. A more detailed explanation of the translation of STDs to STS assertions can be found in [2].

4 Verification Rules

A common technique for formalizing and verifying properties of state transition system executions is temporal logic [11]. For the state machines of § 3 we are not interested in general temporal logic properties, but only in two special cases: invariants for *safety* properties and *leadsto* properties for liveness. This section introduces verification rules for these two property classes. Soundness proofs of these and other rules —expressed in a UNITY-like formalism— can be found in [2].

Note that both invariance and leadsto properties relate single states in an STS execution; in § 5 these properties are used to express properties about the complete communication history of executions.

4.1 Invariance Properties

To show that a STS $\mathcal{S}$ fulfills a safety property, we use invariants. For a system $\mathcal{S} = (I, O, A, \mathcal{I}, \mathcal{T})$, an assertion Φ with $\mathsf{free}(\Phi) \subseteq I \cup O \cup A$ is an invariant, written as $\mathcal{S} \models \Box\Phi$, if Φ evaluates to true for each state in all executions of $\mathcal{S}$:

$$\mathcal{S} \models \Box\Phi \quad\Leftrightarrow\quad \forall\, \xi \in \langle\!\langle \mathcal{S} \rangle\!\rangle \bullet \forall\, k \bullet \xi.k \models \Phi$$

To prove Φ to be an invariant, we have to show that Φ holds initially, and remains true under each transition $\tau \in \mathcal{T}$ as well as under the environment transition τ_ϵ:

$$\begin{array}{|l}
\mathcal{I} \Rightarrow \Phi \\
\Phi \wedge \tau \Rightarrow \Phi' \quad \text{for all } \tau \in \mathcal{T} \\
\Phi \wedge \tau_\epsilon \Rightarrow \Phi' \\
\hline
\mathcal{S} \models \Box\Phi
\end{array}$$

Example. For the sender, the output on channel x is always equal to the sequence of messages from i that have already been consumed:

$$Sender \models \Box\ x = i^\circ$$

The first condition of the invariant rule is fulfilled, since for the sender initially both x and i° are empty (see § 3.4). The other two premises are fulfilled since the sender transition τ_1 appends a single message to both x and i°; for transitions τ_2 and τ_ϵ we observe that both x and i° remain unchanged.

4.2 Leadsto Properties

Progress of a system can be expressed using the leadsto operator $\Phi \leadsto \Psi$, which states that whenever Φ is true for a state in an execution, then Ψ will be true in the same or in a subsequent state in the execution. Usually, the leadsto operator is defined in temporal logic as $\Box(\Phi \Rightarrow \Diamond\Psi)$, but for our purposes the following semantic definition of $\mathcal{S} \models \Phi \leadsto \Psi$ is sufficient:

$$\mathcal{S} \models \Phi \leadsto \Psi \quad \Leftrightarrow \quad \forall\, k \bullet (\xi.k \models \Phi) \Rightarrow (\exists\, l \geq k \bullet \xi.l \models \Psi)$$

For the leadsto operator, too, there are verification rules:

$$\begin{array}{|l}
\text{For all transitions } \tau \in \mathcal{T} \cup \{\tau_\epsilon\}\text{:} \\
\quad \Phi \wedge \neg\, \Psi \wedge \tau \Rightarrow \Phi' \vee \Psi' \\
\text{For a transition } \tau \in \mathcal{T}\text{:} \\
\quad \Phi \wedge \neg\, \Psi \Rightarrow \mathsf{En}(\tau) \\
\qquad \text{and} \\
\quad \Phi \wedge \neg\, \Psi \wedge \tau \Rightarrow \Psi' \\
\hline
\mathcal{S} \models \Phi \leadsto \Psi
\end{array}
\qquad
\begin{array}{|l}
\text{For a transition } \tau \in \mathcal{T}\text{:} \\
\quad \#o = k \wedge k < \mathcal{L} \Rightarrow \mathsf{En}(\tau) \\
\qquad \text{and} \\
\quad \#o = k \wedge k < \mathcal{L} \wedge \tau \Rightarrow \#o' > k \\
\hline
\mathcal{S} \models \#o = k \wedge k < \mathcal{L} \ \leadsto\ \#o > k
\end{array}$$

The first rule is a standard verification rule for liveness under weak fairness [10, 11]: There is a helpful transition $\tau \in \mathcal{T}$ which is enabled in all states where Φ holds, and which leads into a state where Ψ holds (second premise). The other transitions are not harmful in that they leave Φ invariant. Thus, the helpful transition remains enabled until it is, by weak fairness, executed. The second rule, the *output extension rule*, is a specialization of the first rule. It is used to prove that an output stream exceeds a certain length k provided that sufficient input is available. This can be described by an $\mathbb{N}$-valued length expression $\mathcal{L}$ with

$\mathsf{free}(\mathcal{L}) \subseteq I$ which is monotonic in its free variables. The main difference to the first rule is that it is not necessary to show the safety premises of the first rule: For this special case they hold trivially, since channel valuations are monotonic with respect to $\sqsubseteq$, and due to its monotonicity the length expression $\mathcal{L}$ can be proven to be nondecreasing [2]. The left hand side of the output extension's conclusion rule can be strengthened by an arbitrary predicate Ψ, if the left hand sides of the premises are also strengthended by Ψ.

Besides the two rules above, there are a number of additional rules for the leadsto operator: transitivity, weakening of the right hand side, strengthening of the left hand side. The *disjunction rule* combines two leadsto properties: If $\mathcal{S} \models \Phi_1 \leadsto \Psi$ and $\mathcal{S} \models \Phi_2 \leadsto \Psi$, then also $\mathcal{S} \models (\Phi_1 \vee \Phi_2) \leadsto \Psi$. Moreover, invariants can be introduced and eliminated on both sides of the operator.

Example. Again regarding the sender, we want to show

$$Sender \models \#x = k \wedge k < min(\#i, 1 + \#ack) \leadsto \#x > k$$

which expresses that the output on x is extended, provided there is sufficient input on i and ack expressing that the length of the output on x is reaching at least the limit $min(\#i, 1 + \#ack)$.

For $\sigma = Transmit$, we use the output extension rule with τ_1 as the helpful transition, since it produces output on x. The last condition of the rule is easy to prove, since τ_1 implies the extension of x by $x' = x \frown \langle d \rangle$, so that $\#x = k \wedge \tau_1 \leadsto \#x' > k$ is trivial. For the second condition we have to prove that τ_1 is enabled. If we assume $\sigma = Transmit$, it is enabled iff there is some message on the channel i, i.e. iff i is longer than its consumed part i°. Using the safety invariant from above, this can be derived as follows:

$$\#i \geq min(\#i, 1 + \#ack) > k = \#x = \#i^\circ$$

For $\sigma = WaitAck$, transition τ_1 is not enabled. Instead, we use the standard weak fairness rule to show that by transition τ_2 state $WaitAck$ is entered. The two results can be combined with the transitivity and disjunction rules to derive the property

$$\begin{aligned} Sender \models \big((\sigma = WaitAck \vee \sigma = Transmit) \\ \wedge \#x = k \wedge k < min(\#i, 1 + \#ack)\big) \quad \leadsto \quad \#x > k \end{aligned}$$

It can be shown that $\sigma = WaitAck \vee \sigma = Transmit$ is an invariant; its elimination results in the property above [2].

5 History Properties

We introduced two ways to specify reactive systems: history relations and state machines. The two views describe quite different views on a system: Using the black box views of history relations, we model the I/O behavior with streams;

the relations do not refer to any internals of the components and do not describe how this behavior is achieved. Using state machines we concentrate on single steps of the system, referring to the component internals. In this section, we close the gap between state machines and black box views.

Within a state machine execution ξ, changes in the valuations for the input and output variables in $I \cup O$ are restricted to extensions. Thus the valuations of each input and output variable within an execution form a chain, and for each execution and each variable $v \in I \cup O$ there is a least upper bound

$$\lceil \xi \rceil (v) \stackrel{\text{df}}{=} \bigsqcup \{ \ (\xi.k)(v) \ \mid \ k \in \mathbb{N} \ \}$$

Note that $\lceil \xi \rceil (v)$ is only defined for the input and output variables, not for the attribute variables A of a state machine.

The *black box view* of a state machine is a set of valuations for the variables $I \cup O$. It is denoted by $[\![\mathcal{S}]\!]$ and defined via the least upper bounds of the input and output histories of the machine's executions. A valuation in $[\![\mathcal{S}]\!]$ assigns those streams to input and output channels that can appear 'in infinity' on the channels in an infinite execution of the system:

$$[\![\mathcal{S}]\!] \stackrel{\text{df}}{=} \{ \ \alpha \ \mid \ \exists \, \xi \in \langle\!\langle \mathcal{S} \rangle\!\rangle \bullet \bigwedge_{i \in I} \alpha(i) = \lceil \xi \rceil (i) \wedge \bigwedge_{o \in O} \alpha(o) = \lceil \xi \rceil (o) \ \}$$

Since both the proper transitions $\tau \in \mathcal{T}$ and the environment transition τ_ϵ of a state machine allow arbitrary extension of the input variable valuations, it is possible to successively approximate an arbitrary input history. This means that the black box view $[\![\mathcal{S}]\!]$ is total with respect to the input variables of $\mathcal{S}$: For an arbitrary input there is always some reaction of the system. Formally, this reads as: For each valuation α for the variables $I \cup O$ there exists a valuation β for $I \cup O$ such that

$$\alpha \stackrel{I}{=} \beta \quad \text{and} \quad \beta \in [\![\mathcal{S}]\!]$$

5.1 Safety Properties

In practice, it is difficult to directly use the black box semantics $[\![\mathcal{S}]\!]$ of a state machine. Instead, we derive properties of the black box view from properties of the state machine. Technically, a property of the black box view $[\![\mathcal{S}]\!]$ is a predicate Φ with $\mathsf{free}(\Phi) \subseteq I \cup O$ which is valid for each valuation in a system's black box view:

$$\forall \, \alpha \in [\![\mathcal{S}]\!] \bullet \alpha \models \Phi$$

We then write $[\![\mathcal{S}]\!] \Rightarrow \Phi$.

If Φ is an admissible invariance property of a state machine, it holds not only in every state of a system run, but also for the complete communication history:

$$\begin{array}{|l}
\mathsf{free}(\Phi) \subseteq I \cup O \\
\mathsf{adm}\ \Phi \\
\mathcal{S} \models \Box\Phi \\
\hline
[\![\mathcal{S}]\!] \Rightarrow \Phi
\end{array}$$

The validity of the rule follows from the fact that the valuations of the channel variables I and O form a chain. Because of it is invariant, Φ holds for every element of the chain. Because of admissibility, it also holds in the limit.

Example. In § 4.1 we showed that $x = i^\circ$ is an invariant of the sender. Moreover, we have $i^\circ \sqsubseteq i$, and thus $x \sqsubseteq i$ is also an invariant. This predicate is also admissible [12], and thus we can directly conclude

$$[\![Sender]\!] \Rightarrow x \sqsubseteq i$$

This means that the sender STD implies the first half of the sender's history specification in § 2.2. Similarly, we can show $[\![Sender]\!] \Rightarrow \#x \leq 1 + \#ack$.

5.2 Progress Properties

In general, progress properties expressed with the leadsto operator $\leadsto$ cannot be lifted to complete executions. However, output extension properties (§ 4.2) can be used to derive liveness properties of a state machine's black box view. In the following rule, $\mathcal{L}$ is a monotonic $\mathbb{N}$-valued expression with $\mathsf{free}(\mathcal{L}) \subseteq I$, as used in the output extension rule.

$$\begin{array}{|l}
\mathcal{S} \models \#o = k \wedge k < \mathcal{L} \leadsto \#o > k \\
\hline
[\![\mathcal{S}]\!] \Rightarrow \#o \geq \mathcal{L}
\end{array}$$

To see the validity of the rule, assume that the premise holds, but not the conclusion. Thus, there is an execution ξ of $\mathcal{S}$ such that the length of the limit of the channel valuations for o is strictly less than the limit of the valuations of $\mathcal{L}$ in each state; in particular, it is equal to a natural number k. This means that there is an earliest state $\xi.n$ in the execution where the length of the output valuation for o reaches k. Moreover, there is a state $\xi.m$ where $\mathcal{L}$ is larger than k. Since channel valuations cannot become shorter, and $\mathcal{L}$ is monotonic, this means that in all states $\xi.p$, where $p \geq max(n, m)$ the left hand side of the premise is fulfilled, but the right hand side never holds. This violates the assumption that the premise is valid.

Example. From the result of our example in § 4.2, we can directly use the above rule to derive

$$[\![Sender]\!] \Rightarrow \#x \geq min(\#i, 1 + \#ack)$$

Together with the safety properties shown above, this implies the second part of the sender's history specification.

6 Black Box Composition

We now have a closer look on the complete transmission system of Fig. 1. The sender pushes data to the queue and waits for acknowledgments and the receiver requests data from the queue; the queue itself stores up to N ($N \geq 2$) data messages.

The behavior of the three components is defined in Fig. 2 by STDs. Using the techniques of this paper, we can show that the receiver and the queue imply the following history relations:

Queue(N)

in x : Msg, req : Signal
out ack : Signal, y : Msg

$$
\begin{aligned}
& y \sqsubseteq x \\
& \#y \geq min(\#x, \#req) \\
& \#ack = min(\#x, \#req + N - 1)
\end{aligned}
$$

Receiver

in y : Msg
out req : Signal, o : Signal

$$
\begin{aligned}
& o \sqsubseteq y \\
& \#o \geq \#y \\
& \#req = 1 + \#y
\end{aligned}
$$

By black box composition, the history relation of the complete system is specified as follows. The behaviour is simply described by the conjunction of the components properties.

System(N)

in i : Msg
out o : Signal, x : Msg, ack : Signal, y : Msg, req : Signal

$$
\begin{aligned}
& x \sqsubseteq i \\
& y \sqsubseteq x \\
& o \sqsubseteq y \\
& \#x = min(\#i, 1 + \#ack) \\
& \#y \geq min(\#x, \#req) \\
& \#ack = min(\#x, \#req + N - 1) \\
& \#o \geq \#y \\
& \#req = 1 + \#y
\end{aligned}
$$

From the specification of $System(N)$ above, we can immediately see that the output is a prefix of the input: $o \sqsubseteq y \sqsubseteq x \sqsubseteq i$. Using the inequalities it can also be shown by some case analysis that the length of the output equals the length of the input. Together, this implies

$$o = i$$

for all input streams i. As expected, the system implements the identity relation.

The same result could have been obtained by first composing the three component state machines, and then deriving $o \sqsubseteq i$ and $\#o \geq \#i$; the number of verification conditions for the invariance and leadsto properties would have been much higher, however. For the composition of dataflow properties, history relations seem to be the more adequate abstraction level.

7 Conclusion

In this paper we showed how state-based and history-based specification and verification techniques for safety and liveness properties of distributed systems can be combined. State machine properties are expressed using a standard linear temporal logic; history properties are expressed as relations between input and output streams.

In a related technical report [2] we also allow composition at the level of state machines; properties proven for the combined system are shown to hold also for the black box composition of a system. That our system is compositional is due to the dataflow nature of our systems: Components cannot disable transitions of other components, thus the system is interference free. This is quite useful in practice, since it is often hard to find suitable history predicates for each component, although the complete system behavior can be succinctly specified in this way. State machine composition also helps to circumvent the mismatch between purely relational dataflow specifications and the operational intuition that was discovered by Brock and Ackermann [3].

Proofs for larger systems, especially for leadsto properties, are often quite complex. A solution might be to use verification diagrams along the lines of [4, 11], which reduce temporal reasoning to simple first-order verification conditions. Since the number of verification conditions for concrete systems can be quite large, some kind of tool support is needed. As an experiment, the safety properties of the communication system example have been verified using the STeP [1] proof environment; currently, we are formalizing our approach in Isabelle/HOL [13].

Our specification and proof techniques are so far only suited for time independent systems. The extension of history-based specifications raises some interesting questions [5]. A straightforward solution might be to explicitly include "time ticks" in the message streams. Such time ticks can also be used to ensure progress of a state machine. But also without explicit time, progress is not restricted to the weak fairness condition of § 3.3. An alternative would be to just demand that some transition is taken whenever at least one transition is persistently enabled; some classes of components, in particular *fair merge* components would then require additional oracle inputs.

Acknowledgments This report benefited from many stimulating discussions with Manfred Broy. We thank Katharina Spies for comments on a draft version of this report, and one anonymous referee for his very detailed remarks.

References

1. N. Bjørner, A. Browne, E. Chang, M. Colón, A. Kapur, Z. Manna, H. B. Sipma, and T. E. Uribe. STeP: Deductive-Algorithmic Verification of Reactive and Real-time Systems. In *CAV'96. Lecture Notes in Computer Science 1102*, pages 415–418, 1996.

2. M. Breitling and J. Philipps. Black Box Views of State Machines. Technical Report TUM-I9916, Institut für Informatik, Technische Universität München, 1999.
3. J. D. Brock and W. B. Ackermann. Scenarios: A model of nondeterministic computation. In J. Diaz and I.Ramos, editors, *Lecture Notes in Computer Science 107*, pages 225–259, 1981.
4. I. A. Browne, Z. Manna, and H. B. Sipma. Generalized temporal verification diagrams. In *Lecture Notes in Computer Science 1026*, pages 484–498, 1995.
5. M. Broy. Functional specification of time sensitive communicating systems. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *Models, Formalism, Correctness. Lecture Notes in Computer Science 430*, pages 153–179. Springer, 1990.
6. M. Broy. From states to histories. In *Engineering Theories of Software Construction*. NATO Science Series F, Marktoberdorf Summer School, 2000. To be published.
7. M. Broy, F. Dederichs, C. Dendorfer, M. Fuchs, T. F. Gritzner, and R. Weber. The Design of Distributed Systems: An Introduction to Focus—Revised Version. Technical Report TUM-I9202-2, Institut für Informatik, Technische Universität München, 1993.
8. M. Broy, F. Huber, B. Paech, B. Rumpe, and K. Spies. Software and system modeling based on a unified formal semantics. In M. Broy and B. Rumpe, editors, *Requirements Targeting Software and Systems Engineering, International Workshop RTSE'97. Lecture Notes in Computer Science 1526*. Springer, 1998.
9. F. Huber, B. Schätz, A. Schmidt, and K. Spies. Autofocus—a tool for distributed systems specification. In *Proceedings FTRTFT'96 — Formal Techniques in Real-Time and Fault-Tolerant Systems. Lecture Notes in Computer Science 1135*, 1996.
10. L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages*, 6(3):872–923, May 1994.
11. Z. Manna and A. Pnueli. Models for reactivity. *Acta Informatica*, 30:609–678, 1993.
12. L. C. Paulson. *Logic and Computation*. Cambridge University Press, 1987.
13. L. C. Paulson. *Isabelle: A Generic Theorem Prover. Lecture Notes in Computer Science 828*. Springer, 1994.