TECHNISCHE
UNIVERSITÄT
MÜNCHEN

INSTITUT FÜR INFORMATIK

Sonderforschungsbereich 342:
Methoden und Werkzeuge für die Nutzung
paralleler Rechnerarchitekturen

Verification Diagrams for Dataflow Properties

Max Breitling and Jan Philipps

TUM-I0005
SFB-Bericht Nr. 342/03/00 A
März 00

TUM–INFO–03-I0005-80/1.–FI

Alle Rechte vorbehalten
Nachdruck auch auszugsweise verboten

©2000 SFB 342 Methoden und Werkzeuge für
die Nutzung paralleler Architekturen

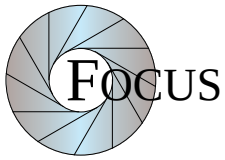
Anforderungen an: Prof. Dr. A. Bode
Sprecher SFB 342
Institut für Informatik
Technische Universität München
D-80290 München, Germany

Druck: Fakultät für Informatik der
Technischen Universität München

Verification Diagrams for Dataflow Properties *

Max Breitling Jan Philipps

Institut für Informatik
Technische Universität München
D-80290 München



www.in.tum.de/~{breitlin|philipps}

Abstract

State-based specification and verification techniques can be used to derive properties of the data flow I/O relation of distributed systems. Safety properties of the I/O relation are typically expressed as a prefix relation on streams; they can be derived from state machine invariants. Liveness properties are typically formulated as a lower bound for the length of output streams; they can be derived from response or leadsto properties of state machines.

While the proof principles for invariance and leadsto properties are well known, proofs for larger systems tend to be rather complex: It is often difficult to get an overview over the complete proof structure, although each single proof step itself is quite simple and usually consists only of the verification of a predicate logic formula. This report shows how verification diagrams can be used to structure the proofs of invariance and leadsto properties. To provide some tool support, the approach is formalized in the theorem prover Isabelle.

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

Contents

1 Introduction

"Once you get into this great stream of history, you can't get out"

Richard Nixon

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 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 data flow components and systems can be concisely formulated as relations over the communication history of components [6, 7]; 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 a related report [1] we show 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 UNITY-like linear temporal logic; history properties are expressed as relations between input and output streams. Still, proofs for larger systems tend to be rather complex; it is often difficult to get an overview over the complete proof structure, although each single proof step itself is quite simple and usually consists only of the verification of a predicate logic formula.

Verification diagrams [4, 13] visualize the proof structure of temporal logic proofs and reduce temporal reasoning to the proof of verification conditions in first-order predicate logic. In this report, we introduce verification diagrams for the invariance and leadsto properties used in the derivation of black box properties for state machines. To help in the book keeping of the verification conditions, we present a formalization of our approach in Isabelle/HOL [17].

This report is structured as follows. In Section 2 we summarize the definitions for state machines and proof principles for the derivation of data flow properties of state machines; we refer to [1, 5] for a more detailed presentation. In Section 3 we introduce verification rules and verification diagrams for invariance and leadsto properties. Section 4 outlines the embedding into the theorem prover Isabelle. Section 5 contains a small example, and Section 6 concludes.

2 State Machines

A system in our framework is a network of components. Each component has input and output ports. The ports are connected by directed channels used for the communication between the components of a system. Component behavior can be specified by state machines that define the reaction of a component depending on the current input and the current state of the machine.

In this section, after repeating some preliminaries in 2.1, we introduce state transition systems and their executions (Section 2.2 and 2.3) as the mathematical basis of state machines. A subset of temporal logic is introduced in Section 2.4 to describe properties of state machine executions. Section 2.5 presents a graphical notation for state machines. Section 2.6 introduces state machine composition, and in Section 2.7 we summarize how state machine properties can be used to derive properties of the I/O history of state machines.

2.1 Streams and Valuations

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 $s \neq \langle\,\rangle$, 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 $s \neq \langle\,\rangle$.

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 $V' \stackrel{\mathsf{df}}{=} \{\ v' \mid v \in V\ \}$ and to valuations: Given a valuation α of variables in Var, α' is a valuation of variables in V' with $\alpha'(v') = \alpha(v)$ 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 $\alpha \overset{V}{=} \beta$.

2.2 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 $V \overset{\text{df}}{=} I \cup O \cup A$. $\mathcal{I}$ 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 $i^\circ \sqsubseteq i$ holds in all reachable states. We define i^+ indirectly as the part of the message history that has not yet been processed by requiring $i = i^\circ \frown i^+$.

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

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

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 allows the environment to extend the input, while it leaves the controlled

variables $v \in O \cup A$ unchanged:

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

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

Example. As an example, we consider a simple buffer: The buffer has two input channels i and r, and one output channel o (Figure 1). 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.

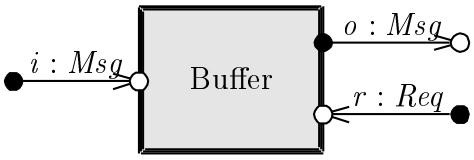


Figure 1: Simple Buffer

As local attributes we choose (besides to the variables i° and r° and σ for the control state) an integer variable c to count pending request, and a sequence variable q to store the sequence of messages stored in the buffer: When the buffer receives a message, we append it at the end of q. If we receive a request, we output the first element of q, and remove it from q. If there are no messages in q that can be sent, we count the pending request by incrementing c. If we receive some message later, we can immediately forward it and decrement c in this case. Obviously, the initial values are $c = 0$ and $q = \langle\,\rangle$. Thus, we have the following variable sets:

$$I = \{i, r\} \qquad O = \{o\} \qquad A = \{i^\circ, r^\circ, \sigma, c, q\}$$

The initial condition $\mathcal{I}$ is formalized as an example for a state predicate in Section 2.4. A convenient way to describe $\mathcal{T}$ of a STS is by state transition diagrams, so we describe the detailed behavior of the buffer in Section 2.5.

2.3 Executions

An *execution* of an 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 two 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 $\mathcal{S}$.

2.4 Predicates and Properties

State machine properties are expressed using assertions that relate communication histories and the values of the attribute variables.

A *state predicate* of a state machine $\mathcal{S} = (I, O, A, \mathcal{I}, \mathcal{T})$ is an assertion Φ where the free variables range over the variables in $V = I \cup O \cup A$.

An example for a state predicate is the initialization assertion $\mathcal{I}$ of the state machine *Buffer* (Figure 1):

$$\sigma = \mathsf{Empty} \wedge q = \langle\,\rangle \wedge c = 0 \wedge i^+ = i \wedge i^\circ = \langle\,\rangle \wedge r^+ = r \wedge r^\circ = \langle\,\rangle \wedge o = \langle\,\rangle$$

State predicates relate the communication histories and state variables only at a given point in a system execution. To express properties about the complete execution, predicates are lifted to executions by one of the following two operators:

- **initially** Φ holds for a state machine $\mathcal{S}$ and a state predicate Φ, iff Φ is true under the variable valuation of the first time point of each system run:

  $$\forall\, \xi \in \langle\!\langle \mathcal{S} \rangle\!\rangle \bullet \xi.1 \models \Phi$$

  This is denoted by $\mathcal{S} \models$ **initially** Φ. It holds if the characterization of the initial states imply Φ, i.e. if $\mathcal{I} \Rightarrow \Phi$ is valid.

- Φ **co** Ψ holds for a state machine $\mathcal{S}$ and state predicates Φ and Ψ (Φ *constrains* Ψ), iff whenever Φ evaluates to true at a point in a system execution, then so does Ψ at the subsequent point:

  $$\forall\, \xi \in \langle\!\langle \mathcal{S} \rangle\!\rangle \bullet \forall\, k \bullet \big(\xi.k \models \Phi \Rightarrow \xi.(k+1) \models \Psi\big)$$

  This is denoted by $\mathcal{S} \models \Phi$ **co** Ψ. The operator **co** is defined to have a weaker binding than all other logical operators.

We also use the following abbreviations:

$$\mathcal{S} \models \textbf{stable}\ \Phi \stackrel{\text{df}}{\Leftrightarrow} \mathcal{S} \models \Phi\ \textbf{co}\ \Phi$$

$$\mathcal{S} \models \textbf{inv}\ \Phi \quad \stackrel{\text{df}}{\Leftrightarrow} \mathcal{S} \models \textbf{stable}\ \Phi \text{ and } \mathcal{S} \models \textbf{initially}\ \Phi$$

Informally, a predicate is stable if its validity is preserved by all transitions of a system, and we call it an invariant, if it holds in all reachable states.

Progress is expressed by the leadsto operator $\mapsto$. Intuitively, $\Phi \mapsto \Psi$ means that whenever in a state machine execution a state is reached where Φ holds, at the same or at a later point in the execution a state is reached where Ψ holds.

The semantic definition of $\mathcal{S} \models \Phi \mapsto \Psi$ is as follows. For all $\xi \in \langle\!\langle \mathcal{S} \rangle\!\rangle$,

$$\forall\, k \bullet (\xi.k \models \Phi) \Rightarrow (\exists\, l \geq k \bullet \xi.l \models \Psi)$$

From the semantic definition it follows immediately that $\mapsto$ is transitive, and that whenever $\Phi \Rightarrow \Psi$, then also $\Phi \mapsto \Psi$.

2.5 State Transition Diagrams

Typically, an STS is not specified by defining formally all elements of the quintuple, but by a *state transition diagram* (STD). 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, a transition (with the name *name*) is labeled using the following schema:

$$name :: \{Precondition\}\ Inputs \rhd Outputs\ \{Postcondition\}$$

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 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.

Example The behavior of the buffer is specified by the STD in Figure 2. We start in the state *Empty*. If we receive some data on i, we store this data in q, and move to the state *Store*. In this state, receiving a request, we output the first element of q, and stay in this state or move back to *Empty*, depending on the length of q. If we receive further data in the state *Store*, we append these in q. If there are no stored messages (in the state *Empty*), but a request arrives, we have to remember this open request, and do this by incrementing c. If $c > 0$, we are in state *Count*. If we get some data now, we immediately forward it on o, and decrement c, until there are no more pending requests and we return to the state *Empty*.

The buffer can also be specified differently, with fewer control states or fewer transitions. We chose this specification since it leads to verifications diagrams that are more interesting but still not too difficult.

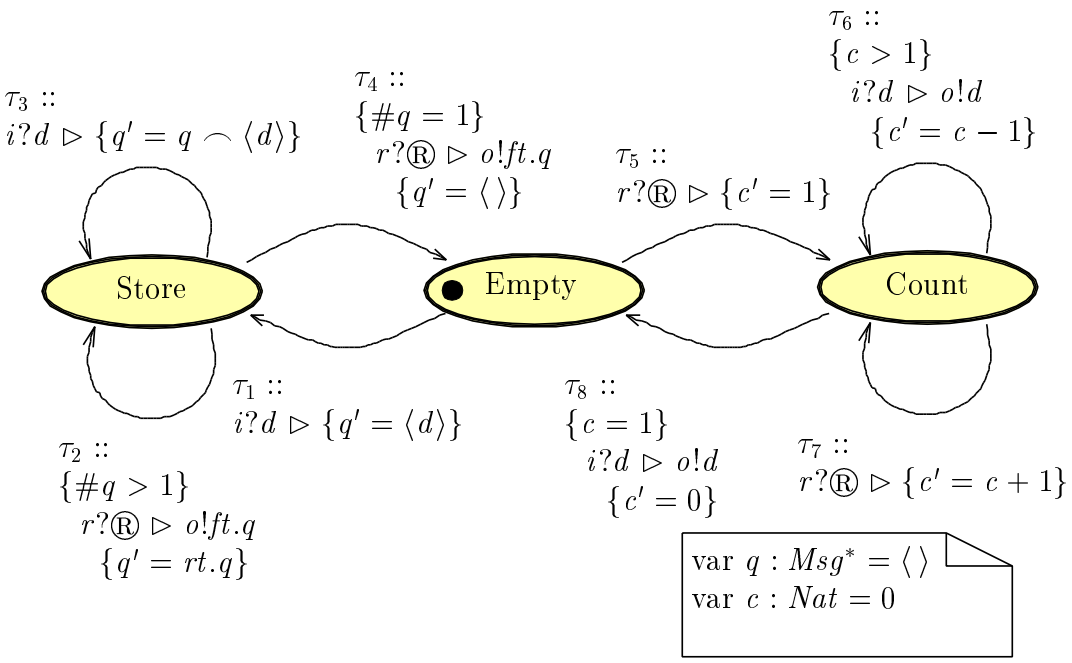


Figure 2: STD for the Buffer

2.6 Composition of State Machines

Two state machines can be composed if they are compatible: The controlled variables must be disjoint, and no machine may read the internals of the other. Both components can interact since one component may read the output of the other. A transition of the composed system consists of the conjunction of a transition of one component with the environment transition of the other. The composed components operate in an interleaved manner.

The formal definition of this interleaving composition is in [1], where it is also shown that a composed system inherits the invariance and progress properties of its components.

2.7 Black Box Views of State Machines

An STS $\mathcal{S}$ describes operationally how component or system behavior is realized step by step. But sometimes a more abstract black box view of component behavior is desirable. It models a component as a relation over its possible input and output communication histories. This relation is denoted by $[\![\mathcal{S}]\!]$; it is described by predicates, where all free variables belong to $I \cup O$.

The black box specification of the buffer is as follows:

$$[\![\mathit{Buffer}]\!] \stackrel{\text{df}}{\Leftrightarrow} o \sqsubseteq i \wedge \#o \leq \#r \wedge \#o \geq \mathit{min}(\#i, \#r)$$

Note that the formula only refers to the input and output message streams, but not to the internal attributes σ, q or c, nor to the internal history variables i°, r° representing the already processed input.

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

The first two conjuncts are safety properties: They restrict the messages on the output channel o, as well as the number of messages transmitted over o. They hold even if the buffer produces no output at all. The third conjunct is a liveness property. It gives a lower bound for the length of the output; thus, the buffer must produce output.

Black box views for an STS $\mathcal{S}$ can be derived systematically from temporal logic properties. When Φ is an admissible predicate with free variables from $I \cup O$, it is sufficient to show that Φ is an invariant:

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

This technique is used for the safety part of a black box specification. The liveness part is typically expressed as inequalities of the form $\#u \geq f(v_1, \ldots, v_n)$ for an output channel u, input channels $v_1, \ldots, v_n$ and a function f from the input channel histories to $\mathbb{N}$; the function f is assumed to be monotonic. Such inequalities can be shown by proving leadsto properties for the state machine (where $k \in \mathbb{N}$ is a constant distinct from all channel names):

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

In this paper, we do not treat this topic further, but concentrate on state machines and their properties together with proof techniques. The transition from state machines to black box views is formally treated in [1, 3]: The black box view of a STS is defined as a valuation for $I \cup O$ that coincides with the least upper bounds of the valuations in an execution of the STS.

Due to the asynchronous message passing between components in a composed system, it is possible to describe a systems black box behavior as the conjunction of the components

black box properties. Frequently, the combination of the black box properties is easier to understand than the composition of the underlying state machines, which leads to a complex state machine with a large state space. As a methodological conclusion for building systems from components, we suggest to handle the composition on the more abstract level of black box views, and analyze the behaviors of single components separately using the techniques of this paper.

3 Verification Diagrams

Black box properties of a component are derived from invariance and leadsto properties of the component state machine. In order to prove invariance and leadsto properties for state machines, one can use a number of verification rules. These rules reduce properties to simpler properties, and finally to a number of verification conditions in predicate logic. 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 have been introduced as a graphical means to help in the representation of property proofs [13, 12, 4]. They can easily be adapted to our framework.

In this section, we first present some verification rules for the constrains and leadsto operators. In Section 3.2 we introduce verification diagrams for the graphical proof outline. Sections 3.3 and 3.4 present specific diagrams for invariance and leadsto properties, proof obligations and examples for each diagram class.

3.1 Verification Rules

Figures 3 and 4 contain typical verification rules for constrains and leadsto properties. They correspond to the UNITY verification rules of [15, 14]. In our framework, **co** and $\mapsto$ are defined over state machine executions, while in [15, 14] they are defined over the transition relation of a state machine. We refer to [1] for a more detailed discussion and a justification of the rules.

3.2 Verification Diagrams

A verification diagram is always used in the context of a state transition system $\mathcal{S} = (I, O, A, \mathcal{I}, \mathcal{T})$. Similar to state transition diagrams, a verification diagram is a directed graph. Verification diagrams may not contain 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 $\tau \in \mathcal{T}$. A transition from a node Φ_a to a node Φ_b labeled with a list of transitions $\tau_k, \ldots, \tau_m$ are a shorthand for a group of transitions between Φ_a and Φ_b labeled with τ_k to τ_m.

With each verification diagram we associate a set of verification conditions for the STS $\mathcal{S}$; if all verification conditions are valid, we say that the diagram is valid. Depending on the structure of the diagram and the associated verification conditions, a valid diagram implies either a constrains or a leadsto property.

$$\frac{\mathcal{I} \Rightarrow \Phi}{\mathcal{S} \models \mathbf{initially}\, \Phi}$$

(a) Initiality

$$\frac{\Phi \wedge \tau_\epsilon \Rightarrow \Psi' \quad \Phi \wedge \tau \Rightarrow \Psi' \quad \text{for all } \tau \in \mathcal{T}}{\mathcal{S} \models \Phi \ \mathbf{co}\ \Psi}$$

(b) Consecution

$$\frac{\mathcal{S} \models \Phi_1 \ \mathbf{co}\ \Psi_1 \quad \mathcal{S} \models \Phi_2 \ \mathbf{co}\ \Psi_2}{\mathcal{S} \models \Phi_1 \wedge \Phi_2 \ \mathbf{co}\ \Psi_1 \wedge \Psi_2 \quad \mathcal{S} \models \Phi_1 \vee \Phi_2 \ \mathbf{co}\ \Psi_1 \vee \Psi_2}$$

(c) Conjunction and Disjunction

$$\frac{\mathcal{S} \models \Phi \ \mathbf{co}\ \Psi \quad \mathcal{S} \models \Psi \ \mathbf{co}\ \chi}{\mathcal{S} \models \Phi \ \mathbf{co}\ \chi}$$

(d) Transitivity

$$\frac{\mathcal{S} \models \Phi \ \mathbf{co}\ \Psi}{\mathcal{S} \models \Phi \wedge \chi \ \mathbf{co}\ \Psi}$$

(e) LHS Strengthening

$$\frac{\mathcal{S} \models \Phi \ \mathbf{co}\ \Psi}{\mathcal{S} \models \Phi \ \mathbf{co}\ \Psi \vee \chi}$$

(f) RHS Weakening

$$\frac{\mathcal{S} \models \mathbf{inv}\, \chi \quad \mathcal{S} \models \Phi \wedge \chi \ \mathbf{co}\ \Psi}{\mathcal{S} \models \Phi \ \mathbf{co}\ \Psi}$$

(g) LHS Invariant Elimination

$$\frac{\mathcal{S} \models \mathbf{inv}\, \chi \quad \mathcal{S} \models \Phi \ \mathbf{co}\ \Psi}{\mathcal{S} \models \Phi \ \mathbf{co}\ \Psi \wedge \chi}$$

(h) RHS Invariant Introduction

Figure 3: Verification rules for **co**

$$\frac{\begin{array}{l}\mathcal{S} \models \Phi \textbf{ co } \Phi \vee \Psi \\ \text{For a transition } \tau \in \mathcal{T}\text{:} \\ \qquad \Phi \Rightarrow \mathsf{En}(\tau) \\ \qquad\quad \text{and} \\ \qquad \Phi \wedge \tau \Rightarrow \Psi'\end{array}}{\mathcal{S} \models \Phi \mapsto \Psi}$$

(a) Ensure

$$\frac{\mathcal{S} \models \Phi \mapsto \Psi \qquad \mathcal{S} \models \Psi \mapsto \chi}{\mathcal{S} \models \Phi \mapsto \chi}$$

(b) Transitivity

$$\frac{\mathcal{S} \models \Phi(x) \mapsto \Psi \quad \text{for all } x \in X}{\mathcal{S} \models \big(\exists\, x \in X \bullet \Phi(x)\big) \mapsto \Psi}$$

(c) Disjunction

$$\frac{\Phi \Rightarrow \Psi}{\mathcal{S} \models \Phi \mapsto \Psi}$$

(d) Implication

$$\frac{\mathcal{S} \models \Phi \mapsto \Psi}{\mathcal{S} \models \Phi \wedge \chi \mapsto \Psi}$$

(e) LHS Strengthening

$$\frac{\mathcal{S} \models \Phi \mapsto \Psi}{\mathcal{S} \models \Phi \mapsto \Psi \vee \chi}$$

(f) RHS Weakening

$$\frac{\mathcal{S} \models \textbf{inv}\, \chi \qquad \mathcal{S} \models \Phi \wedge \chi \mapsto \Psi}{\mathcal{S} \models \Phi \mapsto \Psi}$$

(g) LHS Invariant Elimination

$$\frac{\mathcal{S} \models \textbf{inv}\, \chi \qquad \mathcal{S} \models \Phi \mapsto \Psi}{\mathcal{S} \models \Phi \mapsto \Psi \wedge \chi}$$

(h) RHS Invariant Introduction

Figure 4: Verification rules for $\mapsto$

Verification diagrams can be hierarchical: A node can contain a sub-diagram. Hierarchical diagrams can be flattened, so that assertions from a node higher in the hierarchy are conjoined with the assertion of the nodes below it; arrows entering or exiting higher-level nodes are connected to all lower-level nodes. Figure 5 shows the hierarchical and flattened version of a part of a verification diagram.

Since all hierarchical verification diagrams are equivalent to a flattened diagram, we discuss only flattened diagrams in the next section.

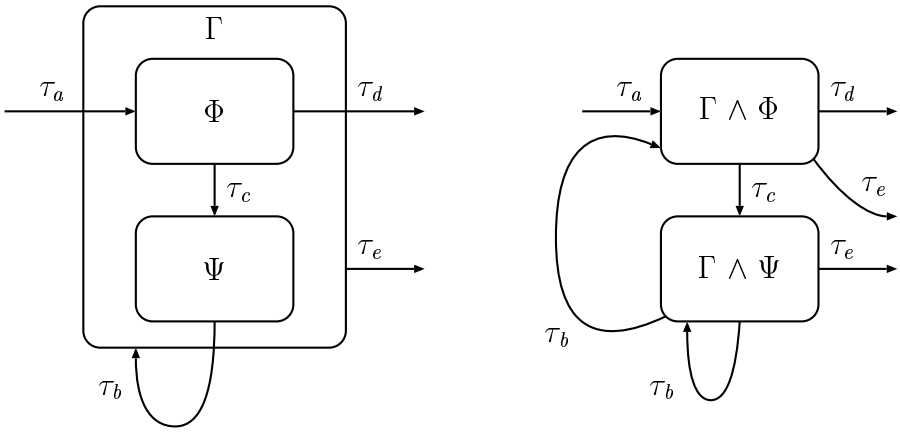


Figure 5: Hierarchical and Flattened Verification Diagrams

3.3 Invariance Diagrams

A invariance diagram is a verification diagram which contains no terminal node.

Verification conditions. The following verification conditions are associated with each node Φ of an invariance diagram:

- For each transition $\tau \in \mathcal{T}$ with τ-labeled edges leaving Φ and entering nodes $\Phi_k, \ldots, \Phi_m$:

$$\Phi \wedge \tau \Rightarrow \Phi'_k \vee \ldots \vee \Phi'_m$$

- For each transition $\tau \in \mathcal{T}$ with no τ-labeled edge leaving Φ:

$$\Phi \wedge \tau \Rightarrow \Phi'$$

- Finally, for the environment transition τ_ϵ:

$$\Phi \wedge \tau_\epsilon \Rightarrow \Phi'$$

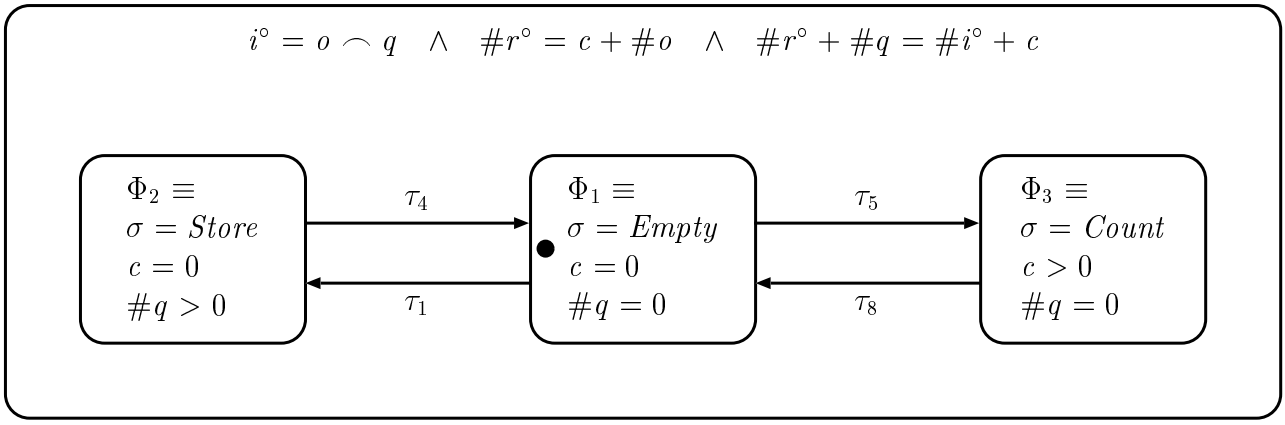


Figure 6: Invariance Diagram for the Buffer

The three classes of verification conditions cover for each node the complete set of transitions $\mathcal{T}$ as well as the environment transition τ_ϵ of $\mathcal{S}$. The right hand side of each implication can be weakened to include all node labels $\Phi_0, \ldots, \Phi_n$. Thus, the validity of an invariance diagram implies for the each assertion Φ:

$$\mathcal{S} \models \Phi \textbf{ co } \Phi_0 \vee \ldots \vee \Phi_n$$

Using the disjunction rule for **co**, this means that a valid invariance diagram implies the following property:

$$\mathcal{S} \models \left(\bigvee_{0 \le i \le n} \Phi_i \right) \textbf{ co } \left(\bigvee_{0 \le i \le n} \Phi_i \right)$$

If, in addition

$$\mathcal{S} \models \textbf{initially} \left(\bigvee_{0 \le i \le n} \Phi_i \right)$$

then

$$\mathcal{S} \models \textbf{inv} \left(\bigvee_{0 \le i \le n} \Phi_i \right)$$

Frequently, the initiality assertion of the state transition system is sufficiently strong that there is a subset $N \subseteq \{0, \ldots, n\}$ with

$$\mathcal{S} \models \textbf{initially} \left(\bigvee_{i \in N} \Phi_i \right)$$

In this case, the nodes from N can be marked as initial nodes to further clarify the proof structure.

Example. Figure 6 shows an invariance diagram for the buffer. The following formula is an invariant:

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

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:

- 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 three nodes in our diagram:

$$\begin{aligned}
\Psi_1 &\stackrel{\text{df}}{=} \Psi \ \wedge \ \sigma = \mathit{Empty} \wedge c = 0 \wedge \#q = 0 \\
\Psi_2 &\stackrel{\text{df}}{=} \Psi \ \wedge \ \sigma = \mathit{Store} \wedge c = 0 \wedge \#q > 0 \\
\Psi_3 &\stackrel{\text{df}}{=} \Psi \ \wedge \ \sigma = \mathit{Count} \wedge c > 0 \wedge \#q = 0
\end{aligned}$$

All in all, there are 27 proof obligations associated with the diagram:

- All transitions between nodes are correct:

$$\begin{aligned}
&\Psi_1 \wedge \tau_1 \Rightarrow \Psi_2' \\
&\Psi_1 \wedge \tau_5 \Rightarrow \Psi_3' \\
&\Psi_2 \wedge \tau_4 \Rightarrow \Psi_1' \\
&\Psi_3 \wedge \tau_8 \Rightarrow \Psi_1'
\end{aligned}$$

- If there are no edges leaving a node, the corresponding transitions do not invalidate that node's assertion:

$$\begin{array}{lll}
\Psi_1 \wedge \tau_2 \Rightarrow \Psi_1' & \Psi_2 \wedge \tau_1 \Rightarrow \Psi_2' & \Psi_3 \wedge \tau_1 \Rightarrow \Psi_3' \\
\Psi_1 \wedge \tau_3 \Rightarrow \Psi_1' & \Psi_2 \wedge \tau_2 \Rightarrow \Psi_2' & \Psi_3 \wedge \tau_2 \Rightarrow \Psi_3' \\
\Psi_1 \wedge \tau_4 \Rightarrow \Psi_1' & \Psi_2 \wedge \tau_3 \Rightarrow \Psi_2' & \Psi_3 \wedge \tau_3 \Rightarrow \Psi_3' \\
\Psi_1 \wedge \tau_6 \Rightarrow \Psi_1' & \Psi_2 \wedge \tau_5 \Rightarrow \Psi_2' & \Psi_3 \wedge \tau_4 \Rightarrow \Psi_3' \\
\Psi_1 \wedge \tau_7 \Rightarrow \Psi_1' & \Psi_2 \wedge \tau_6 \Rightarrow \Psi_2' & \Psi_3 \wedge \tau_5 \Rightarrow \Psi_3' \\
\Psi_1 \wedge \tau_8 \Rightarrow \Psi_1' & \Psi_2 \wedge \tau_7 \Rightarrow \Psi_2' & \Psi_3 \wedge \tau_6 \Rightarrow \Psi_3' \\
 & \Psi_2 \wedge \tau_8 \Rightarrow \Psi_2' & \Psi_3 \wedge \tau_7 \Rightarrow \Psi_3'
\end{array}$$

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

$$\begin{aligned}
&\Psi_1 \wedge \tau_\epsilon \Rightarrow \Psi_1' \\
&\Psi_2 \wedge \tau_\epsilon \Rightarrow \Psi_2' \\
&\Psi_3 \wedge \tau_\epsilon \Rightarrow \Psi_3'
\end{aligned}$$

The invariance diagrams is shown to be valid in Section 4.3. Moreover, the initialization predicate of the buffer implies Ψ_1, i.e.

$$\mathit{Buffer} \models \mathbf{initially}(\Psi_1 \vee \Psi_2 \vee \Psi_3)$$

Thus, the disjunction of the node assertion is an invariant:

$$\mathit{Buffer} \models \mathbf{inv}(\Psi_1 \vee \Psi_2 \vee \Psi_3)$$

and, since $\Psi_1 \vee \Psi_2 \vee \Psi_3 \Rightarrow \Psi$, also

$$\mathit{Buffer} \models \mathbf{inv}\,\Psi$$

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 $\Psi \Rightarrow \chi$ with

$$\chi \stackrel{\text{df}}{\Leftrightarrow} o \sqsubseteq i \wedge \#o \leq \#r$$

This means that also

$$\mathit{Buffer} \models \mathbf{inv}\,\chi$$

Since the free variables of χ are channel variables of the buffer, and since χ is admissible, we can conclude (see Section 2.7) that it holds not only in each state of a buffer's execution, but also for the complete I/O history:

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

3.4 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.

Verification conditions. No verification conditions are associated with the terminal node Φ_0 of the diagram. For each non-terminal nodes Φ_i, with $i > 0$, the following verification conditions are associated:

- Let $\tau_k, \ldots \tau_m$ be the labels of the edges leaving Φ_i. Then at least one of the corresponding transitions must be enabled in a state where Φ_i holds:

  $$\Phi_i \Rightarrow \mathsf{En}(\tau_k) \vee \ldots \vee \mathsf{En}(\tau_m)$$

  Note that since $i > 0$ there is at least one outgoing edge from Φ_i.

- For each transition $\tau \in \mathcal{T}$ with τ-labeled edges leaving Φ_i and entering nodes $\Phi_k, \ldots, \Phi_m$:

  $$\Phi_i \wedge \tau \Rightarrow \Phi'_k \vee \ldots \vee \Phi'_m$$

  Since the diagram is acyclic, the node indices $k, \ldots, m$ are less than i.

- For each transition $\tau \in \mathcal{T}$ with no τ-labeled edge leaving Φ_i:

  $$\Phi_i \wedge \tau \Rightarrow \Phi'_i$$

- Finally, for the environment transition τ_ϵ:

  $$\Phi_i \wedge \tau_\epsilon \Rightarrow \Phi'_i$$

The verification conditions cover for each node the complete set of transitions $\mathcal{T}$ as well as the environment transition τ_ϵ of $\mathcal{S}$. Thus, the following constrains property can be shown to hold for each node Φ_i, $i > 0$, where $\Phi_k, \ldots, \Phi_m$ are the nodes reachable from Φ_i by one edge $(k, \ldots, m < i)$:

$$\mathcal{S} \models \Phi_i \textbf{ co } \Phi_i \vee \Phi_k \vee \ldots \vee \Phi_m$$

Using the ensure rule, this property together with the first two verification conditions implies:

$$\mathcal{S} \models \Phi_i \mapsto \Phi_k \vee \ldots \vee \Phi_m$$

and thus, by weakening of the right hand side,

$$\mathcal{S} \models \Phi_i \mapsto \left(\bigvee_{j<i} \Phi_j \right) \qquad (\dagger)$$

By induction we now show that for all $i > 0$:

$$\mathcal{S} \models \Phi_i \mapsto \Phi_0$$

- For $i = 1$, the property above immediately implies

  $$\mathcal{S} \models \Phi_1 \mapsto \Phi_0$$

- For a node Φ_i with $i > 1$, we know from the induction hypothesis that for all $j < i$,

  $$\mathcal{S} \models \Phi_j \mapsto \Phi_0$$

  By the disjunction rule,

  $$\mathcal{S} \models \left(\bigvee_{j<i} \Phi_j \right) \mapsto \Phi_0$$

  and thus by transitivity with $(\dagger)$,

  $$\mathcal{S} \models \Phi_i \mapsto \Phi_0$$

By the disjunction rule of $\mapsto$, this implies:

$$\mathcal{S} \models \left(\bigvee_{0 \leq i \leq n} \Phi_i \right) \mapsto \Phi_0$$

For two properties Φ and Ψ with

$$\Phi \Rightarrow \left(\bigvee_{0 \leq i \leq n} \Phi_i \right) \quad \text{and} \quad \Phi_0 \Rightarrow \Psi$$

the verification diagram then implies

$$\mathcal{S} \models \Phi \mapsto \Psi$$

because of the weakening and strengthening rules for $\mapsto$.

Example. Figure 7 shows the response diagram for our example of the simple buffer needed to show the following property, which states that the buffer outputs a message on channel o provided there are enough message inputs and requests:

$$\#o = k \wedge k < min(\#i, \#r) \mapsto \#o > k$$

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.

For example, initially the buffer would be in a state that satisfies Φ_5. Now, since $\#o < min(\#i, \#r)$, both transition τ_1 (which reads a message from i and stores it in the queue) and transition τ_5 (which increments the number of pending requests) are enabled. Assume the buffer executes τ_1; it then moves into a state that satisfies Φ_2. Executing τ_4 from this state would read a request from r and output the (only) message in the queue. Executing τ_3, on the other hand, moves the system into a state that satisfies Φ_1. In such a state, we have two choices: Either reading more input from i into the queue (repeating τ_3, again ending in a state that satisfies Φ_1), or answering a request on r with output on o (by τ_2). Note that in a state that satisfies Φ_1 (and the assertions higher in the node hierarchy), we have

$$\#r^\circ + \#q = \#i^\circ = \#(o \frown q) = \#o + \#q$$

and therefore $\#r^\circ = \#o$. Because $\#o = k < min(\#i, \#r) \leq \#r$ we have $\#r^\circ < \#r$, which implies that τ_2 is enabled. Because of the fairness assumption of state transition systems, the transition τ_2 has to be executed at some point of the execution: The buffer produces output on o.

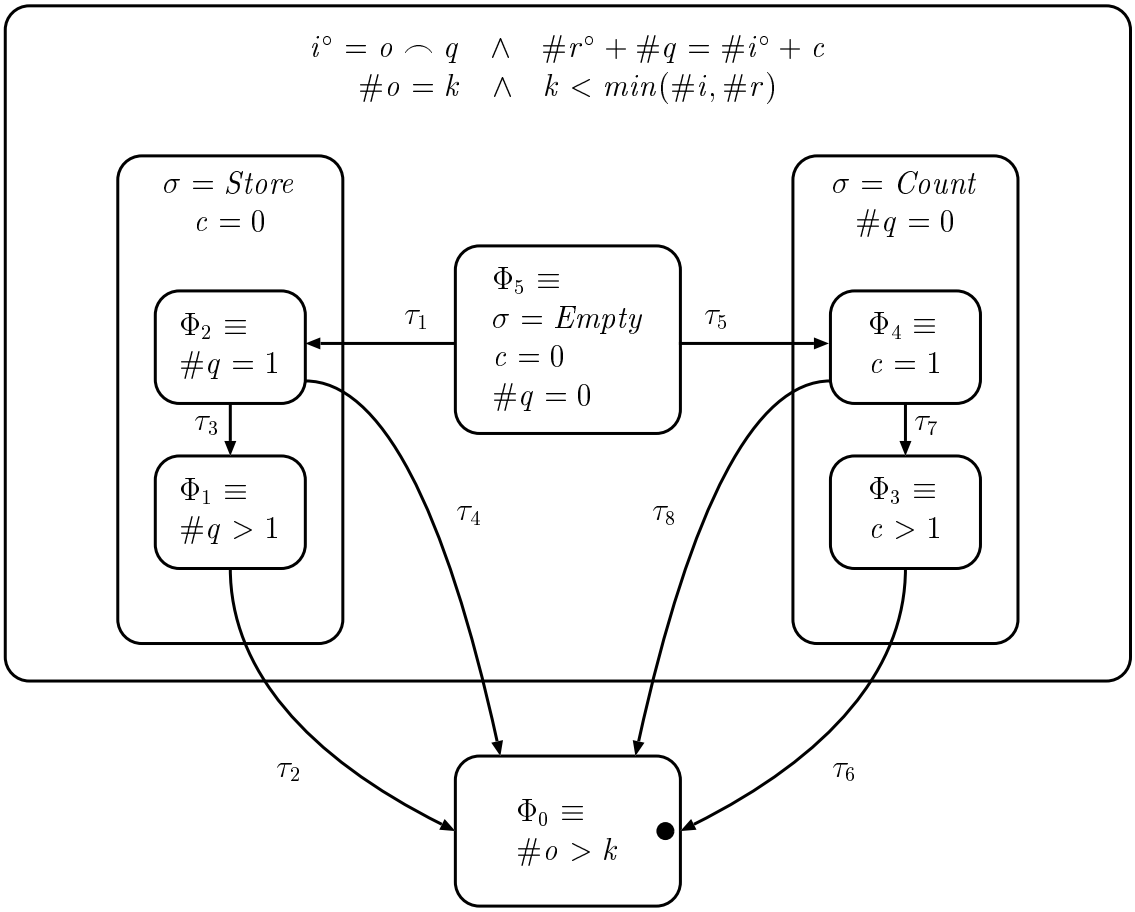


Figure 7: Response Diagram for the Buffer

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

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

where Ψ_1, Ψ_2, Ψ_3 are the predicates from the buffer's invariance diagram (see Figure 6).

Note that since $\Psi_1 \vee \Psi_2 \vee \Psi_3$ is an invariant of the buffer, we can use the invariance elimination rule (Figure 4(g)) to derive

$$\#o = k \wedge k < min(\#i, \#r) \mapsto \#o > k$$

Like the invariant of the previous section, this property also tells us something about the buffer's I/O behavior, namely that

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

holds for the buffer's black box view (see Section 2.7 and [1] for details).

Finding response diagrams is not easy: One needs an operational understanding of the system. Nevertheless, we think that these diagrams are quite intuitive.

The following 50 proof obligations are associated with the diagram:

- From each node, at least one of the departing transitions is enabled:

$$\Phi_5 \Rightarrow \mathsf{En}(\tau_1) \vee \mathsf{En}(\tau_5)$$
$$\Phi_4 \Rightarrow \mathsf{En}(\tau_7) \vee \mathsf{En}(\tau_8)$$
$$\Phi_3 \Rightarrow \mathsf{En}(\tau_6)$$
$$\Phi_2 \Rightarrow \mathsf{En}(\tau_3) \vee \mathsf{En}(\tau_4)$$
$$\Phi_1 \Rightarrow \mathsf{En}(\tau_2)$$

- The transitions between two nodes are correct:

$$\Phi_5 \wedge \tau_1 \Rightarrow \Phi'_2 \qquad \Phi_3 \wedge \tau_6 \Rightarrow \Phi'_0$$
$$\Phi_5 \wedge \tau_5 \Rightarrow \Phi'_4 \qquad \Phi_2 \wedge \tau_3 \Rightarrow \Phi'_1$$
$$\Phi_4 \wedge \tau_7 \Rightarrow \Phi'_3 \qquad \Phi_2 \wedge \tau_4 \Rightarrow \Phi'_0$$
$$\Phi_4 \wedge \tau_8 \Rightarrow \Phi'_0 \qquad \Phi_1 \wedge \tau_2 \Rightarrow \Phi'_0$$

- For each transition $\tau \in \mathcal{T}$ with no τ-labeled edge leaving a node Φ_i, the node assertion remains valid:

$$\Phi_5 \wedge \tau_2 \Rightarrow \Phi'_5 \quad \Phi_4 \wedge \tau_1 \Rightarrow \Phi'_4 \quad \Phi_2 \wedge \tau_1 \Rightarrow \Phi'_2$$
$$\Phi_5 \wedge \tau_3 \Rightarrow \Phi'_5 \quad \Phi_4 \wedge \tau_2 \Rightarrow \Phi'_4 \quad \Phi_2 \wedge \tau_2 \Rightarrow \Phi'_2$$
$$\Phi_5 \wedge \tau_4 \Rightarrow \Phi'_5 \quad \Phi_4 \wedge \tau_3 \Rightarrow \Phi'_4 \quad \Phi_2 \wedge \tau_5 \Rightarrow \Phi'_2$$
$$\Phi_5 \wedge \tau_6 \Rightarrow \Phi'_5 \quad \Phi_4 \wedge \tau_4 \Rightarrow \Phi'_4 \quad \Phi_2 \wedge \tau_6 \Rightarrow \Phi'_2$$
$$\Phi_5 \wedge \tau_7 \Rightarrow \Phi'_5 \quad \Phi_4 \wedge \tau_5 \Rightarrow \Phi'_4 \quad \Phi_2 \wedge \tau_7 \Rightarrow \Phi'_2$$
$$\Phi_5 \wedge \tau_8 \Rightarrow \Phi'_5 \quad \Phi_4 \wedge \tau_6 \Rightarrow \Phi'_4 \quad \Phi_2 \wedge \tau_8 \Rightarrow \Phi'_2$$

$$\Phi_3 \wedge \tau_1 \Rightarrow \Phi'_3 \qquad \Phi_1 \wedge \tau_1 \Rightarrow \Phi'_1$$
$$\Phi_3 \wedge \tau_2 \Rightarrow \Phi'_3 \qquad \Phi_1 \wedge \tau_3 \Rightarrow \Phi'_1$$
$$\Phi_3 \wedge \tau_3 \Rightarrow \Phi'_3 \qquad \Phi_1 \wedge \tau_4 \Rightarrow \Phi'_1$$
$$\Phi_3 \wedge \tau_4 \Rightarrow \Phi'_3 \qquad \Phi_1 \wedge \tau_5 \Rightarrow \Phi'_1$$
$$\Phi_3 \wedge \tau_5 \Rightarrow \Phi'_3 \qquad \Phi_1 \wedge \tau_6 \Rightarrow \Phi'_1$$
$$\Phi_3 \wedge \tau_7 \Rightarrow \Phi'_3 \qquad \Phi_1 \wedge \tau_7 \Rightarrow \Phi'_1$$
$$\Phi_3 \wedge \tau_8 \Rightarrow \Phi'_3 \qquad \Phi_1 \wedge \tau_8 \Rightarrow \Phi'_1$$

- Finally, the environment transitions do not invalidate the node assertions:

$$\Phi_1 \wedge \tau_\epsilon \Rightarrow \Phi'_1$$
$$\Phi_2 \wedge \tau_\epsilon \Rightarrow \Phi'_2$$
$$\Phi_3 \wedge \tau_\epsilon \Rightarrow \Phi'_3$$
$$\Phi_4 \wedge \tau_\epsilon \Rightarrow \Phi'_4$$
$$\Phi_5 \wedge \tau_\epsilon \Rightarrow \Phi'_5$$

Generalized Response Diagrams. Response diagrams are based on the transitivity of $\mapsto$; thus, they allow only proofs of properties $\Phi \mapsto \Psi$ where a state for which Ψ holds is reached in finitely many transitions from a state where Φ holds.

A variation of the response diagrams are *generalized response diagrams*, which are based on the induction rule of $\mapsto$:

$$\frac{\mathcal{S} \models \big(p \wedge M = m\big) \mapsto \big(p \wedge M < m\big) \vee q \quad \text{for all } m \in W}{\mathcal{S} \models p \mapsto q}$$

Here nodes are labeled with a ranking function from the state variables to elements of a well-founded order. Generalized response diagrams may contain cycles, but it is required that in each transition the ranking function decreases.

3.5 Invariants as Lemmas

Frequently, the node assertions in response diagrams also imply an invariance property. For example, the formula

$$i^\circ = o \frown q \quad \wedge \quad \#r^\circ + \#q = \#i^\circ + c$$

at the top of Figure 7 is part of the invariant from the invariance diagram of Figure 6. This means that part of the proof effort for this invariant is repeated in the proof of the liveness property.

For verification purposes, it is desirable to use previously proven invariants as lemmas for invariance and response diagrams. Note that verification diagrams actually represent formulas such Φ **co** Φ and $\Phi \mapsto \Psi$, respectively. For both constrains and leadsto properties invariants can be introduced and removed on the left hand side (see Figures 3(e), 3(g) and 4(e), 4(g)). Consequently, given an invariant Γ of a state machine, for each verification condition of the form

$$\Phi \wedge \tau \Rightarrow \Psi' \quad \text{and} \quad \Phi \Rightarrow \mathsf{En}(\tau)$$

it is sufficient to prove

$$\Gamma \wedge \Phi \wedge \tau \Rightarrow \Psi' \quad \text{and} \quad \Gamma \wedge \Phi \Rightarrow \mathsf{En}(\tau)$$

instead.

4 Formalization in Isabelle

Even for the simple buffer example, verification diagrams require the proof of a large number of verification conditions: For a non-hierarchical verification diagram with n nodes and a state machine with m transitions, about $n \times m$ verification conditions have to be proved. Many of these conditions are trivial: The precondition of a transition τ originating from a control state c is obviously not valid for a node assertion Φ which implies $\sigma \neq c$; thus, the validity of any verification condition of the form $\Phi \wedge \tau \Rightarrow \Psi'$ is immediate. Also, many verification conditions are quite similar, so that their proofs are almost identical. Still, some kind of tool support is necessary to discharge the remaining proof obligations and also to check whether a set of verification conditions is complete, so that indeed the verification diagram is valid.

As a first step towards tool support for state machines and verification diagrams, we have formalized the state machine theory from Section 2 in the HOL instantiation of the theorem prover Isabelle [17]. Section 4.1 contains an overview over the resulting Isabelle theory files. As an example of how to use the formalization, Section 4.2 shows how state machines are encoded in Isabelle; Section 4.3 demonstrates how proof obligations from verification diagrams are discharged and combined to show the validity of the verification diagram.

We only give part of the Isabelle formalization; the theory files and proof scripts can be accessed electronically [2]. Introductory texts to Isabelle are also available electronically [11].

4.1 Theory Overview

Our state machine formalization is essentially an adaption of Shankar's work on compositional state machine verification with PVS [19]. While Shankar treats only invariants, however, we also included support for fairness assumptions and liveness properties. The formalization is split into five theories:

- `Executions.thy` defines state machine executions, predicates over states and pairs of states, as well as invariance and leadsto properties for executions.
- `Machines.thy` formalizes state machines, fairness of state machine executions and the set of fair executions of a state machine. Invariance and leadsto properties are lifted from single executions to state machines.
- `Composition.thy` defines state machines composition.
- `IOMachines.thy` adds definitions for input and output over directed channels. It is based on the theory `Prefix.thy`, which provides a prefix operator $\sqsubseteq$ for lists.
- `Focus.thy` combines the other theories and adds some specialized tactics for state machine verification and verification diagrams.

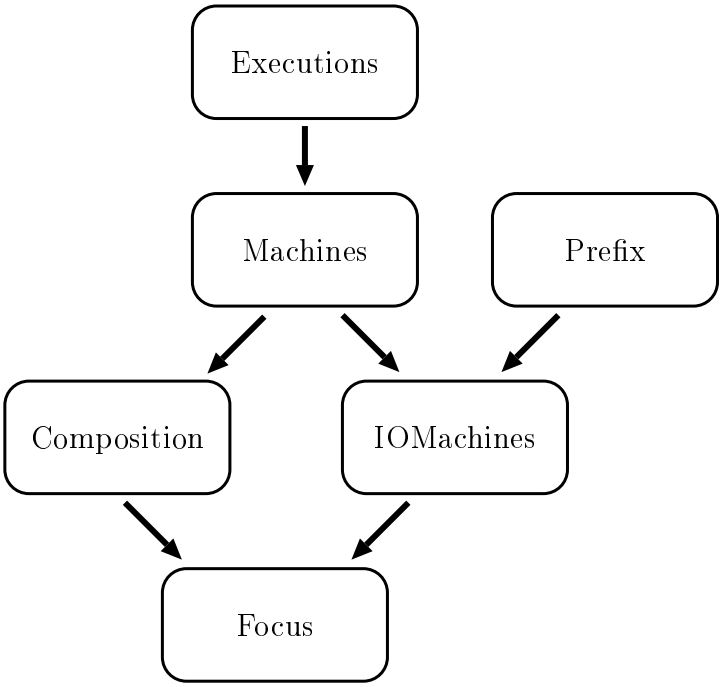


Figure 8: Theory Graph

Figure 8 shows the theory structure of our formalization. The arrows denote dependencies between the theories.

Executions

The theory `Executions.thy` (Figure 9) defines an uninterpreted type *state*. A state machine variable is represented by a state function, which maps from states to the variable's domain. State predicates and actions are defined as predicates over states and pairs of states, respectively. An execution is an infinite sequence of states. A state predicate P is invariant in an execution if it holds in every state of the execution. Leadsto properties $P \mapsto Q$ for single executions are expressed using `resp` (for "response"): Each state of the execution where P holds is followed by a state where Q holds.

An action *act* is enabled in a state s, if there is a state t such that $act(s, t)$ is true. In a formalization based on an uninterpreted state type, proving enabledness by finding a suitable witness for t is impossible for nontrivial actions. The special syntax `basevars <`$v_1, \ldots, v_n$`>` generates axioms

$$\forall\, c_1, \ldots, c_n \bullet \exists\, s \bullet v_1(s) = c_1 \wedge \ldots \wedge v_n(s) = c_n$$

which postulate the existence of a state s for each possible valuation of distinct state functions $v_1, \ldots, v_n$. From these axioms, theorems for proving enabledness properties can be derived:

$$\big(\exists\, x_1, \ldots, x_n \bullet \forall\, t \bullet v_1(t) = x_1 \wedge \ldots \wedge v_n(t) = x_n \Rightarrow act(s, t)\big) \Rightarrow \exists\, t.act(s, t)$$

Instead of finding a witness for a state t, it is then sufficient to find witnesses for values of the state functions $v_1, \ldots, v_n$.

```
Executions = Main +

types
  state
  exec     = "nat => state"
  'a stfun = "state => 'a"
  stpred   = "bool stfun"
  'a trfun = "state => state => 'a"
  action   = "bool trfun"
  expred   = "exec => bool"

arities
  state :: term

syntax
  "_bv"    :: "idts => bool"   ("basevars <_>")

consts
  Enabled  :: action => stpred
  inv      :: stpred => exec  => bool
  resp     :: stpred => stpred => exec  => bool

defs

Enabled_def
        "Enabled act s == (? t. act s t)"
inv_def
        "inv  P ex ==   (! k. P (ex k))"
resp_def
        "resp P Q ex == (! k . P (ex k) --> (? l . k <= l & Q (ex l)))"
end
```

Figure 9: `Executions.thy`

Machines

State machines are formalized in `Machines.thy` as records of an initialization predicate, an action `strans` for proper transitions, an action `etrans` for environment transitions, and a fairness set. The action `strans` usually is equal to the disjunction $\bigvee_{\tau \in \mathcal{T}} \tau$ of all state machine transitions. Note that in our formalization, state machines have no explicit state variable set: Machine variables must be explicitly defined as state functions. This is the reason we need an environment action `etrans`: state functions that represent input channel variables are not immediately recognizable.

An execution is fair if each action in the fairness set is either persistently disabled from a given state on, or infinitely often executed. The fairness set usually is equal to the set

of proper transitions, but allows the modeling of other fairness assumptions. When it is equal to $\bigvee_{\tau \in \mathcal{T}} \tau$, we obtain a minimal progress model where any enabled transition may be taken.

Runs of a state machine are state sequences that respect the machine's initialization predicate and transition actions; executions are runs that in addition are fair according to the machine's fairness set. Invariance and leadsto properties are lifted to the set of executions of a state machine. `WCo` represents the constrains operator **co**.

```
Machines = Executions +

record mach =
  init     :: stpred
  strans   :: action
  etrans   :: action
  fairness :: action set

consts
  isfair :: action => exec => bool
  isrun  :: mach   => exec => bool
  isexec :: mach   => exec => bool

  WCo    :: mach => stpred => stpred => bool
  Inv    :: mach => stpred => bool
  Resp   :: mach => stpred => stpred => bool

defs
isfair_def
        "isfair act ex ==
                (! k . ? l. k <= l & ~ (Enabled act (ex l))) |
                (! k . ? l. k <= l & act (ex l) (ex (Suc l)))"
isrun_def
        "isrun M ex ==  ((init M) (ex 0)) &
                        (! k .  ((strans M) (ex k) (ex (Suc k))) |
                                ((etrans M) (ex k) (ex (Suc k))))"
isexec_def
        "isexec M ex == isrun M ex & (! a : (fairness M) . isfair a ex)"
WCo_def
        "WCo M P Q   == (! ex k. (isexec M ex) -->
                                    (P (ex k) --> Q (ex (Suc k))))"
Inv_def
        "Inv M P == (! ex . (isexec M ex) --> (inv P ex))"
Resp_def
        "Resp M P Q == (! ex . (isexec M ex) --> (resp P Q ex))"
end
```

Figure 10: `Machines.thy`

From the machine theory, numerous theorems can be derived. In particular, all verification rules from Section 3.1 (Figures 3 and 4) are proven.

The Isabelle formalization of the ensure rule looks as follows:

```
[| WCo M P (%s. P s | Q s);
   act : (fairness M);
   ! s t. act s t --> (strans M) s t;
   ! s.   P s --> Enabled act s;
   ! s t. P s & act s t --> Q t
|]
   ==> Resp M P Q
```

This rule is more general than the ensures rule of Figure 4(a), because of the fairness sets of our state machines. Instead of demanding there is a single transition that leads to a state where Q holds, this rule is parameterized by an action *act* which is both in the fairness set, and respects the machine's transition relation. Thus, this rule can be used not only under weak fairness, but also under minimal progress, where both the fairness set `fairness M` and the transition relation `strans M` are equal to $\bigvee_{\tau \in \mathcal{T}} \tau$. In this case, the only choice for act is `strans M`: All enabled transitions must lead to a state where Q holds.

Composition

The composition of two state machines is again a state machine (Figure 11). The initialization predicate is the conjunction of the component initialization predicates; a transition consists of a proper transition of one component machine and environment transition of the other; the environment transition is an environment transition of both component machines; fairness is defined as the union of the fairness sets of the component machines.

```
Composition = Machines +

consts
   compos :: mach => mach => mach

defs
compos_def
   "compos M1 M2 == (|  init =
                          % s. (init M1 s) & (init M2 s),
                        strans =
                          % s t. ((strans M1 s t) & (etrans M2 s t)) |
                                 ((strans M2 s t) & (etrans M1 s t)),
                        etrans =
                          % s t. (etrans M1 s t) & (etrans M2 s t),
                        fairness = (fairness M1) Un (fairness M2)
                     |)"
end
```

Figure 11: `Composition.thy`

The main properties of state machine composition is that it is commutative and associative, and that each execution of a composed machine is also an execution of each component machine. As a consequence, invariance and leadsto properties can be lifted from single components to compositions:

$$\frac{\mathcal{S}_1 \models \mathbf{inv}\,\Phi}{\mathcal{S}_1 \| \mathcal{S}_2 \models \mathbf{inv}\,\Phi} \qquad \frac{\mathcal{S}_1 \models \Phi \mapsto \Psi}{\mathcal{S}_1 \| \mathcal{S}_2 \models \Phi \mapsto \Psi}$$

IOMachines

The theory `IOMachines.thy` (Figure 12) introduces input and output on channels. We use the convention that for a channel c, the part of a channel history that has already been processed is denoted by `r_`c. The theory defines the following predicates as state function, state predicate or action:

- `Current` is a state function that returns the n-th element of the channel history x, where $n = \#x^\circ$. If $x^\circ \neq x$, this is the first message on x that has not yet been processed.
- `Input` is an action that appends the first unread message from x to x^+; it requires that $\#x^\circ \leq x$, so that this message indeed exists.
- `Output` is an action that appends a value a to the channel history x.
- `isEmpty` is a state predicate that is true when there are no unread messages on a channel x.
- `Unch` is an action that forces a state function to remain unchanged.
- `Extend` is an action that allows a list-valued state function to be extended. This action is used to model environment transitions.

Focus

The theory `Focus.thy` combines the previous theories. It introduces no new definitions, but defines a number of verification tactics for state machines. In Section 4.3, we use these tactics for the formal verification of the buffer example.

4.2 The Buffer in Isabelle

In this section we formalize the buffer in a theory file `BUFFER.thy` based on the state machine theories from the previous section. We give a detailed description of the formalization, and only omit redundant aspects, such as some of the transition definitions.

```
IOMachines = Machines + Prefix +

consts
 Current   :: "'a list stfun => 'a list stfun => 'a stfun"
 Input     :: "'a list stfun => 'a list stfun => action"
 Output    :: "'a list stfun => 'a => action"
 isEmpty   :: "'a list stfun => 'a list stfun => stpred"
 Unch      :: "'a stfun => action"
 Extend    :: "'a list stfun => action"

defs
Current_def
        "Current r_x x s == (x s) ! (length (r_x s))"
Input_def
        "Input r_x x s t == (length (r_x s) < length (x s)) &
                            (r_x s) <= (x s) &
                            (r_x t) = (r_x s) @ [Current r_x x s]"
Output_def
        "Output x a s t == (x t = x s @ [a])"
isEmpty_def
        "isEmpty r_x x s == length (r_x s) = length (x s)"
Unch_def
        "Unch v s t == v t = v s"
Extend_def
        "Extend v s t == v s <= v t"
end
```

Figure 12: `IOMachines.thy`

Besides the formalization of the buffer state machine, we also define the node assertions of the buffer's invariance and response verificiation diagrams (Figures 6 and 7).

Buffer State Machine

First, we declare the theory based on the theories `Focus` and a theory `FIFO`. `FIFO` contains enqueue and dequeue operations on lists, since we need a data structure for buffering data in a First-In-First-Out manner.

We then introduce a data type `State` for the control state of the buffer as well as two types for messages and requests; `rqu` is an arbitrary element of the request type `Rqu`.

```
BUFFER = Focus + FIFO +

datatype State   = Store | Empty | Count

types
  Msg
  Rqu

arities
```

```
  Msg :: term
  Rqu :: term

consts
   req :: Rqu
```

We now add the declarations for the channels i, r and o, and call them `I`, `Req` and `Out`. For the input channels, we also need variables i° and r°, which we call `R_I` and `R_Req`. Additionally, we introduce the variables q, c and σ of the appropriate types. Note that all variables are declared as state functions, which map execution states into the variables' values. The constant k is used later for expressing the progress property.

```
consts
 I     :: Msg list stfun
 R_I   :: Msg list stfun
 Req   :: Rqu list stfun
 R_Req :: Rqu list stfun
 Out   :: Msg list stfun

 Q     :: Msg list stfun
 C     :: nat      stfun
 Sigma :: State    stfun
 k     :: nat
```

The STS is described by an initialization assertion, the transitions as well as the environment transition, and a fairness set. We define all these variables of the appropriate type, and define the transition as the disjunction of the eight transitions of the buffer. The initial values of the variables are defined in the predicate `Init`.

```
consts
STS         :: mach
Init        :: stpred
Trans       :: action
Tau1,Tau2,Tau3,Tau4,Tau5,Tau6,Tau7,Tau8,TauE :: action

defs
STS_def
  "STS == (| init = Init,
             strans = Trans,
             etrans = TauE,
             fairness = { Tau1, Tau2, Tau3, Tau4, Tau5, Tau6, Tau7, Tau8 } |)"

Init_def
  "Init s == R_I s   = []    & R_Req s = [] & Out s   = [] &
             Sigma s = Empty & Q s     = [] & C s     = 0"

Trans_def
  "Trans s t == Tau1 s t | Tau2 s t | Tau3 s t | Tau4 s t |
                Tau5 s t | Tau6 s t | Tau7 s t | Tau8 s t "
```

The transition definitions are based on the theory `IOMachines`; since they are quite similar, we just present two of them, τ_4 and the environment transition τ_ϵ. Remember

that variables are encoded as state functions. For example, `Sigma s` denotes the value of σ in state s, while `Sigma t` is the value of σ in state t. Therefore, the first line of the definition of `Tau4` describes the change of value σ from *Store* to *Empty*. The next line states the precondition $\#q = 1$. While i° remains unchanged, i.e. nothing is read from i, we read from r, and therefore we assume there is some unread message that we append to r°. This is described very concisely using the predicates `Unch` and `Input`. The `Output` conjunct states that the transition produces output on `Out`, namely the first element of q. The input variables `I` and `Req` remain unchanged. Finally, the effect on the data variables is stated: the queue `Q` is set to empty, the number of pending requests `C` remains unchanged. The other seven transitions are all defined similarly (the full theory is available online [2]).

```
defs
  Tau4_def
        "Tau4 s t ==    Sigma s = Store & Sigma t = Empty &
                        length (Q s) = 1 &
                        Unch  R_I       s t &
                        Input R_Req Req s t &
                        Output Out (hd (Q s)) s t &
                        Unch  I   s t &
                        Unch  Req s t &
                        Q t = [] &
                        Unch C s t"
```

The environment transition `tauE` leaves all variables that are controlled by the buffer unchanged. Only the input channels `I` and `Req` may be extended.

```
  TauE_def
        "TauE s t ==    Unch   Sigma s t &
                        Unch   Q     s t &
                        Unch   C     s t &
                        Unch   Out   s t &
                        Unch   R_Req s t &
                        Unch   R_I   s t &
                        Extend I   s t &
                        Extend Req s t "
```

Verification Diagram Definitions

For the property proofs, we use a simple invariant for all input channels, which states that the processed input is a prefix of the complete input, i.e. $i^\circ \sqsubseteq i$ for all $i \in I$. We declare this property as `ChannelInv`; it is proved in the next section.

The axiom `Base` enumerates the state variables of the buffer system. It is used to prove that a transition is enabled in a state.

```
consts
   ChannelInv  :: stpred
```

```
defs
   ChannelInv_def
    "ChannelInv s == R_I s <= I s &
                     length (R_I s) <= length (I s) &
                     R_Req s <= Req s &
                     length (R_Req s) <= length (Req s)"
rules
   Base  "basevars <Sigma Out R_I I R_Req Req Q C>"
```

Below are the node definitions for the invariance diagram. We can reflect the hierarchical structure of the diagram by composing the properties in a similar way: `Psi s` states the topmost assertion of the diagram hierarchy. The three predicates `Psi1`, `Psi2` and `Psi3` include `Psi`. The invariant `BufferInv` is the disjunction of the three predicates, stating that the system always fulfills one of these predicates.

```
consts
 Psi,Psi1,Psi2,Psi3 :: stpred
 BufferInv   :: stpred

defs
 Psi_def "Psi s == (R_I s) = (Out s) @ (Q s) &
                   0 <= (C s) &
                   length (R_Req s) = (C s) + length (Out s) &
                   length (R_Req s) + length (Q s) = (C s) + length (R_I s)"

 Psi1_def "Psi1 s == Psi s & Sigma s = Empty & C s = 0 & length (Q s) = 0 "

 Psi2_def "Psi2 s == Psi s & Sigma s = Store & C s = 0 & 0 < length (Q s)"

 Psi3_def "Psi3 s == Psi s & Sigma s = Count & 0 < (C s) & length (Q s) = 0"

 BufferInv_def "BufferInv s == Psi1 s | Psi2 s | Psi3 s"
```

The response diagram is formalized in a similar way. Again we introduce properties `Phi12`, `Phi34`, and `Phi` for the super-states.

```
consts
 Phi0,Phi1,Phi2,Phi3,Phi4,Phi5,Phi12,Phi34,Phi :: stpred

Phi_def
  "Phi s == (R_I s) = (Out s) @ (Q s) &
            length (R_Req s) + length (Q s) = (C s) + length (R_I s) &
            length (Out s) = k &
            k < min (length (I s)) (length (Req s))"

Phi0_def  "Phi0 s == k < length (Out s)"

Phi12_def "Phi12 s == Phi s & Sigma s = Store & C s = 0"
```

```
Phi1_def  "Phi1 s == Phi12 s & 1 < length (Q s)"

Phi2_def  "Phi2 s == Phi12 s & 1 = length (Q s)"

Phi34_def "Phi34 s == Phi s & Sigma s = Count & 0 = length (Q s)"

Phi3_def  "Phi3 s == Phi34 s & 1 < C s"

Phi4_def  "Phi4 s == Phi34 s & 1 = C s"

Phi5_def  "Phi5 s == Phi s & Sigma s = Empty & 0 = C s & 0 = length (Q s)"
```

This completes the formalization of the buffer state machine and the verification diagram nodes.

The translation of the state transition and verification diagrams to 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.

4.3 Buffer Verification

In this section we describe how the validity of the diagrams can be proved with Isabelle. Again, we go through the whole proof file `BUFFER.ML` and only omit similar proof obligations.

First, we gather the the definitions from the Buffer theory in lists for easier access:

```
val Buffer_defs = [STS_def, BufferInv_def, ChannelInv_def,
                   Init_def, Trans_def, TauE_def,
                   Tau1_def, Tau2_def, Tau3_def, Tau4_def,
                   Tau5_def, Tau6_def, Tau7_def, Tau8_def ];
val Psi_defs    = [Psi1_def,Psi2_def,Psi3_def,Psi_def];
val Phi_defs    = [Phi0_def,Phi1_def,Phi2_def,Phi3_def,Phi4_def,
                   Phi5_def, Phi12_def,Phi34_def,Phi_def];
```

From the base variable axiom, we automatically derive an enabledness theorem for the buffer (see page 25). Below is the resulting theorem `BaseEnabled`:

```
? x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8.
     ALL t.
        Sigma t = x_1 & Out t   = x_2 & R_I t = x_3 &
        I t     = x_4 & R_Req t = x_5 & Req t = x_6 &
        Q t     = x_7 & C t     = x_8 --> P s t
  ==> ? t. P s t" : thm
```

Then, we prove the channel invariant using the specific tactic `channelinv_tac`, which takes the state machine definitions as a parameter:

```
bind_thm("BaseEnabled", base_thm thy Base);

Goal "Inv STS ChannelInv";
by (channelinv_tac Buffer_defs);
qed "ChannelInv";
```

Proof of the Invariance Diagram

The proof obligations for the invariance diagram for the buffer are listen in Section 3.3. Isabelle easily proves all of them. We first define a simple prover function, and then apply it to the list of proof obligations.

```
fun prover g = (writeln g;
                prove_goalw thy (Buffer_defs @ Psi_defs @ Phi_defs) g
                (fn prems => [cut_facts_tac prems 1, chan_tac 1]));

val invariance_vc = map prover [
  "[| Psi1 s ; Tau1 s t |] ==> Psi2 t",
  "[| Psi1 s ; Tau5 s t |] ==> Psi3 t",

...

  "[| Psi2 s ; TauE s t |] ==> Psi2 t",
  "[| Psi3 s ; TauE s t |] ==> Psi3 t"
];
```

The resulting theorems together with the buffer definitions are then used to combine the results into an invariance diagram. The invariance diagram tactic `invdiag_tac` assembles the property

$$\mathit{Buffer} \models \big(\Psi 1 \vee \Psi 2 \vee \Psi 3\big)\ \mathbf{co}\ \big(\Psi 1 \vee \Psi 2 \vee \Psi 3\big)$$

The tactic can use other invariants as lemmas (see Section 3.5); here we include the channel invariant.

Finally, `auto_tac` solves two remaining subgoals: The buffer's initial state satisfies $\Psi 1 \vee \Psi 2 \vee \Psi 3$, and the diagram implies the buffer invariant `BufferInv`.

```
Goal "Inv STS BufferInv";
by (invdiag_tac "STS"
        [STS_def,Trans_def] ["Psi1","Psi2","Psi3"]
        [ChannelInv] invariance_vc);
by (auto_tac (claset(),simpset() addsimps (Buffer_defs @ Psi_defs)));
qed "BufferInv";
```

Proof of the Response Diagram

The 50 proof obligations for the progress diagram are already listed in Section 3.4. Most of them can be proven with the same technique as used for invariants. We apply the prover function to all combinations of node assertions and transitions, with appropriate target nodes:

```
val progress_vc = map prover [
"[| ChannelInv s; BufferInv s; Phi5 s; Tau1 s t |] ==> Phi2 t",
"[| ChannelInv s; BufferInv s; Phi5 s; Tau2 s t |] ==> Phi5 t",
"[| ChannelInv s; BufferInv s; Phi5 s; Tau3 s t |] ==> Phi5 t",

...

"[| ChannelInv s; BufferInv s; Phi1 s; Tau7 s t |] ==> Phi1 t",
"[| ChannelInv s; BufferInv s; Phi1 s; Tau8 s t |] ==> Phi1 t",
"[| ChannelInv s; BufferInv s; Phi1 s; TauE s t |] ==> Phi1 t"
];
```

Note that we strengthened the left hand side of each verification conditions with the invariants `ChannelInv` and `BufferInv`; these invariants are later removed via the invariant elimination rules.

To show the enabledness conditions, we use the tactic `enabled_tac`, which takes the base variable theorem as a parameter. We show only two of the proofs here:

```
Goalw (Buffer_defs @ Phi_defs @ [Enabled_def])
  "[| ChannelInv s; BufferInv s; Phi5 s |] ==> Enabled Tau1 s";
by (enabled_tac BaseEnabled);
qed "EnPhi5Tau1";

...

Goalw (Buffer_defs @ Phi_defs @ [Enabled_def])
  "[| ChannelInv s; BufferInv s; Phi1 s |] ==> Enabled Tau2 s";
by (enabled_tac BaseEnabled);
qed "EnPhi1Tau2";
```

The verification conditions can now be combined to show the response properties. First we show that from each diagram node, we can either reach the terminal node Φ_0, or at least a node that is closer to it. Besides lists of definitions, the tactic `ensures_tac` takes a transition name as a parameter; this is a hint which transitions is used to progress to the target node. The tactic also takes a list of the invariants used in the verification conditions; they are removed via the invariant elimination rules and do not appear in the final property.

```
val ensures_vc = [EnPhi5Tau1,EnPhi5Tau5,EnPhi4Tau8,EnPhi3Tau6,
                  EnPhi2Tau4,EnPhi1Tau2];

Goal "Resp STS Phi1 Phi0";
by (ensures_tac [STS_def,Trans_def] [ChannelInv, BufferInv]
        "Tau2" (progress_vc @ ensures_vc));
qed "r_1_2_0";

Goal "Resp STS Phi3 Phi0";
by (ensures_tac [STS_def,Trans_def] [ChannelInv, BufferInv]
```

```
        "Tau6" (progress_vc @ ensures_vc));
qed "r_3_6_0";

Goal "Resp STS Phi4 (% s. Phi0 s | Phi3 s)";
by (ensures_tac [STS_def,Trans_def] [ChannelInv, BufferInv]
        "Tau8" (progress_vc @ ensures_vc));
qed "r_4_8_03";

Goal "Resp STS Phi2 (% s. Phi0 s | Phi1 s)";
by (ensures_tac [STS_def,Trans_def] [ChannelInv, BufferInv]
        "Tau4" (progress_vc @ ensures_vc));
qed "r_2_4_01";

Goal "Resp STS Phi5 (% s. Phi2 s | Phi4 s)";
by (ensures_tac [STS_def,Trans_def] [ChannelInv, BufferInv]
        "Tau1" (progress_vc @ ensures_vc));
qed "r_5_1_24";
```

Finally, we combine these theorems to prove the main result of the diagram: We reach Φ_0 from *all* nodes. The response diagram tactic `invrespdiag_tac` requires several parameters: the invariant list, the state transition system name, the list of the basic response properties, and a list with the diagram node assertion names. The linear order of this last list has to be compatible with the partial order underlying the response diagram (Section 3.4).

The diagram tactic leaves a subgoal: Using lower-level proof commands we show that $\Phi \Rightarrow \bigvee_{1 \leq i \leq 5} \Phi_i$.

```
Goal "Resp STS Phi Phi0";
by (invrespdiag_tac "STS"
                ["Phi0", "Phi1", "Phi2", "Phi3", "Phi4", "Phi5"]
                [ChannelInv, BufferInv]
                [r_1_2_0, r_3_6_0, r_4_8_03, r_2_4_01, r_5_1_24]);
by (Blast_tac 1);
by (rtac allI 1);
by (simp_tac (simpset() addsimps (Buffer_defs @ Phi_defs @ Psi_defs)) 1);
by (tidy_tac 1);
by (etac disjE 1);
by (etac disjE 2);
by (ALLGOALS Force_tac);
qed "BufferResponse";
```

The tactics we used are just ad-hoc solutions that seem to be powerful enough to prove the obligations that we encountered so far. They certainly need to be improved, made more general, more elegant and more efficient. Given a tool that generates theory files from state transition diagrams and verification diagrams, it would be easy to generate tailored verification tactics that require fewer parameters.

5 Example: Communication System

Figure 13 shows a communication system (originally proposed by the VSE group of the DFKI, Saarbrücken, [10]). The system consists of a sender and a receiver connected via a queue component. The queue's buffer can hold N data elements. To ensure that the buffer does not overflow a handshaking protocol is used. We assume that the sender "pushes" data (it sends a datum, then waits for an acknowledgment from the queue), while the receiver "pulls" data (it sends a request to the queue, then awaits a datum). Request and acknowledgment signals are modeled with the singleton set $\mathsf{Signal} = \{\circledast\}$.

The example is also treated in [1], where parts of the property proofs are presented based on the verification rules for **co** and $\mapsto$. We first give the state machine and black box specifications of the communication system (Sections 5.1 and 5.2), and then present the verification diagrams and some of the proof obligations that imply that the state machines indeed fulfill the safety and liveness parts of the black box specifications (Sections 5.3-5.4). In Section 5.5 we also show that the communication system can be implemented with finite channel buffers for the internal communication channels.

The Isabelle formalization and the proof scripts of the communication system are similar to those of the buffer example. Because of their length we do not include them in this report; all files can be accessed electronically [2]. For some of the proofs we give informal justifications. Isabelle can discharge all of them using the same tactics that were used in the buffer example of Section 4.3.

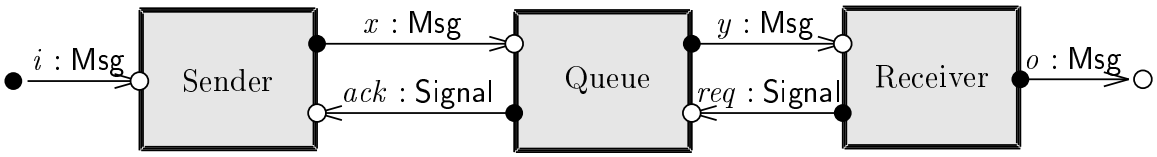


Figure 13: Bounded Buffer

5.1 State Machine Specifications

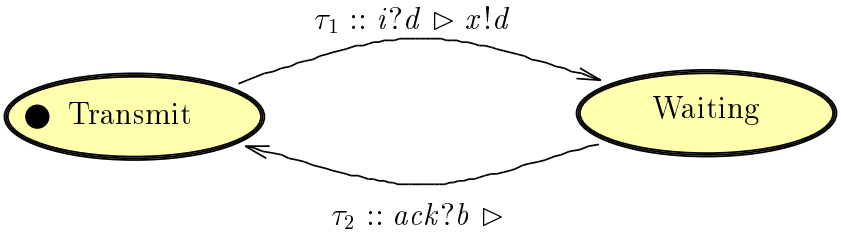
Figure 14 shows the state transitions diagrams of the sender, queue and receiver components. The queue component has an attribute variable q, which holds a finite sequence of messages.

5.2 Black Box Specifications

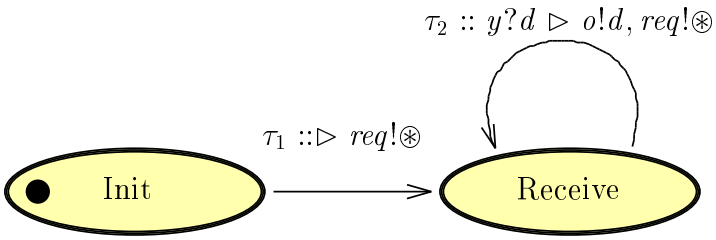
The purpose of the communication system is to realize data transmission with bounded internal channel buffers with a handshaking protocol. Thus, the overall black box behavior should imply

$$o = i$$

Sender



Receiver



Queue

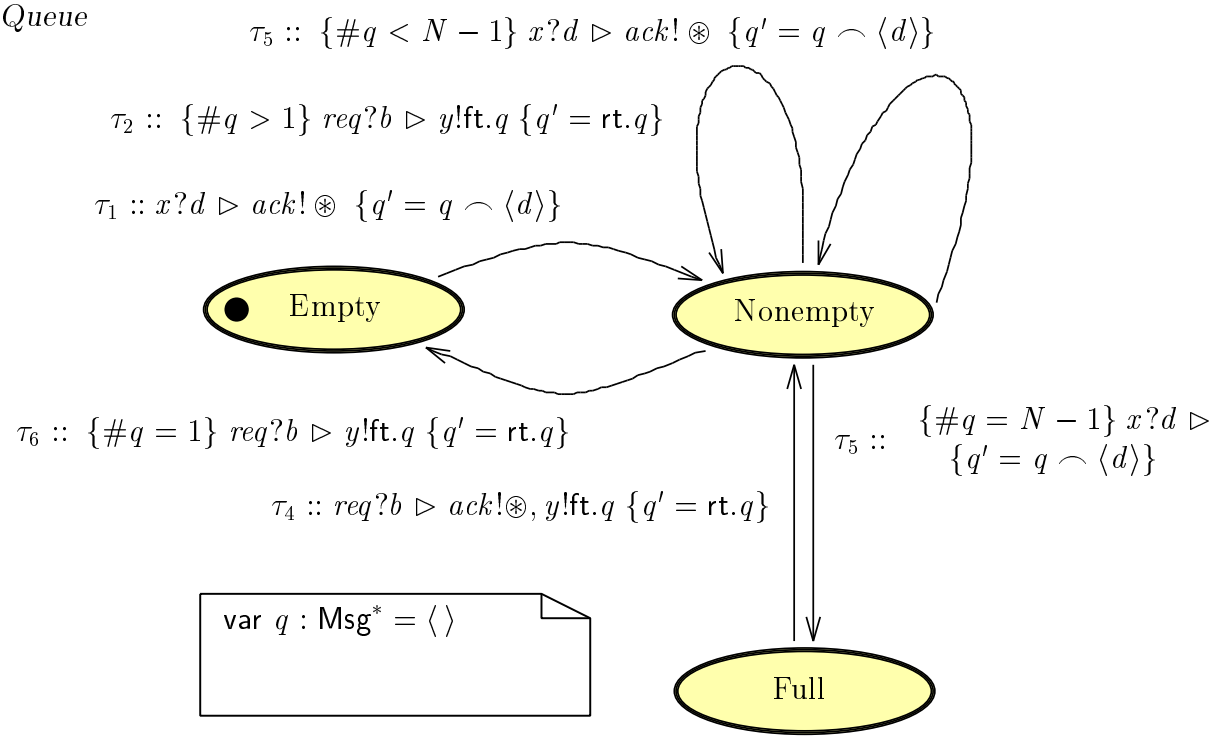


Figure 14: Sender, Receiver and Queue STDs

so that the communication system can be used instead of a simple unidirectional communication channel with unbounded capacity.

Below we give black box specifications for each of the three components. They are divided into prefix (safety) and length (liveness) properties.

Sender

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

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

The prefix part of the sender specification simply states the obvious requirement that the output channel history is a prefix of the input channel history.

The length property of the sender expresses its "push" behavior: The length of the output is one more than the number of acknowledgments received from the queue, provided there is still data from the environment available.

Receiver

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

$o \sqsubseteq y$
$\#o \geq \#y$
$\#req = 1 + \#y$

For the receiver's data output channel, the safety and and liveness properties are similar to the sender. The length property for the request channel expresses the "pull" behavior of the receiver: Immediately after initialization and after each message received from the queue a request is sent.

Note that here the length property for the requests is an equality. This is because it also incorporates the safety property that the length of req must be less than or equal to $1+\#y$; since it is only the number of requests that is relevant, instead of a prefix property a numerical inequality is used as an upper bound for the length of the communication history.

Queue(N)

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

$y \sqsubseteq x$
$\#y \geq min(\#x, \#req)$
$\#ack = min(\#x, \#req + N - 1)$

For the queue's data output channel y, the specification is again split into a prefix and a length property. For the handshake signal ack, safety and liveness aspects are again combined into a single equality.

Black box specifications are composed by conjunction; a precondition is that their output channels are disjoint. Channels that are both input to a component and output from another, become output channels of the complete system. The specification for the composition of sender, queue and receiver in our example is shown below.

System(N)

in i : Msg

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

$$x \sqsubseteq i$$
$$y \sqsubseteq x$$
$$o \sqsubseteq y$$

$$\#x \geq min(\#i, 1 + \#ack)$$
$$\#y \geq min(\#x, \#req)$$
$$\#ack = min(\#x, \#req + N - 1)$$
$$\#o \geq \#y$$
$$\#req = 1 + \#y$$

From the specification of *System*(N) above, we can immediately see that the output is a prefix of the input. By some case analysis it can also be shown that the length of the output equals the length of the input. This implies $o = i$ for all input streams i: The communication system indeed implements the identity relation.

5.3 Safety Properties

For the safety part of the black box specifications, we need to show the following properties:

$$\begin{aligned}
[\![Sender]\!] &\Rightarrow x \sqsubseteq i \\
[\![Receiver]\!] &\Rightarrow o \sqsubseteq y \\
[\![Receiver]\!] &\Rightarrow \#req \leq 1 + \#y \\
[\![Queue]\!] &\Rightarrow y \sqsubseteq x \\
[\![Queue]\!] &\Rightarrow \#ack \leq min(\#x, \#req + N - 1)
\end{aligned}$$

All of these properties are admissible [16], hence it is sufficient to show that the properties are invariants of the state transitions systems. For example, we have to show:

$$Sender \models \mathbf{inv}\, x \sqsubseteq i$$

That these properties are invariants cannot be proven directly. Instead, we derive stronger invariants, which imply the safety properties above. The stronger invariants relate the length of the output channel histories with the length of the already processed part of the input channel histories. Typically, this relation depends on the current control state of a component.

For the sender, the following is a suitable stronger invariant:

$$x = i^\circ$$
$$\big(\sigma = \mathit{Transmit} \wedge \#x = \#\mathit{ack}^\circ\big) \vee \big(\sigma = \mathit{Waiting} \wedge \#x = 1 + \#\mathit{ack}^\circ\big)$$

The proof of this invariant is straightforward; since the sender state machine only has two transitions, the proof is straightforward and we do not use a verification diagram. For the receiver, we use the following invariant; again, the proof is straightforward:

$$o = y^\circ$$
$$\big(\sigma = \mathit{Init} \wedge \#\mathit{req} = \#y^\circ\big) \vee \big(\sigma = \mathit{Receive} \wedge \#\mathit{req} = 1 + \#y^\circ\big)$$

The queue invariant is a bit more elaborate. It is visualized by the invariance diagrams of Figure 15, which follows the structure of the queue's state transition diagram (see Figure 14).

Since there are three nodes in the invariance diagram and the queue has six proper transitions and one environment transition, a total of 21 verification conditions has to be discharged. The verification conditions are then assembled by the invariance diagram tactic to yield the complete queue invariant:

$$x^\circ = y \frown q$$
$$\#\mathit{req}^\circ = y$$
$$\#x^\circ = \#\mathit{req}^\circ + \#q$$
$$\Big(\big(\sigma = \mathit{Empty} \wedge \#q = 0 \wedge \#x^\circ = \#\mathit{ack}\big) \vee$$
$$\big(\sigma = \mathit{Nonempty} \wedge 1 \leq \#q \leq N - 1 \wedge \#x^\circ = \#\mathit{ack}\big) \vee$$
$$\big(\sigma = \mathit{Full} \wedge \#q = N \wedge \#x^\circ = 1 + \#\mathit{ack}\big)\Big)$$

It is easy to show that the sender, receiver and queue invariants indeed imply the required safety properties.

In the Isabelle formalization, the proofs of all three invariants require a channel invariance lemma similar to that of the Buffer in Section 4.2.

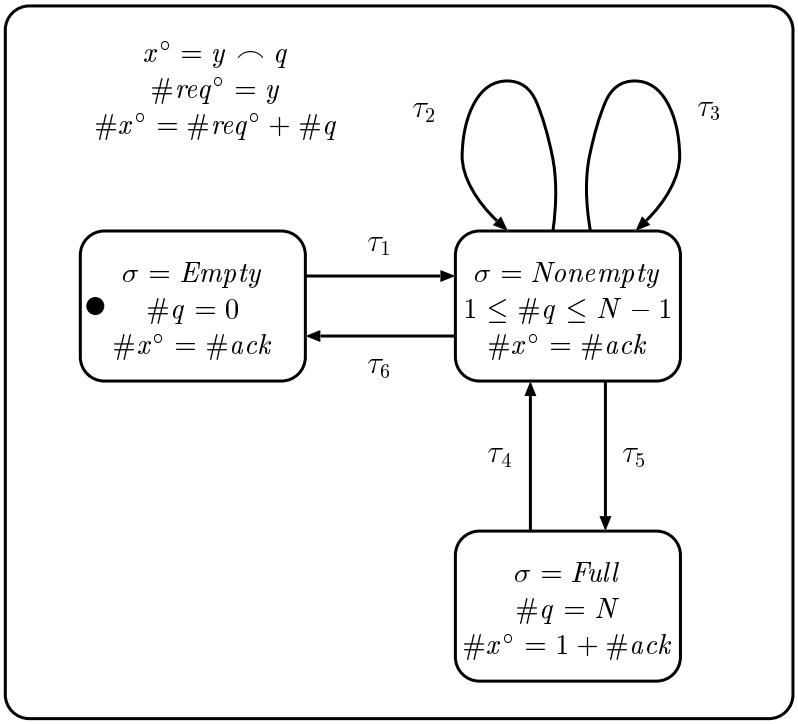


Figure 15: Queue Invariance Diagram

5.4 Liveness Properties

For the liveness part of the black box specifications, we have to show the following properties:

$$
\begin{aligned}
[\![Sender]\!] &\Rightarrow \#x \geq min(\#i, 1 + \#ack) \\
[\![Receiver]\!] &\Rightarrow \#o \geq \#y \\
[\![Receiver]\!] &\Rightarrow \#req \geq 1 + \#y \\
[\![Queue]\!] &\Rightarrow \#ack \geq min(\#x, \#req + N - 1) \\
[\![Queue]\!] &\Rightarrow \#y \geq min(\#x, \#req)
\end{aligned}
$$

These properties all have the form $\#u \geq f(v_1, \ldots, v_n)$ for an output channel u, input channels $v_1, \ldots, v_n$ and a function f from the input channel histories to $\mathbb{N}$; the function f is assumed to be monotonic. According to Section 2.7 such a liveness property can be shown by proving the following leadsto property on the state machine (where k is a constant distinct from all channel names):

$$\#u = k \wedge f(v_1, \ldots, v_n) > k \mapsto \#u > k$$

For the communication system we regard the following leadsto properties:

$$\begin{array}{rcll} \mathit{Sender} & \models & \#x = k \wedge \mathit{min}(\#i, 1 + \#\mathit{ack}) > k & \mapsto \ \#x > k \\ \mathit{Receiver} & \models & \#o = k \wedge \#y > k & \mapsto \ \#o > k \\ \mathit{Receiver} & \models & \#\mathit{req} = k \wedge 1 + \#y > k & \mapsto \ \#\mathit{req} > k \\ \mathit{Queue} & \models & \#y = k \wedge \mathit{min}(\#x, \#\mathit{req}) > k & \mapsto \ \#y > k \\ \mathit{Queue} & \models & \#\mathit{ack} = k \wedge \mathit{min}(\#x, \#\mathit{req} + N - 1) > k & \mapsto \ \#\mathit{ack} > k \end{array}$$

The first property means that the sender's only output channel x is extended; the next two properties imply the output extension of the two receiver output channels; the last two properties imply the output extension for the two output channels of the queue. For each of these properties, we use a verification diagram. The diagrams use the component invariants shown in the previous section as lemmas.

The sender's response diagram is shown in Figure 16. From the sender's invariant we know that the system is either in state "*Transmit*" or in state "*Waiting*". In the former case, transition τ_1 (which copies a message from i to x) immediately leads to the target node; in the other case, we first have to return via τ_2 (which consumes an acknowledgment signal) to "*Transmit*".

The enabledness of the two transitions is easy to show: When the sender is in state "*Transmit*", we know from the invariant that $\#x = \#i^\circ$ and hence $\#i^\circ = \#x = k < \mathit{min}(\#i, 1 + \#\mathit{ack}) \leq \#i$; thus, τ_1 is enabled. When the sender is in state "*Waiting*", we know from the invariant that $\#x = 1 + \#\mathit{ack}^\circ$, and hence $1 + \#\mathit{ack}^\circ = \#x = k < \mathit{min}(\#i, 1 + \#\mathit{ack}) \leq 1 + \#\mathit{ack}$; thus, τ_2 is enabled.

Figure 17 shows the leadsto diagrams for the two liveness properties of the receiver; the reasoning behind these two diagrams is similar to that of the sender's. Note that transition τ_1 is always enabled when the receiver is in its initial state.

Figure 18 shows the two leadsto diagrams for the queue component. The diagram in Figure 18(a) is used to show that acknowledgments are produced when the queue is not completely filled. Similarly, the diagram in Figure 18(b) is used to prove the queue output data on y, provided it is not empty and the receiver sent a request.

In Figure 18, an acknowledgment is produced immediately when a message is received from the sender while the queue is empty (with transition τ_1), or when the queue is full and receives a request from the receiver (transition τ_4). If the queue is neither empty nor completely filled, it can receive and acknowledge further input with transition τ_3; it might also first answer requests by transitions τ_2 until it is empty, before following transition τ_1. The case that the buffer is filled to just below capacity is handled separately: The input of an additional message from the sender does not result immediately in an acknowledgment. The enabledness of the transitions in the diagram is again shown by reasoning with (in)equalities, similar to the sender's diagram above.

The diagram in Figure 18 is quite similar; here the situation that only one message is stored in the queue has to be treated as a special case.

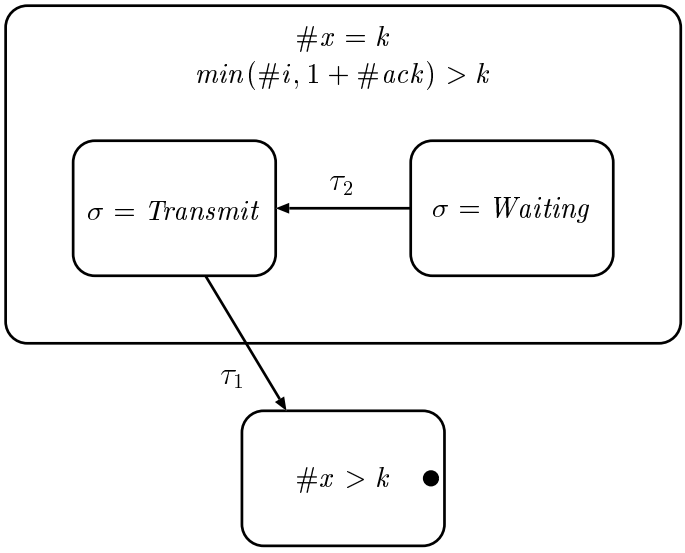


Figure 16: Leadsto Diagram for the Sender

5.5 Boundedness

So far we have shown that the communication system outputs all messages received on i, and that it outputs only those messages.

Now we prove that the system also fulfills its purpose: to realize error-free transmission over channels with finite buffer size. We show the following invariant, which states that each of the internal channel buffers contains at most one message:

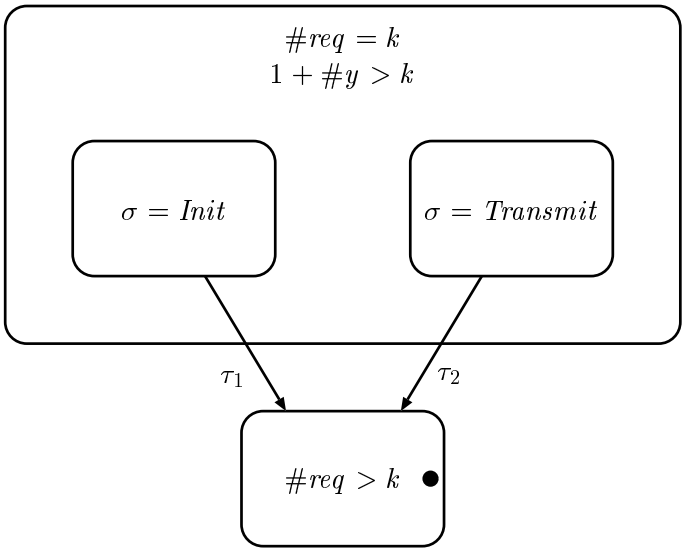
$$\big(Sender \| Queue \| Receiver\big) \models \mathbf{inv}\big(\#x^+ \leq 1 \wedge \#ack^+ \leq 1 \wedge \#y^+ \leq 1 \wedge \#req^+ \leq 1\big)$$

Boundedness is a global system property. Without resorting to assumption/guarantee techniques, it can only be shown for the complete system, which is defined by an interleaving composition of the three components:

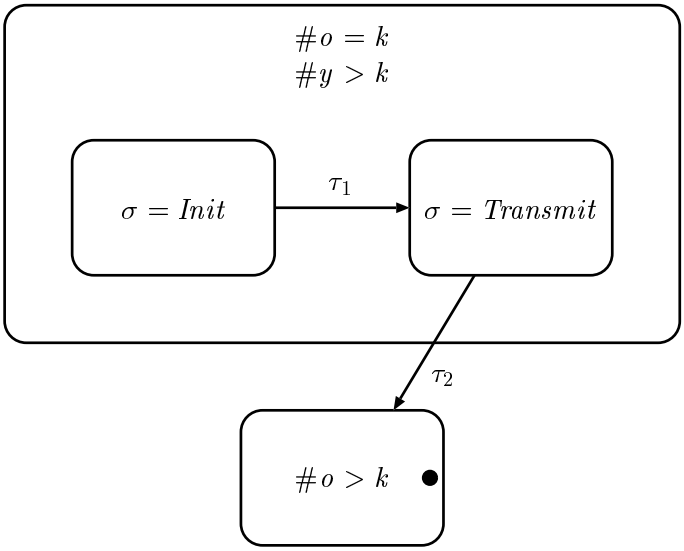
```
SystemSTS_def
        "SystemSTS == compos SenderSTS (compos QueueSTS ReceiverSTS)"
```

Each transition of the complete system is the conjunction of one transition each of sender, queue and receiver; at most one of these transitions is a non-environment transition. Thus, the boundedness invariant follows from the validity of the trivial invariance diagram of Figure 19.

To prove the verification conditions of the diagram, the invariants and channel invariants of each component are used as lemmas; they are lifted to the system level by the compositionality theorems for invariants (see Section 4.1).

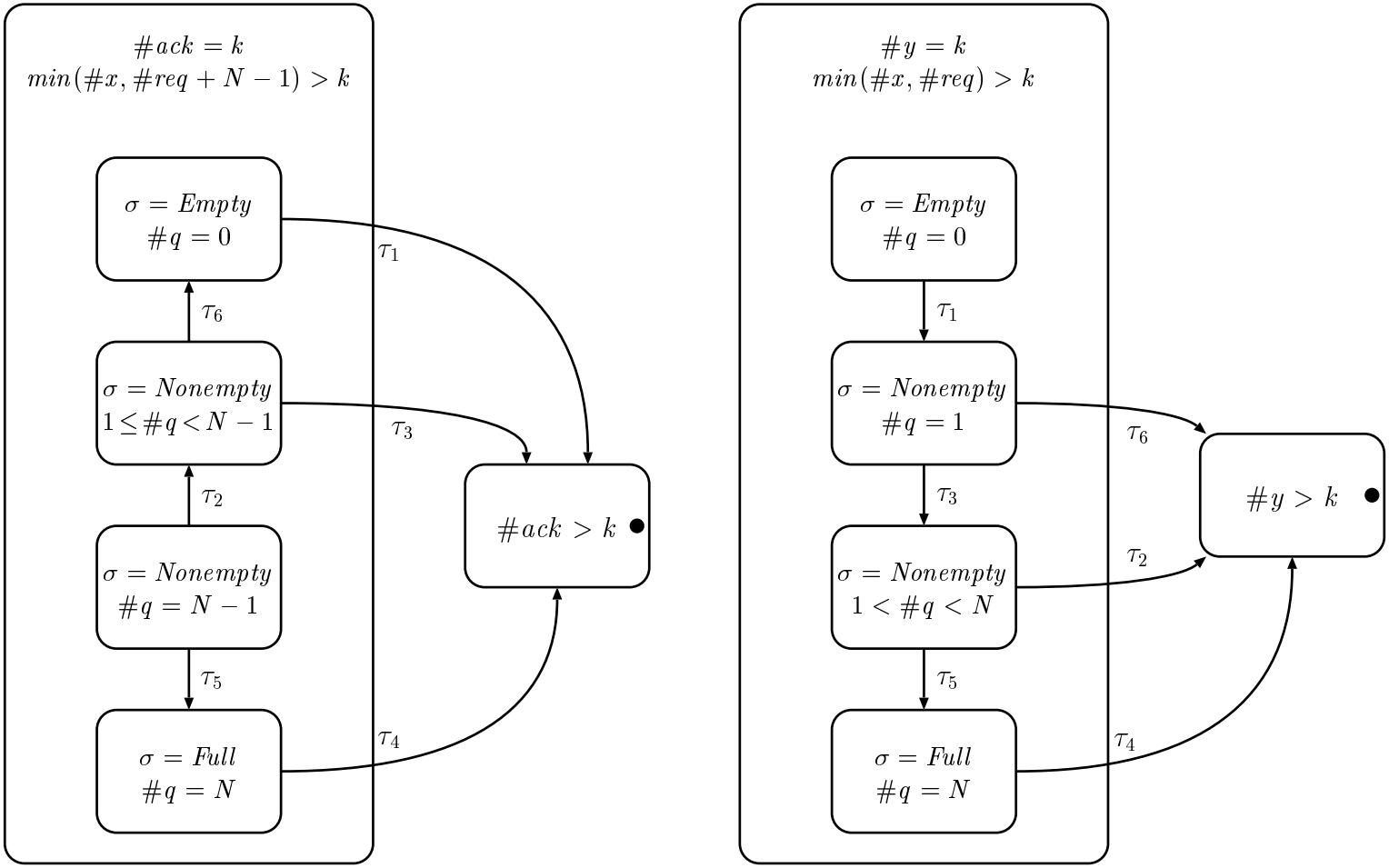


(a) Output extension of "*req*"



(b) Output extension of "*o*"

Figure 17: Leadsto Diagrams for the Receiver



(a) Output extension of "*ack*"

(b) Output extension of "*y*"

Figure 18: Leadsto Diagrams for the Queue

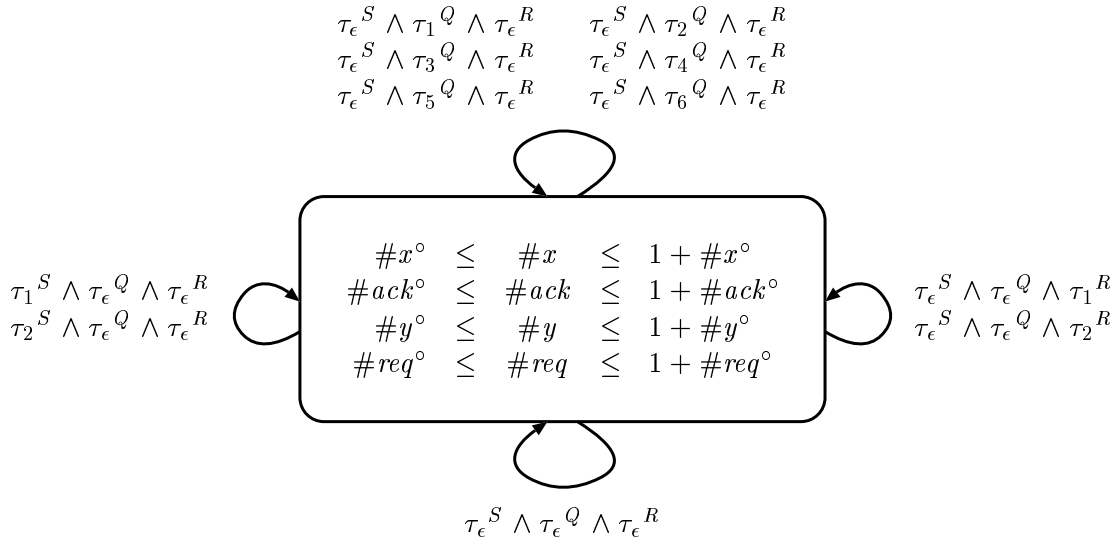


Figure 19: Invariance Diagram for Boundedness

6 Conclusion

In a previous report we showed how to close the gap between relational I/O specifications with temporal logic properties [1]. Verification diagrams and Isabelle tool support move this more theoretical foundation closer to practice.

Verification diagrams are a concise, yet readable, documentation of the structure of some classes of temporal logic proofs. The verification conditions associated with a diagram are comparatively simple formulas in predicate logic. Discharging these proof obligations with a theorem prover is quite feasible, at least with the level of automatization offered by Isabelle.

Our formalization of the theory of state machines provides more than proof support for verification conditions: Verification rules for the temporal logic operators **co**, $\mapsto$ and **inv** are proven correct, and the validity of verification diagrams is derived automatically from verification conditions using tactics that mimic the proof structure from Sections 3.3 and 3.4.

From a tool support technique, the automatic generation of theory files, proof obligations and —as far as possible— proof scripts from system structure, state transition and verification diagrams is obviously desirable. In a recent project [20], the CASE tool AutoFocus [8, 9] has been linked with the theorem proving environment VSE II [18]; similar interfaces between AutoFocus and Isabelle would benefit from this existing work.

In this report, we did not delve into the tactics and proof scripts that we used to solve the verification conditions of the examples or that assemble them to yield invariance and leadsto properties. Much more work is necessary to handle more complicated systems that require a formalization of the data types stored in state attributes and sent over communication channels. Unfortunately, while from a technical point of view, Isabelle tactics are well-documented, from a methodological point of view the art of writing Isabelle tactics seems hardly explored.

Often, compositional reasoning about state machine systems 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 [22, 21]. 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.

References

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

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

[3] Max Breitling and Jan Philipps. Step by step to histories. In *AMAST'2000*, 2000.

[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. From states to histories. In *Marktoberdorf Summer School on Engineering Theories of Software Construction*. (Springer, NATO ASI Series F), 2000. To be published.

[6] 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.

[7] 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.

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

[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] D. Hutter, H. Mantel, G. Rock, and W. Stephan. Nebenläufige verteilte Systeme: Ein Producer/Consumer Beispiel. Internal Manuscript, DFKI, 1998.

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

[12] Z. Manna, A. Browne, H. B. Sipma, and T. E. Uribe. Visual abstractions for temporal verification. volume 1548 of *Lecture Notes in Computer Science*, pages 28–41, 1998.

[13] 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.

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

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

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

[17] L. C. Paulson. *Isabelle: A Generic Theorem Prover. Lecture Notes in Computer Science 828.* Springer, 1994.

[18] G. Rock, W. Stephan, and A. Wolpers. Tool Support for the Compositional Development of Distributed Systems. In *Proc. Formale Beschreibungstechniken für verteilte Systeme,GI/ITG-Fachgespräch.* GMD-Studien Nr. 315, ISBN: 3-88457-514-2, 1997.

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

[20] 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`.

[21] K. Stølen. Assumption/commitment rules for dataflow networks - with an emphasis on completeness. In H. R. Nielson, editor, *Programming Languages and Systems - ESOP'96, Lecture Notes in Computer Science 1058*, pages 356–372. Springer, 1996.

[22] 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.

SFB 342: Methoden und Werkzeuge für die Nutzung paralleler Rechnerarchitekturen

bisher erschienen :

Reihe A

Liste aller erschienenen Berichte von 1990-1994
auf besondere Anforderung

342/01/95 A Hans-Joachim Bungartz: Higher Order Finite Elements on Sparse Grids

342/02/95 A Tao Zhang, Seonglim Kang, Lester R. Lipsky: The Performance of Parallel Computers: Order Statistics and Amdahl's Law

342/03/95 A Lester R. Lipsky, Appie van de Liefvoort: Transformation of the Kronecker Product of Identical Servers to a Reduced Product Space

342/04/95 A Pierre Fiorini, Lester R. Lipsky, Wen-Jung Hsin, Appie van de Liefvoort: Auto-Correlation of Lag-k For Customers Departing From Semi-Markov Processes

342/05/95 A Sascha Hilgenfeldt, Robert Balder, Christoph Zenger: Sparse Grids: Applications to Multi-dimensional Schrödinger Problems

342/06/95 A Maximilian Fuchs: Formal Design of a Model-N Counter

342/07/95 A Hans-Joachim Bungartz, Stefan Schulte: Coupled Problems in Microsystem Technology

342/08/95 A Alexander Pfaffinger: Parallel Communication on Workstation Networks with Complex Topologies

342/09/95 A Ketil Stølen: Assumption/Commitment Rules for Data-flow Networks - with an Emphasis on Completeness

342/10/95 A Ketil Stølen, Max Fuchs: A Formal Method for Hardware/Software Co-Design

342/11/95 A Thomas Schnekenburger: The ALDY Load Distribution System

342/12/95 A Javier Esparza, Stefan Römer, Walter Vogler: An Improvement of McMillan's Unfolding Algorithm

342/13/95 A Stephan Melzer, Javier Esparza: Checking System Properties via Integer Programming

342/14/95 A Radu Grosu, Ketil Stølen: A Denotational Model for Mobile Point-to-Point Dataflow Networks

342/15/95 A Andrei Kovalyov, Javier Esparza: A Polynomial Algorithm to Compute the Concurrency Relation of Free-Choice Signal Transition Graphs

342/16/95 A Bernhard Schätz, Katharina Spies: Formale Syntax zur logischen Kernsprache der Focus-Entwicklungsmethodik

342/17/95 A Georg Stellner: Using CoCheck on a Network of Workstations

342/18/95 A Arndt Bode, Thomas Ludwig, Vaidy Sunderam, Roland Wismüller: Workshop on PVM, MPI, Tools and Applications

Reihe A

342/19/95 A Thomas Schnekenburger: Integration of Load Distribution into ParMod-C

342/20/95 A Ketil Stølen: Refinement Principles Supporting the Transition from Asynchronous to Synchronous Communication

342/21/95 A Andreas Listl, Giannis Bozas: Performance Gains Using Subpages for Cache Coherency Control

342/22/95 A Volker Heun, Ernst W. Mayr: Embedding Graphs with Bounded Treewidth into Optimal Hypercubes

342/23/95 A Petr Jančar, Javier Esparza: Deciding Finiteness of Petri Nets up to Bisimulation

342/24/95 A M. Jung, U. Rüde: Implicit Extrapolation Methods for Variable Coefficient Problems

342/01/96 A Michael Griebel, Tilman Neunhoeffer, Hans Regler: Algebraic Multigrid Methods for the Solution of the Navier-Stokes Equations in Complicated Geometries

342/02/96 A Thomas Grauschopf, Michael Griebel, Hans Regler: Additive Multilevel-Preconditioners based on Bilinear Interpolation, Matrix Dependent Geometric Coarsening and Algebraic-Multigrid Coarsening for Second Order Elliptic PDEs

342/03/96 A Volker Heun, Ernst W. Mayr: Optimal Dynamic Edge-Disjoint Embeddings of Complete Binary Trees into Hypercubes

342/04/96 A Thomas Huckle: Efficient Computation of Sparse Approximate Inverses

342/05/96 A Thomas Ludwig, Roland Wismüller, Vaidy Sunderam, Arndt Bode: OMIS — On-line Monitoring Interface Specification

342/06/96 A Ekkart Kindler: A Compositional Partial Order Semantics for Petri Net Components

342/07/96 A Richard Mayr: Some Results on Basic Parallel Processes

342/08/96 A Ralph Radermacher, Frank Weimer: INSEL Syntax-Bericht

342/09/96 A P.P. Spies, C. Eckert, M. Lange, D. Marek, R. Radermacher, F. Weimer, H.-M. Windisch: Sprachkonzepte zur Konstruktion verteilter Systeme

342/10/96 A Stefan Lamberts, Thomas Ludwig, Christian Röder, Arndt Bode: PFSLib – A File System for Parallel Programming Environments

342/11/96 A Manfred Broy, Gheorghe Ştefănescu: The Algebra of Stream Processing Functions

342/12/96 A Javier Esparza: Reachability in Live and Safe Free-Choice Petri Nets is NP-complete

342/13/96 A Radu Grosu, Ketil Stølen: A Denotational Model for Mobile Many-to-Many Data-flow Networks

342/14/96 A Giannis Bozas, Michael Jaedicke, Andreas Listl, Bernhard Mitschang, Angelika Reiser, Stephan Zimmermann: On Transforming a Sequential SQL-DBMS into a Parallel One: First Results and Experiences of the MIDAS Project

Reihe A

342/15/96 A Richard Mayr: A Tableau System for Model Checking Petri Nets with a Fragment of the Linear Time μ -Calculus

342/16/96 A Ursula Hinkel, Katharina Spies: Anleitung zur Spezifikation von mobilen, dynamischen Focus-Netzen

342/17/96 A Richard Mayr: Model Checking PA-Processes

342/18/96 A Michaela Huhn, Peter Niebert, Frank Wallner: Put your Model Checker on Diet: Verification on Local States

342/01/97 A Tobias Müller, Stefan Lamberts, Ursula Maier, Georg Stellner: Evaluierung der Leistungsf"ahigkeit eines ATM-Netzes mit parallelen Programmierbibliotheken

342/02/97 A Hans-Joachim Bungartz and Thomas Dornseifer: Sparse Grids: Recent Developments for Elliptic Partial Differential Equations

342/03/97 A Bernhard Mitschang: Technologie f"ur Parallele Datenbanken - Bericht zum Workshop

342/04/97 A nicht erschienen

342/05/97 A Hans-Joachim Bungartz, Ralf Ebner, Stefan Schulte: Hierarchische Basen zur effizienten Kopplung substrukturierter Probleme der Strukturmechanik

342/06/97 A Hans-Joachim Bungartz, Anton Frank, Florian Meier, Tilman Neunhoeffer, Stefan Schulte: Fluid Structure Interaction: 3D Numerical Simulation and Visualization of a Micropump

342/07/97 A Javier Esparza, Stephan Melzer: Model Checking LTL using Constraint Programming

342/08/97 A Niels Reimer: Untersuchung von Strategien für verteiltes Last- und Ressourcenmanagement

342/09/97 A Markus Pizka: Design and Implementation of the GNU INSEL-Compiler gic

342/10/97 A Manfred Broy, Franz Regensburger, Bernhard Schätz, Katharina Spies: The Steamboiler Specification - A Case Study in Focus

342/11/97 A Christine Röckl: How to Make Substitution Preserve Strong Bisimilarity

342/12/97 A Christian B. Czech: Architektur und Konzept des Dycos-Kerns

342/13/97 A Jan Philipps, Alexander Schmidt: Traffic Flow by Data Flow

342/14/97 A Norbert Fröhlich, Rolf Schlagenhaft, Josef Fleischmann: Partitioning VLSI-Circuits for Parallel Simulation on Transistor Level

342/15/97 A Frank Weimer: DaViT: Ein System zur interaktiven Ausführung und zur Visualisierung von INSEL-Programmen

342/16/97 A Niels Reimer, Jürgen Rudolph, Katharina Spies: Von FOCUS nach INSEL - Eine Aufzugssteuerung

342/17/97 A Radu Grosu, Ketil Stølen, Manfred Broy: A Denotational Model for Mobile Point-to-Point Data-flow Networks with Channel Sharing

342/18/97 A Christian Röder, Georg Stellner: Design of Load Management for Parallel Applications in Networks of Heterogenous Workstations

Reihe A

342/19/97 A Frank Wallner: Model Checking LTL Using Net Unfoldings

342/20/97 A Andreas Wolf, Andreas Kmoch: Einsatz eines automatischen Theorembeweisers in einer taktikgesteuerten Beweisumgebung zur Lösung eines Beispiels aus der Hardware-Verifikation – Fallstudie –

342/21/97 A Andreas Wolf, Marc Fuchs: Cooperative Parallel Automated Theorem Proving

342/22/97 A T. Ludwig, R. Wismüller, V. Sunderam, A. Bode: OMIS - On-line Monitoring Interface Specification (Version 2.0)

342/23/97 A Stephan Merkel: Verification of Fault Tolerant Algorithms Using PEP

342/24/97 A Manfred Broy, Max Breitling, Bernhard Schätz, Katharina Spies: Summary of Case Studies in Focus - Part II

342/25/97 A Michael Jaedicke, Bernhard Mitschang: A Framework for Parallel Processing of Aggregat and Scalar Functions in Object-Relational DBMS

342/26/97 A Marc Fuchs: Similarity-Based Lemma Generation with Lemma-Delaying Tableau Enumeration

342/27/97 A Max Breitling: Formalizing and Verifying TimeWarp with FOCUS

342/28/97 A Peter Jakobi, Andreas Wolf: DBFW: A Simple DataBase Frame-Work for the Evaluation and Maintenance of Automated Theorem Prover Data (incl. Documentation)

342/29/97 A Radu Grosu, Ketil Stølen: Compositional Specification of Mobile Systems

342/01/98 A A. Bode, A. Ganz, C. Gold, S. Petri, N. Reimer, B. Schiemann, T. Schnekenburger (Herausgeber): "'Anwendungsbezogene Lastverteilung"', ALV'98

342/02/98 A Ursula Hinkel: Home Shopping - Die Spezifikation einer Kommunikationsanwendung in FOCUS

342/03/98 A Katharina Spies: Eine Methode zur formalen Modellierung von Betriebssystemkonzepten

342/04/98 A Stefan Bischof, Ernst-W. Mayr: On-Line Scheduling of Parallel Jobs with Runtime Restrictions

342/05/98 A St. Bischof, R. Ebner, Th. Erlebach: Load Balancing for Problems with Good Bisectors and Applications in Finite Element Simulations: Worst-case Analysis and Practical Results

342/06/98 A Giannis Bozas, Susanne Kober: Logging and Crash Recovery in Shared-Disk Database Systems

342/07/98 A Markus Pizka: Distributed Virtual Address Space Management in the MoDiS-OS

342/08/98 A Niels Reimer: Strategien für ein verteiltes Last- und Ressourcenmanagement

342/09/98 A Javier Esparza, Editor: Proceedings of INFINITY'98

342/10/98 A Richard Mayr: Lossy Counter Machines

342/11/98 A Thomas Huckle: Matrix Multilevel Methods and Preconditioning

Reihe A

342/12/98 A Thomas Huckle: Approximate Sparsity Patterns for the Inverse of a Matrix and Preconditioning

342/13/98 A Antonin Kucera, Richard Mayr: Weak Bisimilarity with Infinite-State Systems can be Decided in Polynomial Time

342/01/99 A Antonin Kucera, Richard Mayr: Simulation Preorder on Simple Process Algebras

342/02/99 A Johann Schumann, Max Breitling: Formalisierung und Beweis einer Verfeinerung aus FOCUS mit automatischen Theorembeweisern – Fallstudie –

342/03/99 A M. Bader, M. Schimper, Chr. Zenger: Hierarchical Bases for the Indefinite Helmholtz Equation

342/04/99 A Frank Strobl, Alexander Wisspeintner: Specification of an Elevator Control System

342/05/99 A Ralf Ebner, Thomas Erlebach, Andreas Ganz, Claudia Gold, Clemens Harlfinger, Roland Wism"uller: A Framework for Recording and Visualizing Event Traces in Parallel Systems with Load Balancing

342/06/99 A Michael Jaedicke, Bernhard Mitschang: The Multi-Operator Method: Integrating Algorithms for the Efficient and Parallel Evaluation of User-Defined Predicates into ORDBMS

342/07/99 A Max Breitling, Jan Philipps: Black Box Views of State Machines

342/08/99 A Clara Nippl, Stephan Zimmermann, Bernhard Mitschang: Design, Implementation and Evaluation of Data Rivers for Efficient Intra-Query Parallelism

342/09/99 A Robert Sandner, Michael Mauderer: Integrierte Beschreibung automatisierter Produktionsanlagen - eine Evaluierung praxisnaher Beschreibungstechniken

342/10/99 A Alexander Sabbah, Robert Sandner: Evaluation of Petri Net and Automata Based Description Techniques: An Industrial Case Study

342/01/00 A Javier Esparza, David Hansel, Peter Rossmanith, Stefan Schwoon: Efficient Algorithm for Model Checking Pushdown Systems

342/02/00 A Barbara König: Hypergraph Construction and Its Application to the Compositional Modelling of Concurrency

342/03/00 A Max Breitling and Jan Philipps: Verification Diagrams for Dataflow Properties

SFB 342 : Methoden und Werkzeuge für die Nutzung paralleler Rechnerarchitekturen

Reihe B

342/1/90 B Wolfgang Reisig: Petri Nets and Algebraic Specifications

342/2/90 B Jörg Desel: On Abstraction of Nets

342/3/90 B Jörg Desel: Reduction and Design of Well-behaved Free-choice Systems

342/4/90 B Franz Abstreiter, Michael Friedrich, Hans-Jürgen Plewan: Das Werkzeug runtime zur Beobachtung verteilter und paralleler Programme

342/1/91 B Barbara Paech1: Concurrency as a Modality

342/2/91 B Birgit Kandler, Markus Pawlowski: SAM: Eine Sortier- Toolbox -Anwenderbeschreibung

342/3/91 B Erwin Loibl, Hans Obermaier, Markus Pawlowski: 2. Workshop über Parallelisierung von Datenbanksystemen

342/4/91 B Werner Pohlmann: A Limitation of Distributed Simulation Methods

342/5/91 B Dominik Gomm, Ekkart Kindler: A Weakly Coherent Virtually Shared Memory Scheme: Formal Specification and Analysis

342/6/91 B Dominik Gomm, Ekkart Kindler: Causality Based Specification and Correctness Proof of a Virtually Shared Memory Scheme

342/7/91 B W. Reisig: Concurrent Temporal Logic

342/1/92 B Malte Grosse, Christian B. Suttner: A Parallel Algorithm for Set-of-Support
Christian B. Suttner: Parallel Computation of Multiple Sets-of-Support

342/2/92 B Arndt Bode, Hartmut Wedekind: Parallelrechner: Theorie, Hardware, Software, Anwendungen

342/1/93 B Max Fuchs: Funktionale Spezifikation einer Geschwindigkeitsregelung

342/2/93 B Ekkart Kindler: Sicherheits- und Lebendigkeitseigenschaften: Ein Literaturüberblick

342/1/94 B Andreas Listl; Thomas Schnekenburger; Michael Friedrich: Zum Entwurf eines Prototypen für MIDAS