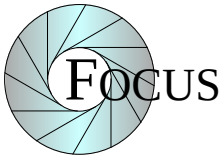
Diagrams for Dataflow *

Max Breitling Jan Philipps
Institut für Informatik
Technische Universität München
D-80290 München
http://www.in.tum.de/~{breitlin|philipps}



Abstract

The behavior of reactive systems can be described by their black box properties as a relation between input and output streams. More operational is the behavior's description by state machines. While the first view enjoys easy compositionality, the second leads directly to implementations. In this paper we show how these two views can be integrated by offering a technique for proving that a state machine indeed has specified safety and liveness properties.

We use graphical description techniques both for specifying the state machines and for structuring the proofs, and sketch how theorem provers help to generate and discharge resulting proof obligations.

1 Introduction

State transition diagrams in various incarnations have become a popular technique to specify software and hardware systems. Their suggestive notation leads to readable design documents for a component's implementation. With temporal logics there are precise property specification and verification techniques for state machines. Proofs in temporal logic often follow the operational intuition behind state machines: Invariance properties, for example, are typically shown using induction over the machine transitions.

Temporal logic and model checking are less successful, however, when the data flow 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 [1]; such properties are inherently modular and allow easy reasoning about the global system behavior.

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

In this paper, we first show how state-based and history-based specification and verification techniques for safety and liveness properties of distributed systems can be combined; then, we adopt Manna and Pnueli's verification diagrams [9] for our properties and describe tool support for verification through the theorem prover Isabelle.

2 Structure Diagrams

To specify the (statical) interface of a system and its internal structure, we can use *system structure diagrams* (SSD). In these, components are represented by boxes that are interconnected by directed arrows, representing communication channels. A channel is labeled with its name and its associated type of messages. Channels not connected to another component are input or output channels of the overall system.

As an example, we consider a simple buffer: It has two input channels i and r, and one output channel o, and its SSD is shown in Figure 1.

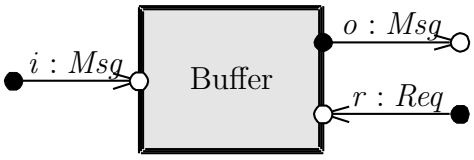


Figure 1: Simple Buffer

The buffer is intended to store all messages it receives on i. For every request message ® it receives on the channel r, it re-sends the stored data in a FIFO-manner via the output channel. A behavior can be specified by the relation of the input and output communication histories that are allowed on its channels. 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

Buffer

in $i : \mathsf{Msg}^\omega, r : \{®\}^\omega$

out $o : \mathsf{Msg}^\omega$

$o \sqsubseteq i$

$\#o \leq \#r$

$\#o \geq min(\#i, \#r)$

Figure 2: Specification of the Buffer

$\exists\, u \in \mathsf{Msg}^\omega \bullet s \frown u = t$. 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 $s \neq \langle\,\rangle$, while $\mathsf{rt}.s$ denotes the rest of a stream when this first element was removed, i.e. $\langle \mathsf{ft}.s \rangle \frown \mathsf{rt}.s = s$ if $s \neq \langle\,\rangle$.

The buffer's interface and behavior can be defined using the format in Figure 2. The top part declares input and output channels, the bottom part specifies the buffer's I/O relation in the style of FOCUS [1]. I/O relations can often be divided into *prefix* properties for safety aspects and *length* properties for liveness aspects. In the example, the first two lines of the behavior specification characterize safety aspects, the third line liveness aspects of the buffer.

3 State Machines as Diagrams

We use the term *state machine* both for the abstract syntax (state transition systems, STS) and for the concrete graphical representation (state transition diagrams, STD).

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 that assigns values to these variables. The variables in I and O represent input and output channel histories, respectively. They range over finite message streams. $\mathcal{I}$ is an assertion 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 that relates a current state with its successor states. While the free variables of $\mathcal{I}$ belong to $V = I \cup O \cup A$, the free variables of each transition assertion τ belong to $V \cup V'$. The primed variables in V' refer to variable valuations in the next state. Each transition τ can only extend the channel histories I and O. In addition, there is an environment transition τ_ϵ that leaves all variables unchanged except those of I — the input histories I may be extended.

For this class of transition systems we define executions as state sequences that respect $\mathcal{I}$, the transitions in $\mathcal{T}$, and a fairness condition (see [2] for details).

Furthermore, we define a simple linear temporal logic that is inspired by UNITY [10, 11]: A property Φ is an *invariant* for a system $\mathcal{S}$ iff Φ is valid in all reachable states in executions of $\mathcal{S}$; this is written as $\mathcal{S} \models \mathbf{inv}\, \Phi$. With $\mathcal{S} \models \Phi \mapsto \Psi$ we describe *response* properties: Whenever in an execution of the state machine $\mathcal{S}$ a state is reached where Φ holds, then at the same or a later state in the execution Ψ also holds.

An STS can be represented by a *state transition diagram*. STDs are directed graphs where the vertices represent (control) states and the edges represent transitions between states. One vertex is a designated *initial state*; graphically this vertex is marked by an opaque circle in its left half. Edges are labeled; each label consists of four parts: A *precondition*, a set of *input statements*, a set of *output statements* and a *postcondition*. In STDs, transition labels are represented with the following schema:

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

Inputs and *Outputs* stand for lists of expressions of the form

$$i?x \quad \text{and} \quad o!\mathit{exp} \quad (i \in I, o \in O)$$

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

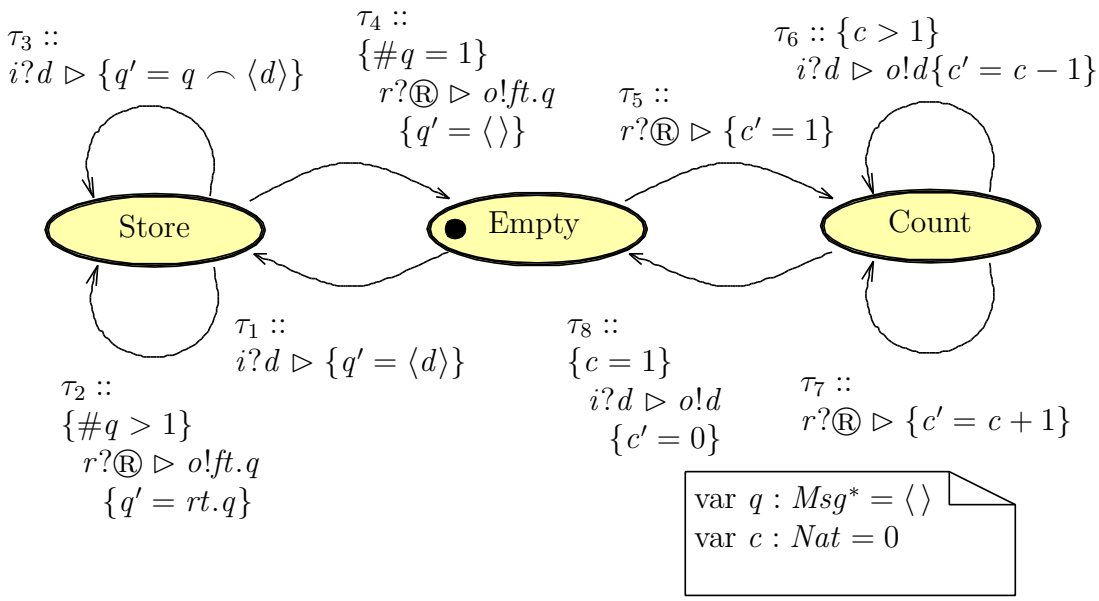


Figure 3: State Transition Diagram for the Buffer

and transition-local variables as free variables, while *Postcondition* and *exp* may additionally contain primed state 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 in the input channels can be matched with *Inputs*, the precondition is and the postcondition can be made true by assigning proper values to the primed variables, the transition is enabled. If it is chosen, the inputs are read, the outputs are written and the postcondition is made true.

A STD for the buffer is given in Fig. 3. It start in the state *Empty*. If some data is received on i, it is stored in q, and the control moves to the state *Store*. In this state, receiving a request, the first element of q is sent. Depending on the length of q, the buffer leaves the control state unchanged or moves back to *Empty*. Receiving further data in the state *Store*, they are appended in q. If there are no stored messages (in the state *Empty*), but a request arrives, this open request has to be remembered by incrementing c. If $c > 0$, the buffer is in the state *Count*. If some data arrives now, it is immediately forwarded on o, decrementing c, until there are no more pending requests and the buffer returns to the state *Empty*.

Since in each execution of an STS $\mathcal{S}$ the valuations of the channel variables I and O can only be extended between adjacent states, they form a chain in the CPO of channel valuations. Thus, there is a unique least upper bound of this chain, which assigns to each channel variable c in $I \cup O$ the complete communication history over c. This least upper bound is the I/O history of the execution; it is denoted by $[\![\mathcal{S}]\!]$.

In [2] we have shown that in order to verify that I/O histories of a state machine fulfill a given FOCUS specification, it is sufficient to prove an *invariance* property for prefix properties of the FOCUS specification, and a temporal logic *response* or *leads-to* property for each length property.

If Φ is an admissible predicate with free variables from $I \cup O$, then

$$[\![\mathcal{S}]\!] \Rightarrow \Phi \quad \text{iff} \quad \mathcal{S} \models \mathbf{inv}\, \Phi$$

For $u \in O$, $v_1, \ldots, v_n \in I$, and a monotonic function f from the input channel histories to $\mathbb{N}$, we have

$$[\![\mathcal{S}]\!] \Rightarrow \#u \geq f(v_1, \ldots, v_n) \quad \text{iff} \quad \mathcal{S} \models \big(\#u = k \wedge f(v_1, \ldots, v_n) > k\big) \mapsto \#u > k$$

4 From Diagrams to Properties

Since invariance and response properties link the state machine and I/O history views of a system, proofs about the I/O history of a system can be reduced to proofs of temporal logic properties. The usual linear presentation of such proofs, however, does not reflect the operational intuition behind the proof and can be confusing and hard to understand.

Verification diagrams [9] visualize the proof structure of temporal logic proofs and reduce temporal reasoning to proofs of verification conditions in first-order predicate logic. Verification diagrams can be tailored to the invariance and response properties used in the derivation of I/O history properties for state machines.

A verification diagram is a directed graph without unreachable nodes. The diagram's nodes are labeled by assertions $\Phi_0, \ldots, \Phi_n$. The free variables of each assertion are a subset of the STS variables $V = I \cup O \cup A$. Nodes marked by opaque circles in the left half are called *initial nodes*. A node marked by an opaque circle in the right half is called the *terminal node*. Initial and terminal nodes are optional, and there must be at most one terminal node. We tacitly assume that all node assertions are syntactically different and logically exclusive, and refer to the nodes by just their assertions. The edges in a verification diagram are labeled by transitions $\tau \in \mathcal{T}$.

Verification diagrams can be hierarchical: A node can contain a sub-diagram. Hierarchical diagrams are equivalent to flattened diagrams, where assertions from a node higher in the hierarchy are conjoined with the assertion of the nodes below it and arrows entering or exiting higher-level nodes are connected to all lower-level nodes.

With each verification diagram, a number of verification conditions are associated. Each condition belongs to one of the two following patterns, where Φ, Ψ are state assertions and τ is a transition: $\Phi \wedge \tau \Rightarrow \Psi'$ (*consequence*) and $\Phi \Rightarrow \mathsf{En}(\tau)$ (*enabledness*). The predicate Ψ' is a "next-state" variant of Ψ obtained by priming all free variables in Ψ.

Invariance Diagrams. An invariance diagram is a verification diagram which contains no terminal node.

Figure 4 shows an invariance diagram for the buffer. It is used to prove that the following formula is an invariant of the buffer:

$$\Psi \quad \stackrel{\text{df}}{=} \quad i^\circ = o \frown q \quad \wedge \quad \#r^\circ = c + \#o \quad \wedge \quad \#r^\circ + \#q = \#i^\circ + c$$

To find such an invariant, an understanding of the operation of the STS is necessary. The intended meaning of the variables q and c must be encoded in the formulas:

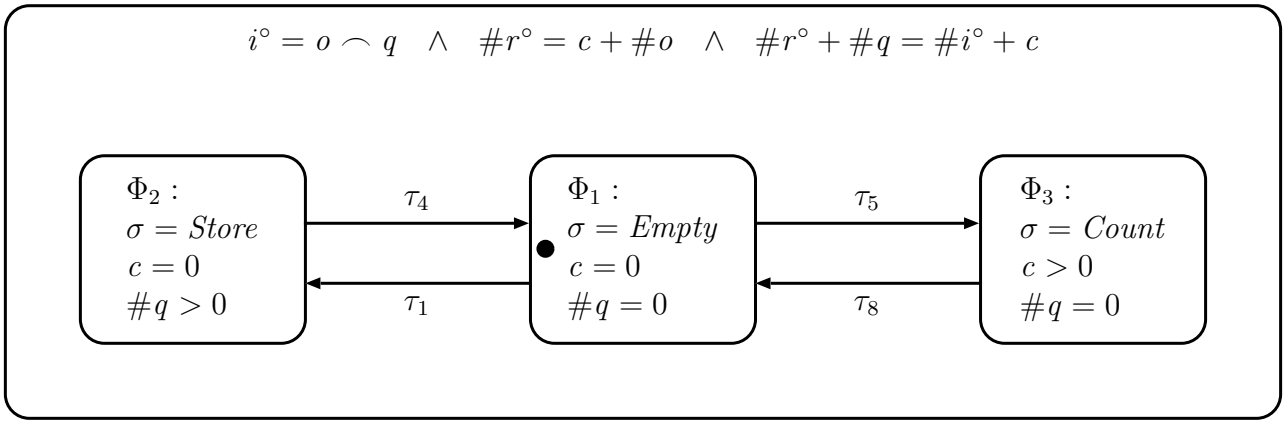


Figure 4: Invariance Diagram for the Buffer

- Messages read from i are either already output on o, or are still stored in q.
- Received requests are either still pending (counted in c) or are already answered (by sending a message on o).
- The difference $\#r^\circ - c$ of the number of received requests and the number of open requests is the number of answered requests, and therefore equal to the number of received messages ($\#i^\circ$) minus the number of messages still buffered ($\#q$).

Thus, the *Buffer* can have messages in q, or it can have can have pending requests, or can be in an balanced state where q is empty and there are no pending requests. These three case are reflected in the three nodes Ψ_1, Ψ_2 and Ψ_3 of our diagram.

All in all, there are 27 proof obligations associated with the diagram. They can be divided into three classes:

- If there is an arrow labeled with a transition τ between two verification diagram nodes, that transition changes the buffer state according to the node assertions. For example,

  $$\Psi_1 \wedge \tau_1 \Rightarrow \Psi_2'$$

- If there are no edges labeled with a transition τ leaving a node, the transition does not invalidate that node's assertion. For example,

  $$\Psi_1 \wedge \tau_2 \Rightarrow \Psi_1'$$

- Finally, the environment transition does not invalidate node assertions. For example,

  $$\Psi_1 \wedge \tau_\epsilon \Rightarrow \Psi_1'$$

From the invariant Ψ we can also deduce properties of the buffer's I/O histories. Note that since $i^\circ \sqsubseteq i$ and $r^\circ \sqsubseteq r$ we have

$$\Psi \Rightarrow \big(o \sqsubseteq i \wedge \#o \leq \#r\big)$$

This means that also

$$\mathit{Buffer} \models \mathbf{inv}\,\big(o \sqsubseteq i \wedge \#o \leq \#r\big)$$

Since the free variables of the invariant are channel variables of the buffer, and since the invariant is admissible, we can conclude that it holds not only in each state of a buffer's execution, but also for the complete I/O history [2]:

$$[\![\mathit{Buffer}]\!] \Rightarrow \big(o \sqsubseteq i \wedge \#o \leq \#r\big)$$

Response Diagrams. A response diagram is a verification diagram that is acyclic: Its nodes can be ordered such that for each pair of nodes Φ_i and Φ_j, if there is an edge from Φ_i to Φ_j, then $i > j$. There is a single node with no outgoing edges. This node is marked as the terminal node and labeled with the assertion Φ_0.

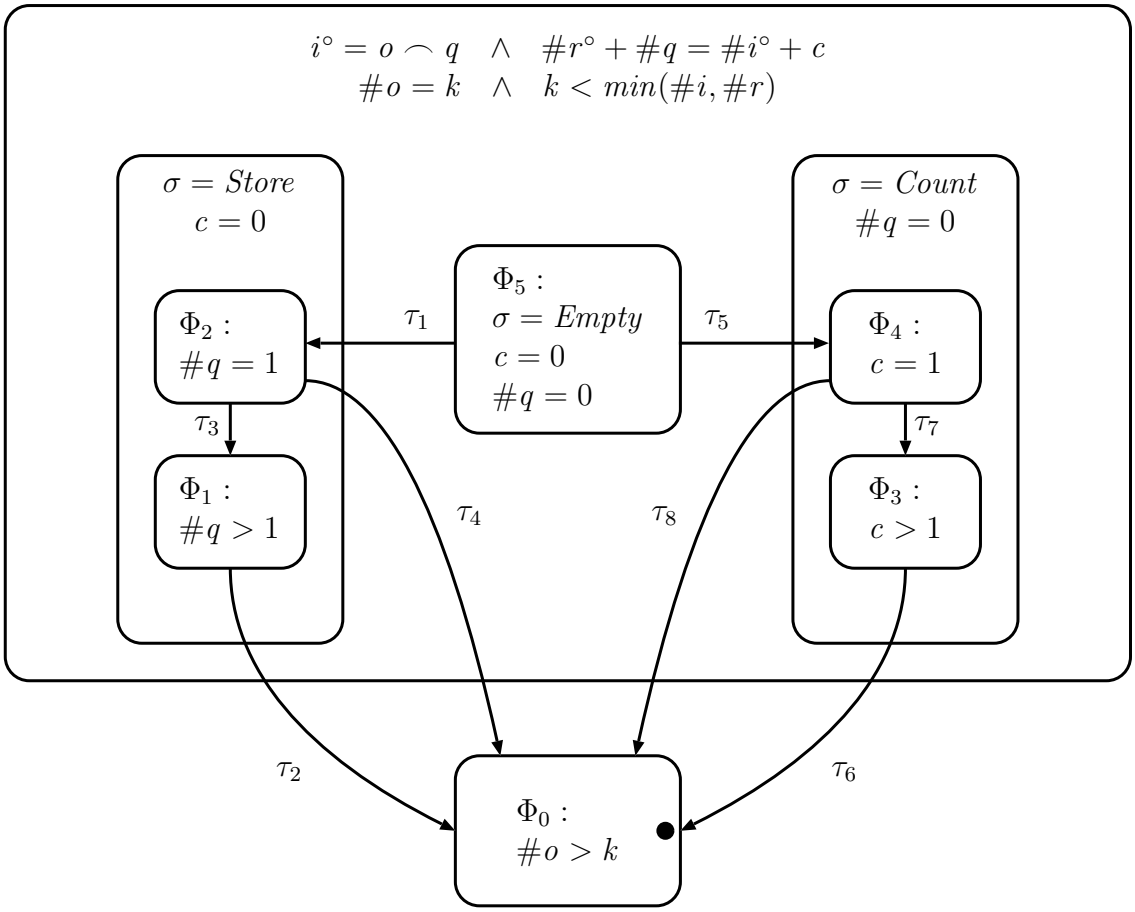


Figure 5: Response Diagram for the Buffer

Figure 5 shows a response diagram for our buffer. It is used to show the following property, which states that the buffer outputs a message on channel o provided there are enough message inputs and requests:

$$\mathit{Buffer} \models \#o = k \wedge k < min(\#i, \#r) \mapsto \#o > k \qquad (\dagger)$$

This property holds immediately for states where only transitions are enabled that produce output (τ_2, τ_4, τ_6 and τ_8). From all other states, the system must move closer to a state where output must be produced. In the verification diagram, the state space is split into five partitions, Φ_1 to Φ_5. The terminal node Φ_0 is the target node, where output on o has been produced. Transitions that send a message on o immediately reach the target node. Other transitions may keep a node assertion valid, or lead to a node closer to the target. Proofs of the enabledness of the transitions depend on the left hand side of the property, which implies that there is input waiting on i or r.

There are 50 proof obligations associated with the response diagrams. They can be divided into four classes:

- From each node, at least one of the departing transitions is enabled. For example,

  $$\Phi_5 \Rightarrow \mathsf{En}(\tau_1) \vee \mathsf{En}(\tau_5)$$

- If there is an arrow labeled with a transition τ between two verification diagram nodes, that transition changes the buffer state according to the node assertions. For example,

  $$\Phi_5 \wedge \tau_1 \Rightarrow \Phi_2'$$

- If there are no edges labeled with a transition τ leaving a node, the transition does not invalidate that node's assertion. For example,

  $$\Phi_5 \wedge \tau_2 \Rightarrow \Phi_5'$$

- Finally, the environment transition does not invalidate the node assertions:

  $$\Phi_1 \wedge \tau_\epsilon \Rightarrow \Phi_1'$$

The last three classes correspond to the invariance diagram verification conditions.

From the diagram we can deduce that the buffer satisfies the following property:

$$\mathit{Buffer} \models \big(\#o = k \wedge k < \mathit{min}(\#i, \#r) \wedge (\Psi_1 \vee \Psi_2 \vee \Psi_3)\big) \mapsto \#o > k$$

where Ψ_1, Ψ_2, Ψ_3 are the predicates from the buffer's invariance diagram (see Figure 4). This implies the property (†) above, since $\Psi_1 \vee \Psi_2 \vee \Psi_3$ is an invariant of the buffer and holds in all reachable states. Finally (†) can be lifted to the I/O history level [2]:

$$[\![\mathit{Buffer}]\!] \Rightarrow \#o \geq \mathit{min}(\#i, \#r)$$

5 Tool Support for Diagrams

Verification tools. While the verification conditions associated with a verification diagram are simple, the size of property proofs is still quite formidable: A verification diagram with n nodes for a system with m transitions requires the verification of about $n \times m$ verification

conditions. Obviously, without tool support property verification is not feasible. We formalized our approach in Isabelle/HOL as an extension of Shankar's PVS formalization of state machines [14] to handle liveness properties and asynchronous communication.

Given an Isabelle formalization of a state transition system, verification conditions are proven as theorems. Verification tactics assemble sets of verification conditions according to the structure of invariance and response diagrams to temporal logic formulas. The derivation of the I/O history properties from the temporal logic properties is not handled within our formalization: This would require the use of the much more complicated logic of computable functions [12].

The Isabelle formalization is documented in [4]. The theory files and proof scripts can be accessed electronically [3]. Introductory texts to Isabelle are also available electronically [8].

Design tools. The translation of the state transition and verification diagrams into an Isabelle theory is quite schematic and straightforward, yet error-prone when done by hand. Clearly, the automatic generation of the theories from a CASE tool or diagram editor is desirable.

In a recent project [15], the CASE tool AutoFocus [6, 7] has been linked with the theorem proving environment VSE II [13]. While that work uses a synchronous execution model instead of the asynchronous model presented here, it can form a basis for the automatic generation of state machine formalizations and verification conditions for Isabelle from system models and verification diagrams.

The main remaining effort then is to prove theorems about the concrete data types used for messages and component state attributes. Most verification conditions can be automatically discharged by Isabelle.

6 Conclusions

State based and I/O history based views of systems can be linked by temporal logical formulas for invariance and response. The simple structure of the temporal logic properties is well-suited for verification diagrams in order to structure property proofs. Preliminary tool support for verification is available through the Isabelle theorem prover.

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.

Compositional reasoning about state machine systems often requires a separation of component properties into assumptions about their input and guarantees about their output in relation to the input. For a component's input/output relation, such assumption/guarantee (A/G) specifications and their logical properties are well known [16]. As future work, we attempt to find temporal logic formula classes that can be used to derive black box A/G specifications, and that have simple proof rules and suggestive verification diagrams.

Finally, our techniques can be adapted to different state-based description techniques. SDL and ROOM, in particular, would be good candidates for a concrete state machine syntax.

References

[1] M. Breitling, U. Hinkel, and K. Spies. Formale Entwicklung verteilter reaktiver Systeme mit Focus. In *Formale Beschreibungstechniken für verteilte Systeme, 8. GI/ITG Fachgespräch*, 1998.

[2] M. Breitling and J. Philipps. Black Box Views of State Machines. Technical Report TUM-I9916, Institut für Informatik, Technische Universität München, 1999.

[3] M. Breitling and J. Philipps. State machine theories and proof scripts for Isabelle/HOL. `http://www4.in.tum.de/~philipps/BBV`, 2000.

[4] M. Breitling and J. Philipps. Verification Diagrams for Dataflow Properties. Technical Report TUM-I0005, Institut für Informatik, Technische Universität München, 2000.

[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, F. Huber, and B. Schätz. AutoFocus – ein Werkzeugprototyp zur Entwicklung eingebetteter Systeme. *Informatik Forschung und Entwicklung*, 14(3):121–134, 1999.

[7] 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.

[8] Isabelle home page. `http://isabelle.in.tum.de`.

[9] Z. Manna and A. Pnueli. Temporal verification diagrams. In *International Symposium on Theoretical Aspects of Computer Software, Lecture Notes in Computer Science 789*, pages 726–765, 1994.

[10] J. Misra. A logic for concurrent programming: Progress. *Journal of Computer and Software Engineering*, 3(2):273–300, 1995.

[11] J. Misra. A logic for concurrent programming: Safety. *Journal of Computer and Software Engineering*, 3(2):239–272, 1995.

[12] L. C. Paulson. *Logic and Computation*. Cambridge University Press, 1987.

[13] G. Rock, W. Stephan, and A. Wolpers. Tool Support for the Compositional Development of Distributed Systems. In *Formale Beschreibungstechniken für verteilte Systeme, 7. GI/ITG Fachgespräch*, 1997.

[14] N. Shankar. A lazy approach to compositional verification. Technical Report CSL-93-08, Computer Science Laboratory, SRI, 1993.

[15] O. Slotosch. Overview over the project Quest. In *FM-Trends '98, LNCS 1641*, 1998. Project home page at `http://www4.in.tum.de/proj/quest`.

[16] K. Stølen, F. Dederichs, and R. Weber. Specification and refinement of networks of asynchronously communicating agents using the assumption/commitment paradigm. *Formal Aspects of Computing*, 8(2), 1995.